

# Estruturas de Dados

Prof. Guilherme N. Ramos

Todo programa é a implementação de um algoritmo computacional, e manipula dados por definição (ao rodar em um computador de *programa armazenado*), e variáveis e constantes são os objetos de dados básicos manipulados em um programa [1]. Uma *estrutura de dados* é uma forma particular de organizar os dados de modo a facilitar o acesso e a manipulação destes.

“Acerte as estruturas de dados primeiro, e o resto do programa se escreverá sozinho.”

David Jones

## 1 Representação de Dados





As operações que podem ser realizadas nos dados são definidas por seus tipos, e os mais comuns são numéricos, simbólicos e lógicos. Os dados são armazenados em bits (*binary digits*) na memória, que pode ser vista como um conjunto ordenado de bits. Cada bit representa um estado binário cujo valor é:

“ligado” é representado pelo símbolo 1, e

“desligado” é representado pelo símbolo 0.

Um bit define apenas 2 estados distintos (0/1) e mutuamente exclusivos [2], mas se considerarmos 2 bits são 4 estados (00/01/11/10). 3 bits definem 8 estados, e assim sucessivamente; então sabe-se que  $n$  bits possibilitam  $2^n$  estados distintos. Para facilitar, há uma nomenclatura específica para lidar com a quantidade de bits: 8 bits compõem 1 byte, e  $10^6$  bytes são 1 MB.

Considerando 1 byte, pode-se definir 256 estados diferentes, mas *o que representa cada estado?*

	$\mathbb{N}$	$\mathbb{Z}$	Letra	Imagem	...
00000000	0	0	A		...
00000001	1	1	B		...
00000010	2	2	C		...
⋮	⋮	⋮	⋮	⋮	⋮
11111111	255	-127	?		...

Na memória do computador, a representação física dos dados é uma só: *binária*<sup>1</sup>. Mas a *interpretação* dos bits define a informação. Assim como dados diferentes podem ser armazenados como um mesmo conjunto de bits, conjuntos diferentes de bits podem ser interpretados como o mesmo dado.

## 2 Sistemas Numéricos

Bits, assim como algarismos, podem representar números pelo sistema numérico posicional que é baseado na soma ponderada dos valores dos símbolos da base, de acordo com sua posição. Por exemplo, na base decimal o valor do número representados pelos símbolos 123 é dado por:

$$100 + 20 + 3 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

<sup>1</sup>Há 10 tipos de pessoas: as que percebem código binário e as que não.

O valor de qualquer número em uma representação posicional depende de cada algarismo que o compõe e de sua posição. Os algarismos dependem da base numérica, mas o valor em qualquer base pode ser facilmente obtido com a seguinte fórmula:

$$a_n a_{n-1} \cdots a_2 a_1 a_0 = a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \cdots + a_2 \cdot b^2 + a_1 \cdot b^1 + a_0$$

Onde  $b$  é a base numérica e  $a_i$  é o algarismo na  $i$ -ésima posição do número, sendo  $0 \leq a_i < b$ . Em computação, as bases mais utilizadas são *hexadecimal*, com os algarismos  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ , *decimal*, com  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ , *octal*, com  $\{0, 1, 2, 3, 4, 5, 6, 7\}$ , e *binária*, com  $\{0, 1\}$ . Assim, tem-se:

$$7B_{16} = 123_{10} = 173_8 = 1111011_2$$

## 2.1 Números Naturais

Os números naturais são facilmente representados pelo sistema de numeração binário. Por exemplo, 3 bits podem representar:

$$000 = 0 \cdot 2^2 + 0 \cdot 2 + 0 = 0$$

$$001 = 0 \cdot 2^2 + 0 \cdot 2 + 1 = 1$$

$$010 = 0 \cdot 2^2 + 1 \cdot 2 + 0 = 2$$

$$011 = 0 \cdot 2^2 + 1 \cdot 2 + 1 = 3$$

$$100 = 1 \cdot 2^2 + 0 \cdot 2 + 0 = 4$$

$$101 = 1 \cdot 2^2 + 0 \cdot 2 + 1 = 5$$

$$110 = 1 \cdot 2^2 + 1 \cdot 2 + 0 = 6$$

$$111 = 1 \cdot 2^2 + 1 \cdot 2 + 1 = 7$$

É importante notar que, como são armazenados na memória, os valores dos números são limitados pela quantidade de bits utilizada.

## 2.2 Números Inteiros

Como os naturais, é fácil representar os números inteiros em binário com a mesma lógica posicional. Entretanto, estes números podem ter valores negativos, e é preciso considerar esta informação.

A solução encontrada foi utilizar um bit (arbitrariamente, sempre o mais a esquerda) para indicar o sinal do número: 0 implica que o número é *positivo*, 1 que é negativo. Há três formas distintas de interpretar estes números.

Sinal e Magnitude considera o bit mais a esquerda como indicador de sinal e os bits restantes diretamente pelo sistema posicional (como um número natural).

$$\begin{aligned}
000 &\rightarrow +(0 \cdot 2 + 0) = +0 \\
001 &\rightarrow +(0 \cdot 2 + 1) = +1 \\
010 &\rightarrow +(1 \cdot 2 + 0) = +2 \\
011 &\rightarrow +(1 \cdot 2 + 1) = +3 \\
100 &\rightarrow -(0 \cdot 2 + 0) = -0 \\
101 &\rightarrow -(0 \cdot 2 + 1) = -1 \\
110 &\rightarrow -(1 \cdot 2 + 0) = -2 \\
111 &\rightarrow -(1 \cdot 2 + 1) = -3
\end{aligned}$$

Complemento de um considera o bit mais a esquerda como indicador de sinal. Se *positivo*, os bits restantes indicam o valor pelo sistema posicional; se *negativo*, é preciso inverter todos os bits antes de considerar o sistema posicional.

$$\begin{aligned}
000 &\rightarrow +00 = +(0 \cdot 2 + 0) = +0 \\
001 &\rightarrow +01 = +(0 \cdot 2 + 1) = +1 \\
010 &\rightarrow +10 = +(1 \cdot 2 + 0) = +2 \\
011 &\rightarrow +11 = +(1 \cdot 2 + 1) = +3 \\
100 &\rightarrow -11 = -(1 \cdot 2 + 1) = -3 \\
101 &\rightarrow -10 = -(1 \cdot 2 + 0) = -2 \\
110 &\rightarrow -01 = -(0 \cdot 2 + 1) = -1 \\
111 &\rightarrow -00 = -(0 \cdot 2 + 0) = -0
\end{aligned}$$

Por fim, a forma mais utilizada é complemento de dois, que considera o bit mais a esquerda como indicador de sinal. Se *positivo*, os bits restantes indicam o valor pelo sistema posicional; se *negativo*, é preciso inverter todos os bits e incrementar em 1 o resultado antes de considerar o sistema posicional. Embora pareça mais complicada, esta forma tem uma série de vantagens (como quantidade de valores distintos e facilidade na computação de operações matemáticas).

$$\begin{aligned}
000 &\rightarrow 000 \rightarrow +000 = +(0 \cdot 2^2 + 0 \cdot 2 + 0) = +0 \\
001 &\rightarrow 001 \rightarrow +001 = +(0 \cdot 2^2 + 0 \cdot 2 + 1) = +1 \\
010 &\rightarrow 010 \rightarrow +010 = +(0 \cdot 2^2 + 1 \cdot 2 + 0) = +2 \\
011 &\rightarrow 011 \rightarrow +011 = +(0 \cdot 2^2 + 1 \cdot 2 + 1) = +3 \\
100 &\xrightarrow{inv} 011 \xrightarrow{+1} -100 = -(1 \cdot 2^2 + 0 \cdot 2 + 0) = -4 \\
101 &\xrightarrow{inv} 010 \xrightarrow{+1} -011 = -(0 \cdot 2^2 + 1 \cdot 2 + 1) = -3 \\
110 &\xrightarrow{inv} 001 \xrightarrow{+1} -010 = -(0 \cdot 2^2 + 1 \cdot 2 + 0) = -2 \\
111 &\xrightarrow{inv} 000 \xrightarrow{+1} -001 = -(0 \cdot 2^2 + 0 \cdot 2 + 1) = -1
\end{aligned}$$

É interessante notar que, dependendo da interpretação, um mesmo conjunto de bits pode ter valores numéricos diferentes (ex: 111). Além disso, o conjunto dos números inteiros é infinito, mas a memória tem apenas uma quantidade finita de bits. Considere o código para calcular a raiz de um número inteiro:

00-raiz2-4.c

```

1 int erro(int r, int n) {
2     return abs(r*r - n);

```

```

3 }
4
5 int raiz2(int n) {
6     int r = n/2;
7
8     if(n < 2)
9         return (n < 0 ? -1 : n);
10
11     while(erro(r, n) > r)
12         r = (r+(n/r))/2;
13
14     return r;
15 }

```

Sabendo que, na linguagem C, o tipo `int` usa 32 bits (complemento de 2), o que acontece quando se tenta calcular a raiz de  $10^{20}$ ? Veja `02-limites.c`, e pense no que ocorre na linha 2.

Em contraste, na linguagem Python, o inteiro também utiliza uma representação em complemento de 2, mas com uma quantidade diferente de bits para armazenar o número. Veja `02-limites.py`.

## 2.3 Números Reais

Os números reais também podem ser representados como binários pelo sistema posicional, basta estender a lógica para os valores não inteiros. Por exemplo:

$$13,125 = 1 \cdot 10^1 + 3 \cdot 10^0 + 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

Este valor pode ser representado com a mesma lógica, agora em binário:

$$1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1101,001$$

A representação com ponto fixo define que, dada uma quantidade  $Q$  de bits para representar o número, há uma quantidade fixa  $m$  de bits que armazenam a parte inteira e outra quantidade  $f$  que armazena a parte fracionária do número (tais que  $Q = m + f$ ). Por exemplo, supondo  $Qm.f = Q5.3$ , o número 13,125 seria representado pelos bits 01101001 (“significando” 01101,001). Supondo  $Q4.4$ , seriam os bits 11010010.

Já a representação em ponto flutuante aproveita as vantagens da notação científica (por exemplo,  $13,125 = 0,13125 \cdot 10^2$ ). Da mesma forma, qualquer número binário pode ser representado como  $m \cdot 2^e$ , sendo  $m$  o valor da *mantissa* e  $e$  o *expoente*.

A ideia é simples: define-se uma quantidade fixa de bits para armazenar a mantissa, e o restante para armazenar o expoente. Esta representação oferece maior flexibilidade e alcance para números reais (comparada ao ponto fixo). O padrão dos computadores modernos é o IEEE 754, que considera precisão simples (32 bits) e dupla (64 bits), e o define o valor armazenado pela equação:

$$(-1)^{sinal} \cdot (1 + mantissa) \cdot 2^{expoente - offset}$$

No caso da precisão simples, o primeiro bit (mais a esquerda) indica o sinal, os 8 bits seguintes o expoente, e os 23 bits restantes a mantissa (parte fracionária); o *offset* tem valor 127 (por que?). Considerando os 32 bits **1 01111110 100000000000000000000000**, tem-se:

$$\begin{aligned}
 & (-1)^1 \cdot 1,1 \cdot 2^{126-127} \\
 &= -1,1 \cdot 2^{-1} \\
 &= -0,11 \\
 &= -(1 \cdot 2^{-1} + 1 \cdot 2^{-2}) \\
 &= -(0,5 + 0,25) \\
 &= -0,75
 \end{aligned}$$

sinal		expoente											mantissa																		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

O mesmo processo é realizado para precisão dupla, mas neste caso o expoente tem 11 bits, a mantissa 52 bits, e o *offset* é 1023. Este tipo de representação possibilita o uso de números muito pequenos/grande (veja `02-limites.c`). Como exercício, tente definir a representação do valor 576,73 em ponto flutuante de precisão simples.

O uso de ponto flutuante oferece diversas vantagens, principalmente a representação de valores absolutos muito grandes ou pequenos [2]; mas há um “pequeno problema” (veja `03-precisao_float.c`). Suponha o código a seguir, com cuja representação do tipo `float` é no padrão IEEE 754 (precisão simples). Qual mensagem seria mostrada por sua execução?

`05-precisao_float.c`

```

1  float soma = 0;
2  for(i = 0; i < 10; ++i)
3      soma += 0.1;
4
5  if(soma == 1)
6      printf("soma == 1\n\n");
7  else
8      printf("soma != 1\n\n");

```

Se você entendeu direito a representação na memória, deve ter percebido que o comportamento esperado (e correto!) é que a mensagem seja: “soma != 1”. Se não concorda, tente representar o valor 0,1 em ponto flutuante. Se ainda assim não estiver muito claro, analise este código: `04-precisao_float.c`.

Esta imprecisão tem uma série de implicações. A mais clara é que você não deve comparar diretamente variáveis com este tipo, pois valores que “deveriam” ser iguais não são [necessariamente]. A solução é considerar uma tolerância aceitável entre os valores. Outra implicação é que é preciso muita atenção a possibilidade de acúmulo de erro, pois podem ser realmente significativos.

A linguagem C oferece diversos tipos numéricos, cujos tamanhos (quantidade de bits utilizados para armazenagem) *depende da implementação*; isso permite que o programador explore as vantagens do hardware [3]. Isso pode ser um problema, pois afeta a portabilidade do código.

Na linguagem Python, há apenas o tipo `float`, que é representado na memória no padrão IEEE754 (64 bits).

### 3 Símbolos

Símbolos são uma forma extremamente versátil de comunicar informações; O alfabeto define um pequeno conjunto de símbolos que, juntos, podem expressar quase tudo que se deseja. E, claro, símbolos também podem ser representados por bits.

Uma vez estabelecido um padrão de *codificação de caracteres* (associação [arbitrária] de bits a certos caracteres), pode-se armazenar estes símbolos na memória (e recuperá-los). Há diversas formas de se codificar caracteres: EBCDIC, Unicode (veja isso), ente outros.

Nesta disciplina, o foco é a representação em ASCII, uma forma extremamente compacta e a mais utilizada no padrão ANSI. Neste contexto, um `char` é um pequeno inteiro, de modo que pode ser livremente usado em expressões aritméticas [1] (veja `00-ascii.c` e `01-ascii.c`).

A versão 3 da linguagem Python utiliza outra abordagem, cada caractere é representado como Unicode, mas os valores numéricos/simbólicos podem ser utilizados por intermédio das funções `ord` e `chr`, respectivamente (veja `00-ascii.py` e `01-ascii.py`).

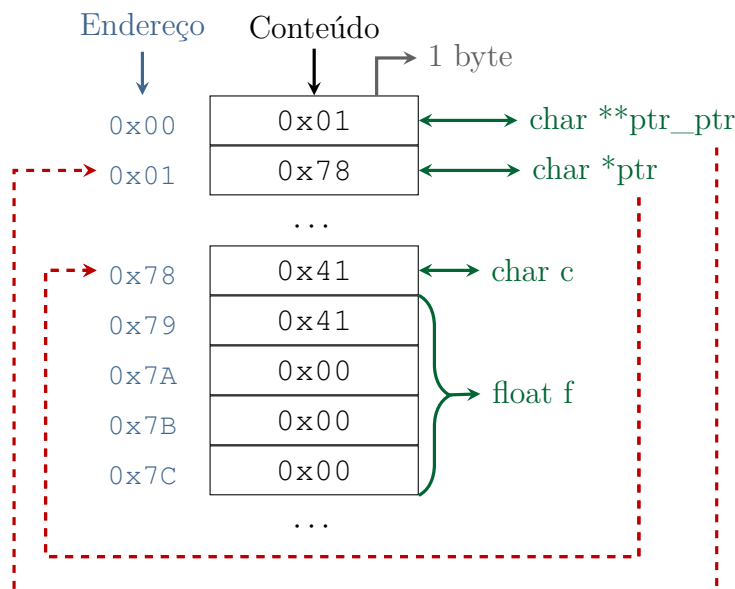
## 4 Ponteiros

Cada variável declarada ocupa um espaço na memória, conforme seu tipo, e nome da variável é apenas uma forma “amigável” de lidar com o endereço deste espaço. Ponteiro (ou *apontador*) é um tipo de dado que armazena um *endereço de memória*, possibilitando leitura e escrita deste endereço. São muito usado na linguagem C por serem, as vezes, a única forma de expressar uma computação [1] (e geralmente tornarem o código mais compacto e eficiente). Simplificando, para um tipo T, a variável  $T^* p$  é do tipo *ponteiro para T* e pode armazenar o endereço de um objeto do tipo T [3].

### 4.1 Ponteiros na Linguagem C

Um ponteiro na linguagem C é declarado da seguinte forma: `tipo *identificador`. Por exemplo:

```
1  int*   ptr_int;      /* ponteiro para inteiro */
2  float* ptr_float;    /* ponteiro para real */
3  char*   ptr_char;     /* ponteiro para caractere */
4
5  char**  ptr_ptr_char; /* ponteiro para (ponteiro para caractere) */
```



O endereço de memória identifica o espaço físico na memória dos bytes que armazenam a informação. Por exemplo, na figura acima, o *endereço* 0x78 indica o byte identificado como *c*. *c* é uma variável do tipo `char`, que neste exemplo ocupa 1 byte de memória. Ao analisar o *conteúdo* de 0x78, vê-se que o valor armazenado é 0x41 (símbolo ‘A’ na tabela ASCII).

*Atenção a diferença conceitual entre **endereço** e **conteúdo**. O endereço indica a localização na memória (onde está armazenado), o conteúdo indica o valor dos bits (o que está armazenado).*

Da mesma forma, o byte no endereço 0x79 é identificado como *f*, e como a variável é do tipo `float`, o computador sabe que ela ocupa (neste exemplo) 4 bytes de memória ( $0x41000000 = 8$ ).

O byte no endereço 0x01 é identificado como *ptr*, do tipo `char *` (ponteiro para `char`) que, neste exemplo, ocupa 1 byte. Por ser um ponteiro, o conteúdo deste identificador é interpretado como um número natural que *aponta* para um endereço de memória (no exemplo, para o endereço 0x78). Por ser um ponteiro para `char`, o conteúdo deste endereço é tratado como `char`.

Sabendo o endereço de memória de acesso aleatório, pode-se acessar diretamente a posição indicada pelo ponteiro e verificar seu conteúdo e dizer, por exemplo, *qual o caractere armazenado no endereço apontado por ptr*?

Uma das vantagens de se utilizar ponteiros é lidar com endereços de memória, de forma “independente” do tipo (os bytes têm endereços representados da mesma forma, independentemente do que armazenam). Por exemplo, assim como *ptr*, *ptr\_ptr* identifica 1 byte e armazena um endereço de memória; mas por ser um ponteiro para ponteiro, *ptr\_ptr*, aponta para um endereço de memória que também armazena um endereço de memória, no caso um *ponteiro para char*.

00-ponteiro.c

```

1 char c = 'A';
2 char* ptr = &c; /* Armazena o endereço de c */
3
4 /* O conteúdo de c é: */
5 printf("  c = %c\n", c);
6 /* O conteúdo de ptr é: */
7 printf(" ptr = %p\n", ptr);
8 /* O conteúdo do endereço apontado por ptr é: */
9 printf("*ptr = %c\n", *ptr);
10 /* O endereço de ptr é: */
11 printf("&ptr = %p\n", &ptr);

```

01-tipos.c

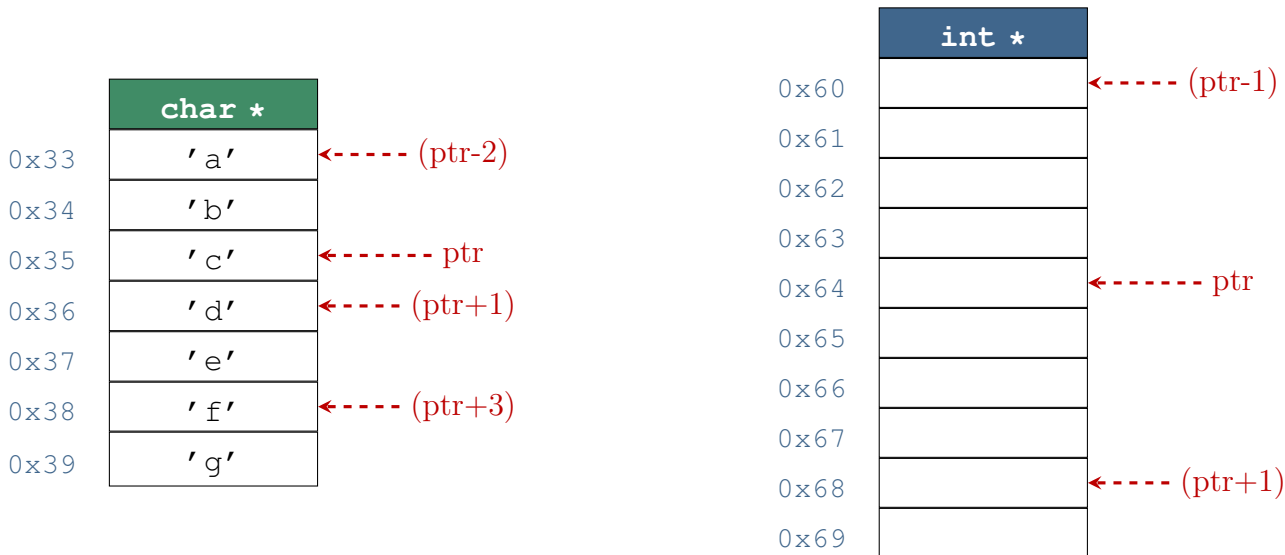
```

1 int i = 10;
2 char c = 'A';
3 float f = 1.5;
4 double d = 3.14;
5 int* pi = &i;
6 char* pc = &c;
7 float* pf = &f;
8 double* pd = &d;

```

Na linguagem C, o conteúdo de uma variável é acessado por seu identificador, e o mesmo é válido para ponteiros. Entretanto, muitas vezes deseja-se acessar o conteúdo do endereço apontado pelo ponteiro, para tanto utiliza-se o operador unário *\**. Para obter o endereço do conteúdo de um identificador, utiliza-se o operador unário *&*.

A linguagem C tem tipos estáticos e fortes, portanto ao declarar um ponteiro já se sabe, em tempo de compilação, o tipo de dado a ser apontado e, conseqüentemente, a quantidade de bytes que cada conteúdo ocupa. Isso têm efeitos interessantes na aritmética de ponteiros. Por armazenarem números naturais, pode-se adicionar ou subtrair de ponteiros para acessar outros elementos na memória, e conforme a tipagem, sabe-se exatamente quantos bytes equivalem a “uma unidade” do tipo. Desta forma, alterar o conteúdo de um ponteiro para *char* em uma unidade seria o equivalente a deslocá-lo 1 byte, mas seriam 4 no caso de um ponteiro para *int* (veja 03-aritmetica.c).



É preciso muito cuidado ao utilizar aritmética de ponteiros, pois você pode lidar com uma referência inválida de memória.

Como qualquer outro tipo, ponteiros podem ser utilizados como argumentos de funções. Um ponteiro dado como argumento é armazenado no escopo da função, mas seu conteúdo pode indicar um endereço de memória de outro escopo, possibilitando aplicações muito mais interessantes. Compare o código de troca a seguir com o visto em 03-escopo.c.

```

1 /* Troca os conteúdos dos inteiros. */
2 void troca_i(int* a, int* b) {
3     int aux = (*a);
4     (*a) = (*b);
5     (*b) = aux;
6 }

```

Esta versão de `troca_i` funciona como esperado, pois embora tenha suas variáveis locais e seu escopo, pode acessar diretamente os endereços de outros escopos para manipular a memória. É assim que as funções `scanf` e `printf` conseguem realizar suas tarefas.

Além disso, sabe-se que a comunicação de dados entre [sub]algoritmos é feita pela passagem de argumentos (se houver) e pelo valor de retorno (se houver). E que na linguagem C, o valor de retorno é sempre uma *saída* da função. Mas pode-se explorar o uso de ponteiro de forma que os argumentos fornecidos passados podem ser considerados de:

**entrada:** são recebidos e processados, mas não alterados;

**saída:** têm seus valores alterados para processamento após a execução da função; e

**entrada/saída:** fornece um valor à função e é alterado por sua execução.

Por exemplo,  $\{a, b, c\}$  de entrada e  $\{r1, r2\}$  de saída:

05-bhaskara.c

```

1 int bhaskara(double a, double b, double c,
2             double *r1, double *r2) {
3     double delta = b*b - 4*a*c;
4     int raizes_reais = (delta >= 0 ? 1 : 0);
5
6     if(raizes_reais) {
7         (*r1) = (-b + sqrt(delta))/(2*a);
8         (*r2) = (-b - sqrt(delta))/(2*a);
9     }
10
11     return raizes_reais;
12 }

```

Por fim, em um computador de programa armazenado, as instruções também são dados guardados na memória que têm endereços para acesso, então por que não usar ponteiros de funções?

06-funcao.c

```

1 #include "../01_Numeros/apc_numeros.h"
2
3 /* Chama a função dada usando os parâmetros dados (a,b) como
4  * argumentos. */
5 int chama(int (*func)(int, int), int a, int b) {
6     return func(a,b);
7 }
8
9 int main() {
10     int a = 1, b = 2;
11
12     printf("chama(max,%d,%d) = %d\n", a,b, chama(max_i,a,b));
13     printf("chama(min,%d,%d) = %d\n", a,b, chama(min_i,a,b));
14
15     a = 7;
16     printf("chama(max,%d,%d) = %d\n", a,b, chama(max_i,b,a));
17     printf("chama(min,%d,%d) = %d\n", a,b, chama(min_i,b,a));
18
19     return 0;
20 }

```



Esta ideia pode parecer um pouco confusa de início, mas possibilita programas extremamente interessantes. Por exemplo, o Método de Newton-Raphson serve para aproximar a raiz de um polinômio qualquer. A primeira implementação vista (`00-Newton-Raphson.c`) define as funções do polinômio (`f` e `fp`) para encontrar a raiz quadrada, e a função `Newton_Raphson` as utiliza para realizar a tarefa.

O método numérico abstrai a implementação das funções, de modo que funciona para quaisquer implementação de `f` e `fp`. Entretanto, a implementação exige que estas estejam bem definidas antes da compilação, prendendo a função de aproximação à definição do polinômio. Uma forma de superar esta limitação é utilizando ponteiros de função (veja `06-funcao.c` e tente resolver o exercício `Newton-Raphson.c`).

*“Brincar com ponteiros é como brincar com fogo. Fogo talvez seja a ferramenta mais importante da humanidade. Usado com cuidado, fogo traz enormes benefícios; mas quando foge ao controle, é desastroso.”*

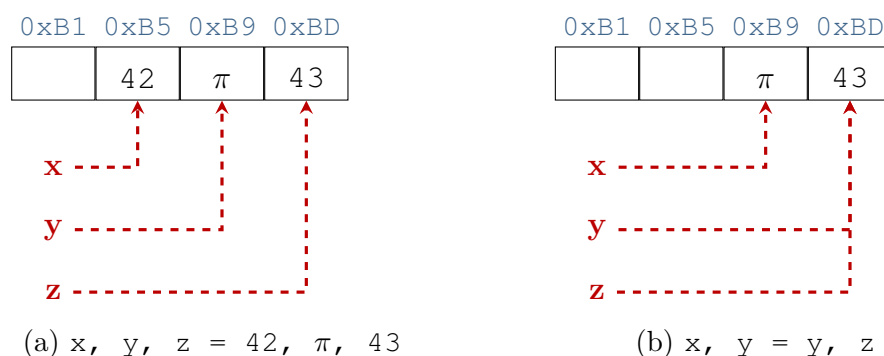
John Barnes

## 4.2 Ponteiros na Linguagem Python

A linguagem Python não utiliza ponteiros diretamente e, por projeto, manipula objetos de forma diferente de C. Em ambas, os argumentos são passados por valor, mas dependendo do tipo de objeto (mutável/imutável), a manipulação dentro do escopo de uma função tem ou não efeito em um escopo externo.

Primeiro, é preciso lembrar alguns detalhes da própria linguagem: os objetos (que ficam na memória) têm os seguintes atributos: *tipo*, que define a informação (e valores) que o objeto pode receber e as operações que podem ser executadas nele; *valor*, que é o endereço de memória ocupado pelo objeto; *nome*, que identifica o objeto diferenciando dos demais elementos na memória; e *tempo de vida*, que é o período de tempo de execução do programa durante o qual o objeto existe. Na linguagem Python lida-se com referências para objetos, as quais desempenham papéis similares a ponteiros.

Os objetos são *imutáveis* (como `int`, `float`, `str`, etc.) ou *mutáveis* (como `lista`, `dict`, etc.). Toda variável é, na verdade, uma referência para algum objeto da memória. Considerando objetos imutáveis, como ilustrado na figura abaixo, o valor de nome ‘`z`’ não varia a não ser que você o mude explicitamente, e o operador `=` não é de *atribuição de valor* (como na linguagem C), mas de *referência*, e o coletor de lixo lida com a informação 42 ao término de seu tempo de vida.



Na linguagem Python, o conteúdo de uma variável é acessado por seu *nome*, e não é permitida a manipulação direta da memória por um endereço. Isso tem uma série de vantagens (simplicidade) e desvantagens (desempenho). Internamente, o sistema acumula identificadores próprios para cada objeto enquanto durar o seu *tempo de vida* (veja `00-identificador.py` e `01-identificador.py`).

A linguagem Python tem tipo dinâmico e forte, portanto ao declarar um objeto se sabe, em tempo de execução, o tipo de dado referenciado e como ele é representado na memória.

No caso da passagem de informações a funções por argumentos, a linguagem Python define a variável local como referência ao valor passado (similar a passar a referência para o valor armazenado). A diferença ocorre na questão de qual é o objeto dado, se é *mutável*, a função recebe uma referência para ele e pode alterá-lo (desde que a referência local não seja variada); se é *imutável*, a função não pode alterá-lo.

A comunicação de dados entre [sub]algoritmos é feita pela passagem de argumentos e pelo valor de retorno, e a linguagem Python possibilita o retorno de múltiplos valores.

05-bhaskara.py

```
1 def bhaskara(a, b, c):
2     delta = (b ** 2) - (4 * a * c)
3     raizes_reais = (delta >= 0)
4
5     r1 = (-b + (delta ** 0.5)) / 2 if raizes_reais else 0.0
6     r2 = (-b - (delta ** 0.5)) / 2 if raizes_reais else 0.0
7
8     return (raizes_reais, r1, r2)
```

Na linguagem Python, funções também são objetos, e podem ser fornecidas como argumentos para outras funções de forma mais simplificada (comparada a linguagem C). Um exemplo simples disso é:

06-funcao.py

```
1 def chama(func, a, b):
2     '''Chama a função dada usando os parâmetros dados (a,b)
3     como argumentos.'''
4     return func(a, b)
5
6
7 a, b = 1, 2
8
9 # As funções max e min já são definidas em Python
10 print('chama(max, %d, %d) = %d' % (a, b, chama(max, a, b)))
11 print('chama(min, %d, %d) = %d' % (a, b, chama(min, a, b)))
```

Da mesma forma, pode-se abstrair o método de aproximação de raízes e passar a função do polinômio (Newton-Raphson.py).

## 5 Vetores

É fácil manipular um dado para resolver um problema:

```
1 z = min(x, y);
2
```

Mas e  $n$  problemas?

```
1 z = min(x1, min(x2, min(x3, min(x4, min(x5, /* ... */ min(xk, xn)/* ... */))));
2
```

Não é factível considerar escrever tanto código, nem pensar nas dificuldades de eventuais alterações. Felizmente há uma solução mais eficiente. Suponha que você tenha um ponteiro com o endereço de um caractere na memória, e que saiba que os  $n$  bytes imediatamente seguintes estão alocados para você armazenar outros caracteres. Você poderia identificá-los como  $\{c_0, c_1, \dots, c_{n-1}\}$  e lidar com estes  $n$  elementos no código, ou, dado que sabe o endereço do primeiro, utilizar aritmética de ponteiros para acessar qualquer outro caractere em função do deslocamento em relação a posição inicial.

```

1 printf("c0 = %c\n", c0);
2 printf("c1 = %c\n", c1);
3 /* ... */
1000 printf("c999 = %c\n", c999); /* n==1000 */

```

```

1 for(i = 0; i < n; ++i)
2     printf("c%d = %c\n", i, *(c+i));

```

Um vetor (array) é um conjunto finito e ordenado (em relação a posição) de elementos homogêneos (do mesmo tipo). É um modo particular de organizar dados para facilitar o acesso e manipulação dos dados, caracterizado pelas operações sobre os dados, e não pelo tipo do dado (já que se baseia na aritmética de ponteiros).

Desta forma, é possível ter um vetor de qualquer tipo de dado. Considerando  $n$  caracteres, o computador aloca um bloco de memória de  $n$  bytes (supondo que um `char` seja armazenado em 1 byte). Analogamente, supondo  $n$  inteiros (de 4 bytes cada), serão alocados  $4n$  bytes de memória (que equivalem a  $n$  unidades [de memória] de inteiros).

0	1	2	3	4	5	6	7	8	9
?	?	?	?	?	?	?	?	?	?

Portanto, para lidar com um vetor basta saber duas coisas: o endereço do primeiro elemento e quantos são os elementos armazenados. Na linguagem C, fica fácil declarar um vetor:

```

1 int    inteiros[1000];
2 float  reais[50];
3 char   caracteres[8];

```

Considerando o vetor de caracteres, o que acontece na execução é que o computador armazena o ponteiro para caracteres (`c`) (em algum lugar da memória) e um espaço do tamanho desejado ( $8 * \text{sizeof}(\text{char})$ ) em outro lugar. O acesso a cada elemento, dado o endereço do primeiro, é direto com aritmética de ponteiros ou com o operador `[ ]`:

```

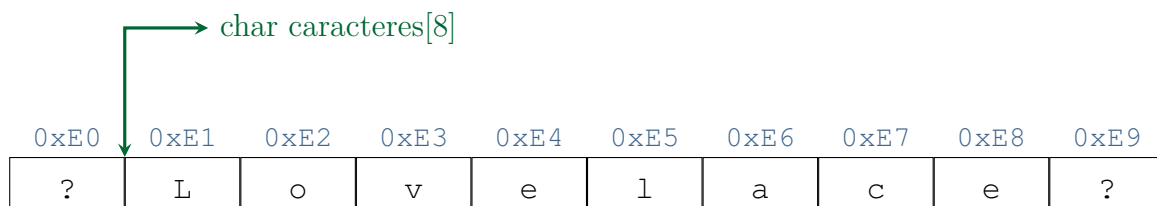
1 for(i = 0; i < n; ++i)
2     printf("c%d = %c\n", i, *(c+i));

```

```

1 for(i = 0; i < n; ++i)
2     printf("c%d = %c\n", i, c[i]);

```



A linguagem Python tem uma abordagem diferente, no contexto desta disciplina, usaremos listas como vetores [4]. Embora a implementação seja distinta, o funcionamento é bastante similar.

```

1 inteiros = [0, 1, 2, 3]
2 reais = [0.0] * 10
3 caracteres = ['L', 'o', 'v', 'e', 'l', 'a', 'c', 'e']

```

```

1 for i in range(len(inteiros)):
2     print(inteiros[i])

```

```

1 for c in caracteres:
2     print(c);

```

O acesso aos elementos depende da posição deles em relação ao endereço de início. Dependendo da linguagem, o primeiro elemento tem índice 0 (como em C, Java, Python, e Lisp) ou 1 (Fortran, COBOL, e Lua), mas por uma questão de simplicidade e facilidade de uso, a numeração deveria começar em zero [5].

O acesso correto aos elementos depende também do tamanho do vetor. Na linguagem C, os  $n$  elementos do vetor estão localizados em um bloco contínuo de bytes (com deslocamento em  $[0, n)$ ,

mas a aritmética de ponteiros permite que se acesse outras posições [talvez inválidas] de memória. Por exemplo, considerando a figura, a expressão  $*(c-1)$  (equivalente a  $c[-1]$ ) é uma expressão sintaticamente correta, e resulta em endereço de memória válido (pois existe), mas é semanticamente incorreta pois acessa um bloco de memória que não foi alocado para o vetor em questão.

#### 00-vetor.c

```
1  int vetor[10];
2  int i, soma = 0;
3
4  for(i = 0; i < 10; ++i) {
5      printf("Digite o %d-ésimo elemento: ", i);
6      scanf("%d", vetor+i);
7  }
8
9  printf("\nOs elementos são: ");
10 for(i = 0; i < 10; ++i)
11     printf("%d ", vetor[i]);
12
13 for(i = 9; i >= 0; --i)
14     soma += vetor[i];
15 printf("\nE a soma deles é: %d\n", soma);
```

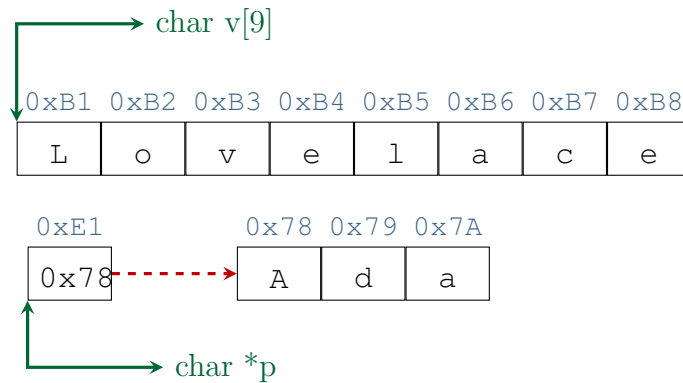
Na linguagem Python, os  $n$  elementos sus podem ser acessados pelo índice correto:  $[0, n)$ , mas - obviamente - não há aritmética de ponteiros. Entretanto, há certas variações que facilitam a vida. Por exemplo, o uso da função `len` e o uso de índices negativos para acessar os elementos em ordem inversa. No caso,

```
1 lista = [x for x in range(1, 10)] # lista = [1, 2, 3, 4, 5, 6, 7, 8, 9]
2
3 len(lista) # 10
4 lista.append(10) # lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
5
6 lista[0] # 1
7 lista[-0] # 1
8 lista[-1] # 10
9 lista[1:3] # [2, 3]
10
11 for i in range(1, len(lista) + 1):
12     print(lista[-i], end=' ') # Ordem inversa: 10 9 8 7 6 5 4 3 2 1
```

Vetores não são ponteiros, mas na linguagem C eles são praticamente equivalentes, e a relação é tão forte que eles devem ser discutidos juntos [1]. O vetor é conjunto contíguo de elementos homogêneos pré-alocados com posição e tamanho fixos, já o ponteiro é uma referência para um [tipo específico de] dado qualquer. Um ponteiro é uma variável, então  $a = p$  e  $p++$  são operações válidas; já um vetor não é uma variável, então  $a = v$  e  $v++$  *não* são válidas [6].

#### 01-vetor.c

```
1  char v[] = "Lovelace";
2  char* p = "Ada";
3
4  printf("%s (%p)\n", v, v);
5  printf("%s (%p)\n\n", p, p);
6
7  /* "v" é tipo 'vetor de 9 caracteres' */
8  /* "&v" é tipo 'ponteiro para ponteiro de caractere' */
9  printf(" sizeof(v) = %ld\n", sizeof(v));
10 printf("sizeof(&v) = %ld\n\n", sizeof(&v));
11
12 /* "p" é tipo 'ponteiro para caractere' */
13 /* "&p" é tipo 'ponteiro para ponteiro para caractere' */
14 printf(" sizeof(p) = %ld\n", sizeof(p));
15 printf("sizeof(&p) = %ld\n\n", sizeof(&p));
```



Na linguagem Python, listas são objetos *mutáveis*, portanto uma função pode alterar os elementos de uma lista dada como argumento:

```
1 def troca_primeiros(lista):
2     lista[0], lista[1] = lista[1], lista[0]
3
4 lista = [x for x in range(1, 10)]
5 troca_primeiros(lista)
6 lista # [2, 1, 3, 4, 5, 6, 7, 8, 9]
```

Na linguagem C, muito cuidado ao utilizá-los como argumento de funções - a situação de “equivalência” pode gerar uma série de problemas.

apc\_vetor.h

```
1 void mostra_i(int *vetor, int n) {
2     int i;
3
4     printf("%d", vetor[0]);
5     for(i = 1; i < n; ++i)
6         printf(", %d", vetor[i]);
7     printf("\n");
8 }
```

02-vetor.c

```
1 /* Retorna o maior elemento do vetor. */
2 int maior(int vetor[TAM]) {
3     /* O tipo é "vetor de TAM elementos". */
4     int i, maior = 0;
5
6     for(i = 1; i < TAM; ++i)
7         if(vetor[i] > vetor[maior])
8             maior = vetor[i];
9
10    return maior;
11 }
```

Certas aplicações interessantes de vetores exigem que seus elementos estejam ordenados (em relação ao conteúdo). Existem diversas formas de se fazer isso (consegue elaborar um algoritmo de ordenação?), cada uma com certas características. Por exemplo, para ordenar em ordem crescente, pode-se considerar o primeiro elemento e compará-lo com todos os demais, sempre que o primeiro for menor que o comparado, troca-se os conteúdos de posição. Desta forma, ao final de todas as comparações, o elemento na primeira posição será o menor de todos. Então basta repetir este procedimento para cada posição subsequente e, ao final da execução, tem-se o vetor ordenado.

```
1 for i in range(len(vetor) - 1):
2     for j in range(i + 1, len(vetor)):
3         if(vetor[i] > vetor[j])
4             vetor[i], vetor[j] = vetor[j], vetor[i]
```

Este simples algoritmo pode ser facilmente adaptado para ordenar o vetor em ordem decrescente. Na verdade, é possível abstrair ainda mais este procedimento com uma função de comparação, que indica uma ordem entre dois elementos. Então, supondo uma função `em_ordem` que indica se os elementos estão ordenados, o algoritmo pode ser redefinido como:

03-vetor.c

```
1 /* Aplica a função de comparação dada em todos os elementos
2  * do vetor, alterando suas posições de modo que fique
3  * ordenado. */
```

```

4 void ordena(int vetor[TAM], int (*em_ordem)(int, int)) {
5     int i, j;
6     for(i = 0; i < TAM; ++i)
7         for(j = i + 1; j < TAM; ++j)
8             if(!em_ordem(vetor[i], vetor[j]))
9                 troca_i(vetor+i, vetor+j);
10 }

```

Assim o algoritmo abstrai a função de comparação, e pode ser usado para realizar ordenações diferentes:

### 03-vetor.c

```

1 int crescente(int a, int b) { return (a < b); }
2 int decrescente(int a, int b) { return (a > b); }
3
4 int main() {
5     ordena(vetor, crescente);
6     ordena(vetor, decrescente);
7
8     return 0;
9 }

```

Por fim, a linguagem C tem “declarações complicadas” (seção 5.12 de [1]), é melhor entender o funcionamento que [tentar] adivinhar o tipo. Por exemplo:

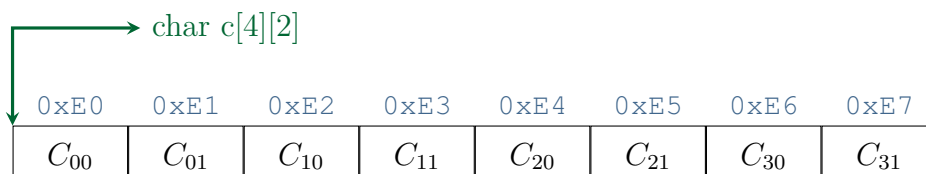
```

1 char **argv
2 /* argv: ponteiro para ponteiro para char */
3
4 int (* tabdia)[13]
5 /* tabdia: ponteiro para vetor[13] de int */
6
7 int * tabdia[13]
8 /* tabdia: vetor[13] de ponteiro para int */
9
10 void *comp()
11 /* comp: função retornando ponteiro para void */
12
13 void (* comp)()
14 /* comp: ponteiro para função retornando void */
15
16 char (*(*x())[1])()
17 /* x: função retornando ponteiro para vetor[] de ponteiro para função retornando char */
18
19 char (*(*x[3])[1])()[5]
20 /* x: vetor[3] de ponteiro para função retornando ponteiro para vetor[5] de char */

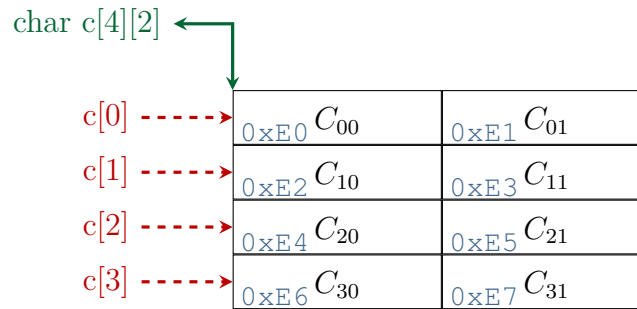
```

## 5.1 Vetores Bidimensionais

Um vetor é um bloco de memória de  $N$  elementos quaisquer, assim como um ponteiro armazena o endereço de memória de um elemento qualquer. Se há ponteiro para ponteiro, por que não um vetor de vetores?



Para o computador, o que ocorre é a criação de um vetor,  $c$ , de 4 elementos. Cada elemento é, por sua vez um vetor de 2 caracteres. Como a máquina conhece o tipo do elemento, a aritmética de ponteiros funciona como esperado.



São inúmeras as aplicações deste tipo de representação<sup>2</sup>. Um mapa pode ser representado como coordenadas em um plano cartesiano que pode ser representado por um vetor bidimensional (por exemplo, para implementar um jogo de Batalha Naval). Um texto, pode ser implementado como um vetor de strings (vetor de caracteres). Um programa pode ser representado como uma sequência de comandos (strings).

O padrão ANSI aceita duas variações de implementação da função `main`:

```
1 int main()
2 int main(int argc, char **argv)
```

A primeira versão não aceita argumentos, é a mais simples de implementar. Mas muitas vezes deseja-se que o programa lide com certos valores passados pela linha de comando, então utiliza-se a segunda versão, que recebe o inteiro `argc` (*argument count*), a quantidade de argumentos dada, e o vetor de strings `argv` (*argument values*), que contém o valor de cada argumento.

```
$ gcc hello_world.c -o hello_world
$ ./hello_world
$ cd workspace
```

Neste exemplo, o primeiro recebe 4 argumentos, o segundo comando apenas 1 e o terceiro 2. É simples manipular estes valores, e esta possibilidade de comunicação entre programas diferentes possibilita inúmeras aplicações.

10-main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char** argv) {
5     /* Assume que todos os argumentos são inteiros. */
6
7     int i, soma = 0;
8     for(i = 1; i < argc; ++i)
9         soma += atoi(argv[i]);
10
11     return soma;
12 }
```

Este programa, junto a um de subtração, multiplicação, e divisão poderia compor um programa *calculadora*. Ou algo mais complicado como o exemplo abaixo (ou o `gcc`).

11-main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int resultado;
```

<sup>2</sup>Mas na prática, são menos usados que vetores de ponteiros [1].

```

6  resultado = system("gcc -Wall -ansi 10-main.c -o soma");
7  printf("Resultado de gcc: %d \n", WEXITSTATUS(resultado));
8
9  if(resultado == EXIT_SUCCESS) {
10     resultado = system("./soma 50 -1 6 -47 2");
11     printf("Resultado da soma: %d\n", WEXITSTATUS(resultado));
12 }
13
14 return EXIT_SUCCESS;
15 }

```

“Grandes poderes trazem grandes responsabilidades.”

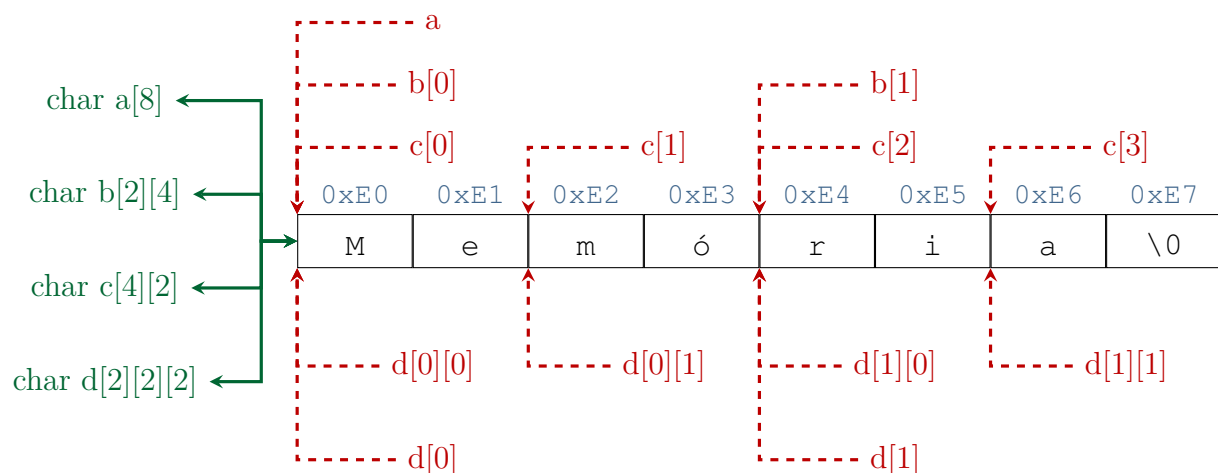
Ben Parker

O acesso a blocos de memória com ponteiros é algo extremamente útil se feito com a devida cautela (e sem maldade ou malícia - veja 12-main.c).

## 5.2 Vetores N-dimensionais

Um vetor é um bloco de memória de  $M$  elementos quaisquer, assim como um ponteiro armazena o endereço de memória de um elemento qualquer. Se há ponteiro para ponteiro para ponteiro, por que não um vetor de vetor de vetores?

Os mesmos princípios que se aplicam a 2, se aplicam a  $N$  vetores.



Vetores não são ponteiros, mas na linguagem C eles são praticamente equivalentes. Suponha uma referência  $v$  para um objeto do tipo  $T$ . Ao ser usada em um expressão,  $v$  é tratado como um ponteiro para seu 1º elemento, **exceto** se:

1.  $v$  é operando de `sizeof` ou `&` (veja 01-vetor.c); ou
2.  $v$  é um identificador para um vetor que está sendo inicializado (veja 02-vetor.c).

Ou seja, na chamada de uma função, ao passar um vetor, o que a função recebe é considerado um ponteiro para o tipo do vetor (o endereço do primeiro elemento). Isso não causa problemas em vetores unidimensionais, mas na passagem de vetores multidimensionais, por exemplo `int b[5][10]`, o vetor é considerado, dentro da função, como um ponteiro para inteiro (`int *b`), interferindo com a aritmética de ponteiros. Veja (com atenção) exemplos disso em 13-ponteiros.c).

## 6 Registros

Vetores são conjuntos de dados homogêneos, mas muitas vezes deseja-se lidar com dados heterogêneos. Um registro é uma estrutura que pode armazenar diferentes tipos de dados em um bloco de memória.



```

1 Algoritmo LeFuncionários
2 Definições
3     funcionario : registro (nome, endereço : string;
4                             sexo : caractere;
5                             código : inteiro;
6                             salário : real)
7 Variáveis
8     funcionários : vetor[1000] de funcionario
9     total, p100 : real
10 Início
11     total ← 0
12     Para i de 0 a 999 Faça
13         Leia(funcionários[i])
14         total ← total + funcionários[i].salário
15     FimPara
16     Para i de 0 a 999 Faça
17         p100 ← 100 * funcionários[i].salário / total
18         Escreva(funcionários[i].nome, "recebe ", p100, "%")
19     FimPara
20 Fim

```

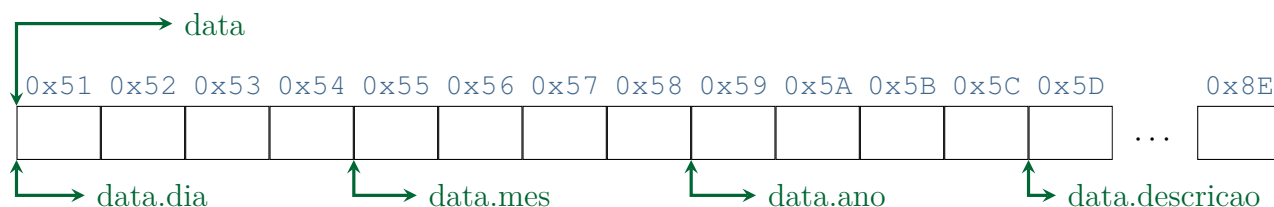
Na linguagem C, o registro é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, colocadas juntas sob um único nome para manipulação conveniente [1]. É definido pela palavra-chave `struct`, e o acesso a seus componentes pelo identificador e o caractere `'.'`.

00-data.c

```

1  /* Definição da estrutura do registro (identificado como "data"): */
2  struct {
3      int dia, mes, ano; /* A "data" em si. */
4      char descricao[50]; /* Uma descrição da data. */
5  } data;
6  printf("Digite a descrição: ");
7  scanf("%[^\n]", data.descricao);
8  printf("Digite o ano: ");
9  scanf("%d", &(data.ano));
10 printf("Digite o mês: ");
11 scanf("%d", &(data.mes));
12 printf("Digite o dia: ");
13 scanf("%d", &(data.dia));
14
15 /* Acesse as informações armazenadas e mostre-as. */
16 printf("%s:\n%02d/%02d/%04d\n", data.descricao,
17                                     data.dia,
18                                     data.mes,
19                                     data.ano);

```



A disposição dos elementos na memória é sequencial, *na mesma ordem em que são declarados* na definição do registro. A codificação pode ser facilitada pela definição um tipo (que deve ser adequadamente identificado). Um registro pode ser composto por outros registros, utilizado como argumento de função ou elemento de um vetor, ser manipulado por ponteiros, etc. (leia com atenção o arquivo 01-multiplos.c).

Um aspecto interessante é que, por ser uma coleção arbitrária, um registro não pode ser comparado a outro diretamente [1], mas é bem simples definir uma ordem sobre eles:

## 02-ordem.c

```
1 /* Indica ordem alfabética, utilizando a cronológica em caso
2  * de igualdade. */
3 int alfabetica_cronologica(aniversario_t a, aniversario_t b){
4     int em_ordem = alfabetica(a,b);
5     return (em_ordem ? em_ordem : cronologica(a, b));
6 }
```

Em linguagem C, para evitar cópia de muitos bytes de memória, geralmente é mais eficiente se passar um ponteiro de um registro para uma função [1]. Muita atenção ao conceito de registro e de funções associadas a sua manipulação, são o primeiro passo rumo a Orientação a Objetos (na verdade, o conceito de registro em C é simplesmente uma variante do conceito de classe em C++ [3])...

A linguagem Python não tem a mesma abordagem da linguagem C. Geralmente uma coleção de dados [heterogêneos] é definida com a estrutura de dados `dict`, mas outra abordagem é a utilização do mecanismo de classes (que acaba utilizando um `dict`, mas os detalhes estão fora do escopo). Uma classe é definida pela palavra-chave `class`, a função `__init__` e seus componentes, que são acessados pelo identificador e o caractere `'.'`.

## 00-data.py

```
1 # Definição do registro:
2 class Data():
3     def __init__(self):
4         # Função de inicialização de valores, é executada
5         # sempre que uma nova instância é criada.
6         dia, mes, ano = 0, 0, 0 # A "data" em si.
7         descricao = ''         # Uma descrição da data.
8
9 data = Data()
10
11 data.descricao = input('Digite a descrição: ')
12 data.ano = int(input('Digite o ano: '))
13 data.mes = int(input('Digite o mês: '))
14 data.dia = int(input('Digite o dia: '))
15
16 # Acesse as informações armazenadas e mostre-as.
17 print('%s:\n%02d/%02d/%04d\n' % (data.descricao,
18                                 data.dia,
19                                 data.mes,
20                                 data.ano))
```

Diferentemente da linguagem C, a disposição dos elementos na memória não é necessariamente é sequencial, mas um registro em Python pode, entre outras coisas, ser composto por outros registros (veja o arquivo 01-multiplos.py). A comparação direta também é possível se houver uma ordem sobre eles:

## 02-ordem.py

```
1 def cronologica(a, b):
2     if a.data.ano != b.data.ano:
3         return a.data.ano < b.data.ano
4     if a.data.mes != b.data.mes:
5         return a.data.mes < b.data.mes
6     return a.data.dia <= b.data.dia
```

## 7 Arquivos

Um arquivo de computador é um espaço de memória cujas características dependem do tipo de dispositivo em que está armazenado. De forma abstrata, um arquivo é uma sequência de blocos [de

bytes] de dados. Cada arquivo está associado a um nome, que, junto ao seu *caminho*, o identifica unicamente no sistema operacional.

As operações mais comuns são *leitura/escrita* de dados. A manipulação de memória é gerenciada pelo sistema, mas *é responsabilidade do programador iniciar e finalizar o processo*. Nem sempre as operações são bem sucedidas, é preciso ficar atento a erros.

Na linguagem C, os arquivos são manipulados por um ponteiro para o tipo FILE, que identifica a sequência de bytes do arquivo. O conjunto de dados é finalizado com o byte EOF (*end of file*). As principais funções para manipulação de arquivos estão na biblioteca `stdio.h`, mas lembre-se que antes de ser lido/gravado, um arquivo deve ser *aberto* [1].

**Acesso:** (`fopen/fclose` e `fseek`) A ação de abrir um arquivo permite ao usuário localizar e abrir um canal de comunicação com o meio de armazenamento [do arquivo] e pode inviabilizar alterações por outros processos; e fechar o arquivo interrompe este canal de comunicação.

**E/S:** (`fread/fwrite` e `fscanf/fprintf`) Uma vez aberto o canal, é relativamente simples ler/escrever uma quantidade determinada de bytes nele.

00-binario.c

```
1 FILE *fp = fopen(arquivo, "wb+"); /* "b" de "binário"... */
2 if((fp == NULL) && (!fp) && (fp == 0)) {
3     /* Os testes são equivalentes. */
4     printf("Não foi possível abrir \"%s\".\n", arquivo);
5     return EXIT_FAILURE;
6 }
7
8 /* Escreve no arquivo na ordem: string -> real -> inteiro. */
9 fwrite(str, sizeof(str), 1, fp);
10 fwrite(&d, sizeof(d), 1, fp);
11 fwrite(&i, sizeof(i), 1, fp);
12
13 /* O programador é responsável pelo arquivo. */
14 fclose(fp);
```

Como de costume, bastam o endereço inicial e a quantidade de bytes a ser manipulada para que tudo funcione como esperado. Mas nem tudo são flores. O arquivo existe? O usuário tem permissão para abrir o arquivo? O arquivo está disponível para acesso? Estas (entre outras) situações implicam que a instrução (linha 1) pode não ser bem sucedida...

Não se preocupe (muito), é fácil lidar com isto já que só há duas possibilidades a serem tratadas (linhas 2 a 6). O que fazer em caso de não ser possível abrir o arquivo depende da aplicação e do desenvolvedor; no entanto:

*Uma vez aberto o arquivo, é responsabilidade do programador garantir que ele seja fechado corretamente.*

Portanto a linha 14 é essencial...

Na linguagem Python, os arquivos são manipulados por um objeto *file*, que identifica a sequência de bytes do arquivo. O conjunto de dados é finalizado com o byte EOF. As principais funções para manipulação de arquivos fazem parte do objeto, e como nas demais linguagens, antes de ser lido/gravado, um arquivo deve ser *aberto* (e depois *fechado*).

**Acesso:** (`open/close` e `seek`) A ação de abrir um arquivo permite ao usuário localizar e abrir um canal de comunicação com o meio de armazenamento [do arquivo] e pode inviabilizar alterações por outros processos; e fechar o arquivo interrompe este canal de comunicação.

**E/S:** (`read/write`) Uma vez aberto o canal, é extremamente simples ler/escrever informações nele.

## 00-binario.py

```

1 f = open(arquivo, 'wb') # 'b' de 'binário'...
2 if(f.closed):
3     print('Não foi possível abrir \'%s\'' % arquivo)
4     return False
5
6 # Escreve no arquivo na ordem: string -> real -> inteiro.
7 f.write(struct.pack('50s', msg.encode('ASCII')))
8 f.write(struct.pack('d', d))
9 f.write(struct.pack('i', i))
10
11 # O programador é responsável pelo arquivo.
12 f.close()

```

Como em C, o arquivo é aberto no início, e basta indicar a quantidade de bytes a ser manipulada para que tudo funcione como esperado. É possível verificar se foi possível abrir o arquivo. Mas Python tem como *boa prática de programação* o uso do comando `with`, o que garante que o arquivo seja fechado quando a execução sair do bloco (dentro do `with`). Como é responsabilidade do programador garantir que ele seja fechado corretamente, é fácil perceber esta vantagem. A linguagem Python também tem uma abordagem diferente indicada para lidar com possíveis erros:

“É mais fácil pedir perdão do que permissão.”

Grace Hopper

Eis um exemplo:

## 00-binario.py

```

1 try:
2     with open(arquivo, 'rb') as f:
3         msg = f.read(50) # tem de ser do tamanho certo!
4         d = struct.unpack('d', f.read(8))
5         i = struct.unpack('i', f.read(4))
6
7         print('string = %s' % msg.decode('ASCII'))
8         print('double = %lf' % d)
9         print('int = %d' % i)
10 except:
11     print('Erro ao manipular o arquivo \'' + arquivo + '\'.')

```

Ou seja, *tente* (`try`) realizar uma operação e, caso ocorra um *erro*, lide com esta situação excepcional adequadamente (no exemplo, com uma mensagem dada na linha 11).

Os arquivos são armazenados na memória na forma binária, claro, mas para facilitar a interação com usuários humanos, é possível utilizar formas mais amigáveis (veja também os outros exemplos):

## 03-texto.c

```

1 const string str = "Algoritmos e Programação de Computadores";
2 const double d = 12.23;
3 const int i = 101;
4
5 fprintf(fp, "%s\n%lf\n%d", str, d, i);

```

## 04-texto.py

```

1 with open(arquivo, 'r') as f:
2     conteudo_original = f.read()
3
4 backup = 'copia_de_' + arquivo
5 with open(backup, 'w') as f:
6     f.write('# -*- coding: utf-8 -*-\n'
7           '# @package: ' + backup + '\n')

```

```
8          '# @author: [gerado automaticamente]\n'
9          '# @disciplina: Algoritmos e Programação de Computadores\n'
10         '#\n'
11         '# Exemplo de manipulação de arquivos. Este arquivo foi\n'
12         '# gerado a partir de \'' + arquivo + '\'. \n\n')
13         f.write(conteudo_original)
```

## Referências

- [1] Brian W. Kernighan and Dennis M. Ritchie. *C: a linguagem de programação padrão ANSI*. Campus, Rio de Janeiro, 1989.
- [2] Aaron M Tenenbaum, Yedidyah Langsam, and Moshe Augenstein. *Estruturas de dados usando C*. Pearson Makron Books, São Paulo (SP), 1995.
- [3] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, Mass., 3. ed., 20. print edition, 2004.
- [4] Allen Downey, Chris Meyer, and Jeffrey Elkner. *How to think like a computer scientist: learning with Python*. Green Tea Press, Wellsley, Mass., 2002.
- [5] Edsger Wybe Dijkstra. Why numbering should start at zero. <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html>, August 1982. Acessado em: 2015-02-14.
- [6] Peter VanDerLinden. *Expert C programming: deep C secrets*. SunSoft Press, Mountain View, Calif, 1994.