

SCC 202 - Algoritmos e Estruturas de Dados I

TAD: Tipo Abstrato de Dados (2)
Exemplo e Prática

5/8/2010

Um TAD matriz

- ◆ A maioria dos programadores usa os vetores e matrizes oferecidos pelas linguagens.
- ◆ Não param para pensar na natureza de tais estruturas.
- ◆ Aqui, vamos parar para pensar, pois iremos implementar um TAD matriz, com algumas operações.

- Link: <http://www.cs.miami.edu/~geoff/Courses/MTH517-00S/Content/ArrayBasedADTs/ArrayADT.html>

- ◆ Matrizes são armazenadas de forma linear com 2 opções:
 - Ordenado por linha ou por coluna
 - ◆ Fixa linha e varia a coluna (C, Java)
 - ◆ Fixa a coluna e varia a linha (Fortran)
- ◆ Vamos guardar também o nro de linhas e o nro de colunas (2 inteiros), e a linearização dos elementos.
- ◆ Espaço dos elementos = produto das dimensões

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3
4,1	4,2	4,3

Figure 3: A 4x3 two-dimensional array.



Figure 4: Row order storage for the array in Figure 3.



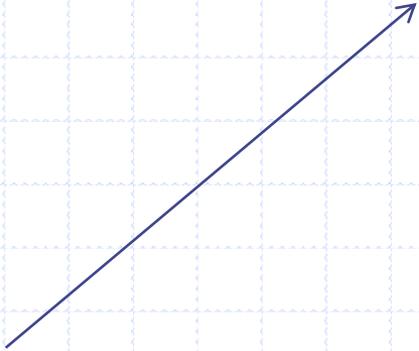
Figure 5: Column order storage for the array in Figure 3.

Assumindo um endereço por elemento e a base = alpha

◆ Matrizes ordenadas por linhas:

- Array [NumberOfRows][NumberOfColumns]
- Offset for preceding rows is $\text{Row} * \text{NumberOfColumns}$
- Offset in row is Column
- **Address** is $\text{alpha} + (\text{Row} - 1) * \text{NumberOfColumns} + \text{Column}$

```
struct matriz {  
    int lin;  
    int col;  
    float* v;  
};
```



Ajustei para não pular espaço,
nem ultrapassar o espaço
armazenado

Ordenação por colunas:
 $(\text{Column} - 1) * \text{NumberOfRows} + \text{Row}$

TADs em C: Exemplo

- Note que a composição da estrutura Matriz (struct matriz) não está explícita.
- Dessa forma, os demais módulos que usarem esse TAD não poderão acessar diretamente os campos dessa estrutura.
- Os clientes desse TAD só terão acesso às informações que possam ser obtidas através das funções exportadas.

```
/* TAD: Matriz m por n */

/* Tipo Exportado */
typedef struct matriz Matriz;

OU      typedef struct matriz *Matriz; e dai não usa
ponteiros nas funções

/* Funções Exportadas */

/* Função cria - Aloca e retorna matriz m por n */
Matriz* cria (int m, int n);

/* Função libera - Libera a memória de uma matriz */
void libera (Matriz* mat);

/* Continua... */
```

```
/* Continuação... */

/* Função acessa - Retorna o valor do elemento [i][j]
 */
float acessa (Matriz* mat, int i, int j);

/* Função atribui - Atribui valor ao elemento [i][j]
 */
void atribui (Matriz* mat, int i, int j, float v);

/* Função linhas - Retorna o no. de linhas da matriz
 */
int linhas (Matriz* mat);

/* Função colunas - Retorna o no. de colunas da matriz
 */
int colunas (Matriz* mat);
```

Arquivo matriz.h

- ◆ **OBSERVAÇÃO:** Note que as funções recebem e retornam PONTEIROS para Matriz.
- ◆ Isso porque o cliente não conseguirá declarar uma variável do tipo Matriz, pois seu tamanho e composição são desconhecidos para o cliente.
- ◆ No entanto, o cliente consegue declarar um ponteiro para Matriz, pois o ponteiro é uma variável cujo tamanho independe do tipo de dado que aponta, já que armazena apenas um endereço de memória.

TADs em C: Exemplo

```
#include <stdlib.h> /* malloc, free, exit */
#include <stdio.h> /* printf */
#include "matriz.h"
```

```
struct matriz {
    int lin;
    int col;
    float* v;
};
```

```
void libera (Matriz* mat){
    free (mat->v);
    free (mat);
}
```

Deve estar incluído no .c correspondente

TADs em C: Exemplo

```
/* Continuação... */  
  
Matriz* cria (int m, int n) {  
    Matriz* mat = (Matriz*) malloc(sizeof(Matriz));  
    if (mat == NULL) {  
        printf("Memória insuficiente!\n");  
        exit(1);  
    }  
    mat->lin = m;  
    mat->col = n;  
    mat->v = (float*) malloc(m*n*sizeof(float));  
    return mat;  
}
```



MSG:
Má escolha!

TADs em C: Exemplo

```
/* Continuação... */

float acessa (Matriz* mat, int i, int j) {
    int k; /* índice do elemento no vetor */
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    k = (i - 1)*mat->col + j;
    return mat->v[k];
}

int linhas (Matriz* mat) {
    return mat->lin;
}
```



MSG:
Má escolha!

TADs em C: Exemplo

```
/* Continuação... */

void atribui (Matriz* mat, int i, int j, float v) {
    int k; /* índice do elemento no vetor */
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
        printf("Atribuição inválida!\n");
        exit(1);
    }
    k = (i - 1)*mat->col + j;
    mat->v[k] = v;
}

int colunas (Matriz* mat) {
    return mat->col;
}
```



MSG:
Má escolha!

Programa cliente – que usa o TAD

```
#include <stdio.h>
#include <stdlib.h>
#include "matriz.h"

int main(int argc, char *argv[])
{
    float a,b,c,d;
    Matriz *M;

    // criação de uma matriz
    M = cria(5,5);

    // inserção de valores na matriz
    atribui(M,1,2,40);
    atribui(M,2,3,3);
    atribui(M,3,0,15);
    atribui(M,4,1,21);
```

Programa cliente – que usa o TAD

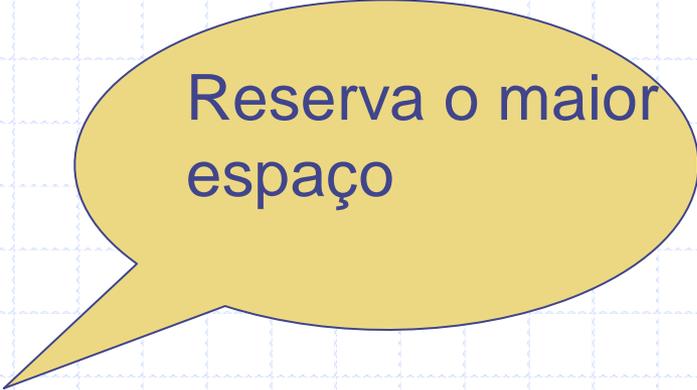
```
/* Continuação... */  
  
// verificando se a inserção foi feita corretamente  
a = acessa(M,1,2);  
b = acessa(M,2,3);  
c = acessa(M,3,0);  
d = acessa(M,4,1);  
  
printf ("M[1][2]: %4.2f \n", a);  
printf ("M[2][3]: %4.2f \n", b);  
printf ("M[3][0]: %4.2f \n", c);  
printf ("M[4][1]: %4.2f \n", d);  
  
libera(M);  
system("PAUSE");  
return 0;  
}
```

Restrição da solução

- ◆ Com a implementação dada só podemos ter uma matriz de floats
- ◆ Como faríamos para ter uma matriz de inteiros ou strings?
 - ◆ Poderíamos ir no .c e .h e alterar o tipo e passagem de alguns parâmetros relativos ao conteúdo da matriz
 - ou
 - ◆ poderíamos criar uma **union** com os 3 tipos de dados acima para definir os elementos da Matriz
- ◆ Veremos estas duas opções no curso.

Union

- ◆ Uma declaração **union** determina uma *única* localização de memória onde podem estar armazenadas várias variáveis diferentes.
 - ```
union angulo{
 float graus;
 float radianos;
};
```
  - ```
union angulo{  
    int graus;  
    float radianos;  
};
```
- ◆ Responsabilidade do programador em garantir o tipo recentemente atualizado.



Reserva o maior espaço