Правила оформления кода для контеста по геометрии

28 июля 2022 г.

1 Обязательные требования

1.1

Хранение и работа с основными геометрическими объектами должна производиться через классы ниже:

```
struct point {
    int x;
    int y;
};

struct line {
    int a;
    int b;
    int c;
};
```

Причем point должен использоваться как для точек, так и для векторов. Не стоит разделять их.

1.2

Большинство задач предлагает вам либо посчитать какую-то величину (длину вектора, расстояние между объектами), либо проверить какой-то предикат (точка лежит на отрезке, отрезки пересекаются) и многие из этих задач зависимы друг от друга. Поэтому требуется писать такой код, который вам самим будет проще использовать в других задачах. Каждое из таких вычислений следует оборачивать в отдельную функцию с понятным названием.

Примеры, расстояние от точки до прямой:

1. Минимально приемлимый вариант

```
long double distance_point_to_line(point p, line l) {
    return abs(l.a * p.x + l.b * p.y + l.c)
        /  sqrtl(l.a * l.a + l.b * l.b);
}
```

2. Вынесем подстановку точки в прямую, как оператор

```
struct line {
    int a;
    int b;
    int c;

    //Переопределим оператор "Круглые скобки"
    //Во избежание ненужных копирований точку принимаем в виде const ссылки
    /*
    const после объявления метода говорит, что метод никак не изменяет объект
    Без этого оператор можно было бы вызывать лишь у не const объектов,
    и функция ниже бы не скомпилировалась
    */
    int operator()(const point& p) const {
        return a * p.x + b * p.y + c;
    }
};

//Можно использовать сокращения, но название все еще должно быть читаемым
long double dist_point_line(const point& p, const line& l) {
    return abs(l(p)) / sqrtl(l.a * l.a + l.b * l.b);
}
```

3. Выделим у прямой нормаль, а у вектора реализуем взятие длины

```
struct point {
    int x;
    int y;

    //Длина вектора
    long double length() const {
        return sqrtl(x * x + y * y);
    }

    //Скалярное произведение
    int operator*(const point& p) const {
        return x * p.x + y * p.y;
    }
};

struct line {
    //Теперь первые 2 коэффициента храним в виде вектора
    point norm;
    int c;

    //Но можно сделать для них геттеры
    int a() const {
        return norm.a;
    }

    /*
    Данный метод, в отличие от предыдущего, возвращает изменяемую ссылку
    Благодаря этому можно делать, например, так:
    line l;
    l.a() = 1;
    */
    int& a() {
        return norm.a;
    }

    int b() const {
        return norm.b;
    }

    int& b() {
        return norm.b;
    }

    int operator()(const point& p) const {
        return norm * p + c;
    }
};

long double dist_point_line(const point& p, const line& l) {
    return abs(l(p)) / l.norm.length();
}
```

1.3

Минимизируйте использование глобальных переменных. Передавайте все, что нужно, в качестве аргументов функций. Вывод ответа делайте только в main.
Примеры, как делать не надо:

1. Непонятное название, функцию нельзя применить к другим данным

```
point p;
line l;
long double ans;

void otvet() {
    ans = abs(l.a * p.x + l.b * p.y + l.c)
        / sqrtl(l.a * l.a + l.b * l.b);
}

int main() {
    cin >> p.x >> p.y;
    cin >> l.a >> l.b >> l.c;

    otvet();

    cout << setprecision(10) << fixed << ans << '\n';

    return 0;
}
```

2. Функцию невозможно применить для промежуточных вычислений

```
void dist(point p, line l) {
    long double ans = abs(l.a * p.x + l.b * p.y + l.c)
        / sqrtl(l.a * l.a + l.b * l.b);
    cout << setprecision(10) << fixed << ans << '\n';
}
```

1.4

Используйте написанные ранее функции при написании новых.
Пример написания лишнего кода:

```
long double dist_point_line_points(const point& p, const point& a, const point& b) {
    line l(a, b); //Конструктор от двух точек
    return abs(l.a * p.x + l.b * p.y + l.c)
        /  sqrtl(l.a * l.a + l.b * l.b); //Копипаста!!!
    /*
    Можно заменить на
    return dist_point_line(p, l);
    */
}

long double dist_point_ray(const point& p, const point& a, const point& b) {
    point ab = b - a; //Разность тоже можно перегрузить
    point ap = p - a;
    //Это еще и к тому, почему удобнее не разделять точки и векторы
    if (ap * ab > 0) { //Скалярное произведение
        line l(a, b);
        return abs(l.a * p.x + l.b * p.y + l.c)
            /  sqrtl(l.a * l.a + l.b * l.b); //Копипаста!!!
        /*
        Можно заменить на
        return dist_point_line_points(p, a, b);
        */
    } else {
        return ap.length();
    }
}
```

В итоге формула подсчета расстояния от точки до прямой будет написана в коде вместо трех раз, всего один.

2 Дополнительные советы и пожелания

2.1

Избегайте ненужного перехода к вещественным числам, если не хотите потом страдать из-за точности. Большинство задач вычислительной геометрии, не включая само вычисление расстояний и углов, решается в интах и лонгах при интовых входных данных.
Храните точки и прямые в интовых координатах, если нет крайней необходимости в обратном.
Полезно помимо функции длины вектора сделать также и функцию квадрата его длины, так как данная величина целая, и в некоторых задачах достаточно и этого.

2.2

Мы не требуем изучения шаблонных классов, но часто бывает, что в разных задачах хочется резко поменять все инты в координатах и функциях на лонги или даблы. Возможность такой модицикации можно добавить довольно просто:

```
using geom_num = int;
//или typedef int geom_num;
//typedef - это стиль C, а в C++ принято все же использовать using

struct point {
    /*
    Теперь geom_num можно использовать во всех местах,
    где может резко понадобиться сменить тип
    */
    geom_num x;
    geom_num y;
};
```

2.3

Эталонное решение задачи: расстояние от точки до прямой, заданной двумя точками:

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

using gtype = int;
using ld = long double;

struct point{
    gtype x;
    gtype y;

    //Несколько стандартных конструкторов, про которые можно погуглить
    point(const point& rhs) = default;
    point(point&& rhs) = default;
    point& operator=(const point& rhs) = default;
    point& operator=(point&& rhs) = default;

    //Конструктор от двух координат (по умолчанию, нулевых)
    explicit point(gtype x = 0, gtype y = 0):
            x(x),
            y(y) {}

    //Вся векторная арифметика
    //Сложение
    point operator+(const point& p) const {
        return point(x + p.x, y + p.y);
    }

    point& operator+=(const point& p) {
        x += p.x;
        y += p.y;
        return *this;
    }

    //Вычитание
    point operator-(const point& p) const {
        return point(x - p.x, y - p.y);
    }

    point& operator-=(const point& p) {
        x -= p.x;
        y -= p.y;
        return *this;
    }

    //Унарный минус
    point operator-() const {
        return point(-x, -y);
    }
```

```
//Умножение на число
point operator*(gtype k) const {
    return point(x * k, y * k);
}

point& operator*=(gtype k) {
    x *= k;
    y *= k;
    return *this;
}

//Деление на число
point operator/(gtype k) const {
    return point(x / k, y / k);
}

point& operator/=(gtype k) {
    x /= k;
    y /= k;
    return *this;
}

//Скалярное произведение
gtype operator*(const point& p) const {
    return x * p.x + y * p.y;
}

//Косое произведение
gtype operator%(const point& p) const {
    return x * p.y - y * p.x;
}

//Операторы сравнения (как у пары)
bool operator==(const point& a) const {
    return x == a.x && y == a.y;
}

bool operator!=(const point& a) const {
    return x != a.x || y != a.y;
}

bool operator>(const point& a) const {
    return x > a.x || x == a.x && y > a.y;
}

bool operator>=(const point& a) const {
    return x > a.x || x == a.x && y >= a.y;
}

bool operator<(const point& a) const {
    return x < a.x || x == a.x && y < a.y;
}

bool operator<=(const point& a) const {
    return x < a.x || x == a.x && y <= a.y;
}
```

```
};

//Квадрат длины вектора
gtype sqr(const point& p) {
    return p * p;
}

//Длина вектора
ld len(const point& p) {
    return sqrtl(sqr(p));
}

//Операторы ввода и вывода для точки
istream& operator>>(istream& in, point& p) {
    in >> p.x >> p.y;
    return in;
}

ostream& operator<<(ostream& out, const point& p) {
    out << p.x << ' ' << p.y;
    return out;
}

struct line {
    gtype a;
    gtype b;
    gtype c;

    line(const line& rhs) = default;
    line(line&& rhs) = default;
    line& operator=(const line& rhs) = default;
    line& operator=(line&& rhs) = default;

    //Конструктор от коэффициентов
    explicit line(gtype a = 0, gtype b = 0, gtype c = 0):
            a(a),
            b(b),
            c(c) {}

    //Конструктор от двух точек
    line(const point& p1, const point& p2):
            a(p2.y - p1.y),
            b(p1.x - p2.x),
            c(p2.x * p1.y - p1.x * p2.y) {}

    //Подстановка точки в прямую
    gtype operator()(const point& p) const {
        return a * p.x + b * p.y + c;
    }
};

istream& operator>>(istream& in, line& l) {
    in >> l.a >> l.b >> l.c;
    return in;
}
```

```
ostream& operator<<(ostream& out, const line& l) {
    out << l.a << ' ' << l.b << ' ' << l.c;
    return out;
}

//Основной код начинается здесь
ld dist_point_line(const point& p, const line& l) {
    return abs(l(p)) / sqrtl(l.a * l.a + l.b * l.b);
}

ld dist_point_line(const point& p, const point& a, const point& b) {
    return dist_point_line(p, line(a, b));
}

int main() {
    point p, a, b;
    cin >> p >> a >> b;

    cout << setprecision(10) << fixed << dist_point_line(p, a, b) << '\n';

    return 0;
}
```

И не пугайтесь размера шапки. Чем более подробно она написана, тем удобнее писать основной код.
Еще один вариант реализации шапки, уже с шаблонами:
https://github.com/Nartovdima/cp/blob/master/geometry/geom_header.cpp