**1. Introdução.** A seguir encontram-se as implementações em `C` de algumas operações primitivas que serão muito úteis em diversos algoritmos geométricos.

**2. Representação de um ponto.** Todos os pontos serão representados por um vetor de dimensão apropriada (a dimensão usada no código abaixo é 2) ao invés de usarmos registros.

```
#define X 0
#define Y 1
#define DIM 2    /* dimensão dos pontos */
  typedef enum {
    FALSE, TRUE
  } bool;
  typedef int tPointi[DIM];    /* tipo ponto inteiro */
  typedef double tPointd[DIM];    /* tipo ponto real */
```

**3. Representação de um polígono.** Um polígono pode ser representado por um vetor de pontos ou uma lista ligada (linear ou circular, simples ou duplamente ligada) de pontos. Listas ligadas permitem alocação dinâmica de memória bem como inserções e remoções simples (em tempo $O(1)$). Entretanto, no código abaixo usaremos a estrutura de vetor para guardar os pontos de um polígono: isso tornará o código mais claro, pois vetores facilitam a estrutura de *loops* e incremento de índices.

```
#define PMAX 1000    /* máximo número de pontos em um polígono */
  typedef tPointi tPolygoni[PMAX];    /* polígono de pontos inteiros */
```

**4. Cálculo de área de um triângulo.** A função *Area2* Retorna duas vezes o valor da área 'sinalizada' de um triângulo determinado pelos pontos $a, b$ e $c$ (triângulo $\triangle abc$). Se $a, b, c$ são orientados no sentido anti-horário o valor é positivo, caso contrário é negativo.

```
  int Area2(tPointi a, b, c)
  {
    return a[X] * b[Y] - a[Y] * b[X] + a[Y] * c[X] - a[X] * c[Y] + b[X] * c[Y] - c[X] * b[Y];
  }
```

**5. Cálculo da área de um polígono.** Retorna o dobro da área de um polígono $P$.

```
  int AreaPoly2(int n, tPolygoni P)
  {
    int i;
    int sum = 0;    /* soma das áreas parciais */
    for (i = 1; i < n - 1; i++) sum += Area2(P[0], P[i], P[i + 1]);
    return sum;
  }
```

**6. Predicado Left.** A intersecção de dois segmentos de reta pode ser decidida através do uso do predicado *Left*, que determina se um ponto está a esquerda ou direita de uma reta orientada. Veremos mais adiante como *Left* será usado para decidirmos a intersecção de dois segmentos de reta.

Uma reta orientada é determinada por um vetor $\vec{ab}$, onde $a$ e $b$ são pontos. Se um ponto $c$ está a esquerda dessa linha orientada então a tripla $(a, b, c)$ forma um circuito no sentido anti-horário. Um ponto $c$ esta a esquerda da linha orientada determinada pelo vetor $\vec{ab}$ se e somente se a orientação do triângulo $\triangle abc$ é positiva.

```
  bool Left(tPointi a, b, c)
  {
    return Area2(a, b, c) > 0;
  }
```

**7. Predicado Lefton.** O quê acontece se o ponto $c$ está sobre a reta orientada determinada por $\vec{ab}$? Então a area do $\triangle abc$ é nula. O predicado *LeftOn* devolve verdadeiro se o ponto $c$ estiver a esquerda ou sobre a reta orientada determinada pelo vetor $\vec{ab}$.

**bool** *LeftOn*(**tPointi** $a, b, c$)
{
  **return** *Area2*$(a, b, c) \geq 0$;
}

**8. Predicado Collinear.** Este Predicado é verdadeiro se os pontos $a, b$ e $c$ são colineares.

**bool** *Collinear*(**tPointi** $a, b, c$)
{
  **return** *Area2*$(a, b, c) \equiv 0$;
}

**9. Predicado IntersecProp (intersecção própria).** A função *IntersecProp* returna `TRUE` se a os segmento de reta $ab$ intersecta o segmento $cd$ própriamente: existe um único ponto em comum no interior de ambos os segmentos. Para identificar intersecção própria usamos o predicado *Left*.

**bool** *IntersectProp*(**tPointi** $a, b, c, d$){  /∗ Elimina casos impróprios ∗/
  **if** ( $(Collinear(a, b, c) \lor Collinear(a, b, d) \lor Collinear(c, d, a) \lor Collinear(c, d, b))$ **return** `FALSE`;
  **return** $Xor(Left(a, b, c), Left(a, b, c)) \land Xor(Left(c, d, a), Left(c, d, b))$; }

**10. Predicado Between.** Retorna `TRUE` se e somente se os pontos $a, b$ e $c$ são colineares e $c$ é um ponto no segmento fechado $ab$.

**bool** *Between*(**tPointi** $a$, **tPointi** $b$, **tPointi** $c$)
{
  **tPointi** $ba$, $ca$;
  **if** $(\neg Collinear(a, b, c))$ **return** `FALSE`;
  **if** $(a[X] \neq b[X])$  /∗ Se $ab$ não é vertical então verifica se $c$ está entre $a$ e $b$ ∗/
    **return** $((a[X] \leq c[X]) \land (c[X] \leq b[X])) \lor ((a[X] \geq c[X]) \land (c[X] \geq b[X]))$;
  **else return** $((a[Y] \leq c[Y]) \land (c[Y] \leq b[Y])) \lor ((a[Y] \geq c[Y]) \land (c[Y] \geq b[Y]))$;
}

**11. Predicado Intersect.** Retorna `TRUE` se e somente se o segmento $ab$ e o segmento $cd$ intersectam-se, propriamente ou impropriamente.

**bool** *Intersect*(**tPointi** $a, b, c, d$)
{
  **if** $(IntersectProp(a, b, c, d))$ **return** `TRUE`;
  **else if** $(Between(a, b, c) \lor Between(a, b, d) \lor Between(c, d, a) \lor Between(c, d, b))$ **return** `TRUE`;
  **else return** `FALSE`;
}

**12. Predicado Diagonalie.** Retorna `TRUE` se $v_i v_j$ é uma diagonal própria interna ou externa do polígono $P$, ou $\mid i - j \mid = 2$ e $v_i$ e $v_j$ são colineares com o vértice do meio.

**bool** *Diagonalie*(**int** $i$, **int** $j$, **int** $n$, **tPolygoni** $P$)
{
  **int** $k$;
  **int** *k1*;  /∗ k + 1 ∗/
  **for** $(k = 0;\ k < n;\ k{+}{+})$ {  /∗ para cada aresta $v_k v_{k+1}$ de $P$ ∗/
    $k1 = (k + 1) \,\%\, n$;
    **if** $(\neg((k \equiv i) \lor (k1 \equiv i) \lor (k \equiv j) \lor (k1 \equiv j)))$  /∗ Pula a aresta incidente a $i$ ou $j$ ∗/

```
      if (Intersect(P[i], P[j], P[k], P[k1])) return FALSE;
    }
    return TRUE;
  }
```

**13. Predicado InCone.** Seja $v_0, v_1, \ldots, v_{n-1}$ os vértices de um polígono listados no sentido anti-horária e suponha que $s = v_i v_j$, $i \neq j$, não intesecta arestas do polígono $P$ que não sejam incidentes a $v_i$ nem a $v_j$. Se $s$ é interior ao polígono $P$ nas vizinhanças de $v_i$ e $v_j$ então $s$ é interior ao polígono $P$. Assim o teste para decidir se um segmento $s$ é interior a um polígono pode ser divido em: $(i)$ verificar se $s = v_i v_j$ intersecta arestas de $P$; e $(ii)$ testar se o segmento $s$ é interior a $P$ na vizinhança de $v_i$ e $v_j$. Podemos testar se $s$ está no interior de $P$ na vizinhança de $v_i$ ou $v_j$ em tempo constante, como pode ser visto abaixo.

```
  bool InCone(int i, int j, int n, tPolygoni P)
  {
    int i1;    /* i + 1 */
    int in1;    /* i − 1 */
    i1 = (i + 1) % n;
    in1 = (i + n − 1) % n;
    if (LeftOn(P[in1], P[i], P[i1]))
        /* se P[i] é um vértice convexo (v_{i+1} está à esquerda ou sobre o segmento v_{i−1}v_i) */
      return Left(P[i], P[j], P[in1]) ∧ Left(P[j], P[i], P[i1]);
    else    /* Assuma v_{i−1}, v_i, v_{i+1} não colineares. P[i] é vértice reflexo. */
      return ¬(Lefton(P[i], P[j], P[i1]) ∧ LeftOn(P[j], P[i], P[in1]));
  }
```

**14. Primitiva Diagonal.** Returno `TRUE` se e somente se $v_i v_j$ é uma diagonal do polígono $P$.

```
  bool Diagonal(int i, int j, int n, tPolygoni P)
  {
    return InCone(i, j, n, P) ∧ Diagonalie(i, j, n, P);
  }
```

25 de agosto de 1997, às 09:43

# Primitivas



Os códigos em `C` foram escritos por Joseph O'Rourke.

Este documento foi escrito em `CWEB`.