

Estruturas e arrays de estruturas

Pedro Guerreiro

Universidade do Algarve

8 de março de 2021

As *estruturas* são o último ingrediente significativo que ainda nos falta na programação com C. Nas palavras da “bíblia” do C, o livro *The C Programming Language*, de Kernighan e Ritchie, “*a structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling*”.

Portanto, é simples: uma estrutura é um grupo de variáveis que achamos fazer sentido estarem juntas. “Fazer sentido” significa aqui que as estruturas simplificam a escrita dos programas e, também, (muito importante!), que nos ajudam a refletir sobre o problema que o programa pretende resolver.

A ideia de “estrutura” em programação é poderosa: uma *estrutura*, agrupando variáveis, é ela própria uma variável. Temos assim, um esquema recursivo. Quer dizer, algumas das variáveis agrupadas numa estrutura poderão ser elas próprias estruturas, com muita generalidade.

Uma vez que cada estrutura é uma variável, iremos atribuir-lhes um tipo explícito sistematicamente, usando o mecanismo das declarações `typedef`.

Estudemos este assunto, no contexto de um problema de programação.

Problema: ponto dentro de triângulo

Queremos um programa que, dados um triângulo e um ponto, verifica se o ponto está no interior do triângulo. Quando dispusermos de tipo `Triangle`, que represente triângulos, e de um tipo `Point`, que represente pontos, aquilo de que precisamos é de uma função com o seguinte protótipo:

```
int inside(Point p, Triangle t)
```

A função retornará 0, se o ponto não estiver dentro do triângulo, e 1, se estiver.

O problema do *ponto dentro do triângulo* é um problema clássico de programação geométrica.

Começemos por declarar o tipo `Point`. É uma estrutura que agrupa dois números, um para a coordenada x e outro para a coordenada y :

```
typedef struct {
    double x;
    double y;
} Point;
```

Dizemos que as variáveis `x` e `y` constituem os *membros da estrutura Point*.

Uma vez definida um tipo estrutura, como o tipo `Point`, é conveniente equipá-lo com algumas funções habituais. A primeira é o chamado *construtor*, que “constrói” um objeto de tipo `Point`. Tipicamente, damos ao construtor o nome do tipo, mas com minúscula inicial. Observe:

```
Point point(double x, double y)
{
    Point result = {x, y};
    return result;
}
```

Em funções de teste, havemos de querer consultar os valores dos campos da estrutura. Preparemos já duas funções para isso:

```
void point_print(Point p)
{
    printf("<%g %g>", p.x, p.y);
}

void point_println(Point p)
{
    point_print(p);
    printf("\n");
}
```

Observamos, na função `point_print` a forma de aceder aos membros da estrutura, usando o operador ponto `.`. Em geral, a escrita `x.y` representa o membro `x` da estrutura `y`. Recorde que uma estrutura é uma variável que agrupa variáveis.

Além dos construtores e das funções de escrita, haverá, para cada tipo estrutura, outras operações específicas do tipo. Por exemplo, para os pontos, podemos definir uma função para a distância entre dois pontos:

```
//Euclidean distance
double distance(Point p, Point q)
{
    return sqrt(dbl_square(p.x-q.x) + dbl_square(p.y-q.y));
}
```

Esta é a distância euclidiana. Em certos problemas de programação, usa-se a distância de Manhattan, assim definida:

```
//Manhattan distance
double distance_manhattan(Point p, Point q)
{
    return fabs(p.x-q.x) + fabs(p.y-q.y);
}
```

Mais um exemplo: uma função para verificar se três pontos são colineares. Recordemos da matemática que três pontos, (x_1, y_1) , (x_2, y_2) , (x_3, y_3) , são colineares se $(y_2-y_1)/(x_2-x_1) == (y_3-y_2)/(x_3-x_2)$, ou, com mais generalidade, evitando divisões por zero, se $(y_2-y_1) * (x_3-x_2) == (x_2-x_1) * (y_3-y_2)$. Programa-se assim:

```
int collinear(Point p, Point q, Point r)
{
    return (q.y-p.y) * (r.x-q.x) == (q.x-p.x) * (r.y-q.y);
}
```

Cada uma destas funções vem acompanhada da respetiva função de teste unitário, como de costume. Ei-las:

```
void unit_test_distance(void)
{
    Point p1 = point(2.0, 4.0);
    Point p2 = point(5.0, 8.0);
    assert(distance(p1, p2) == 5.0);
    Point p3 = point(0,0);
    Point p4 = point(1,1);
    assert(distance(p3, p4) == sqrt(2.0));
    assert(distance(p1, p3) == sqrt(20.0));
}
```

```

    assert(distance(p1, p1) == 0.0);
    assert(distance(p2, p4) == sqrt(65.0));
    Point p5 = point(1.5, 2.0);
    Point p6 = point(2.25, 3.0);
    assert(distance(p5, p6) == 1.25);
}

void unit_test_distance_manhattan(void)
{
    Point p1 = point(2.0, 4.0);
    Point p2 = point(5.0, 8.0);
    assert(distance_manhattan(p1, p2) == 7.0);
    Point p3 = point(0,0);
    Point p4 = point(1,1);
    assert(distance_manhattan(p3, p4) == 2.0);
    assert(distance_manhattan(p1, p3) == 6.0);
    assert(distance_manhattan(p1, p1) == 0.0);
    assert(distance_manhattan(p2, p4) == 11.0);
    Point p5 = point(1.5, 2.0);
    Point p6 = point(2.25, 3.0);
    assert(distance_manhattan(p5, p6) == 1.75);
}

void unit_test_collinear(void)
{
    Point p1 = point(2.0, 4.0);
    Point p2 = point(5.0, 8.0);
    Point p3 = point(8.0, 12.0);
    assert(collinear(p1, p2, p3));
    Point p4 = point(0.0, 0.0);
    Point p5 = point(1.0, 1.0);
    Point p6 = point(4.0, 4.0);
    assert(collinear(p4, p5, p6));
    Point p7 = point(1000.0, 4.0);
    assert(collinear(p1, p6, p7));
    assert(!collinear(p4, p1, p2));
}

```

Para exercitar estas funções, usamos a seguinte função de teste interativo:

```

void test_points(void)
{
    double x1, y1, x2, y2, x3, y3;
    while (scanf("%lf%lf%lf%lf%lf%lf", &x1, &y1, &x2, &y2, &x3,
&y3) != EOF)
    {

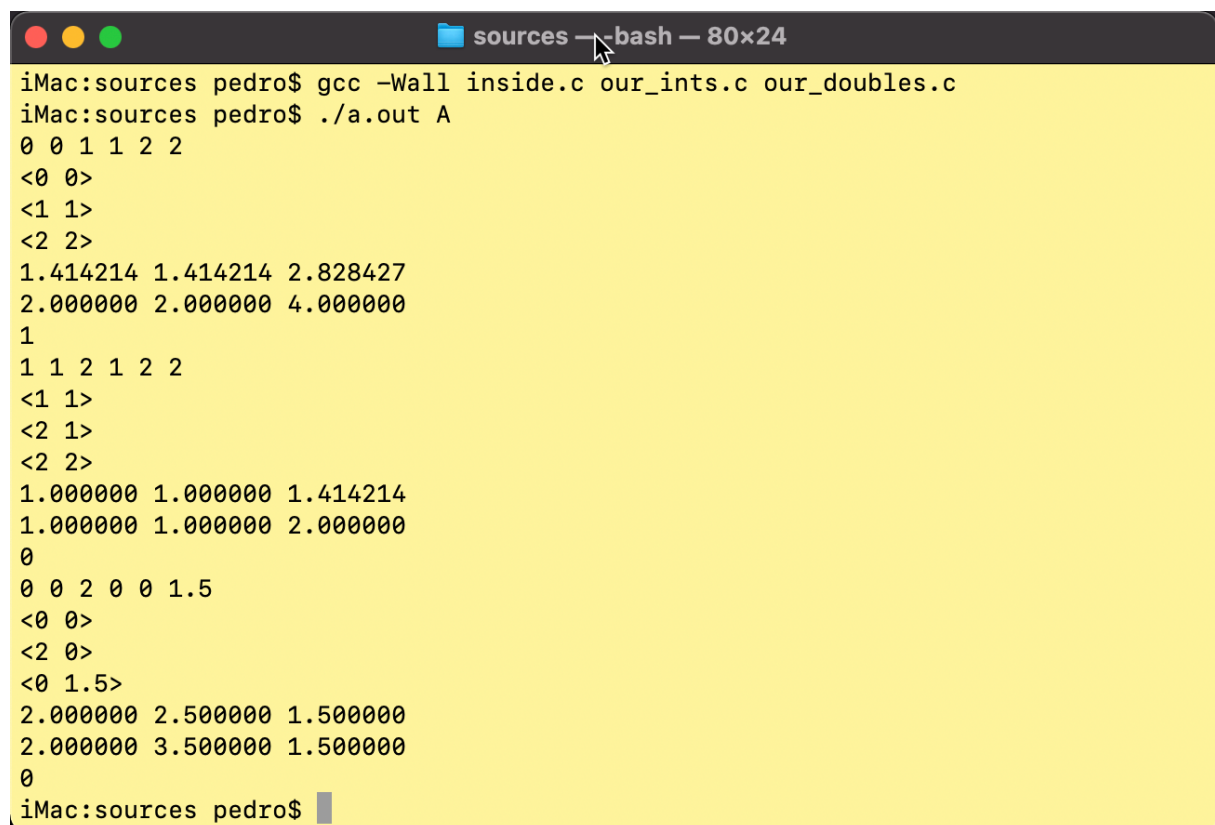
```

```

    Point p1 = point(x1, y1);
    Point p2 = point(x2, y2);
    Point p3 = point(x3, y3);
    point_println(p1);
    point_println(p2);
    point_println(p3);
    double d12 = distance(p1, p2);
    double d23 = distance(p2, p3);
    double d31 = distance(p3, p1);
    double dm12 = distance_manhattan(p1, p2);
    double dm23 = distance_manhattan(p2, p3);
    double dm31 = distance_manhattan(p3, p1);
    int c = collinear(p1, p2, p3);
    printf("%f %f %f\n", d12, d23, d31);
    printf("%f %f %f\n", dm12, dm23, dm31);
    printf("%d\n", c);
}
}

```

Eis o registo de uma sessão de teste:



```

sources — bash — 80x24
iMac:sources pedro$ gcc -Wall inside.c our_ints.c our_doubles.c
iMac:sources pedro$ ./a.out A
0 0 1 1 2 2
<0 0>
<1 1>
<2 2>
1.414214 1.414214 2.828427
2.000000 2.000000 4.000000
1
1 1 2 1 2 2
<1 1>
<2 1>
<2 2>
1.000000 1.000000 1.414214
1.000000 1.000000 2.000000
0
0 0 2 0 0 1.5
<0 0>
<2 0>
<0 1.5>
2.000000 2.500000 1.500000
2.000000 3.500000 1.500000
0
iMac:sources pedro$

```

Podemos usar de novo a técnica das estruturas para definir o tipo [Triangle](#):

```

typedef struct {
    Point v[3];

```

```
} Triangle;
```

Eis o construtor e as funções de escrita:

```
Triangle triangle(Point p, Point q, Point r)
{
    Triangle result = {p, q, r};
    return result;
}
```

```
void triangle_print(Triangle t)
{
    printf("[");
    for (int i = 0; i < 3; i++)
        point_print(t.v[i]);
    printf("]");
}
```

```
void triangle_println(Triangle t)
{
    triangle_print(t);
    printf("\n");
}
```

Vejamos agora algumas funções: o perímetro, a área e uma função para verificar se o triângulo é um triângulo retângulo. Para o perímetro baseamo-nos na distância entre pontos:

```
double perimeter(Triangle t)
{
    return
        distance(t.v[0], t.v[1]) +
        distance(t.v[1], t.v[2]) +
        distance(t.v[2], t.v[0]);
}
```

Para confirmar, a respetiva função de teste unitário:

```
void unit_test_perimeter(void)
{
    Triangle t1 = {{{0, 0}, {4, 0}, {0, 3}}};
    assert(perimeter(t1) == 12.0);
    Triangle t2 = {point(0.0, 0.0), point(0.5, 0.0), point(0.0,
0.375)};
    assert(perimeter(t2) == 1.5);
}
```

```

Triangle t3 = {point(10.0, 20.0), point(10.5, 20.0),
point(10.0, 20.375)};
assert(perimeter(t3) == 1.5);
}

```

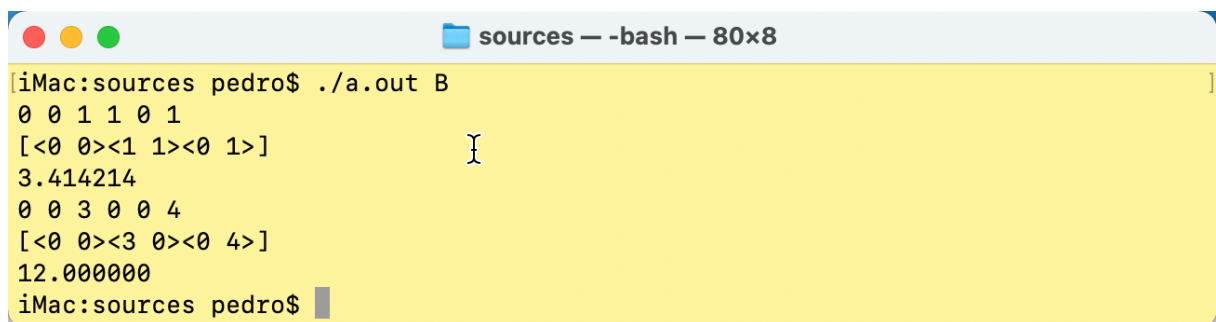
Experimentemos, com a seguinte função de teste, a qual, sucessivamente lê três pares de coordenadas, cria um triângulo, mostra o triângulo e calcula o perímetro:

```

void test_triangle(void)
{
    double p1x, p1y, p2x, p2y, p3x, p3y;
    while (scanf("%lf%lf%lf%lf%lf", &p1x, &p1y, &p2x, &p2y,
&p3x, &p3y) != EOF)
    {
        Triangle t = triangle(point(p1x, p1y),point(p2x,
p2y),point(p3x, p3y));
        triangle_println(t);
        double p = perimeter(t);
        printf("%f\n", p);
    }
}

```

Eis o resultado de uma sessão de teste com a função anterior:



```

iMac:sources pedro$ ./a.out B
0 0 1 1 0 1
[<0 0><1 1><0 1>]
3.414214
0 0 3 0 0 4
[<0 0><3 0><0 4>]
12.000000
iMac:sources pedro$

```

Para a área, temos a seguinte função, que calcula a área “com sinal”, recorrendo a uma fórmula menos conhecida, mas muito interessante:

```

double signed_area(Triangle t)
{
    return (
        t.v[0].x * (t.v[1].y - t.v[2].y) +
        t.v[1].x * (t.v[2].y - t.v[0].y) +
        t.v[2].x * (t.v[0].y - t.v[1].y)) / 2;
}

```

Repare na escrita `t.v[0].x`: `t` é a estrutura `Triangle`, que tem um só membro, o array `v`; `v[0]` é o primeiro elemento do array, e representa o primeiro vértice do triângulo; o vértice é uma estrutura de tipo `Point` e `x` é o primeiro membro da estrutura.

Eis a função de teste unitário:

```
void unit_test_signed_area(void)
{
    Triangle t1 = {{{0, 0}, {4, 0}, {0, 3}}};
    assert(signed_area(t1) == 6.0);
    Triangle t2 = {{{0, 0}, {0, 3}, {4, 0}}};
    assert(signed_area(t2) == -6.0);
    Triangle t3 = {{{1, 1}, {5, 1}, {3, 11}}};
    assert(signed_area(t3) == 20.0);
    Triangle t4 = {{{1, 1}, {3, 11}, {5, 1}}};
    assert(signed_area(t4) == -20.0);
}
```

Tal como a função de teste unitário ilustra, a área é positiva se os vértices estiverem no sentido contrário aos ponteiros do relógio e negativa se os vértices estiverem no sentido dos ponteiros do relógio.

Nota cultural: em vez de dizer “sentido dos ponteiros do relógio”, que, sendo inequívoco, é muito prolixo, pode dizer-se mais concisamente, “sentido dextrógiro”. E, em vez de “sentido contrário aos ponteiros do relógio”, pode dizer-se “sentido sinistrógiro”. Estas interessantes palavras, “dextrógiro” e “sinistrógiro”, são de origem latina e significam “que gira para a direita” e “que gira para a esquerda”, respetivamente.

Para ter a área, no sentido habitual, sempre um número positivo (ou zero, no caso de ser um triângulo degenerado), usa-se a seguinte função:

```
double area(Triangle t)
{
    return fabs(signed_area(t));
}
```

Para captar este conceito de virar à esquerda ou virar à direita, que surgiu a propósito da área com sinal, a respeito da sequência dos lados, faz-nos falta o conceito de *vetor*. No plano **XY**, um vetor representa um *deslocamento*, e pode ser ca-

racterizado por um par de números reais, o primeiro para o descolamento segundo o eixo dos **X** e o segundo para o deslocamento segundo o eixo dos **Y**.

Logo, o `typedef` é análogo ao do tipo `Point`:

```
typedef struct {
    double dx;
    double dy;
} Vector;
```

Eis o construtor:

```
Vector vector(double dx, double dy)
{
    Vector result = {dx, dy};
    return result;
}
```

Os vetores relacionam-se com os pontos por meio de duas operações: *ponto menos ponto dá vetor* e *ponto mais vetor dá ponto*. Mais concretamente, dados dois pontos **P** e **Q**, a diferença **Q-P** é o vetor que representa o deslocamento relativo de **P** para **Q**. Inversamente, dado um ponto **P** e um vetor **V**, **P+V** é o ponto **Q** tal que **Q-P** é **V**.

Podemos programar estas duas funções.

```
Vector minus(Point p, Point q)
{
    return vector(p.x-q.x, p.y-q.y);
}

Point plus(Point p, Vector u)
{
    return point(p.x + u.dx, p.y + u.dy);
}
```

Há duas operações sobre vetores de interesse computacional: o produto interno e o produto externo. Para vetores dos nossos, que são bidimensionais, programam-se a assim:

```
double dot(Vector v, Vector u) // dot product
{
    return v.dx * u.dx + v.dy * u.dy;
```

```

}

double cross(Vector v, Vector u) // cross product
{
    return v.dx * u.dy - u.dx * v.dy;
}

```

Em inglês o *dot product* é o produto interno e o *cross product* é o produto externo.

Em rigor, o resultado no produto interno é um número e o resultado do produto externo é um vetor. No nosso caso, em que ambos os vetores estão no plano **XY**, o vetor resultante do produto externo é um vetor perpendicular ao plano **XY**, pelo que as componentes segundo o eixo do **X** e segundo o eixo dos **Y** serão ambas zero. Assim, o resultado que a função `cross` produz representa a componente segundo o eixo dos **Z**.

Ora, o sinal do resultado do produto externo, positivo, negativo ou zero, depende da “orientação” relativa dos dois vetores: se for sinistrógira, o resultado é positivo; se for dextrógira, é negativo; se os vetores estiverem alinhados, é zero. Eis duas funções para analisar pares de vetores quanto à sua orientação relativa:

```

// Is `u` counterclockwise from `v` (and not on the same
// line)?
int ccw(Vector v, Vector u)
{
    return cross(v, u) > 0;
}

// Is `u` clockwise from `v` (and not on the same line)?
int cw(Vector v, Vector u)
{
    return cross(v, u) < 0;
}

```

Se os dois vetores forem “paralelos”, o produto externo dá zero. Nesse caso, há duas hipóteses: ou os vetores estão alinhados no mesmo sentido, ou estão alinhados em sentidos opostos. A distinção faz-se pelo sinal do produto externo:

```

int alligned(Vector v, Vector u)
{
    return cross(v, u) == 0 && dot(v, u) >= 0;
}

```

```
int opposite(Vector v, Vector u)
{
    return cross(v, u) == 0 && dot(v, u) <= 0;
}
```

O produto interno será zero se os dois vetores forem perpendiculares:

```
// Are `v` and `u` perpendicular?
int is_perpendicular(Vector v, Vector u)
{
    return dot(v, u) == 0;
}
```

A propósito, isto permite-nos programar simplesmente a função que verifica se um triângulo é um triângulo retângulo:

```
int is_right(Triangle t)
{
    int a[3];
    for (int i = 0; i < 3; i++)
        a[i] = dot(minus(t.v[(i+2)%3], t.v[(i+1)%3]),
                    minus(t.v[(i+1)%3], t.v[i])) == 0.0;
    return ints_count(a, 3, 1) == 1;
}
```

Apesar de simples, a função é muito “densa”. Por isso, é sensato confirmarmos, com uma função de teste unitário:

```
void unit_test_is_right(void)
{
    assert(is_right(triangle(point(0, 0), point(4, 0), point(0,
3)))));
    assert(is_right(triangle(point(1, 1), point(2, 2), point(-1,
5)))));
    assert(is_right(triangle(point(2, 2), point(6, 1), point(7,
5)))));
    assert(!is_right(triangle(point(0, 1), point(4, 0), point(0,
3)))));
    assert(!is_right(triangle(point(1, -1), point(2, 2),
point(-1, 5))));
    assert(!is_right(triangle(point(2, 2), point(6, 1), point(7,
6)))));
}
```

Equipados com os três tipos, **Point**, **Triangle** e **Vector**, já podemos resolver o problema que nos propusemos de início: verificar se um ponto está no interior de um triângulo. A estratégia é a seguinte: sejam **P**, **Q** e **R** os vértices do triângulo-

lo e **C** o ponto que está ou não no interior do triângulo. Admitamos, para simplificar, que os pontos estão do sentido sinistrógiro. Então, se o ponto estiver do lado esquerdo de cada lado, quando percorremos os lados, no sentido dos vértices (isto é, no sentido por que os vértices estão no array), isso significa que o ponto está no interior do triângulo. Inversamente, se estiver do lado direito de algum dos lados, então o ponto não está no interior do triângulo. Em rigor, “estar do lado esquerdo do lado” significa “pertencer ao semiplano que fica do lado esquerdo da reta que contém o lado, relativamente ao sentido que vai do primeiro ponto do lado para o segundo ponto”.

Verificar se o ponto **C** está à esquerda do lado **PQ**, equivale a verificar se os vetores **Q-P** e **C-Q** são um par sinistrógiro.

Portanto, primeiro o subproblema de verificar se um triângulo é sinistrógiro: :

```
int is_ccw(Triangle t)
{
    return cross(minus(t.v[1], t.v[0]),
                 minus(t.v[2], t.v[1])) > 0.0;
}
```

Na verdade, também poderíamos calcular a área com sinal e verificar se é positiva, mas isso daria mais trabalho computacional.

Eis a função de teste unitário, que usa os mesmos triângulos que a função de teste unitário para a área com sinal:

```
void unit_test_is_ccw(void)
{
    Triangle t1 = {{{0, 0}, {4, 0}, {0, 3}}};
    assert(is_ccw(t1));
    Triangle t2 = {{{0, 0}, {0, 3}, {4, 0}}};
    assert(!is_ccw(t2));
    Triangle t3 = {{{1, 1}, {5, 1}, {3, 11}}};
    assert(is_ccw(t3));
    Triangle t4 = {{{1, 1}, {3, 11}, {5, 1}}};
    assert(!is_ccw(t4));
}
```

De seguida, a função que verifica se ponto está no interior de um triângulo sinistrógiro:

```

int inside_ccw(Point p, Triangle t)
{
    assert(is_ccw(t));
    int a[3];
    for (int i = 0; i < 3; i++)
        a[i] = dbl_sign(cross(minus(t.v[(i+1)%3], t.v[i]),
                               minus(p, t.v[(i+1)%3])));
    return ints_count(a, 3, 1) == 3;
}

```

Finalmente, para um triângulo qualquer:

```

int inside(Point p, Triangle t)
{
    if (!is_ccw(t))
        t = triangle(t.v[0], t.v[2], t.v[1]);
    return inside_ccw(p, t);
}

```

Terminamos com a função de teste unitário:

```

void unit_test_inside(void)
{
    // ccw triangle
    Triangle t1 = {{0, 0}, {4, 0}, {0, 3}};
    assert(inside(point(1, 1), t1));
    assert(!inside(point(5, 5), t1));
    assert(!inside(point(-1, -1), t1));
    assert(!inside(point(0, 0), t1));
    // cw triangle
    Triangle t2 = {point(1, 1), point(5, 1), point(3, 5)};
    assert(inside(point(2, 2), t2));
    assert(inside(point(4, 2), t2));
    assert(inside(point(3, 4), t2));
    assert(!inside(point(0, 0), t2));
    assert(!inside(point(5, 1), t2));
}

```

Já que a função `inside` constituiu o pretexto para este exercício, eis a função de teste, para ser usada interativamente na consola. Primeiro, aceita as coordenadas dos vértices do triângulo e depois, sucessivamente, coordenadas de pontos. Para cada ponto, verifica se está no interior do triângulo:

```

void test_inside(void)
{

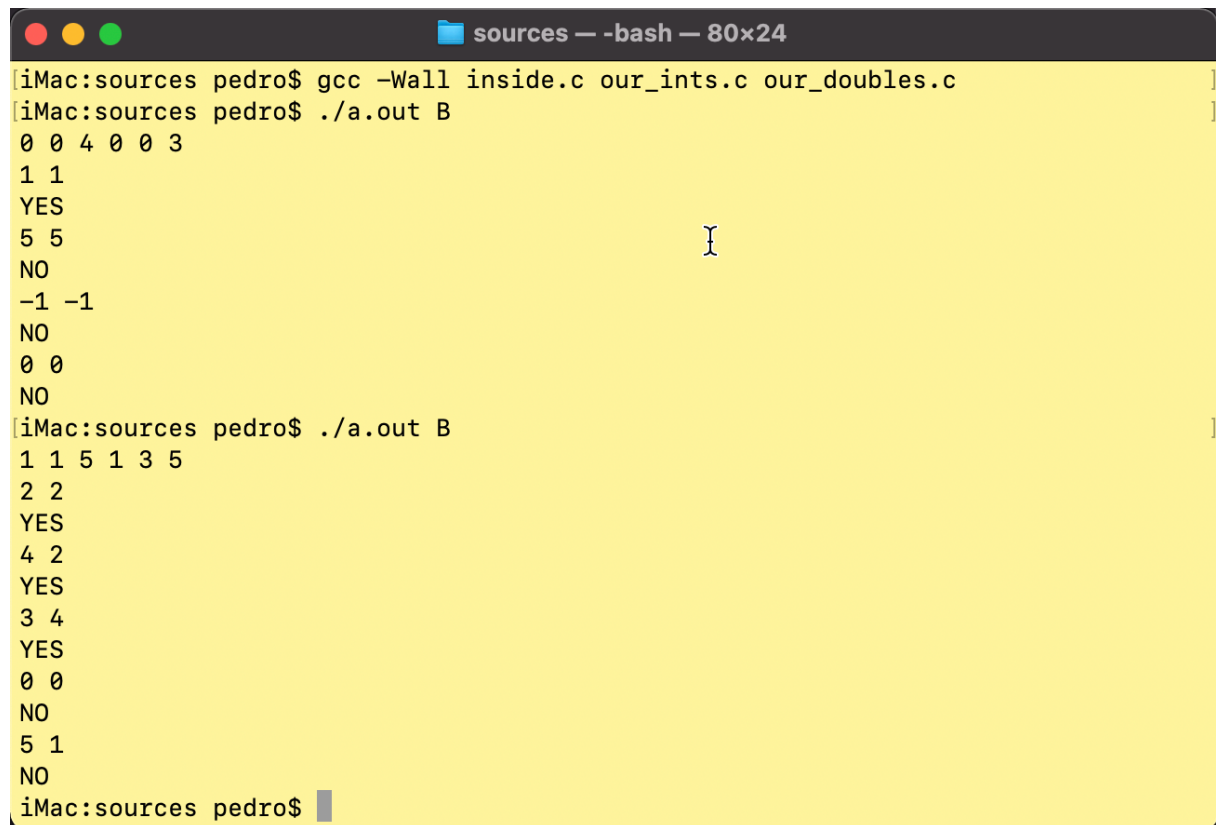
```

```

    double p1x, p1y, p2x, p2y, p3x, p3y;
    scanf("%lf%lf%lf%lf%lf%lf", &p1x, &p1y, &p2x, &p2y, &p3x,
    &p3y);
    Triangle t = triangle(point(p1x, p1y), point(p2x,
    p2y), point(p3x, p3y));
    double x, y;
    while (scanf("%lf%lf", &x, &y) != EOF)
    {
        int z = inside(point(x,y), t);
        printf("%s\n", message[z]);
    }
}

```

Na seguinte sessão de teste, usámos os valores da função de teste unitário:



```

sources — -bash — 80x24
iMac:sources pedro$ gcc -Wall inside.c our_ints.c our_doubles.c
iMac:sources pedro$ ./a.out B
0 0 4 0 0 3
1 1
YES
5 5
NO
-1 -1
NO
0 0
NO
iMac:sources pedro$ ./a.out B
1 1 5 1 3 5
2 2
YES
4 2
YES
3 4
YES
0 0
NO
5 1
NO
iMac:sources pedro$

```

Para referência, a função `unit_tests` e a função `main`:

```

void unit_tests(void)
{
    unit_test_distance();
    unit_test_collinear();
    unit_test_inside();
    unit_test_perimeter();
    unit_test_signed_area();
    unit_test_is_right();
}

```

```

    unit_test_is_ccw();
}

int main(int argc, char **argv)
{
    unit_tests();
    char x = 'A';
    if (argc > 1)
        x = *argv[1];
    if (x == 'A')
        test_points();
    else if (x == 'B')
        test_triangle();
    else if (x == 'C')
        test_inside();

    else if (x == 'U')
        printf("All unit tests PASSED\n");
    else
        printf("%s: Invalid option.\n", argv[1]);
    return 0;
}

```

Nota técnica:

No início do programa, usamos os seguintes três diretivas *pragma* :

```

#pragma GCC diagnostic error "-Wall"
#pragma GCC diagnostic error "-Wextra"
#pragma GCC diagnostic error "-Wconversion"

```

Estes “pragmas” fazem o compilador ligar as opções indicadas — `-Wall`, `-WExtra` e `-Conversion` —, automaticamente. A primeira é nossa conhecida. A segunda é do mesmo género, mas deteta anomalias que a opção `-Wall` ignora. A terceira é mais específica e assinala caso de conversões numéricas em que pode haver perda de precisão. Eis um exemplo:

```

void pragma_conversion_demo(void)
{
    double pi = 3.14;
    int x = pi;
    assert(x == 3);
}

```

Este programa compilaria, sem a opção `-Conversion`. Com a opção `-Conversion`, dá erro: `Implicit conversion turns floating-point number into integer: 'double' to 'int'`. De facto, na conversão de `double` para `int`, a parte fracionária do número perde-se. Se quisermos mesmo realizar a conversão, devemos torná-la explícita, assim:

```
void pragma_conversion_demo(void)
{
    double pi = 3.14;
    int x = (int)pi;
    assert(x == 3);
}
```

As opções de compilação podem ser indicadas na linha de comando, tal como temos feito até agora em relação à opção `-Wall`, mas, como queremos usar aquelas três sempre, fica mais prático tê-las no texto do programa.