



### Раздел «Язык Си» . OOP-struct\_contC :

- Сложные структуры в языке C:
  - Координаты
  -  Задачи 1
  - Треугольники.
  -  Задачи 2

### Сложные структуры в языке C:

Часто при реализации задачи удобно мыслить объектами, которые участвуют в постановке задачи.

Например, при решении различных геометрических задач удобно использовать именно геометрические объекты: точка, координаты, прямая, треугольник и т.д.

#### Маленькая задача "про треугольник".\*

Треугольники задаются координатами своих вершин на плоскости. Все координаты – целые числа.

Необходимо определить сколько различных треугольников задано на плоскости. (Треугольники считаются одинаковыми по геометрическому правилу сравнения треугольников).

Геометрическое решение задачи очень простое: нужно просто сравнить все стороны одного треугольника с другим. И так для всех треугольников, чтобы выяснить какие из них не равны.

Осознаем, что необходимо иметь и уметь для решения задачи "геометрически":

- для каждого треугольника нужно иметь все длины его сторон, для этого:
  - иметь координаты вершин,
  - вычислять расстояние между вершинами по координатам этих вершин;
- нужно уметь сравнивать треугольники по сторонам с наложением, отображением и поворотами.

### Координаты

Понятно, что сначала нужно научиться работать с координатами:

Опишем структуру **Coord** и примеры функций работы с ней.

Заметим, что и точек, и треугольников для нашей задачи нужно множество, поэтому следует рассмотреть пример использования массива структур.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// Предполагаем, что где-то на плоскости есть начало (0,0)
// структура, описывающая координаты точки на плоскости
typedef struct Crd{
    int x,y;
}Coord;

// Функция, которая двигет точку по плоскости
// по оси X на x и по оси Y на y.
// Возвращает новые координаты точки

Coord movePoint(Coord point, int x, int y){
    point.x += x;
    point.y +=y;
    return point;
};
```

### Поиск

### Раздел «Язык Си»

[Главная](#)  
[Зачем учить C?](#)  
[Определения](#)

#### Инструменты:

[Поиск](#)  
[Изменения](#)  
[Index](#)  
[Статистика](#)

### Разделы

[Информация](#)  
[Алгоритмы](#)  
[Язык Си](#)  
[Язык Ruby](#)  
[Язык](#)  
[Ассемблера](#)  
[El Judge](#)  
[Парадигмы](#)  
[Образование](#)  
[Сети](#)  
[Objective C](#)

[Login>>](#)

```
// Печать координат
void printPoint(Coord point){
    printf("(%d,%d)\n",point.x, point.y);
};

int main(){
    /* Описываем массив из трех точек:
    Сразу инициализируем (по два числа в фигурных скобках
    - значения для каждого элемента)
    А можно и так задавать значения:
    points[0].x = 3;
    points[0].y = 5;
    и т.д.
    */
    Coord points[3]={3,10},{2,5},{0,0}};
    int i;

    // Двигаем все точки на (10,10):

    for(i = 0; i < 3; i++){
        points[i] = movePoint(points[i],10,10);
    }

    // Печатаем все новые координаты точек:
    for( i = 0; i < 3; i++)
        printPoint(points[i]);

    return 0;
}
```

## Задачи 1

1. Реализовать функцию подсчета квадрата расстояния между точками

```
// иногда разумно использовать именно квадрат (не извлекать корень)
// для точности сравнений и экономии вычислений
int sizePoint(Coord a, Coord b);
```

2. Реализовать функции, описанные ниже при этом использовать функцию из предыдущей задачи:

```
//структура, описывающая координаты точки на плоскости
typedef struct Crd{
    int x,y;
}Coord;

// Структура для хранения расстояния между точками:
typedef struct{
    int *m; // динамический массив для хранения всех расстояний
    int n; // количество расстояний
}Distances;

// Заполнение структуры для хранения расстояний:
Distances sizeFill(Coord* points, int k){ // k - количество точек
// Описание объекта для Distances:
    Distances ds;
// Выделение динамической памяти для расстояний:
    ds.m = (int*) calloc(n, sizeof(int)); // - память для int
    ds.n = n; // - количество расстояний

// Здесь написать нужный код:

    return ds;
};

// Печать всех расстояний:
```

```

void printD(Distances d){

// Здесь нужно написать код для печати всех расстояний.

};

// Поиск минимального расстояния:
int minDistance(Distances ds){

// Здесь написать нужный код:

};

// Пример использования этих функций
int main(){

// Массив точек:
Coord points[3];
int i, x, y;
int min;

// Расстояния между точками:
Distances ds;

// Прочитать значения координат
for(i = 0; i < 3; i++){
    scanf("%d%d", &x,&y);
    points[i].x = x;
    points[i].y = y;
}

// Заполнение структуры расстояний:
ds = sizeFill(points, 3);

// печать всех расстояний:
printD(ds);

// получение минимального расстояния:
min = minDistance(ds);

}

```

## Треугольники.

Теперь можно заняться треугольниками. Вспомним о чем мечталось при постановке задачи. У каждого треугольника должны быть:

- три вершины, которые задаются координатами
- длины сторон треугольника
- полезно еще иметь заранее вычисленную площадь (на всякий случай).

Чтобы само решение совсем выглядело прстым хочется иметь функции, полезные для решения задачи:

- для заполнения данными переменных-треугольников.
- для сравнения треугольников

Вообще треугольник – полноценный участник решения, а раз так, назовем его **объект**

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// Вот это мы уже умеем :)
//структура, описывающая координаты точки на плоскости
typedef struct Crd{
    int x,y;
}Coord;

// Треугольник на плоскости задается координатами
// своих вершин

```

```
typedef struct Tr{
    Coord vertices[3]; // используем координаты
    int lenSQ[3]; // длины сторон
    // можно добавить и другие свойства,
    // чтобы лишний раз их не вычислять
    float square;
}Triangle;

// "Шплинт шпонки шпульки шпинделя моталки" :)
// или
// "на высоком дубу - ларец, в ларце - заяц,
// в зайце - утка, в утке - яйцо, в яйце - игла ..."

// Печать атрибутов треугольника:

void printTriangle(Triangle tri){
    int i;
    printf("{");
    // Печатаем координаты всех вершин по-очереди:
    for (i = 0; i < 3; i++){
        // Обращение к треугольнику, в нем к вершине с нужным номером,
        // в вершине - к координате ...
        // tri.vertices[i].x
        printf("(%d,%d)", tri.vertices[i].x, tri.vertices[i].y);
    }
    printf("}\n");
};

// Установка параметров треугольника:
// можно возвращать объект треугольник,
// можно передавать как параметр указатель на треугольник,
// можно возвращать указатель на треугольник

// Функция возвращает объект треугольник:

Triangle fillTri_1(Coord *vert){

    // Прежде чем отдать объект, его нужно иметь
    Triangle tmp; // временный треугольник

    int i;

    for (i = 0 ; i < 3; i++){
        tmp.vertices[i].x = vert[i].x;
        tmp.vertices[i].y = vert[i].y;
    }

    // А площадь нам лень подсчитывать :)

    // вернули треугольничек (сомнительный)
    return tmp;

};

/*
В первой функциях тругольник - " совсем сомнительный", потому что
треугольник может существовать не всегда (см. учебник по геометрии)
или массив с точками также может быть неудачный.
Полноценная функция должна проверять эти случаи.
*/

// Функция получает указатель на объект треугольник:
// за существование треугольника отвечает пользователь функции
// Если треугольник можно безопасно записать данными -
// функция возвращает 1, если что-то не так - 0.

int fillTri2(Triangle* tri, Coord *vert){
    int i;
```

```
// Проверяем можем ли использовать значения из переданных
// параметров:
if(vert == 0) // могут быть и другие критерии
    return 0;

for (i = 0 ; i < 3; i++){
/*
Если есть указатель на структурную переменную (объект),
то обращение к полям "через стрелочку":
    tri->square = 22.12;

Здесь мы также имеем указатель и обращаемся к полям:
*/
    tri->vertices[i].x = vert[i].x;
    tri->vertices[i].y = vert[i].y;
}

// Возвращаем 1. Хотя треугольник тоже получается
// "сомнительный".
return 1;
};

/*
В первых двух функциях треугольники - "сомнительные", потому что
массив с точками может быть неудачным, так что треугольник окажется
вырожденным (см. учебник по геометрии)

Во втором случае переменная tri не используется в функции, при
возвращении 0. Значит поля переменной будут заполнены "мусором".
То есть значения переменной не предсказуемы.
Очень "сомнительный треугольник".
Полноценная функция должна проверять и учитывать все эти случаи.
*/

// Функция возвращает треугольник, память на который выделяется динамически.
// Если треугольник не может иметь такие параметры, то вместо него
// возвращаемый указатель принимает значение 0

Triangle* createTri(Coord *vert){
    int i;
    // Прежде чем что-то отдать, это нужно создать
    Triangle *tmp;

    // Проверяем можем ли использовать значения из переданных
    // параметров:
    if(vert == 0) // а могут быть и другие критерии
        return 0;
    // Здесь предлагается написать самостоятельный код
    // для проверки существования треугольника (не вырожденный).

    /*
    Проверяем ...
    */

    // Треугольнику нужна память.
    tmp = (Triangle*) malloc(sizeof(Triangle));

    for (i = 0 ; i < 3; i++){
        tmp->vertices[i].x = vert[i].x;
        tmp->vertices[i].y = vert[i].y;
    }

    return tmp;
};
```

```

int main(){
// Массив точек. Данные могут лежать в файле
Coord points[18];

// Описываем массив из трех треугольников:
Triangle tri[3];

// А третий - вот такой:
Triangle *ptri;

int i, x, y;

// Заполняем массив точек:

for(i = 0; i < 9; i++){
    scanf("%d%d",&x, &y);
    points[i].x = x;
    points[i].y = y;
}

// _____ Создаем треугольник _____

// Первая функция:
tri[1] = fillTri_1(points);
printTriangle(tri[1]);

// Вторая функция:
// Очень удобно - можно проверить получился треугольник или нет!
if (fillTri2(tri,points + 3) == 0){
    printf("Не получился треугольник!! :(");
    exit(1);
};
printTriangle(tri[0]);

// Третья функция:
ptri = createTri(points + 6);

/* И здесь проверяем. Если что, указатель на объект будет равен 0
То есть даже память не выделится.
Даже если не писать проверку, то попытка использовать
такой треугольник вызовет крах программы.
Это очень печально :(, но зато не вводит в заблуждение
"мусорными" данными.
Становится понятно, что нужно принимать меры
*/
if (ptri == 0){
    printf("Не получился треугольник!! :(");
    exit(1);
}

printTriangle(*ptri);

// Можно сделать и так:
tri[2] = *ptri;

// Не нужен больше треугольник, освободи память.
free(ptri);
return 0;
}

```

## Задачи 2

1. В описанную выше функцию печати атрибутов треугольника добавить печать расстояний и площади
2. В функциях `int fillTri2(Triangle* tri, Coord* vert)` и `Triangle * createTri(Coord *vert)` добавить проверку на существование треугольника и подсчет площади. Рекомендуется использовать написанную для координат функцию вычисления расстояния между точками на плоскости.

3. Добавить функцию движения треугольника `_void moveTri(Triangle *tri, int x, int y)`, которая перемещает весь треугольник на `x` по оси `x` и на `y` по оси `y`.
4. Добавить функцию `int cmpTri(Triangle a, Triangle b)` для сравнения двух треугольников. Если треугольники равны, функция возвращает 1, если нет – 0.  
***Два треугольника считаются равными, если стороны этих треугольников попарно равны.***

(с) Материалы раздела "Язык Си" публикуются под лицензией GNU Free Documentation License.