

## ALOCAÇÃO DINÂMICA DE MEMÓRIA – Lista 10

A linguagem C/C++ possui recursos para alocação dinâmica de memória.

As funções que trabalham com alocação de memória se encontram na biblioteca **<stdlib.h>**

<b>void *calloc(int num, int size);</b>	aloca a memória para um vetor de <b>num</b> elementos, com tamanho <b>size</b> de cada um deles
<b>void free(void *address);</b>	libera a memória especificada por <b>address</b>
<b>void *malloc(int num);</b>	aloca a memória por <b>num</b> bytes
<b>void *realloc(void *address, int newsize);</b>	redimensiona a memória alocada para <b>address</b> par novo tamanho <b>newsize</b>

As funções **calloc()** e **malloc()** procuram por uma área livre (de tamanho desejado) na memória operativa e retornam o endereço do início dessa área.

O tipo de retorno dessas funções é **void \*** – é um ponteiro para um tipo vazio, por isso é recomendada a conversão de tipos explícita.

Para definir o tamanho normalmente é utilizada a função **sizeof( dataType )**, onde **dataType** – é o tipo de dados em questão.

A memória alocada dinamicamente será liberada de forma automática ao sair do programa, mas é considerado como boa prática de programação liberar a memória alocada dinamicamente usando a função **free( aPtr )**, onde o **aPtr** é o ponteiro para memória que foi alocada pelo programa.

### Vetores

Para criar um vetor de tamanho definido pelo usuário basta alocar uma área contínua de memória de tamanho apropriado.

Um caso de alocação dinâmica para vetor de tamanho variado é mostrado no exemplo a seguir.

### Exemplo 1 (1p): Vetor com tamanho definido pelo usuário

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *vPtr; // pointer
7      int arraySize;
8      int i;
9
10     printf("\n Input array size : ");
11     scanf("%d",&arraySize);
12
13     // memory allocation
14     vPtr = (int*) malloc(arraySize*sizeof(int));
15
16     // data input
17     printf("\n Data input : \n");
18     for(i=0; i < arraySize; i++)
19     {
20         printf("v[%d] = ", i);
21         scanf("%d", &vPtr[i]);
22     }
23
24     // data output
25     printf("\n\n Data output : \n");
26     for(i=0; i<arraySize; i++)
27         printf("v[%d] = %d \n",i, vPtr[i]);
28
29     return 0;
30 }
```

Input array size : 2

Data input :

v[0] = 1

v[1] = 2

Data output :

v[0] = 1

v[1] = 2

Para acessar os elementos do vetor alocado dinamicamente podemos também trabalhar com a aritmética dos ponteiros.

**Exemplo 2 (1p):** Vetor com tamanho definido pelo usuário (ponteiros)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *vPtr;    // pointer
7      int *auxPtr;  // pointer
8      int arraySize;
9      int i;
10
11     printf("\n Input array size : ");
12     scanf("%d",&arraySize);
13
14     // memory allocation
15     vPtr = (int*) malloc(arraySize*sizeof(int));
16
17     // data input
18     auxPtr = vPtr;
19     printf("\n Data input : \n");
20     for(i=0; i < arraySize; i++)
21     {
22         printf("v[%d] = ", i);
23         scanf("%d", auxPtr);
24         auxPtr++;
25     }
26
27     // data output
28     auxPtr = vPtr;
29     printf("\n\n Data output : \n");
30     for(i=0; i<arraySize; i++)
31     {
32         printf("v[%d] = %d \n",i, *auxPtr);
33         auxPtr++;
34     }
35
36     return 0;
37 }
```

Input array size : 2

Data input :

v[0] = 1

v[1] = 2

Data output :

v[0] = 1

v[1] = 2

De forma geral, sempre é bom verificar se a operação de alocação da memória foi bem sucedida.

Isso pode ser feito como mostra o exemplo a seguir.

### Exemplo 3 (1p): Vetor alocado dinamicamente com teste de sucesso

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *vPtr; // pointer
7      int arraySize;
8      int i;
9
10     printf("\n Input array size : ");
11     scanf("%d",&arraySize);
12
13     // memory allocation
14     vPtr = (int*) malloc(arraySize*sizeof(int));
15
16     if( vPtr == NULL )
17     {
18         printf("\n Memory allocation error! \n");
19     }
20     else
21     {
22         // data input
23         printf("\n Data input : \n");
24         for(i=0; i < arraySize; i++)
25         {
26             printf("v[%d] = ", i);
27             scanf("%d", &vPtr[i]);
28         }
29
30         // data output
31         printf("\n\n Data output : \n");
32         for(i=0; i<arraySize; i++)
33             printf("v[%d] = %d \n",i, vPtr[i]);
34     }
35
36     free(vPtr);
37     return 0;
38 }
```

Input array size : -1

Memory allocation error!

## Matrizes

Para criar uma matriz alocada dinamicamente (com tamanho definido pelo usuário) a princípio temos duas opções.

### 1. Alocação de vetor $n \times m$

Vamos considerar o seguinte caso: queremos criar dinamicamente uma matriz com  $n$  linhas e  $m$  colunas.

De forma geral, qualquer matriz é armazenada em uma área contínua de memória.

Sendo assim, caso queremos procurar por um elemento com índices  $[i][j]$ , podemos fazer isso de acordo com a formula:

$$\text{index} = i * m + j;$$

onde

$i$  – é o número da linha interesse,

$j$  – é o número da coluna de interesse

$m$  – é a quantidade das colunas

Por exemplo, caso temos uma matriz  $3 \times 4$ :

		j=2		
	0	1	2	3
i=1	4	5	6	7
	8	9	10	11

elemento  $[1][2]$ , ou então com  $i = 1, j = 2$

o elemento em questão pode ser localizado da seguinte forma:

$$\text{index} = 1 * 4 + 2 = 6$$

O tamanho da memória necessária para alocação de uma matriz  $n \times m$  é definida como:

$$n * m * (\text{dataType})$$

onde **dataType** – é o tipo de elementos da matriz (int, float, double,...).

Nesse caso o compilador não é informado de forma explícita que se trata de uma matriz. Por isso as tentativas de acesso aos elementos de forma tradicional, como **mat[i][j]**, serão consideradas incorretas.

O acesso deverá ser feito da forma:

$$* (\text{matPtr} + i * m + j)$$

onde:

**matPtr** – o ponteiro para matriz

**m** – a quantidade das colunas

**i** – a linha

**j** – a coluna

**Exemplo 4 (2p):** Matriz alocada dinamicamente (como vetor unidimensional)

```
6  int *matPtr; // pointer
7  int i, j, n, m;
8
9  printf("\n Quantidade de linhas: ");
10 scanf("%d",&n);
11
12 printf("\n Quantidade de colunas: ");
13 scanf("%d",&m);
14
15 // memory allocation
16 matPtr = (int*) malloc(n*m*sizeof(int));
17
18 // data input
19 printf("\n Entrada de dados : \n");
20 for(i=0; i < n; i++)
21 {
22     for(j=0; j < m; j++)
23     {
24         printf("mat[%d][%d] = ", i,j);
25         scanf("%d", (matPtr + i*m +j) );
26     }
27 }
28
29 // data output
30 printf("\n\n Saida de dados: \n");
31 for(i=0; i < n; i++)
32 {
33     for(j=0; j < m; j++)
34     {
35         printf("%5d ", *(matPtr + i*m +j));
36     }
37     printf("\n");
38 }
```

Quantidade de linhas: 2  
Quantidade de colunas: 3

Entrada de dados :

mat[0][0] = 1  
mat[0][1] = 2  
mat[0][2] = 3  
mat[1][0] = 4  
mat[1][1] = 5  
mat[1][2] = 6

Saida de dados:

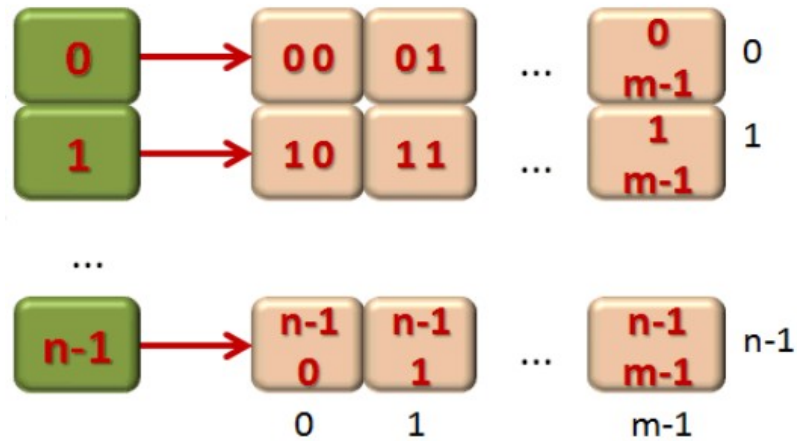
1 2 3  
4 5 6

## 2. Criação de um vetor de ponteiros

Uma outra possibilidade para criação de uma matriz  $n \times m$  é alocação de memória utilizando um vetor de ponteiros.

Para fazer isso é necessário:

- alocar a memória para um vetor de ponteiros
- alocar a memória para vetores unidimensionais (linhas da matriz)
- gravar os endereços das linhas dentro do vetor dos ponteiros



Representação gráfica

Nesse caso o compilador é informado explicitamente sobre a quantidade de linhas e colunas, portanto os elementos poderão ser acessados da forma convencional como **mat[i][j]**.

**Exemplo 5 (2p):** Matriz alocada dinamicamente (como vetor bidimensional)

```
6  int **matPtr; // pointer
7  int i, j, n, m;
8
9  printf("\n Quantidade de linhas: ");
10 scanf("%d",&n);
11
12 printf("\n Quantidade de colunas: ");
13 scanf("%d",&m);
14
15 // memory allocation for lines
16 matPtr = (int**) malloc(n*sizeof(int));
17
18 // data input
19 printf("\n Entrada de dados : \n");
20 for(i=0; i < n; i++)
21 {
22     // memory allocation for coluns
23     matPtr[i] = (int*)malloc(m*sizeof(int));
24     for(j=0; j < m; j++)
25     {
26         printf("mat[%d][%d] = ", i,j);
27         scanf("%d", &matPtr[i][j] );
28     }
29 }
30
31 // data output
32 printf("\n\n Saida de dados: \n");
33 for(i=0; i < n; i++)
34 {
35     for(j=0; j < m; j++)
36     {
37         printf("%5d ", matPtr[i][j]);
38     }
39     printf("\n");
40 }
```

Quantidade de linhas: 2  
Quantidade de colunas: 3

Entrada de dados :

mat[0][0] = 11  
mat[0][1] = 12  
mat[0][2] = 13  
mat[1][0] = 14  
mat[1][1] = 15  
mat[1][2] = 16

Saida de dados:

11 12 13  
14 15 16



## Exercícios:

### Exercício 6 ( 2,5p ):

- criar vetores **a** e **b** com a quantidade dos elementos determinada pelo usuário para cada um deles.
- criar o vetor **c** composto por elementos do vetor **a** e elementos do vetor **b**.

Exemplo

a

1	2
---	---

b

3	4	5
---	---	---

c

1	2	3	4	5
---	---	---	---	---

### Exercício 7 ( 2,5p ):

- criar o vetor **a** com a quantidade dos elementos determinada pelo usuário.
- criar o vetor **b** contendo os elementos pares do vetor **a**.
- criar o vetor **c** contendo os elementos positivos do vetor **a**

### Exercício 8 ( 5p ):

- receber duas matrizes **A** e **B** com tamanhos definidos pelo usuário.
- criar matrizes transpostas para cada matriz.

$$a^T_{ij} = a_{ji}$$

A ( 3 x 2 )

1	2
3	4
5	6

A<sup>T</sup> ( 2 x 3 )

1	3	5
2	4	6

- se for possível, realizar as operações de soma e produto das matrizes **A** e **B**.

A matriz C (m X q) resultante do produto das matrizes A (m x n) e B (n x q) pode ser calculada como:

$$c_{ij} = \sum_{r=1}^n a_{ir} b_{rj} (i=1,2,\dots,m; j=1,2,\dots,q)$$

Exemplo:

A

1	2
2	3
3	4
4	5

B

1	2	3
4	5	6

C

9	12	15
14	19	24
19	26	33
24	33	42