

## 23 Matrizes, Ponteiros e Funções

Ronaldo F. Hashimoto e Carlos H. Morimoto

O objetivo desta aula é relacionar o tipo **matrizes** com ponteiros e assim entender como utilizar matrizes como parâmetros de funções. Ao final dessa aula você deverá saber:

- Descrever como matrizes são armazenadas na memória.
- Descrever a relação entre matrizes e ponteiros.
- Utilizar matrizes como parâmetros de funções.

### 23.1 Matrizes

Vimos na aula anterior que **matrizes** são estruturas indexadas (em forma matricial) utilizadas para armazenar dados de um mesmo tipo: **int**, **char**, **float** ou **double**. Por exemplo, a declaração

```
int M[100][200]; /* declara uma matriz de inteiros
                 * de nome M com 100 linhas e 200 colunas
                 */
```

alocaria uma estrutura de dados da forma:

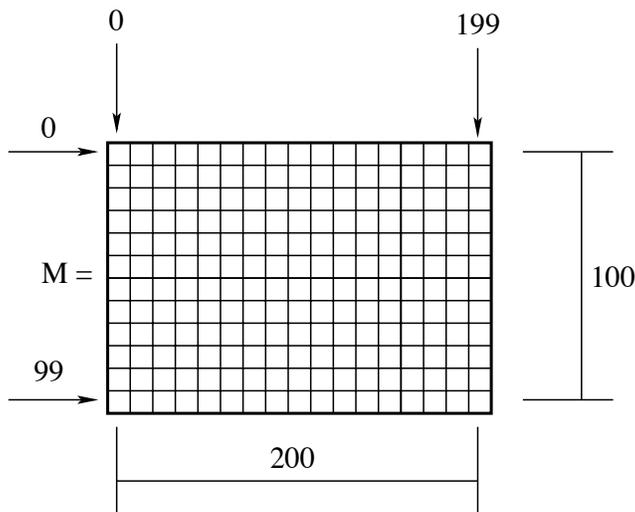


Figura 1: Estrutura de uma matriz `int M[100][200]`.

Uma pergunta que poderíamos fazer é como uma matriz fica armazenada na memória. Como a memória do computador é linear, o armazenamento dos elementos é feito colocando-se cada linha da matriz uma em seguida da outra. Assim, por exemplo, os elementos em uma matriz declarada como `int M[100][200]` são armazenados na memória do computador conforme mostra a figura 2.

Isso significa que para a matriz M, os seus elementos são armazenados na memória da seguinte maneira: os 200 elementos da primeira linha `M[0][0], M[0][1], ..., M[0][199]`, seguidos pelos elementos da segunda linha `M[1][0], M[1][1], ..., M[1][199]` e assim por diante até a última linha `M[99][0], M[99][1], ..., M[99][199]`. Dessa forma, a disposição dos 20.000 elementos da matriz M na memória seria:

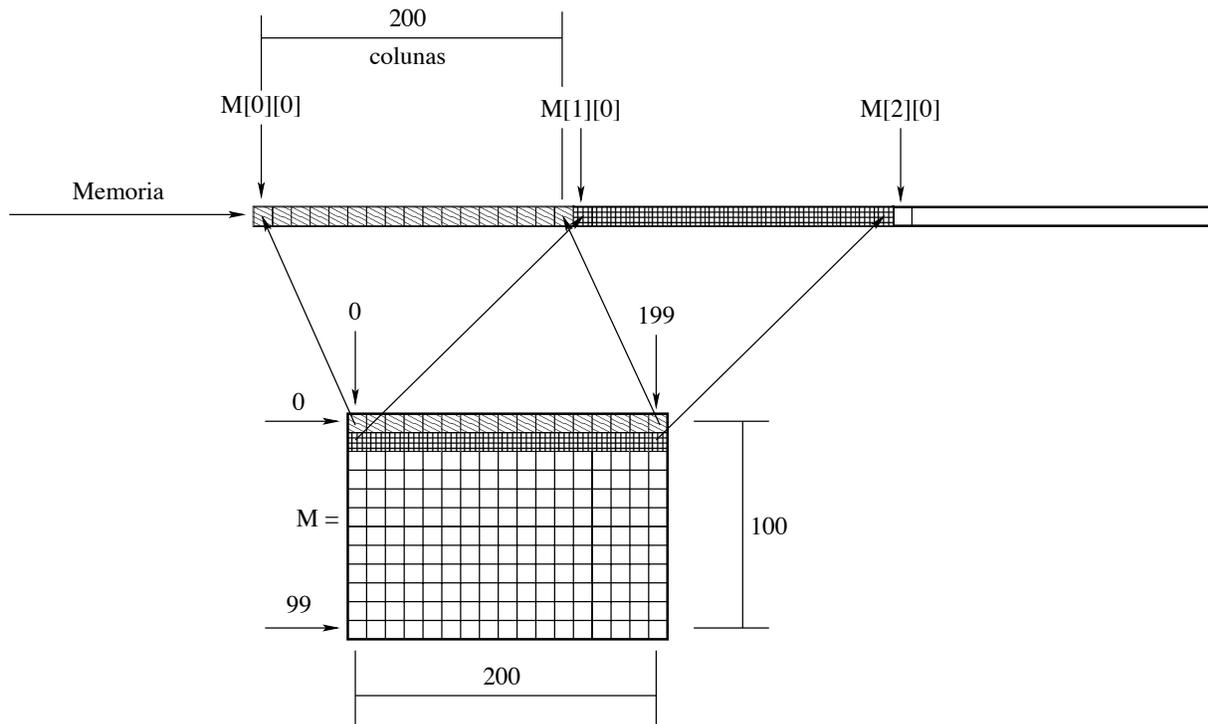


Figura 2: Estrutura da matriz na memória do computador.

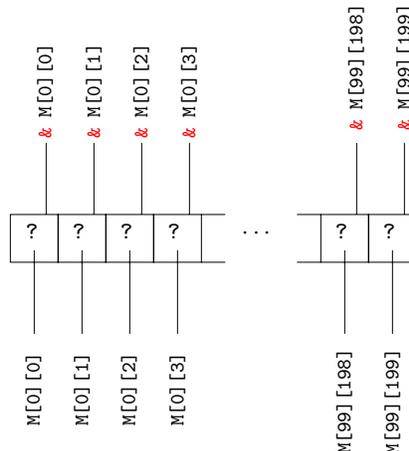


Figura 3: Disposição dos 20.000 elementos da matriz M na memória.

Observando a figura 3, podemos notar que cada elemento da matriz M tem um endereço associado. Para efeitos **didáticos**, vamos supor que o endereço de M[0][0] é um número inteiro, digamos 10, ou seja,  $\&M[0][0] \rightarrow 10$  e que os endereços das casas seguintes são números inteiros consecutivos a partir do endereço 10. Assim, temos, para efeitos **didáticos**, que  $\&M[0][0] \rightarrow 10$ ,  $\&M[0][1] \rightarrow 11$ ,  $\&M[0][2] \rightarrow 12$ , ...,  $\&M[99][198] \rightarrow 20.008$ ,  $\&M[99][199] \rightarrow 20.009$ . Com isto, é possível saber o endereço de qualquer casa de M conhecendo-se o endereço de M[0][0]. Por exemplo, endereço de M[0][78] é  $\&M[0][0] + 78 = 10 + 78 = 88$ . Agora, para ver se você entendeu os conceitos até aqui, vamos fazer uma pergunta para você.

Você saberia me dizer qual é o endereço de M[78][21]?<sup>1</sup>

Para identificar o endereço de memória associado a um determinado elemento M[i][j], é feita internamente uma conta de endereçamento: o endereço do elemento M[i][j] é  $\&M[0][0] + i \cdot nC + j$ , onde nC é o número de colunas

<sup>1</sup> $\&M[0][0] + i \cdot nC + j = 10 + (78 \cdot 200 + 21)$ , onde nC é o número de colunas da matriz.

da matriz. Este é um detalhe interno, que é feito automaticamente pelo compilador. Nos seus programas, você apenas precisa se preocupar em acessar os elementos escrevendo  $M[i][j]$ . O importante é observarmos que o compilador necessita saber o número  $nC$  de colunas da matriz, no nosso exemplo  $nC = 200$ , para fazer a conta de endereçamento. Esta informação nos será útil adiante.

### 23.1.1 Exercício

```
int N[200][100]; /* declara uma matriz de inteiros
                 * de nome N com 200 linhas e 100 colunas
                 */
```

Suponha que o endereço de  $N[0][0]$  é um número inteiro, digamos 10, ou seja,  $\&N[0][0] \rightarrow 10$  e que os endereços das casas seguintes são números inteiros consecutivos a partir do endereço de 10. Assim, temos que  $\&N[0][0] \rightarrow 10$ ,  $\&N[0][1] \rightarrow 11$ ,  $\&N[0][2] \rightarrow 12$ , ...,  $\&N[99][198] \rightarrow 20.008$ ,  $\&N[99][199] \rightarrow 20.009$ .

Você saberia me dizer qual é o endereço de  $N[78][21]$ ?<sup>2</sup> Por que é diferente do endereço de  $M[78][21]$ ?

### 23.1.2 Observação

Na linguagem C não existe verificação de índices fora da matriz. Quem deve controlar o uso correto dos índices é o programador. Além disso, o acesso utilizando um índice errado pode ocasionar o acesso de outra variável na memória. Se o acesso à memória é indevido você recebe a mensagem “segmentation fault”.

## 23.2 Matrizes e Ponteiros

Considere agora o seguinte trecho de código:

```
int A[100][300];
int *p; /* ponteiro para inteiro */
```

A figura 4 ilustra uma possível configuração da memória para as variáveis  $A$  e  $p$ .

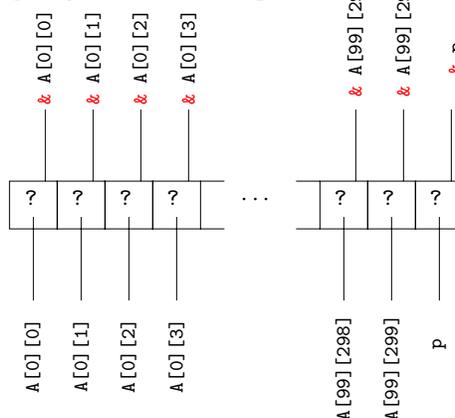


Figura 4: Possível configuração da memória para as variáveis  $A$  e  $p$ .

Podemos utilizar a sintaxe normal para fazer um ponteiro apontar para uma casa da matriz:

```
p = &A[0][0]; /* p aponta para a A[0][0] */
```

O ponteiro  $p$  recebe o endereço de  $A[0][0]$ , ou seja,  $p$  aponta para  $A[0][0]$ .

<sup>2</sup> $10 + (78 * 100 + 21)$

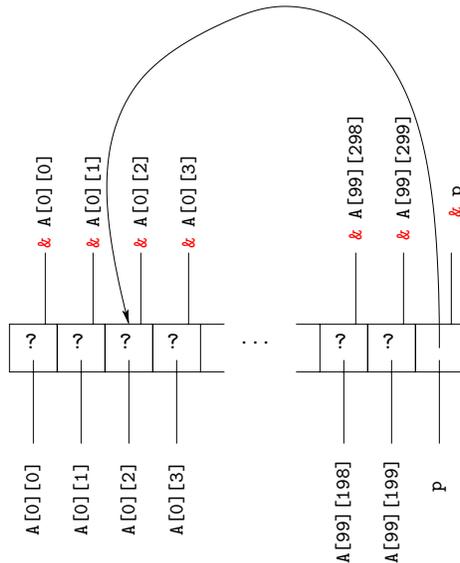


Figura 5: Ponteiro apontando para a matriz (elemento A[0][0]).

Na linguagem C, uma matriz pode ser considerada um vetor de vetores, o que traz várias vantagens aos programadores dessa linguagem.

- Uma das conveniências é utilizar o nome de uma matriz com apenas o primeiro índice para referenciar uma linha em particular, como mostra pedaço de código abaixo:

```
p = A[1];
```

Dessa forma `p` pode ser utilizado para acessar os elementos da linha 1 da matriz `A`. Dada a forma como as matrizes são armazenadas na memória, não é possível referenciar diretamente a uma coluna da matriz usando apenas um índice, ou seja, algo como:

```
p = A[][1]; /* atribuição inválida. */
```

Se for necessário utilizar uma coluna da matriz como vetor, você deve copiar os elementos da coluna para um vetor auxiliar.

- Lembre-se que ponteiros podem ser utilizados para acessar elementos de um vetor usando a mesma sintaxe de vetores (`nome_do_vetor[índice]`) com o ponteiro. Podemos utilizar essa mesma sintaxe também com matrizes. Assim se fizermos:

```
p = A[0];
```

podemos acessar o elemento que está na linha `i` e coluna `j` de `A` fazendo `p[i*300+j]` (300 é o número de colunas da matriz `A`), ou seja, ambos `p[i*300+j]` e `A[i][j]` acessam a casa da linha `i` e coluna `j` da matriz `A`. Exemplo:

```
int A[100][300], i, j;
int *p; /* ponteiro para inteiro */
i = 3; j = 4;
p = A[0]; /* p aponta para o 1o elemento da 1a linha de A */
          /* Equivale a fazer p = &A[0][0] */
p[i*300+j] = 4; /* equivale a fazer M[i][j] = 4 */
```

Mas se fazemos:

```
p = &A[3][0]; /* Equivale a p = A[3] */
```

então,  $p[0]$  é o elemento  $A[3][0]$ ,  $p[1]$  é o elemento  $A[3][1]$ ,  $p[2]$  é o elemento  $A[3][2]$ , e assim por diante.

Observe que a construção  $p[i][j]$  **não** é válida por possuir 2 pares de colchetes. Portanto a atribuição `p = A;` não é correta pois a linguagem C permite que um ponteiro seja utilizado apenas com uma dimensão (ou seja, apenas um par de colchetes), e uma matriz possui 2 dimensões. Uma matriz também não pode ser considerada um ponteiro para ponteiro, também devido a sua estrutura particular onde cada linha é seguida imediatamente por outra. Assim, a atribuição no seguinte pedaço de código:

```
int **p;  
int A[100][100];  
p = A;
```

também não é válida.

### 23.3 Matrizes como Parâmetro de Funções

A forma de declaração de uma matriz como parâmetro de função é a mesma que vimos para declarar matrizes, ou seja, indicando o nome da matriz, e entre colchetes o número de linhas e o número de colunas. Exemplo:

```
# include <math.h>  
float soma_diagonal (float B[300][300], int n) {  
    int i;  
    float r = 0;  
    for (i=0; i<n; i++) {  
        r = r + B[i][i];  
    }  
    return r;  
}
```

Esta função recebe uma matriz de reais  $B$  de tamanho  $300 \times 300$  das quais somente  $n$  linhas e colunas estão sendo consideradas e devolve a soma dos elementos de sua diagonal.

```

1      # include <stdio.h>
2      # include <math.h>
3
4
5      float soma_diagonal (float B[300][300], int n) {
6          int i;
7          float r = 0;
8          for (i=0; i<n; i++) {
9              r = r + B[i][i];
10         }
11         return r;
12     }
13
14     int main () {
15         float C[300][300], soma;
16         int m;
17
18         m = 3;
19         C[0][0] = 2; C[0][1] = -2, C[0][2] = 4;
20         C[1][0] = 3; C[1][1] = -1, C[1][2] = 7;
21         C[2][0] = 5; C[2][1] = -3, C[2][2] = 3;
22
23
24         soma = soma_diagonal (C, m);
25
26         printf ("Soma = %f\n", soma);
27
28         return 0;
29     }

```

B aponta para C[0][0]. Então B[i][i] é C[i][i]

O parâmetro B da função `soma_diagonal` aponta para a variável `C[0][0]` da função `main`. Então `B[i][i]` na função `soma_diagonal` é exatamente `C[i][i]` da função `main`.

## 23.4 Exemplo de Função com Matriz como Parâmetro

O nome de uma matriz dentro de parâmetro de uma função é utilizado como sendo um ponteiro para o primeiro elemento da matriz que está sendo usada na hora de chamar a função.

Exemplo de declaração de funções com matrizes como parâmetros:

```

1      # define MAX 200
2
3
4      float f (float M[MAX][MAX]) {
5          float s;
6          /* declaração da função f */
7          ...
8          M[i][j] = 4;
9          ...
10         return s;
11     }
12
13     int main () {
14         float a, A[MAX][MAX]; /* declaração da variável a e da matriz A */
15         ...
16         /* outras coisas do programa */
17
18         a = f (A); /* observe que a matriz é passada apenas pelo nome */
19
20         ...
21
22         return 0;
23     }
24

```

M aponta para A[0][0].

Na Linha 19, a chamada da função `f` faz com que o ponteiro `M` receba `&A[0][0]`, ou seja, faz com que o ponteiro `M` aponte para `A[0][0]`.

Na Linha 8, temos o comando `M[i][j] = 4`. Como `M` está apontando para `A[0][0]`, então `M[i][j] = 4` é o mesmo que fazer `A[i][j] = 4`. Assim, na Linha 8, dentro da função `f`, estamos mudando o conteúdo da casa de linha `i` e coluna `j` da matriz `A` da função `main`.

### 23.4.1 Exemplo

```

1      # include <stdio.h>
2
3      # define MAX 100
4
5      int f (int M[MAX][MAX], int n) {
6          n = 10;
7          M[2][3] = 4;
8          return 0;
9      }
10
11     int main () {
12         int A[MAX][MAX], m, a;
13         m = 15;
14         A[2][3] = 5;
15         a = f (A, m);
16         return 0;
17     }

```

Na Linha 14, temos o comando `A[2][3] = 5`, ou seja `A[2][3]` contém o valor 5.

Na Linha 15, a chamada da função `f` faz com que o ponteiro `M` receba `&A[0][0]`, ou seja, faz com que o ponteiro `M` aponte para `A[0][0]`.

Na Linha 7, temos o comando `M[2][3] = 4`. Como `M` está apontando para `A[0][0]`, então `M[2][3] = 4` é o mesmo que fazer `A[2][3] = 4`. Assim, na Linha 7, dentro da função `f`, estamos mudando o conteúdo da casa de linha 2

e coluna 3 da matriz A da função main. Dessa forma, depois da execução da função f, A[2][3] conterà o valor 4 e não 5 como foi inicializado.

## 23.5 Problema

- (a) Faça uma função que recebe como entrada um inteiro  $n$ , uma matriz inteira  $A_{n \times n}$  e devolve três inteiros:  $k$ ,  $Lin$  e  $Col$ . O inteiro  $k$  é um maior elemento de  $A$  e é igual a  $A[Lin][Col]$ .

Obs.: Se o elemento máximo ocorrer mais de uma vez, indique em  $Lin$  e  $Col$  qualquer uma das possíveis posições.

Exemplo: se  $A = \begin{pmatrix} 3 & 7 & 1 \\ 1 & 2 & 8 \\ 5 & 3 & 4 \end{pmatrix}$  então  $\begin{cases} k = 8 \\ Lin = 1 \\ Col = 2 \end{cases}$

Solução: vamos chamar esta função de maior. A função maior deve receber uma matriz  $A$  (vamos supor que a matriz não vai ter mais que 100 linhas e colunas) e um inteiro  $n$  indicando (das 100 linhas e colunas reservadas) quantas linhas e colunas de fato estão sendo utilizadas. Como esta função deve devolver três valores ( $k$ ,  $Lin$  e  $Col$ ), então vamos usar três ponteiros. Dessa forma, o protótipo da função maior é

```
# define MAX 100

void maior (int A[MAX][MAX], int n, int *k, int *Lin, int *Col);
```

Esta função recebe uma matriz A e um inteiro n e devolve três valores \*k, \*Lin e \*Col, via ponteiros. Para isto, vamos percorrer a matriz A (usando o padrão de percorrimento de matriz por linha) e verificando se cada elemento A[i][j] da matriz é maior que \*k. Em caso afirmativo, \*k, \*Lin e \*Col são atualizados para A[i][j], i e j, respectivamente.

```
# define MAX 100

void maior (int A[MAX][MAX], int n, int *k, int *Lin, int *Col) {
    int i, j;

    /* percorrimento da matriz A por linha */
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            if (A[i][j] > *k) {
                *k = A[i][j];
                *Lin = i;
                *Col = j;
            }
        }
    }
}
```

Se você observou bem a solução acima, faltou inicializar \*k, \*Lin e \*Col. Para esta função encontrar o maior corretamente, \*k deve ser inicializado com qualquer elemento da matriz A. Assim, vamos inicializar \*k com o primeiro elemento da matriz. Assim, \*k, \*Lin e \*Col são inicializados com A[0][0], 0 e 0, respectivamente.

```

# define MAX 100

void maior (int A[MAX][MAX], int n, int *k, int *Lin, int *Col) {
    int i, j;

    *k = A[0][0];
    *Lin = *Col = 0;

    /* percorrimo da matriz A por linha */
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            if (A[i][j] > *k) {
                *k = A[i][j];
                *Lin = i;
                *Col = j;
            }
        }
    }
}

```

- (b) Faça um programa que, dado um inteiro  $n$  e uma matriz quadrada de ordem  $n$ , cujos elementos são todos **inteiros positivos**, imprime uma tabela onde os elementos são listados em ordem decrescente, acompanhados da indicação de linha e coluna a que pertencem. Havendo repetições de elementos na matriz, a ordem é irrelevante. Utilize obrigatoriamente a função da parte (a), mesmo que você não a tenha feito.

Exemplo: No caso da matriz  $A = \begin{pmatrix} 3 & 7 & 1 \\ 1 & 2 & 8 \\ 5 & 3 & 4 \end{pmatrix}$ , a saída poderia ser:

Elem	Linha	Coluna
8	1	2
7	0	1
5	2	0
4	2	2
3	0	0
3	2	1
2	1	1
1	0	2
1	1	0

Como a matriz  $A$  somente tem valores **inteiros positivos**, uma estratégia para solucionar este problema é usar a função `maior` que devolve o maior elemento da matriz e sua respectiva posição e colocar nesta posição um inteiro negativo, digamos  $-1$ . Chama a função `maior` novamente e devolve o maior elemento da matriz e sua respectiva posição (não vai devolver o primeiro maior, pois esta posição foi marcada com  $-1$ ); colocar nesta posição  $-1$  e chamar a função `maior` novamente; até que todos os elementos da matriz sejam negativos (quando a função `maior` devolver o número  $-1$ ). Vejamos então um trecho que programa que faz esta parte:

```

printf ("Elem  Linha  Coluna\n");
k = 1; /* qualquer valor positivo */
while (k != -1) {
    maior (A, n, &k, &L, &C);
    if (k != -1) {
        printf ("%d %d %d\n", k, L, C);
        A[L][C] = -1;
    }
}

```

Fica como exercício você colocar este trecho acima em um programa completo.