МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЬЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №3 по курсу объектно-ориентированное программирование I семестр, 2019/20 уч. год

Студент *Попов Данила Андреевич, группа 08-208Б-18*

Преподаватель *Журавлёв Андрей Андреевич*

Условие

Задание №1: написать программу, которая реализует работу с фигурами:

1. Вставка
2. Удаление
3. Печать фигуры
4. Печать всех фигур

Описание программы

Код программы состоит из 5-ти файлов:

1. app/main.cpp: исходный код с точкой входа
2. src/figure.hpp: объявление и реализация generic структур
3. src/figure.cpp: реализация функций
4. src/point.hpp: объявление и реализация generic структур
5. src/point.cpp: реализация функций

Выводы

Узнавать тонкости выполнения работы и не делать в одно утро.

Исходный код

figure.hpp

```cpp
#pragma once

#include <utility>
#include <tuple>
#include <vector>
#include <iostream>

#include "point.hpp"

auto constexpr CONST_PI = 3.141592653589793238462643383279502884L;

template<typename _Point>
struct figure {
    using point         = _Point;
    using pointer       = point*;
    using const_pointer = const point*;

    using iterator       = pointer;
    using const_iterator = const_pointer;

    virtual void rotate(point vertex, double angle) = 0;

    /* Returns center of figure
        Has no overflow guard
    */
    virtual point center() const {
        constexpr size_t point_size = point::size();

        auto   begin = this->begin();
        auto   end   = this->end();
        size_t size  = this->size();

        point result = *begin;
        for (auto it = begin + 1; it < end; ++it) {
            for (size_t i = 0; i < point_size; ++i) {
                result[i] += (*it)[i];
            }
        }
        for (size_t i = 0; i < point_size; ++i) {
            result[i] /= size;
        }

        return result;
    }

    virtual double square() {
        auto constexpr x = 0;
        auto constexpr y = 1;

        auto begin = this->begin();
        auto end = this->end();

        double result = 0.0;

        for (auto it = begin + 1; it < end - 1; ++it) {
            result += (*it)[x] * ((*(it + 1))[y] - (*(it - 1))[y]);
        }
        auto first = begin;
        auto last = end - 1;
        result += (*first)[x] * ((*(first + 1))[y] - (*last)[y]);
        result += (*last)[x] * ((*first)[y] - (*(last - 1))[y]);
        result /= 2;

        return result;
    }

    virtual iterator begin() = 0;
    virtual const_iterator begin() const { return const_cast<figure&>(*this).begin(); }

    virtual iterator end() = 0;
    virtual const_iterator end() const { return const_cast<figure&>(*this).end(); }

    size_t size() const {
        return size_t(end() - begin());
    }
};

template<typename _Type>
std::ostream& operator<<(std::ostream& stream, const figure<_Type>& fig) {
    stream << "\"" << typeid(fig).name() << "\":{ ";
    for (const auto& p : fig) {
        stream << p << " ";
    }
    stream << "}";

    return stream;
}

template<typename _Type>
std::istream& operator<<(std::istream& stream, figure<_Type>& fig) {
    for (auto& p : fig) {
        stream >> p;
    }
    return stream;
}

// Examples:
void rotate(figure<point2d>& fig, point2d vertex, double phi);

struct rhombus final : public figure<point2d> {
    using point    = figure::point;
    using iterator = figure::iterator;

    point points[4];

    rhombus()
        : points{ 0 }
    {}

    rhombus(std::istream& stream, double precision = 0.000000001) {
        for (auto& p : points) {
            stream >> p;
        }
        if (stream.fail()) {
            return;
        }

        double dist = distance(points[0], points[3]);
        for (size_t i = 0; i < 3; i++) {
            double next = distance(points[i], points[i + 1]);
            if (std::abs(dist - next) > precision) {
                stream.setstate(std::ios::failbit);
                break;
            }
        }
    }

    rhombus(point2d center, double horizontal, double vertical) {
        constexpr size_t x = 0;
        constexpr size_t y = 1;

        points[0] = { center[x], center[y] + vertical / 2 };
        points[1] = { center[x] - horizontal / 2, center[y] };
        points[2] = { center[x], center[y] - vertical / 2 };
        points[3] = { center[x] + horizontal / 2, center[y] };
    }

    void rotate(point vertex, double angle) override {
        ::rotate(*this, vertex, angle);
    }

    iterator begin() override {
        return &points[0];
    }

    iterator end() override {
        return &points[sizeof(points) / sizeof(*points)];
    }
};

std::istream& operator>>(std::istream& stream, rhombus& fig);

template<size_t _Num>
struct ngon final : figure<point2d> {
    using point = figure::point;
    using iterator = figure::iterator;

    point points[_Num];

    ngon()
        : points{ 0 }
    {}

    ngon(std::istream& stream) {
        for (auto& p : points) {
            stream >> p;
        }
    }

    ngon(point center, double radius)
        : points{ 0 } {
        auto constexpr x = 0;
        auto constexpr y = 1;

        double angle = 0.0;
        double phi = (2 * CONST_PI) / _Num;

        for (auto& p : points) {
            p = rotate_point2d({ center[x], center[y] + radius }, center, angle);
            angle += phi;
        }
    }

    void rotate(point vertex, double angle) override {
        ::rotate(*this, vertex, angle);
    }

    iterator begin() override {
        return &points[0];
    }

    iterator end() override {
        return &points[sizeof(points) / sizeof(*points)];
    }
};

template<size_t _Num>
std::istream& operator>>(std::istream& stream, ngon<_Num>& fig) {
    double x, y, r;
    stream >> x >> y >> r;
    fig = ngon<_Num>({ x, y }, r);

    return stream;
}

using pentagon = ngon<5>;
using hexagon = ngon<6>;
```

figure.cpp

```cpp
#include "figure.hpp"

void rotate(figure<point2d>& fig, point2d vertex, double phi) {
    for (auto& p : fig) {
        p = rotate_point2d(p, vertex, phi);
    }
}

std::istream& operator>>(std::istream& stream, rhombus& fig) {
    double x, y, h, v;
    stream >> x >> y >> h >> v;
    fig = rhombus({ x, y }, h, v);

    return stream;
}
```

point.hpp

```
#pragma once

#include <iostream>
#include <cstddef>
#include <cassert>
#include <cmath>

template<typename _Type, size_t _Dimensions>
struct point {
    static_assert(_Dimensions != 0, "can not create 0d point");

    using type            = _Type;
    using reference       = type&;
    using const_reference = const type&;
    using pointer         = type*;
    using const_pointer   = const type*;

    using iterator       = pointer;
    using const_iterator = const_pointer;

    type dots[_Dimensions];

    type& operator[](size_t ix) noexcept {
        return dots[ix];
    }

    const type& operator[](size_t ix) const noexcept {
        return const_cast<point&>(*this).operator[](ix);
    }

    iterator begin() noexcept {
        return &dots[0];
    }

    const_iterator begin() const noexcept {
        return const_cast<point&>(*this).begin();
    }

    iterator end() noexcept {
        return &dots[_Dimensions];
    }
    const_iterator end() const noexcept {
        return const_cast<point&>(*this).end();
    }

    static constexpr size_t size() noexcept {
        return _Dimensions;
    }

    point operator+(const point& other) {
        point result = *this;

        for (size_t i = 0; i < result.size(); i++) {
            result[i] += other[i];
        }

        return result;
    }

    point operator-(const point& other) {
        point result = *this;

        for (size_t i = 0; i < result.size(); i++) {
            result[i] -= other[i];
        }

        return result;
    }
};

template<typename _Type, size_t _Dims>
std::ostream& operator<<(std::ostream& stream, const point<_Type, _Dims>& p) {
    stream << "{ ";
    for (const auto& d : p) {
        stream << d << " ";
    }
    stream << "}";

    return stream;
}

template<typename _Type, size_t _Dims>
```

```
std::istream& operator>>(std::istream& stream, point<_Type, _Dims>& p) {
    for (auto& d : p) {
        stream >> d;
    }

    return stream;
}

// Examples:
using point2d = point<double, 2>;

point2d rotate_point2d(point2d p, point2d vertex, double phi);
inline double distance(const point2d& left, const point2d& right) {
    double x = left[0] - right[0];
    double y = left[1] - right[1];
    return std::sqrt((x * x) + (y * y));
}
```

point.cpp

```
#include "point.hpp"

point2d rotate_point2d(point2d p, point2d vertex, double phi) {
    auto constexpr x = 0;
    auto constexpr y = 1;

    point2d vector = p - vertex;
    vector = {
        vector[x] * std::cos(phi) - vector[y] * std::sin(phi),
        vector[x] * std::sin(phi) + vector[y] * std::cos(phi)
    };

    return vector + vertex;
}
```