

Fundamentos de Programação

Matrizes e alocação dinâmica de memória

Dainf - UTFPR

Profa. Leyza B. Dorini
Prof. Bogdan T. Nassu

Tamanho máximo!

Até agora, ao trabalhar com matrizes, era assumido que elas tinham tamanho constante:

```
1 #define NL 5
```

```
2 #define NC 2
```

```
3
```

```
4 int main(){
```


```
5     int m[NL][NC];
```

```
6
```

```
7     return 0;
```

```
8 }
```

Aqui a matriz é definida
com tamanho constante.



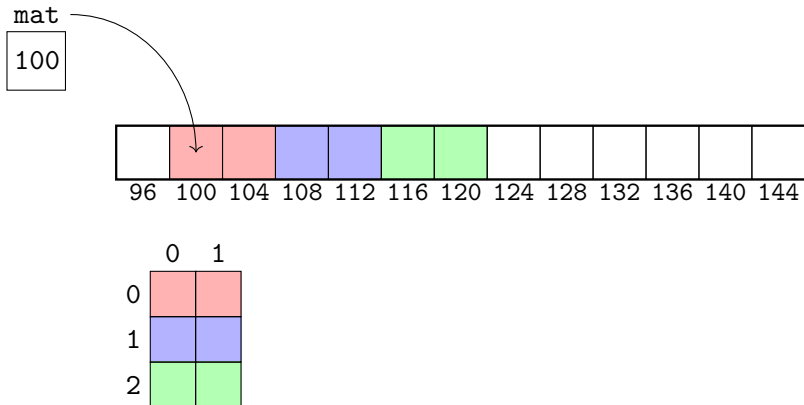
Se

- NL ou NC maior que o necessário - desperdício de espaço.
- NL ou NC menor que o necessário → precisamos recompilar o programa com outro valor para TAM_MAX.

Matrizes estáticas × dinâmicas

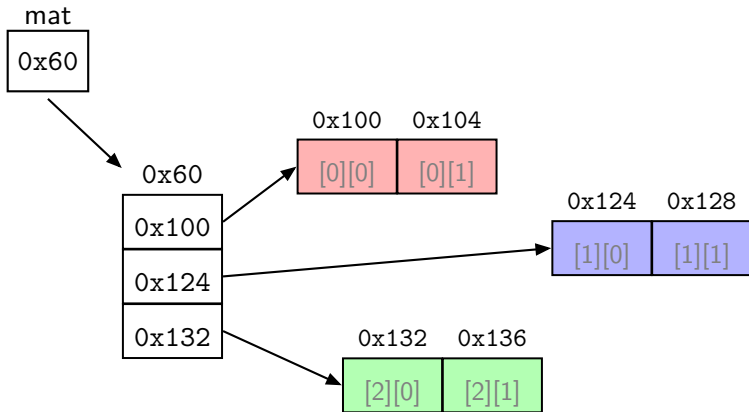
Matrizes estáticas

Como já discutido em materiais anteriores, ao declarar uma matriz `int mat[3][2]`, por exemplo, serão alocadas 6 posições contíguas de memória. A variável `mat` é um ponteiro que armazena o endereço da primeira posição de memória alocada.



Alocação dinâmica de matrizes

Para alocar matrizes dinamicamente, criaremos estrutura abaixo. Note que agora a variável `mat` é um “ponteiro para um vetor de ponteiros para vetores”.



Alocação dinâmica de matrizes

Alocação dinâmica de matrizes

Em resumo, os passos são os seguintes:

Passo 1 Criar um ponteiro para ponteiro.

Passo 2 Associar a ele um vetor de ponteiros alocado dinamicamente, cujo tamanho é o número de **linhas** da matriz.

Passo 3 A cada posição deste vetor de ponteiros associar um outro vetor do tipo a ser armazenado. Cada um destes vetores irá corresponder a uma linha da matriz (portanto possui tamanho igual ao número de **colunas**.)

Passo 4 Usar o espaço como uma matriz.

Passo 5 Chamar a função que libera o espaço.

Passo 1

O primeiro passo consiste em declarar um ponteiro para ponteiro.

```
1  int  **teste;
```

Declaração de um ponteiro para ponteiro
(o tipo deve ser igual àquele dos elementos
que serão armazenados na matriz)

teste



Passo 2

Declarar um vetor de ponteiros. É a partir dele que teremos acesso à cada linha da matriz (portanto, seu tamanho deve ser igual à quantidade desejada de linhas).

Usamos o comando `sizeof` para obter o tamanho em bytes de cada posição. Multiplicamos pela quantidade de posições desejadas para calcular o total de bytes que devem ser alocados.

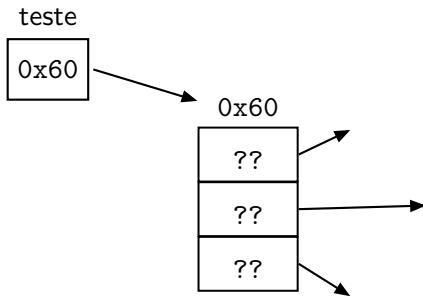
`nl` armazena a quantidade de linhas (pode ser uma variável)

```
1 teste = (int **) malloc (nl * sizeof(int *));
```



Passo 2


```
1  int **teste;  
2  
3  teste = (int **) malloc (nl * sizeof(int *));  
4  
5  //suponha nl=3
```



Passo 3

A cada posição deste vetor de ponteiros associar um outro vetor do tipo a ser armazenado. Cada um destes vetores irá corresponder a uma linha da matriz (portanto possui tamanho igual ao número de colunas).

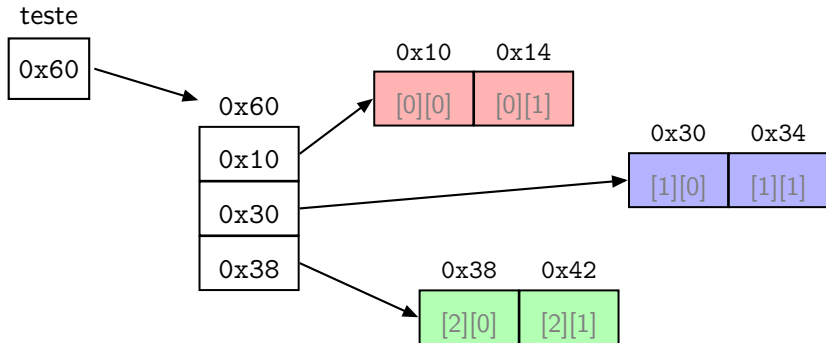
Cada posição do vetor de ponteiros
é associada a um vetor de tamanho igual
à quantidade desejada de colunas
(e do tipo dos dados que devem ser armazenados).



```
1 for ( i = 0; i < nl; i++ )
2     teste[i] = (int *) malloc (nc * sizeof(int));
3
```

Passo 3

```
1  int **teste;  
2  
3  teste = (int **) malloc (nl * sizeof(int *));  
4  for ( i = 0; i < nl; i++ )  
5      teste[i] = (int *) malloc (nc * sizeof(int));
```



Cuidado com o *type casting*

Sempre verifique o *type casting* está compatível com o tipo da variável que está recebendo tal atribuição.

```
1 int **teste, i,  
2   nl = 5,  
3   nc = 4;
```

Como teste é um ponteiro para ponteiro
(dado que vai armazenar a
referência para um vetor de ponteiros),
aqui precisamos fazer um casting
de void* para int**

```
4  
5 teste = (int **) malloc (nl * sizeof(int *));  
6  
7 for ( i = 0; i < nl; i++ )  
8     teste[i] = (int *) malloc (nc * sizeof(int));  
9
```

Aqui precisamos fazer um casting de void* para int*
(afinal, teste[i] armazena a referência para o início
de um vetor de inteiros)

Atenção para a compatibilidade de tipos

Atenção! O tipo da matriz irá determinar o tipo de várias coisas no código, tal como *type casting* e o argumento da função `sizeof()`.

Atenção para a compatibilidade de tipos

```
1 int **teste, 1,  
2 nl = 5,  
3 nc = 4;  
4  
5 teste = (int **) malloc (nl * sizeof(int *));  
6  
7 for ( i = 0; i < nl; i++ )  
8     teste[i] = (int *) malloc (nc * sizeof(int));  
9
```

Passo 4

Importante

O acesso é idêntico a uma matriz qualquer! Inclusive, também podemos passar por parâmetro para uma função.

```
1
2  for (i=0; i<nl; i++){
3      for (j=0; j<nc; j++){
4          printf("%d ", teste[i][j]);
```

Na função, podemos optar por utilizar o operador de indexação ou ponteiros (lembre-se que `m[i][j]` equivale a `*(*(m+i)+j)`)

Passo 5: não se esqueça do free()

Lembre-se: toda memória alocada dinamicamente deve ser corretamente desalocada! No caso de matrizes, é preciso desalocar todos os vetores alocados!

Inicialmente, é preciso desalocar
cada um dos vetores (onde
os dados são de fato armazenados)...

```
1  for (i = 0; i < nl; i++ )  
2      free(teste[i]);  
3  
4  free(teste);
```

... e só então o vetor de ponteiros
é desalocado!

Passando matrizes alocadas dinamicamente por parâmetro para funções

Agora, como não é preciso fazer linearização de índices, o protótipo da função não precisa especificar o número de colunas!

Exemplo: alocando e passando matrizes dinâmicas por parâmetro [1/2]

```
1  int main(){
2      int **matriz, i,
3          nl = 5,
4          nc = 4;
5
6      matriz = (int **) malloc (nl * sizeof(int *))
7      for (i = 0; i < nl; i++)
8          matriz[i] = (int *) malloc (nc * sizeof(int));
9
10     imprimeMatriz(matriz, nl, nc);
11     return 0;
12 }
```

Passamos como parâmetro o endereço da primeira posição de memória alocada (que está armazenada na “variável matriz”)

Exemplo: alocando e passando matrizes dinâmicas por parâmetro [2/2]

No protótipo, recebemos o endereço em um ponteiro para ponteiro

```
1 void imprime(int **m, int nl, int nc){  
2     int i, j;  
3     for (i=0; i<nl; i++){  
4         for (j=0; i<nc; j++){  
5             printf("%d ", m[i][j]);  
6         }  
7     }
```

Na função, podemos optar por utilizar o operador de indexação ou ponteiros (lembre-se que `m[i][j]` equivale a `*(*(m+i)+j)`)

Funções que retornam matrizes

Exemplo: função que aloca e retorna uma matriz (int)

```
1  int** criaMatrizdeInt (int nl, int nc){
2
3      int ** teste;
4      teste = (int **) malloc (nl * sizeof(int *));
5      for ( i = 0; i < nl; i++ )
6          teste[i] = (int *) malloc (nc * sizeof(int));
7
8      return teste;
9  }
10
11 int main(){
12     int nl, nc, **matriz, i;
13
14     scanf("%d %d", &nl, &nc);
15     matriz = criaMatrizdeInt(nl, nc);
16     for (i = 0; i < nl; i++ )
17         free(matriz[i]);
18     free(matriz);
19     return 0;
```

Observe o tipo da função!

Um ponteiro para ponteiro recebe o retorno da função

Nunca se esqueça do free()!

Exemplo: função que aloca e retorna uma matriz (double)

Observe o tipo da função e o uso do malloc!

```
1  double** alocaMatriz (int nl, int nc){
2
3      double ** matriz;
4      matriz = (double **) malloc (nl * sizeof(double *));
5      for ( i = 0; i < nl; i++ )
6          matriz[i] = (double *) malloc (nc * sizeof(double));
7
8      return matriz;
9  }
10
11  int main(){
12      int nl, nc, i;
13      double **m;
14
15      scanf("%d %d", &nl, &nc);
16      m = alocaMatriz(nl, nc);
17      \\free....
18      return 0;
19  }
```

O ponteiro que recebe o retorno da função também precisa ser compatível

Exemplo: utilizando a matriz retornada

```
1  int** criaMatrizdeInt (int nl, int nc){
2  ...
3  }
4
5  int main(){
6      int nl, nc, **matriz, i, j;
7
8      scanf("%d %d", &nl, &nc);
9      matriz = criaMatrizdeInt(nl, nc);
10
11     for(i=0; i<nl; i++)
12     {
13         for(j=0; j<nc; j++)
14             printf("%d ", matriz[i][j]);
15         printf("\n");
16     }
17
18     for (i = 0; i < nl; i++ )
19         free(matriz[i]);
20     free(matriz);
21     return 0;
```

Um ponteiro para ponteiro
recebe o retorno da função

Acessa "normalmente"

É melhor usar matrizes de tamanho estático
ou alocadas dinamicamente?

Depende...

Explorando a linearização de índices

Como uma matriz estática está alocada sequencialmente na memória, sua manipulação pode ser mais eficiente! Contudo, caso você não saiba o tamanho da matriz, pode explorar a linearização de índices da seguinte forma:

- Para uma matriz de dimensões $n_l \times n_c$, crie dinamicamente um vetor unidimensional de $n_l \times n_c$ posições
- Use linearização de índices para trabalhar com o vetor como se fosse uma matriz (ou seja, acesse $m[i][j]$ fazendo $m + i * n_c + j$)

Desta forma, tem-se um melhor aproveitamento da cache pois a matriz inteira está sequencialmente em memória.

Agora é contigo: faça a lista de exercícios!