

Métodos e Técnicas de Programação ::: Ponteiros, Vetores e Strings :::

Prof. Igor Peretta

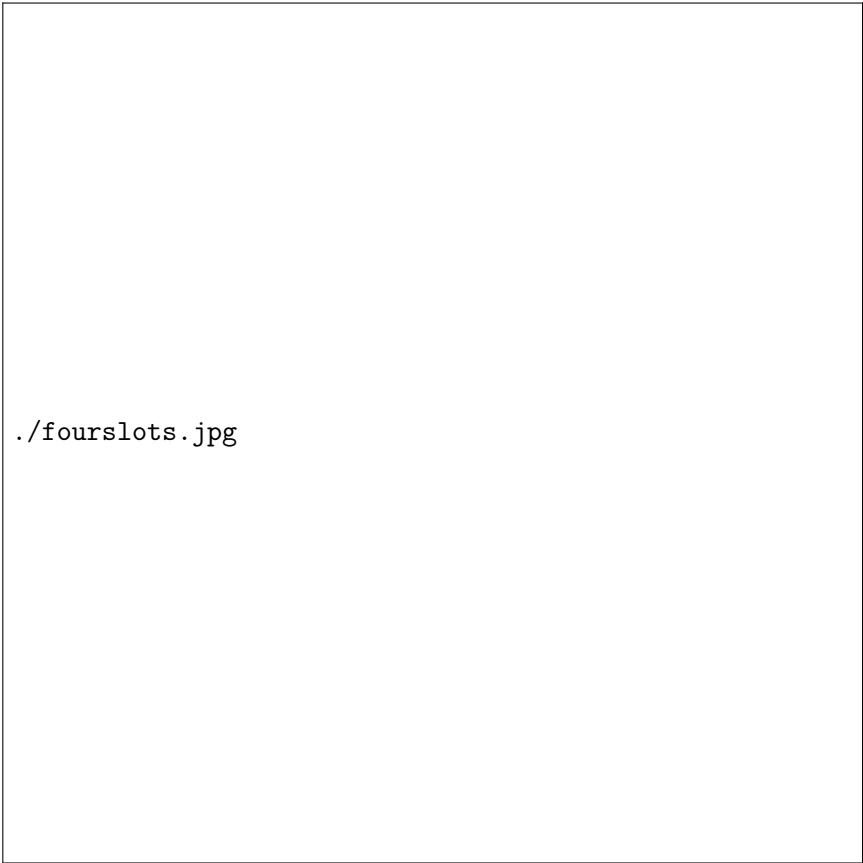
2018-2

Contents

1	Memória do computador	1
1.1	Célula de memória	4
1.2	Esquemático	6
1.3	Endereçamento	6
2	Ponteiros	6
2.1	Abstração em C	6
2.2	Síntese (Declaração, Inicialização, Acesso)	8
2.3	Aritmética de ponteiros	9
2.3.1	Exemplo	10
3	Array (vetor)	11
3.1	Declaração	11
3.2	Inicialização	11
3.3	Acesso	12
3.4	Tamanho de um vetor	12
3.5	Diferença com ponteiros	13
4	Strings	13
5	Síntese	14

1 Memória do computador

Os circuitos integrados de memória mais usados hoje em dia são da classe DDR, sigla que significa *Double data rate synchronous dynamic random-*



`./fourslots.jpg`

Figure 1: Memória do computador: pentes encaixados nos respectivos *slots* da placa mãe.



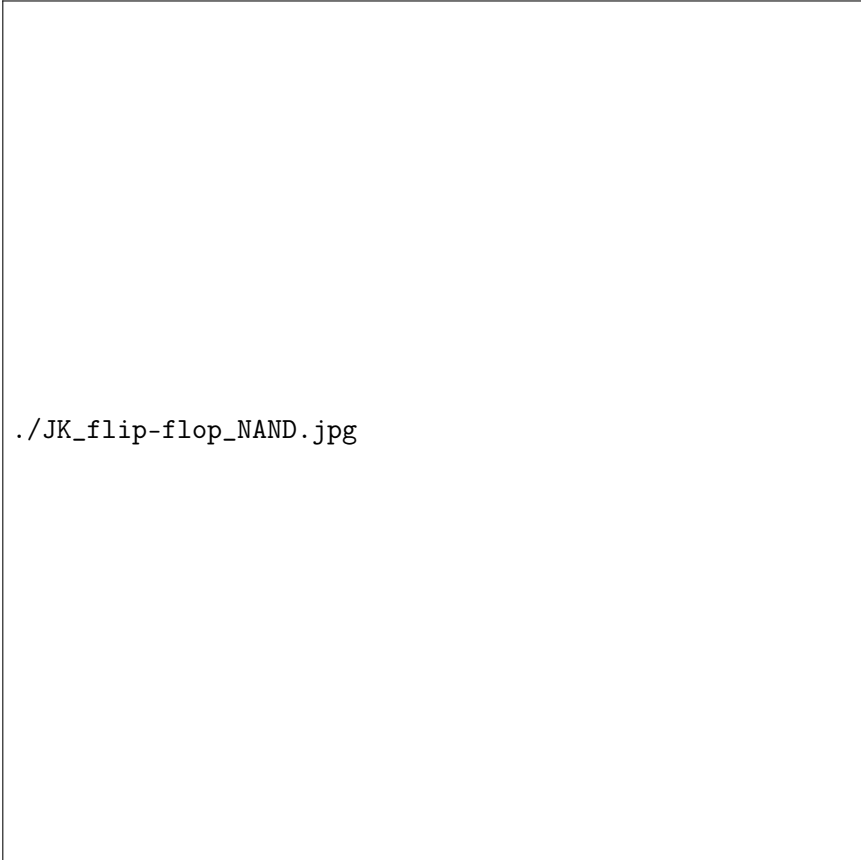
./memDDR2DDR3.jpg

Figure 2: Diferenças no desenho de memórias do tipo DDR2 e DDR3.

access memory ou **DDR SDRAM**. As versões são identificadas por DDR1, DDR2, DDR3 (ainda muito usadas) e DDR4 (as mais modernas).

1.1 Célula de memória

A célula de memória é um circuito eletrônico que armazena **um bit** de informação binária. O comando *set* armazena o valor lógico 1 (nível alto de tensão) e comando *reset* armazena o valor lógico 0 (nível baixo de tensão). Assumindo que o computador não desligue, qualquer desses valores é mantido enquanto não for mudado por um processo de *set* ou *reset*. Esse valor também pode ser acessado por um comando de leitura.



./JK_flip-flop_NAND.jpg

Figure 3: O flip-flop permitiu o desenvolvimento de memórias síncronas. Esquemático com portas lógicas NAND.

./SRAM_Cell.jpg

Figure 4: Esquemático de flip-flop com transistores.

Table 1: Tabela da verdade de um flip-flop JK.

J	K	Comentário	Q_{next}
0	0	mantém	Q
0	1	<i>reset</i>	0
1	0	<i>set</i>	1
1	1	troca	Q_{barra}

Um dos parâmetros de memórias é a frequência de acesso (as mais comuns em torno de 1000 a 2000 MHz). É a frequência de *clock* de acesso a suas células.

```
unsigned int nGb = 4;
unsigned long long int Gb = 1024*1024*1024; // em bytes
unsigned long long int nbits = 8*nGb*Gb; // em bits
printf("%u giga bytes [Gb] de memoria equivalem a %llu bits\n", nGb, nbits);
printf("ou aproximadamente %.1f bilhoes de transistores.\n", 6.0f*nbits/1.0e9);
```

4 giga bytes [Gb] de memoria equivalem a 34359738368 bits
ou aproximadamente 206.2 bilhoes de transistores.

1.2 Esquemático

1.3 Endereçamento

A maioria dos computadores modernos endereçam **bytes**, ou seja, cada endereço de memória se refere a um único byte ou 8 bits de armazenamento. Dados que precisem de mais bits para serem armazenados podem residir em múltiplos bytes ocupando uma **sequência de endereços consecutivos**.


O tamanho de uma palavra (word) é característico de cada arquitetura. Denota o número de dígitos que uma CPU pode processar ao mesmo tempo. A maioria dos computadores de uso geral possuem uma palavra de 32 ou 64 bits. Historicamente temos uma variedade grande, incluindo 8, 9, 10, 12, 18, 24, 36, 39, 40, 48 e 60 bits.

Frequentemente, quando nos referimos ao tamanho da palavra de computadores modernos, também nos referimos ao espaço de endereços do mesmo. Por exemplo, um computador com o *hardware* de "32 bits" normalmente permite endereços de memória com 32 bits. Isso significa que é possível endereçar no máximo $2^{32} = 4.294.967.296$ diferentes bytes de memória (4 Gb). Note que arquivos maiores do que isso precisam de uma capacidade maior de endereçamento (ver FAT, FAT32 e NTFS no Windows).

2 Ponteiros

2.1 Abstração em C

Pensar tudo em C em termos de memória pode ajudar. Podemos abstrair que a memória de um computador é um vetor de bytes onde cada endereço pode armazenar 1 byte. Um computador com 64K de memória, por exemplo,



`./schema.jpg`

Figure 5: Diagrama de blocos de uma memória. Note a pinagem: controle (portas de entrada de comandos); registro de endereçamento (entrada de endereços); e dados (entrada/saída de informação na memória).

pode armazenar 65.536 elementos nesse vetor. Podemos falar de ponteiros como se armazenassem os índices desse vetor. Usando o conteúdo desses ponteiros seria como acessar o respectivo elemento desse vetor.

Na realidade, a memória não é necessariamente consecutiva e nem sequencial, mas essa analogia simplifica o jeito de pensarmos memória em C.

```
unsigned int dado = 0xA1B2C3D4;
unsigned char* addr = (unsigned char*) &dado;
printf("Endereco base do dado: %p\n", addr);
printf("Tamanho em bytes da variavel: %lu bytes\n", sizeof(dado));
printf("> endereco: %p contem %X\n", addr, addr[0]);
printf("> endereco: %p contem %X\n", addr+1, addr[1]);
printf("> endereco: %p contem %X\n", addr+2, addr[2]);
printf("> endereco: %p contem %X\n", addr+3, addr[3]);
```

```
Endereco base do dado: 0x7ffc8cbceb3c
Tamanho em bytes da variavel: 4 bytes
> endereco: 0x7ffc8cbceb3c contem D4
> endereco: 0x7ffc8cbceb3d contem C3
> endereco: 0x7ffc8cbceb3e contem B2
> endereco: 0x7ffc8cbceb3f contem A1
```

2.2 Síntese (Declaração, Inicialização, Acesso)

Sobre ponteiros:

Table 2: Entendendo ponteiros em C.

Símbolo	Exemplo	Significado
*	int *p; int i = 5;	declaração: <tipo> *<nome>
&	p = &i; // 0x3AF2	endereço ou referência
*	printf("%i\n", *p);	conteúdo ou dereferência

```
int * p;
int i = 5;
p = &i;
printf("%d\n", *p);
printf("%p\n", p);
```


5

0x7ffd7fad2c7c

Note que **referência** é quando nos referimos diretamente ao identificador do endereço da memória e **dereferência** é quando nos referimos ao valor contido no endereço armazenado.

Para visualizar o conteúdo de um ponteiro, devemos usar o especificador "%p" no `printf()`.

```
int x = 10, y = 20;
int *p = &x;
printf("no endereço %p temos armazenado o dado %d.\n",&x,x);
printf("o ponteiro para %p esta armazenado em %p.\n",p,&p);
p = &y;
printf("no endereço %p temos armazenado o dado %d.\n",&y,y);
printf("o ponteiro para %p esta armazenado em %p.\n",p,&p);
```

no endereço 0x7ffc74730bd8 temos armazenado o dado 10.
o ponteiro para 0x7ffc74730bd8 esta armazenado em 0x7ffc74730be0.
no endereço 0x7ffc74730bdc temos armazenado o dado 20.
o ponteiro para 0x7ffc74730bdc esta armazenado em 0x7ffc74730be0.

2.3 Aritmética de ponteiros

O **tipo** do ponteiro é muito importante para a aritmética de ponteiros. É ele que irá definir a unidade das operações aritméticas. Tal unidade será igual ao `sizeof(*tipo*)` em bytes.

Table 3: Operações de aritmética de ponteiros.

Operador	Exemplo	Significado
+	(p+n)	Referência n "sizeofs" à frente
-	(p-n)	Referencia n "sizeofs" atrás
++	p++	Incrementa o endereço em p
--	p--	Decrementa o endereço em $pn > n$
	p[n]	Mesmo que $*(p+n)$

O valor NULL pode ser atribuído a um ponteiro quando deseja-se que o mesmo aponte para nenhum lugar.

O tipo **void** é aceito para ponteiros e significa que o ponteiro **não** tem um tipo atrelado a ele. Como esses ponteiros não tem um tipo, a aritmética de ponteiros não funciona com eles.

```
int v[] = { 1, 2, 3 };
void *p;// = NULL;
p = v + sizeof(v)/sizeof(int);
printf("%d; %d\n", *(v+1), *(((int *)p)-1));

2; 3
```

2.3.1 Exemplo

Vamos investigar o que existe em 64 bits de memória:

```
unsigned char *p;
unsigned long long int s = 0x65636F7620616C4F;
p = (unsigned char *) &s;
printf("%lld\n", s);
for(; p < ((unsigned char *) &s) + sizeof(s); p++)
    printf("%p : %X (char = '%c')\n", p, *p, *p);
```

```
7305805573665156175
0x7ffe02183838 : 4F (char = 'O')
0x7ffe02183839 : 6C (char = 'l')
0x7ffe0218383a : 61 (char = 'a')
0x7ffe0218383b : 20 (char = ' ')
0x7ffe0218383c : 76 (char = 'v')
0x7ffe0218383d : 6F (char = 'o')
0x7ffe0218383e : 63 (char = 'c')
0x7ffe0218383f : 65 (char = 'e')
```

Podemos notar que o armazenamento em memória vai do byte **menos significativo** ao byte **mais significativo** (formato *little-endian*). Já dentro da célula de memória (byte), a ordem se inverte: começa com o **mais significativo** até chegar ao **menos significativo** (formato *big-endian*). Isso tem a ver com o rojeto do microprocessador.

```
int *p = NULL;
int v[] = {0,1,2,3,4,5}; // 6 elementos consecutivos
p = v;
for(; p < v + 6; p++)
    printf("%p : %X\n", p, *p);
```

```
0x7ffc687a9130 : 0
0x7ffc687a9134 : 1
0x7ffc687a9138 : 2
0x7ffc687a913c : 3
0x7ffc687a9140 : 4
0x7ffc687a9144 : 5
```

3 Array (vetor)

Chamamos de vetor um tipo de estrutura de dados que pode armazenar coleções sequenciais com tamanho fixo de elemento do mesmo tipo. Uma coleção de dados esta que frequentemente é mais útil pensar como uma coleção de variáveis do mesmo tipo.

3.1 Declaração

Vetores são declarados da seguinte maneira: `<tipo> <nome>[<tamanho>...]`

E podem ser multidimensionais:

```
int s[2][3][4];
int i,j,k;
for(i = 0; i < 2; i++)
    for(j = 0; j < 3; j++)
        for(k = 0; k < 4; k++)
            s[i][j][k] = i + j + k;
printf("%i\n",s[1][2][2]);
```

5

3.2 Inicialização

A inicialização de um vetor em C pode ser através de uma única linha:

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Note que o número de elementos entre as chaves `{}` não pode ser maior que o número de elementos declarados entre colchetes `[]`.

Podemos omitir o tamanho, o compilador se encarrega de criar um vetor do tamanho necessário.

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Para alocar valor para um único elemento do vetor (aqui o 5º elemento):

```
balance[4] = 50.0;
```

3.3 Acesso

Os índices de vetor em C começam no 0 e terminam em $n-1$, onde n é o tamanho do vetor. O acesso é similar à maneira de alocar valor para um único elemento:

```
printf("%i\n", balance[4]); // 5º elemento
```

3.4 Tamanho de um vetor

Quando estamos no mesmo escopo da declaração do vetor, podemos fazer uso do `sizeof` para conseguirmos obter o tamanho de um vetor.

```
int v[] = {0,1,2,3,4,5};
int i, tam = sizeof(v)/sizeof(int);
for(i = 0; i < tam; i++)
    printf("%p : %X\n", &v[i], v[i]);
printf("Tamanho do vetor: %d [elementos]\n", tam);
```

```
0x7ffd10d9e220 : 0
0x7ffd10d9e224 : 1
0x7ffd10d9e228 : 2
0x7ffd10d9e22c : 3
0x7ffd10d9e230 : 4
0x7ffd10d9e234 : 5
Tamanho do vetor: 6 [elementos]
```

```
struct sv { int i[4]; double q, e; } v[2];
int tam = sizeof(v);
printf("Tamanho da struct: %d [bytes]\n", tam);
```

```
Tamanho da struct: 64 [bytes]
```

Note que o armazenamento dos elementos do vetor se dá em regiões de memória consecutivas, sendo o primeiro elemento no primeiro endereço reservado.

3.5 Diferença com ponteiros

Praticamente nenhuma. A variável que representa o vetor é em essência o ponteiro para o primeiro elemento (mesmo tipo) cujo índice é 0. Do mesmo jeito que o operador `[]` opera ponteiros, aqui ele é usado como encapsulamento para o índice.

A única diferença prática, e que transforma vetores em possibilidade distinta de ponteiros, é que, ao declarar um vetor, uma parte da memória é reservada para aqueles elementos. Com isso, o vetor passa a ter um tamanho que pode estar sujeito a erros de estouro.

```
int i;
int a[] = {0xA0, 0x3F, 0x11, 0x0A}; // array
int *p; //ponteiro
p = a; // p aponta para o endereço do primeiro elemento de a
for(i = 0; i < 4; i++)
    printf("[%d] a: %02X; p: %02X\n", i, *(a+i), p[i]);

[0] a: A0; p: A0
[1] a: 3F; p: 3F
[2] a: 11; p: 11
[3] a: 0A; p: 0A
```

4 Strings

Um string é um vetor de **char** terminado com o caractere nulo `'\0'`, por convenção. Esse caractere precisa ser contabilizado quando atribuímos caractere por caractere a uma string.

Embora conceitualmente diferentes, um ponteiro que aponta para `NULL` e o caractere final de uma string são ambos 0.

Uma string literal em um programa C precisa estar entre aspas (`"`), enquanto o apóstrofe (`'`) é reservado para delimitar caracteres.

Uma fonte notória de bugs em um programa é tentar alocar mais bytes em uma string do que cabe em seu tamanho reservado. O uso de funções da biblioteca própria (`<string.h>`) evita esse problema. Funções como `strcpy()` (para alocar uma string) e `strcat()` (concatenar strings) servem para isso.

O tamanho de uma string pode ser obtido através da função `strlen()` (da mesma biblioteca) que irá contar todos os caracteres até encontrar o `'\0'` (excluindo-o da contagem). Com isso, o tamanho do vetor de **char** reservado na memória e o tamanho efetivo da string não serão necessariamente os mesmos.

Não se esqueça de reservar um byte do seu vetor de `char` para o caracter `'\0'`, sob o risco do seu programa não reconhecer o final da string e avançar para áreas da memória com "lixo".

```
char str1[] = "Minha string.";
char str2[] = {'0','u','t','r','a','\0'};
char str2a[] = {0x76, 0x6f, 0x63, 0x65, 0x0};
char str3[256];
char str4[3];
int i;
printf("%s\n%s\n%s\n\n", str1, str2, str2a);
strcpy(str3, str1);
printf("%s -- tam: %u ou %u\n", str3, strlen(str3), sizeof(str3)/sizeof(char));
printf("%s -- tam: %u ou %u\n", str2, strlen(str2), sizeof(str2)/sizeof(char));
for(i = 0; i < 3; i++) str4[i] = 'a';
printf("%s\n", str4);
```

```
Minha string.
Outra
voce
```

```
Minha string. -- tam: 13 ou 256
Outra -- tam: 5 ou 6
aaa
```

```
//char str1[] = "Outra";
char str1[] = {'0','u','t','r','a','\0'};
int len = strlen(str1);
printf("%s - tamanho: %d\n", str1, len);
```

```
Outra - tamanho: 5
```

5 Síntese

Sintetizando o acesso ao dado e o endereço da entidade:

tipo	nome	conteúdo	endereço
variável	x	x	&x
ponteiro	p	*p	p
vetor	a	a[i]	a
string*	s	s ou s[i]	s

(*) O acesso ao conteúdo da string só é possível dessa maneira pois o executável percorre a memória até chegar ao caracter `'\0'`. Se acessarmos o conteúdo como vetor, teremos acesso aos caracteres.

```
scanf("%d", &x);
scanf("%d", p);
scanf("%d", a); ou scanf("%d", &a[2]); ou scanf("%d", a+2);
scanf("%s", s); ou scanf("%c", &s[2]); ou scanf("%c", s+2);

char s[] = "Minha terra tem palmeiras...";
printf("%p : %s\n",s,s);

0x7ffcac4765c0 : Minha terra tem palmeiras...
```