

Agora, vamos examinar alguns tópicos relacionados à linguagens de programação. A intenção, aqui, não é ser exaustivo nem completo. Antes, queremos apenas reforçar algumas idéias gerais que permeiam quase todas as linguagens de programação de uso geral. Mais tarde, no decorrer do texto, estudaremos as linguagens C e C# em detalhes.

## Instruções mais comuns

---

Quais instruções básicas são mais comuns entre as linguagens de programação? Há várias delas. Usaremos, nos exemplos, a sintaxe da linguagem C. Dentre as instruções mais comuns podemos citar:

- Instruções de *atribuição*: nesse caso, especificamos uma expressão E cujo valor deve se calculado. Em seguida, o valor calculado é armazenado numa variável A. A instrução seria escrita na forma

A = E

Exemplo:

AREA = (MED+1)\*C

Nesse caso, calculamos o valor da expressão  $(MED+1)*C$ , i.e., somamos uma unidade ao valor armazenado na variável MED e o resultado é multiplicado pelo valor armazenado na variável C. O valor final é então armazenado no local reservado para a variável AREA. Lembre que não é importante saber quais posições de memória, exatamente, correspondem aos locais associados à estas variáveis. Basta lembrar que o compilador e o sistema operacional se encarregam, automaticamente, de designar as posições de memória associadas a cada uma das variáveis que usamos no programa.

- Instruções de *sequenciamento*: indica que uma instrução deve ser executada diretamente após uma outra. As instruções são escritas separadas pelo operador “;”, como indicado no exemplo:

AREA = (MED+1)\*C ; B = 20 ;

Aqui, indicamos que devemos primeiramente executar a instrução de atribuição  $AREA = (MED+1)*C$ . Logo em seguida, devemos executar a segunda instrução de atribuição  $B = 20$ . Também é comum escrever instruções em coluna:

AREA = (MED+1)\*C ;  
B = 20 ;

A duas formas são totalmente equivalentes.

- Instruções de *desvio*: este tipo de instrução indica um desvio na sequência normal de execução das instruções, condicionado ao resultado de uma expressão lógica. Como exemplo:

if (INST1) { INST2 }  
else { INST3 };

Nesse caso, a expressão lógica indicada por INST1 é avaliada. Se o resultado for “verdadeiro”, então devemos executar a instrução INST2. Caso contrário, i.e., se o resultado for “falso”, então devemos executar a instrução indicada por INST3. Ou

seja, qual instrução deve ser executada nesse instante depende da avaliação da instrução **INST1**, que controla o desvio. Repare que podemos voltar a essa mesma instrução de desvio mais tarde durante a execução do programa e, desta vez, o resultado da avaliação da expressão de controle pode ser diferente daquele obtido anteriormente. Isto porque os valores de algumas variáveis envolvidas no cálculo do valor de **INST1** podem ter sido alterados desde a última avaliação dessa expressão.

- Instruções de *iteração*: esse tipo de instrução indica que um bloco de instruções deve ser executado repetidas vezes. Há várias variantes. Uma possibilidade seria:

**while** (**INST1**) **do** { **INST2** };

Desejamos executar a instrução **INST2** enquanto a avaliação da expressão lógica **INST1** resultar em “verdadeiro”. Ou seja, de forma equivalente:

1. Teste o valor da expressão **INST1**. Se for “falso” termine a execução dessa instrução e passe para a próxima; se for “verdadeiro”, prossiga com a linha 2.
2. Execute a instrução **INST2**.
3. Volte para a linha 1.

Portanto, a expressão lógica **INST1** controla quantas vezes a instrução **INST2** será executada.

- ... e muitas outras, como veremos.

## Tipos de dados comuns

---

Quanto bytes de memória são reservados para armazenar o valor de uma variável? Depende de que *tipo de dado* estaremos armazenando na variável. Alguns tipos de dados requerem mais memória que outros. Em geral, todas as variáveis que usamos em um programa devem ser *declaradas* antes que possam ser usadas em instruções do programa. No ato da declaração da variável, devemos especificar que tipo de dados vamos armazenar nela.

Alguns exemplos comuns:

- Variáveis que armazenam *valores inteiros*: tipicamente, ocupam 4 bytes, ou seja 4x8=32 bits. Se **X** é uma dessas variáveis, então podemos escrever instruções tais como:

**X** = -1;

**X** = 2 + **X**

No primeiro caso, armazenamos o inteiro -1 no local reservado para a variável **X**. No segundo caso, tomamos o valor armazenado em **X** e somamos 2 a este valor; o resultado é armazenado nas posições de memória reservadas para a mesma variável **X**. O valor antigo que estava armazenado em **X** é irremediavelmente destruído.

- Variáveis que armazenam *valores fracionários*: semelhante ao caso anterior, apenas que agora lidamos com valores que são números fracionários. Se **Z** for uma dessas variáveis, poderíamos escrever:

**Z** = 3.14;

$Z = \cos(Z/3.0);$

A primeira instrução armazenaria o valor 3.14 nas posições reservadas à Z. A segunda instrução toma o valor armazenado em Z e o divide por 3; em seguida calcula o coseno desse valor – assumindo o ângulo em radianos – e guarda o resultado final nas posições associadas ao próprio Z.

4. Armazenamento de *vetores de elementos de um certo tipo*: esse tipo de dados é um pouco mais sofisticado. Como um exemplo, vamos supor que o nome, ou variável, X foi declarado como um vetor indexado de 1 a 5 e contendo valores inteiros. Então podemos escrever

$X[2] = 0$

e estaremos indicando que na posição 2 do vetor devemos armazenar o valor zero. Supondo ainda que I é uma outra variável que armazena valores inteiros e que, nesse momento, I contém o valor 3. Então a instrução

$X[I] = -1+I$

indica que devemos armazenar o valor  $-1+3 = 2$  na terceira posição associada ao vetor X. A figura abaixo representaria o vetor X nesse instante. A primeira linha indica as posições no vetor. A segunda linha indica os valores armazenados nas várias posições.

1	2	3	4	5
?	0	2	?	?

- Há muitos e muitos outros tipos de dados que podem ser declarados na linguagem C. De fato, o próprio programador pode criar tipos de dados bastante sofisticados na linguagem C.

## Um exemplo mais elaborado

Vamos examinar outro exemplo, agora um pouco mais sofisticado. Suponhamos que temos um vetor, X, com N posições, contendo números inteiros dados. Para fixar idéias, um caso quando N=5 está ilustrado na figura abaixo:

1	2	3	4	5
15	10	18	12	7

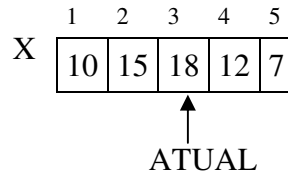
Observe que os dados armazenados em X não estão ordenados de forma crescente, quando lidos da esquerda para a direita. Buscamos um algoritmo para ordenar os dados do vetor X de forma crescente. Assim, ao término do algoritmo, o vetor X deverá conter os mesmos dados, mas na ordem abaixo:

1	2	3	4	5
7	10	12	15	18

Idéias para o algoritmo?!?

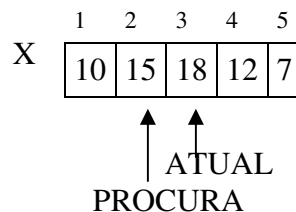
Vamos descrever um método, também chamado de *método indutivo*, que pode ser usado para se desenvolver algoritmos. É um método poderoso, mas não há garantia de que sempre leve a bons resultados. O método compreende os seguintes passos:

1. Supomos que o problema está *parcialmente resolvido*. Nesse caso, supomos que já temos alguns elementos bem posicionados no vetor X. Por exemplo, os três primeiros valores inteiros já estão ordenados, como abaixo:



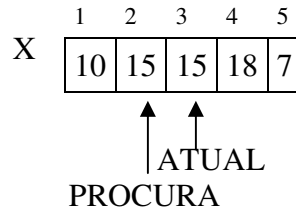
A seta vertical indica a fronteira entre os valores já ordenados, que estão à esquerda e inclusive, e os valores ainda desordenados, que estão à direita dessa fronteira. A variável auxiliar, ATUAL, armazena esse valor 3.

2. Tomamos o primeiro valor ainda não ordenado. Nesse caso seria o 12, na posição 4.
3. Descobrimos a posição correta para o valor escolhido, considerando somente a região do vetor onde os valores já estão ordenados. No caso, o valor 12 deveria entrar antes do valor 15, que está na posição 2 do vetor.

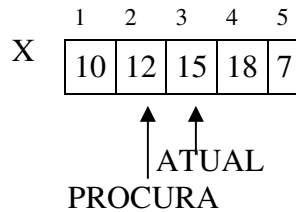


Essa posição é indicada pela primeira seta acima. A variável auxiliar PROCURA é usada para lembrar essa posição.

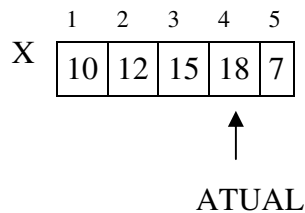
4. Usamos uma variável auxiliar, PROX, para salvar o valor que está agora em  $X[ATUAL+1]$ . Ou seja, PROX armazena o valor 12.
5. Todos os valores em X no intervalo de posições entre PROCURA e ATUAL são copiados para a posição imediatamente à direita de cada um no vetor X. Note como o valor na posição  $X[ATUAL+1]$  foi destruído. Essa é a razão do passo anterior, quando salvamos aquele valor em outra variável. Ficaríamos com:



6. Insere o valor armazenado em PROCURA na posição indicada por PROCURA. Ficaríamos com:



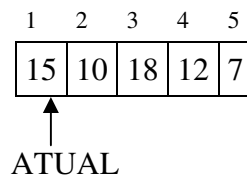
7. Avançamos ATUAL para a próxima posição. Ignorando a variável PROCURA, teríamos:



O que conseguimos? Muita coisa! Note que agora os *quatro* primeiros valores originais de X *estão ordenados*! Ou seja, conseguimos *estender* a região do vetor que já está corretamente ordenada. Esse é o sentido da palavra *indução*, na designação do método. I.e., partindo de uma solução parcial, conseguimos nos aproximar mais da solução final.

Obviamente, se repetirmos os passos 2 a 7 partindo dessa nova posição para ATUAL, ao final teríamos avançado a região corretamente ordenada. Mas, nesse caso, *todo* o vetor X estaria ordenado conforme desejado e *poderíamos parar*.

Logo, pelo mesmo motivo, se partirmos de uma situação na forma:



E repetirmos os passos 2-7 por mais quatro vezes, chegaríamos também na solução final, uma vez que cada iteração dos passos 2-7 estende a ordenação até uma posição à direita da posição indicada pela variável ATUAL.

Note um ponto muito importante: no passo 1 assumimos que o ponto de partida satisfazia a uma propriedade, qual seja, os valores em X até a posição indicada por ATUAL *já estão ordenados*. E, na situação indicada acima, essa propriedade é obviamente satisfeita. Note que, nesse caso, ATUAL=1 e, portanto, até a posição 1 só temos um dos valores de X, qual seja, 15. E esse único valor, obviamente, forma uma seqüência ordenada de valores. Portanto, podemos partir dessa situação e repetidamente executar os passos 2-7, até a solução final.

Veja que uma execução dos passos 2-7, retorna a uma situação onde a propriedade é satisfeita, se partirmos de uma situação que já satisfaz a propriedade. Assim, a cada execução da seqüência 2-7, partindo da situação inicial onde ATUAL vale 1, após quatro execuções dos passos 2-7, a propriedade ainda será válida. Mas, então, o valor de ATUAL será 5 e, nesse caso, a propriedade diz, exatamente, que os valores *até a posição 5 estão ordenados*! Ou seja, ordenamos o vetor X, como desejado. *Voilà!*

É claro que, em geral, o valor de N não é sempre 5. É um inteiro positivo qualquer. Mas é fácil adaptar para essa situação mais geral: basta partir da posição inicial indicada e executar a seqüência de passos 2-7 por (N-1) vezes.

A título de curiosidade apenas, se transcrevêssemos esse algoritmo numa linguagem de programação, teríamos tipicamente algo como:

```
atual = 1;
faça
{ prox = x[atual+1];
  procura = 1;
  enquanto (prox > x[procura]) faça procura = procura+1;
  se (procura <= atual) então
    { desloca = atual;
      enquanto (desloca >= procura) faça
        { x[desloca+1] = x[desloca];
          desloca = desloca-1;
        }
      x[procura] = prox;
    }
} exatamente (N-1) vezes;
```