



# Estruturas de Dados

Prof. Guilherme N. Ramos

## 1 Estruturas de Dados

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]. 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 [3], 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?* Essa é uma decisão arbitrária extremamente importante, pois dados *diferentes* podem ser representados por um mesmo [conjunto de] byte[s]. Por exemplo:

	$\mathbb{N}$	$\mathbb{Z}$	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:

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

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:

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

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

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).

$$000 \rightarrow +(0 \cdot 2 + 0) = +0$$

$$001 \rightarrow +(0 \cdot 2 + 1) = +1$$

$$010 \rightarrow +(1 \cdot 2 + 0) = +2$$

$$011 \rightarrow +(1 \cdot 2 + 1) = +3$$

$$100 \rightarrow -(0 \cdot 2 + 0) = -0$$

$$101 \rightarrow -(0 \cdot 2 + 1) = -1$$

$$110 \rightarrow -(1 \cdot 2 + 0) = -2$$

$$111 \rightarrow -(1 \cdot 2 + 1) = -3$$

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:

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

## 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 `2-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 [3]; mas há um “pequeno problema” (veja `3-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?

```

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: 3-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 [2]. Isso pode ser um problema, pois afeta a portabilidade do código.

## 2.4 Símbolos

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

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, 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]

## 3 Ponteiros

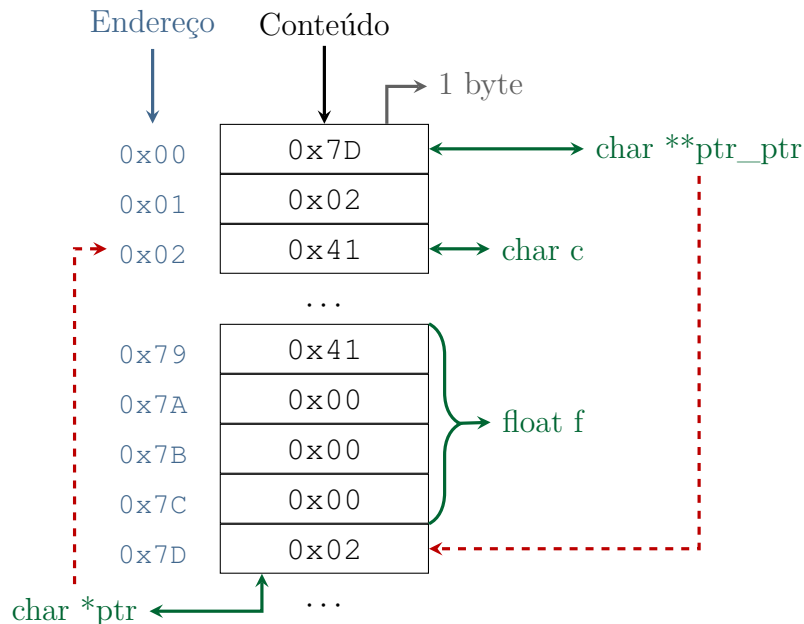
Cada variável declarada ocupa um espaço na memória, conforme seu tipo, e nome da variável é apenas uma forma “amigável” de lidar com o endereço deste espaço. Ponteiro (ou *apontador*) é um tipo de dado que armazena um *endereço de memória*, possibilitando leitura e escrita deste endereço. São muito usado 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 [2].

Em linguagem C, um ponteiro 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  int**  ptr_ptr_int; /* ponteiro para (ponteiro para inteiro) */

```



O endereço de memória identifica o espaço físico na memória dos bytes que armazenam a informação. Por exemplo, na figura acima, o *endereço* 0x02 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 0x02, 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 0x7D 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 0x02). 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*.

#### 0-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);
```

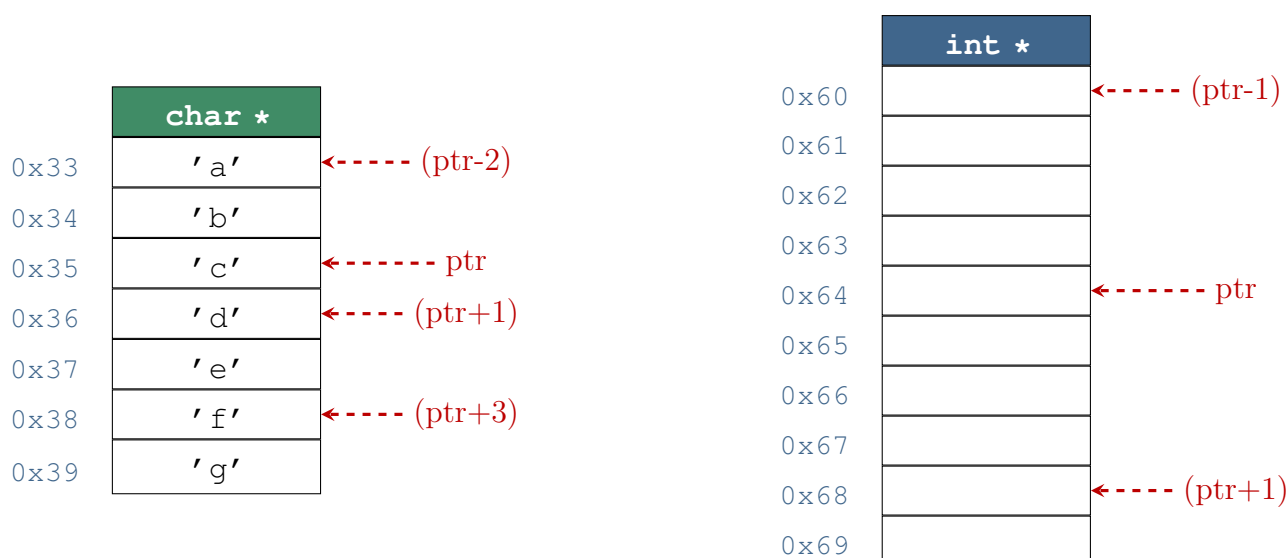
#### 1-tipos.c

```
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 Python, 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. O comportamento de “passagem por referência” pode ser obtido de outras formas.

A linguagem C é fortemente tipada, portanto ao declarar um ponteiro já se sabe 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 2-ponteiro.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 3-escopo.c.

apc\_ponteiro.h

```
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:

4-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;  
8         (*r2) = (-b - sqrt(delta))/2;  
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?

5-funcao.c

```
1 #include "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 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 (`2-raizes.c`).

## 4 Vetores

É fácil manipular um dado para resolver um problema:

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



Mas e  $n$  problemas?

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

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 memória) 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
?	?	?	?	?	?	?	?	?	?

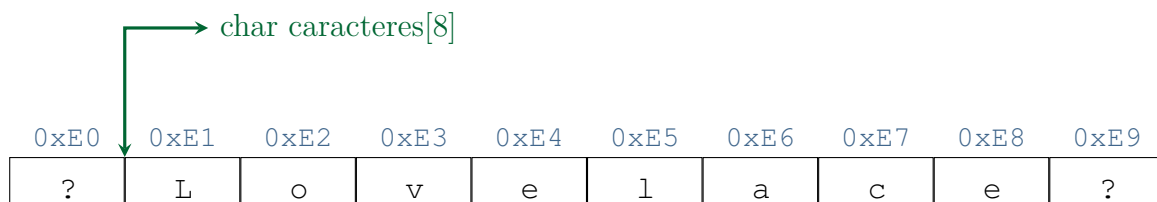
Portanto, para lidar com um vetor basta saber duas coisas: o endereço do primeiro elemento e quantos são os elementos armazenados. Na Linguagem C, fica fácil declarar um vetor:

```
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]);
```



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

O acesso correto aos elementos depende também do tamanho do vetor. 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.

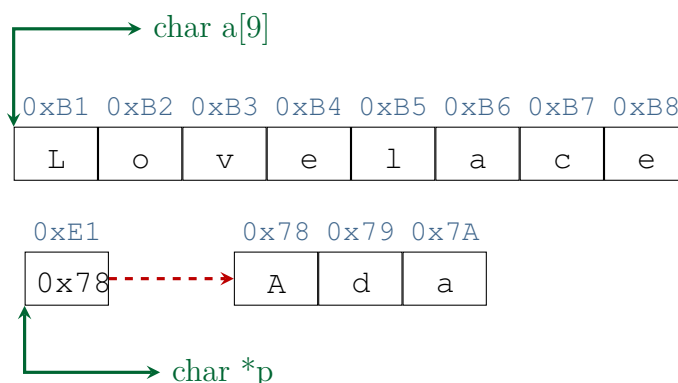
#### 0-vetor.c

```
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("\nA soma deles é: %d\n", soma);
```

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.

#### 1-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));
```



Muito cuidado ao utilizá-los como argumento de funções.

## 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 }
9

```

## 2-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 = 0; i < n; ++i)
2     for(j = i + 1; j < n; ++j)
3         if(vetor[i] > vetor[j])
4             troca_i(vetor+i, vetor+j);

```

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:

## 3-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:

## 3-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 [tentar] entender o funcionamento que 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())[])()
17 /* x: função retornando ponteiro para vetor[] de ponteiro para função retornando
    char */
18
19 char (*(*x[3])[]())[5]
20 /* x: vetor[3] de ponteiro para função retornando ponteiro para vetor[5] de char */

```

## 4.1 Strings

Uma *palavra/frase* é um conjunto finito e ordenado de símbolos. Um *string* é um vetor de caracteres, a “melhor” forma de comunicação com humanos.

4-string.c

```

1  const int TAM = 50;
2  char string[TAM];
3
4  printf("Digite um string: ");
5
6  /* Lê o string. */
7  scanf("%s", string);
8
9  /* Mostra os índices do vetor de caracteres
10 para facilitar a visualização de seu
11 conteúdo. */
12 mostra_indices(TAM, "          ");
13 mostra_n_chars(string, 50);

```

O código acima funciona muito bem, mas tem um comportamento interessante quando se digita uma frase menor que os 50 caracteres definidos. Entretanto, forçar a utilização de frases com tamanho fixo não é uma opção viável. A solução de melhor custo/benefício é obter um vetor “grande o suficiente” para a tarefa em questão, de modo que haja um tamanho restringindo a aplicação (por exemplo, 140 caracteres), e utilizar uma marcação específica para indicar o término dos símbolos de interesse.

## 5-string.c

```

1  const int TAM = 50;
2  char string[TAM];
3
4  printf("Digite um string: ");
5
6  /* Lê o string, limitando a 50 símbolos. */
7  scanf("%50[^\n]", string);
8
9  /* Mostra os índices do vetor de caracteres
10 para facilitar a visualização de seu
11 conteúdo. */
12 mostra_indices(TAM, " ");
13 mostra_n_chars(string, 50);

```

## 6-string.c

```

1  const int TAM = 50;
2  char frase[TAM];
3
4  printf("Digite um string: ");
5  scanf("%[^\n]", frase);
6
7  mostra_indices(TAM, " ");
8  /* Assume-se que o string termina
9 em '.'. */
10 mostra_ate_char(frase, '.');

```

'.' não é uma boa escolha de símbolo terminador (afinal, sempre aparece alguém dizendo “*Hello World!*”), então escolheu-se o símbolo ‘\0’ (o primeiro caractere da tabela ASCII) como terminador de um string para representação interna [1]. Assumindo que *todo string segue este padrão*, pode-se ignorar o tamanho do vetor e simplesmente percorrê-lo até encontrar o caractere de término.

Algumas ideias interessantes para lidar com strings são: contagem do tamanho de uma palavra, de palavras em um texto, comparação de strings, transformação em letras maiúsculas/minúsculas, criptografia, etc..

Considere o código abaixo, e veja se percebe o que está errado:

## 8-memoria.c

```

1  char* buffer;
2
3  printf("Digite uma frase: ");
4  scanf("%s", buffer);
5  printf("Você digitou [%s]\n", buffer);

```

scanf armazena os caracteres digitados no endereço indicado por buffer. Como a variável não foi inicializada, não é possível dizer qual é este endereço. Se buffer é o caractere nulo (NULL), então scanf tentará armazenar os caracteres em um local proibido e a execução do programa será interrompida. Senão, buffer tem um número (que estava já armazenado na memória por algum motivo) que é interpretado como um endereço de memória onde scanf tentará armazenar os caracteres. Embora sintaticamente correto, esta é uma ação perigosa pois tal espaço não foi alocado para seu uso, então o programa estará trabalhando em um local que pode estar sendo usado por outro programa e, portanto, que poderia atrapalhar a execução do seu ou, pior ainda, o seu atrapalhar a execução dele (por exemplo, sobrescrevendo um pedaço do arquivo que contém a declaração do imposto de renda).

Uma solução é definir buffer como um vetor (digamos, de tamanho 50). Mas esta solução é parcialmente correta, pois scanf continuará escrevendo na memória caso o usuário forneça mais caracteres que o vetor pode armazenar. É necessário, então, limitar a quantidade de caracteres que pode ser considerada:

## 8-memoria.c

```

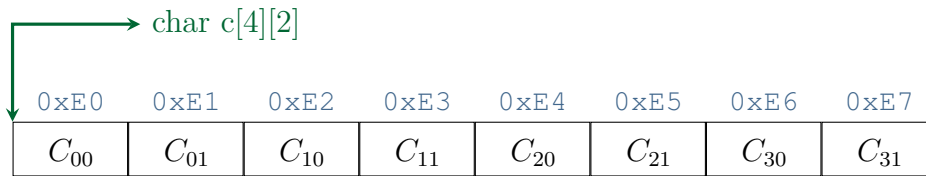
1  char buffer[51]; /* +1 para o caractere de término. */
2
3  printf("Digite uma frase: ");
4  scanf("%50[^\n]", buffer);
5  printf("Você digitou [%s]\n", buffer);

```

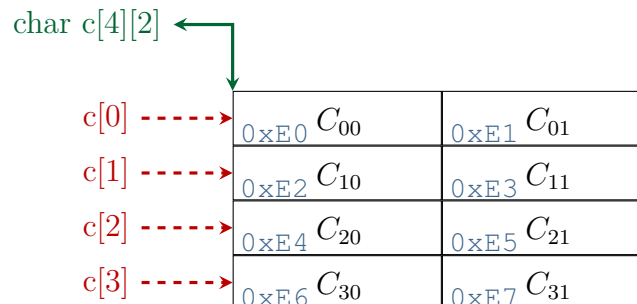
É importante considerar este tipo de programação defensiva, pois nunca se sabe o que o usuário vai fazer (nem como estas vulnerabilidades podem ser exploradas).

## 4.2 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 vector*), 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. */
```

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

```

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;
6
7     resultado = system("gcc -Wall -ansi 10-main.c -o soma");
8     /* Assumindo que o processo terminou sem erros, deve-se usar a macro WEXITSTATUS.
9      Mais detalhes em: http://stackoverflow.com/questions/808541/any-benefit-in-using-wexitstatus-macro-in-c-over-division-by-256-on-exit-status */
10    printf("Resultado de gcc: %d \n", WEXITSTATUS(resultado));
11
12    if(resultado == EXIT_SUCCESS) {
13        resultado = system("./soma 50 -1 6 -47 2");
14        printf("Resultado da soma: %d\n", WEXITSTATUS(resultado));
15    }
16
17    return EXIT_SUCCESS;
18 }

```

“Grandes poderes trazem grandes responsabilidades.”

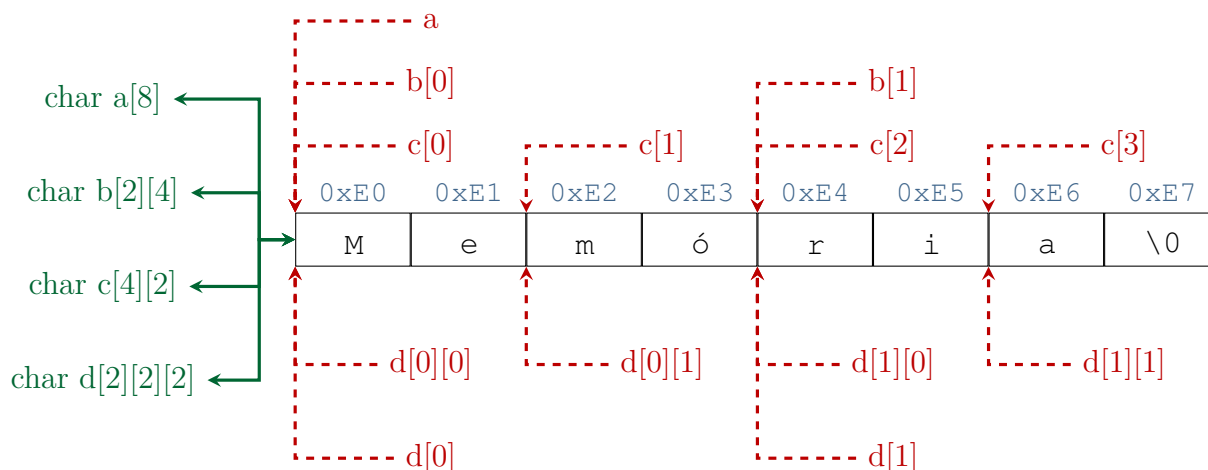
Ben Parker

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).

### 4.3 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 uma expressão,  $v$  é tratado como um ponteiro para seu 1º elemento, **exceto** se:

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

## Referências

- [1] Brian W. Kernighan and Dennis M. Ritchie. *C: a linguagem de programação padrão ANSI*. Campus, Rio de Janeiro, 1989.
- [2] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, Mass., 3. ed., 20. print edition, 2004.
- [3] Aaron M Tenenbaum, Yedidyah Langsam, and Moshe Augenstein. *Estruturas de dados usando C*. Pearson Makron Books, São Paulo (SP), 1995.