

Introdução aos vetores e listas

Nesta seção examinaremos resumidamente como utilizar vetores tanto em *C* quanto em *Python*, mas iniciamos com a questão essencial: por que usar vetor (ou lista, outro possível nome, apesar desse segundo poder implicar em operadores especiais).

1. Por que precisamos de vetores?

Pense em um problema simples, muito comum, que é extrair informações sobre o comportamento médio de um "conjunto" de dados, mas apenas a média pode ser pouco informativo (e.g., metade tem valor 10 e a outra metade 0). Assim, estamos interessados em um *software* que compute a *média* e o *desvio padrão* de um "conjunto" de dados. Lembrando a definição de *desvio padrão (DP)*:

$$DP(v_0, v_1, \dots, v_{n-1}) = ((v_0 - \text{med})^2 + (v_1 - \text{med})^2 + \dots + (v_{n-1} - \text{med})^2)^{1/2}$$

Então não é viável usar variáveis "simples", pois não sabemos *a priori* quantos dados o sistema receberá, além disso não parece razoável investir tempo para desenvolver um sistema que compute a média e desvio padrão para apenas 5, 15 ou qualquer número fixo de valores; mesmo que "hoje" precisemos conhecer da informação apenas para este número fixo, podemos pensar em reaproveitar o *software* "amanhã".

Mas suponhamos ainda que o responsável pelo desenvolvimento desconheça o conceito de *vetores* (algo improvável), neste caso, ele poderia tentar ler *n* e depois, em um laço, os *n* valores para computar a média:

```
s:=0;
repetir para i entre 0 e n { ler(x); s:=s+x; }
imprimir(s/n); // # med = s/n
```

Algoritmo 1. Tentativa de obter a média (sem guardar todos os dados).

Desse modo conseguiria a média (digamos na variável `med`), mas agora para obter o *desvio padrão* seria necessário obter uma soma do quadrado da distância entre cada valor à média! Mas no algoritmo 1 os dados foram lidos, um a um, usados para computar a soma e ao final, teríamos apenas o último valor lido (em `x`).

A saída é o conceito de **vetor**, que é implementado por todas as linguagens de programação de propósito geral, a ideia é reservar uma variável que tenha acesso à várias posições de memória, como exploraremos a seguir. Com o vetor podemos guardar todos os dados lidos no algoritmo 1, possibilitando o cálculo do desvio padrão.

2. Vetores: sequência contígua de variáveis

Usualmente um *vetor* é um agregado de dados de um mesmo tipo básico ocupando posições consecutivas de memória. Isso significa que é possível pegar alternadamente o valor armazenado em qualquer das posições do vetor. Usualmente isso é implementado nas linguagens a partir de um cálculo simples usando a posição inicial do vetor e a posição relativa do elemento desejado (seu *índice*).

Na prática, para examinar ou alterar o conteúdo de qualquer das posições do vetor, as linguagens oferecem um mecanismo de *indexação*, por exemplo, se o nome do agregado de

dados for *vet*, pode-se pegar o primeiro elemento fazendo *vet[0]*, o segundo com *vet[1]* e assim por diante. A figura 1 ilustra isso sob dois pontos de vista, na esquerda (a) representando a estrutura em *baixo nível* com os valores em binário e na direita (b) interpretando os *bits* como valores inteiros.

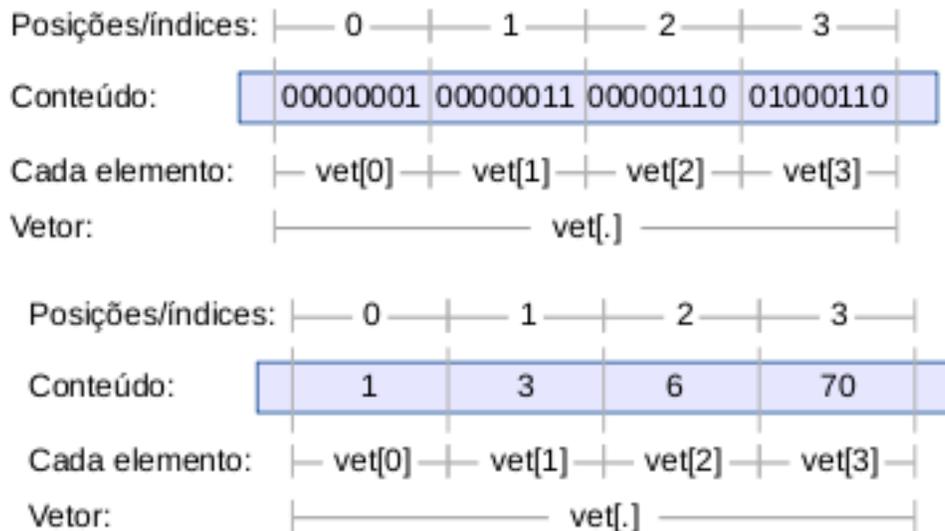


Fig. 1. (a) Representação da RAM com um vetor de 4 posições (conteúdos em binário). (b) Vetor com 4 posições com valores em decimal.

Na figura acima representamos a memória do computador com um vetor (agregado) de variáveis (usando 8 *bits* apenas) e de tamanho 4, ou seja, um vetor para armazenar 4 valores do tipo dado.

Como citado na [explicação sobre funções](#), uma grande vantagem de ter um algoritmo implementado como uma função separada é que o código do programa fica mais organizado e pode-se invocar esse algoritmo (a partir da função) em qualquer outro trecho do programa. Por isso, alguns funções muito úteis são implementadas pelos desenvolvedores para ficarem disponíveis dentro dos ambientes de programação.

Retomando o exemplo do cálculo da *média aritmética* e do *desvio padrão* de *n* valores, poderíamos implementar a função `calcula_media` e a função `calcula_dp`, como abaixo.

```
inteiro calcula_media (vetor, n) { ///  
  s := 0; ///  
  repetir para i entre 0 e n { s := s+vetor[i]; }  
  devolve s/n; ///  
}
real calcula_dp (vetor, med, n) { ///  
  s := 0;  
  repetir para i entre 0 e n {  
    aux := med-vetor[i]; ///  
    s := s + aux*aux; ///  
  }  
  devolve raiz_quadrada(s); ///  
}
```

Algoritmo 2. Calcula média e desvio padrão em funções separadas.

Por exemplo, tanto em C quanto em Python existem implementações para funções matemáticas úteis, como *seno*, *cosseno* ou *raiz quadrada* (respectivamente, *sin*, *cos* e *sqrt*). Mas é preciso indicar ao ambiente que ele deve carregar essas funções, isso é feito com os comandos `#include <math.h>` em C e `import "math"`.

3. Vetores em C

Como em C deve-se sempre declarar o tipo da variável, no caso de vetor deve-se declarar seu tipo e seu tamanho. No exemplo abaixo ilustramos declaração e uso de vetores dos tipos básicos *int*, *char* e *float*.

Vetores em C		
Vetor de inteiros	Vetor de caracteres	Vetor de flutuantes
<pre>#include <stdio.h> #define M 20 // usado para constante int main (void) { int i; // auxiliar, para indexar os vetores int nI; // tamanho util dos vetores int vetI[M]; // Vetor para inteiros (ate' M elementos) scanf("%d", &nI); for (i=0; i< nI; i++) // "Ler" nI inteiros scanf("%d", &vetI[i]); for (i=0; i< nI; i++) // "Imprimir" os elementos printf("%d\n", vetI[i]); return 0; }</pre>	<pre>#include <stdio.h> #define M 20 // usado para constante int main (void) { int i; // auxiliar, para indexar os vetores int nC; // tamanho util dos vetores char vetC[M]; // Vetor para caracteres scanf("%d", &nC); for (i=0; i< nC; i++) // ler nC caracteres scanf(" %c ", &vetC[i]); // USAR branco antes de %c for (i=0; i< nC; i++) printf("%c\n", vetC[i]); return 0; }</pre>	<pre>#include <stdio.h> #define M 20 // usado para constante int main (void) { int i; // Auxiliar, para indexar os vetores int iF; // Tamanho util dos vetores float vetF[M]; // Vetor para flutuantes scanf("%d", &nF); for (i=0; i< nF; i++) // "Ler" nF "floats" scanf("%f", &vetF[i]); for (i=0; i< nF; i++) printf("%f4.1\n", vetF[i]); // Usar 4 posicoes // e return 0; }</pre>

Como já examinado, em C geralmente deve-se carregar a biblioteca `stdio` mas existem outras, como a `stdlib.h` que dispõe, entre outras, da função `qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))` que implementa o método ordenador eficiente denominado [Quick Sort](#) (ou [Ordenador Rápido](#)). Veja um exemplo de código usando o `qsort`:

```
#include <stdio.h> // para funcao 'printf'
#include <stdlib.h> // para a funcao 'qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))'
int vetor[] = { 3, 9, 1, 7, 0, 4, 6, 5, 2, 8 }; // por simplicidade usar globais e manter fixo
int N = 10;

void imprime (int vet[], int n) { // funcao para imprimir em mesma linha
    int i;
    for (i=0; i<n; i++)
        printf(" %d", vet[i]);
    printf("\n");
}
```

```

}

// Esta e' a funcao comparadora: pode-se comparar qualquer coisa, basta
codificar seu comparador
int comparador (const void *a, const void *b) { // precisa de 2 parametros a
serem depois comparados
    return ( *(int*)a - *(int*)b );
}

int main (void) {
    printf("Testar 'qsort' da biblioteca 'stdlib'\n");
    printf("Antes: ");
    imprime(vetor, N);
    printf("Inicio: invocar 'qsort' da biblioteca 'stdlib'\n");
    // Parametros para 'qsort': vetor de dados, numero de elementos no vetor,
tamanho em bytes de cada elemento, endereco da funcao comparadora
    qsort(vetor, N, sizeof(int), &comparador);
    printf("Ordenado: ");
    imprime(vetor, N);
    return 0;
}

```

Atenção. No C padrão NÃO é possível declarar o vetor usando a variável que será usada para informar o número efetivo de posições a serem usadas, ou seja, **não** tente fazer algo como `int n; float vetF[n];`. A razão disso é que C é uma linguagem compilada e ao compilar deve-se reservar o espaço máximo a ser usado pelo vetor. Já a variável `n` só será conhecida durante a execução do programa.

Apesar de algumas implementações de compiladores C permitirem, NÃO usem sob pena de seu programa não funcionar em outros compiladores.

4. Vetores/listas em Python

Geralmente, na literatura de *programação*, o termo *vetor* refere-se a um agregado de dados, de mesmo tipo e em posições consecutivas de memória. Como em *Python*, pode-se misturar tipos, então usa-se o termo **lista** (ou **tupla**). Por exemplo, pode-se declarar `lista = (1, 2, "tres", "VI");` ou `tupla = [1, 2, "tres", "VI"];` e alterar ou imprimir um elemento, como em: `lista[3] = 3; tupla[3] = 3; print("lista=%s, tupla=%s, lista[3]=%d" % (lista,tupla, lista[3]));`.

Como citado na [explicação sobre funções](#), uma grande vantagem de ter um algoritmo genericamente implementado como uma função separada é que o código do programa fica mais organizado e pode-se invocar a função em qualquer outro trecho do programa. Por isso, alguns funções muito úteis acabam sendo implementadas pelos desenvolvedores nos ambientes de programação.

Por exemplo, em *Python* existe a função `sorted(.)` que ordena listas (e tuplas), mas também existe um método sobre lista (não sobre tupla) que também ordena lista:

```
lista = ["tres", "quatro", "um", "dois"];
```

```

print("Ordenado via 'sorted': %s" % sorted(lista)); # ['dois', 'quatro',
'tres', 'um']
lista.sort(); # nao tente imprimir direto "lista.sort()" senao aparece
"None"
print("Ordenado via 'lista.sort()': %s" % lista); # ['dois', 'quatro',
'tres', 'um']

```

Note que o resultado em `lista` será `['dois', 'quatro', 'tres', 'um']`, pois a ordem é lexicográfica (do dicionário).

Outra observação importante, o método `sort()` **não** está definido para *tuplas*, ou seja, os

```

comandos lista = ("tres", "quatro", "um", "dois");

```

```

lista.sort(); print(lista); resultaria erro (AttributeError: 'tuple' object has no
attribute 'sort').

```

No exemplo abaixo ilustramos a declaração e uso de *listas* dos tipos básicos *int*, *char* e *float*.

Nesses exemplos utilizamos uma técnica de pré-alocar todo o espaço para a lista, utilizando um valor máximo para o número de posições a serem utilizadas (o $M = 20$;). Depois solicitamos que o usuário digite o número de elementos que ele deseja inserir na lista (não fazemos um teste, mas nessa técnica esse valor **tem que ser menor que M**), então utilizamos um laço para solicitar a "entrada" de cada elemento, inserindo-o na posição adequada da lista.

Listas em Python com alocação inicial de todo o espaço <i>a priori</i>		
Vetor de caracteres	Vetor de inteiros	Vetor de flutuantes
<pre> M = 20 # usado para dimensionar os vetores def main () : vetC = list(range(0,M)); # Vetor para inteiros (M elementos) nC = int(input()); # Ler quem definira' tamanho for i in range(0, nC) : # Para ler nC caracteres vetI[i] = raw_input(); # no Python 3 usar 'input()' for i in range(0, nC) : # imprimir os elementos print(vetI[i]); </pre>	<pre> M = 20 # usado para dimensionar vetores def main () : vetI = list(range(0,M)); # Vetor de inteiros (M elementos) nI = int(input()); # Ler quem definira' tamanho for i in range(0, nI) : # Para ler nI inteiros vetI[i] = int(input()) for i in range(0, nI) : # Imprimir os elementos print(vetI[i]); </pre>	<pre> M = 20 # Para dimensionar os vetores def main () : vetF = list(range(0,M)); # Vetor para inteiros (M elementos) nF = int(input()); # Ler quem definira' tamanho for i in range(0, nF) : # Para ler nF inteiros vetI[i] = float(input()); for i in range(0, nI) : # Imprimir os elementos print(vetI[i]); </pre>

Vale notar que, como em *Python* o espaço para o vetor é reservado durante a execução, não é necessário pré-dimensionar o espaço para o "vetor" como nos 3 códigos acima, e pode-se usar como linhas 3 e 4: `nI = int(input()); vetI = list(range(0,nI));`.

Por fim, vale destacar um detalhe a respeito de como "digitar" os dados usando o comando `input`. No *Python 2*, Nos códigos acima, se o usuário digitar `SyntaxError: invalid syntax`.

Podemos adotar outra técnica para registro dos dados na lista, desta vez não usando alocação *a priori* de espaço e sim estendendo a lista a cada novo elemento digitado. Mas ainda usando a técnica que obriga o usuário a digitar um único valor por vez (que dizer, digitar um inteiro e teclar `ENTER`). Esse exemplo está à esquerda da tabela abaixo.

Na coluna da direita da tabela mostramos um exemplo distinto, no qual não é necessário digitar *a priori* o número de inteiros que comporão a lista. O usuário deverá digitar todos os dados em uma mesma linha, com um único `ENTER` ao final, e o código decompõe a "string" digitada, utilizando como separadores *espaço em branco* para deduzir quais são os inteiros.

Listas em <i>Python</i> com alocação dinâmica de espaço e tipos de digitação	
Digitando um elemento por vez	Digitando todos os elementos de uma vez
<pre>def alocaodinamica1por_linha () : vetI = []; # Aloca um "vetor" vazio # nI = int(input()); nI = M; for i in range(0, nI) : # ler nI inteiro vetI.append(int(input())); # le novo elemento e estende a lista print("Lista: ", end=""); # imprimir sem mudar de linha for i in range(0, nI) : # imprimir os elementos print("%d " % vetI[i], end=""); # imprimir sem mudar de linha print(); # quebrar a linha ao final</pre>	<pre>def alocaodinamica_varios_por_linha () : # como leremos por linha, pode-se deduzir depois o tamanho da lista linha = raw_input(); # ler como caracteres para poder pegar uma linha inteira - no Python 3 usar 'input()' vetI = map(int, linha.split()); # decompõe a linha em itens e faz um mapeamento, passando cada item para inteiro nI = len(vetI); # pegar tamanho da lista digitada print("Lista: ", end=""); # imprimir sem mudar de linha for i in range(0, nI) : # imprimir os elementos print("%d " % vetI[i], end=""); # imprimir sem mudar de linha print(); # quebrar a linha ao final</pre>

[Leônidas de Oliveira Brandão](http://line.ime.usp.br)

<http://line.ime.usp.br>

Alterações 