Matrizes

Anteriormente, vimos que os vetores são conjuntos de endereços alocados de maneira contínua no espaço de memória e que armazenam tipos homogêneos. Os vetores ficam dispostos de maneira *unidimensional*, conforme a representação:

int x[5] = {7,0,3,9};

7	0	3	9	

Mas, e se quiséssemos ter duas ou mais dimensões? Será que é possível? Como seria? Bom, vamos pensar, se vetores com uma dimensão têm a representação anterior, se pensarmos em elevar para duas dimensões, teríamos:

M[3][3]

1	2	3
4	5	6
9	8	9

Interessante, né? Teríamos linhas e colunas, mas essa representação nos faz lembrar uma estrutura já conhecida, não? As *matrizes*! Pois matrizes possuem linhas e colunas. Isso mesmo!

Matrizes, na programação em C, é um *vetor multidimensional* (com várias dimensões)

No exemplo anterior, **M** possui duas dimensões: as *linhas* e as *colunas*. O acesso a seus elementos, assim como nos vetores, é realizado por meio dos índices, e a sintaxe de sua declaração é dada por:

```
<tipo> nome_variavel [<qtd_linhas>][<qtd_colunas>];
```

Podemos, assim como as demais variáveis, iniciar seu conteúdo no ato da declaração:

```
<tipo> nome_variavel [<qtd_linhas>][<qtd_colunas>] ={<conjunto_de_valores>};
```

<conjunto_de_valores> são elementos, do mesmo tipo, separados por vírgula, que serão armazenados na variável matriz de maneira sequencial.

Sua alocação na memória pode ser realizada de maneira estática ou dinâmica. Aqui na disciplina veremos apenas sua alocação estática, e após este processo, com a variável alocada na memória, o tamanho da sua dimensão (linhas e colunas) não pode ser alterado. Vamos conhecer um pouco sobre sua estrutura?

Estrutura

Fazendo uso do mesmo exemplo inicial:

int M[3][3] = {1,2,3,4,5,6, 9,8,9};

	0	1	2
0	1	2	3
1	4	5	6
2	9	8	9

Como nossa variável M possui 9 espaços na memória, uma maneira de acessá-los é através de índices. Assim como nos vetores os índices das colunas e das linhas de uma matriz *começam com valor zero* e vão até o valor de *<dimensão>-1*.

```
Elemento índice 00 →linha 0 coluna 0 →M[0][0] →1
Elemento índice 10 →linha 1 coluna 0 →M[1][0] →4
Elemento índice 21 →linha 2 coluna 1 →M[2][1] →8
Elemento índice 20 →linha 2 coluna 0 →M[2][0] →9
Elemento índice 12 →linha 1 coluna 2 →M[1][2] →6
```

Atribuição

O processo de atribuição de valores pode ocorrer através de índices. Digamos que queremos alterar o elemento da linha 0 coluna 2 de M. Para alterá-lo é bem semelhante aos vetores, basta informar os índices correspondentes:

```
M[0][2] = 9;
```

Desta forma, M será ajustada para:

	0	1	2
0	1	2	9
1	4	5	6
2	9	8	9

Bom, mas como inicializamos uma matriz? Vamos lá conhecer o processo, é bem semelhante aos vetores.

Inicialização

Assim como os vetor, as matrizes podem ser inicializadas no ato da declaração. Podemos inicializar sem expressarmos nenhum valor para elas:

```
int main(void){
    float y[3][3];
    int x[2][2];
    char v[2][3];
}
```

O espaço na memória será alocado, porém o conteúdo será "*lixo de memória*". Mas podemos iniciar matrizes com elementos definidos:

```
int main(void){
    float y[3][3] = {1,2,3,4,5,6,7,8,9};
    int x[2][2] = {1,3,4,7};
    char v[2][3] = {'a', 'c', 'd', 'u', 'i', 'j'};
}
```

1.0	2.0	3.0
4.0	5.0	6.0
7.0	8.0	9.0

```
float y[3][3] = {1,2,3,4,5,6,7,8,9};
```

1	3
4	7

```
int x[2][2] = {1,3,4,7};
```

a	c	D
u	i	j

```
char v[2][3] = {'a', 'c', 'd', 'u', 'i', 'j'}
```

Ou podemos também iniciar as matrizes com valores "*zerados*", semelhante aos vetores:

```
int main(void){
    float y[3][3] = {};
    int x[2][2] = {};
    char v[2][3] = {};
}
```

Manipulação

A manipulação é semelhante à dos vetores, ocorre por meio de um percurso. Como os elementos podem ser acessados por índices, então podemos ter acesso a cada elemento por meio de sua posição de linha e coluna. E uma estrutura de repetição pode auxiliar nesse percurso. Para exemplo, vamos fazer uso da estrutura *for*.

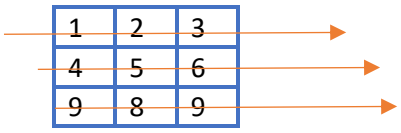
Vamos percorrer a matriz M do nosso exemplo inicial:

```
#define L 3
#define C 3
int main(){
        int M[L][C] = {1,2,3,4,5,6,9,8,9};
        int linha;
        int coluna;

        for(linha=0; linha<L; linha++)
                for(coluna=0; coluna<C; coluna++)
                        printf("Elemento %d, linha %d, coluna %d \n", M[linha][coluna], linha, coluna);
}
```

Observe que o percurso parte da *linha 0* e executa todo o *for* interno, ou seja, a variável *coluna* varia de *0* a *L-1*, para então a variável *linha* ser incrementada para *1*. Como saída do código anterior, teremos:

```
Elemento 1, linha 0, coluna 0
Elemento 2, linha 0, coluna 1
Elemento 3, linha 0, coluna 2
Elemento 4, linha 1, coluna 0
Elemento 5, linha 1, coluna 1
Elemento 6, linha 1, coluna 2
Elemento 9, linha 2, coluna 0
Elemento 8, linha 2, coluna 1
Elemento 9, linha 2, coluna 2
```



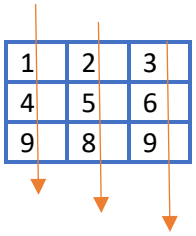
Dessa forma, da maneira como está implementado o percurso nessa matriz ocorreu *linha a linha*, da *esquerda para a direita*. O percurso poderia ser dado analisando *coluna a coluna*:

```
#define L 3
#define C 3
int main(){

	int M[L][C] = {1,2,3,4,5,6,9,8,9};
	int linha;
	int coluna;

	for(coluna=0; coluna<C; coluna++)
		for(linha=0; linha<L; linha++)
			printf("Elemento %d, linha %d, coluna %d \n", M[linha][coluna], linha, coluna);
}
```

```
Elemento 1, linha 0, coluna 0
Elemento 4, linha 1, coluna 0
Elemento 9, linha 2, coluna 0
Elemento 2, linha 0, coluna 1
Elemento 5, linha 1, coluna 1
Elemento 8, linha 2, coluna 1
Elemento 3, linha 0, coluna 2
Elemento 6, linha 1, coluna 2
Elemento 9, linha 2, coluna 2
```



Dessa maneira, é possível observar as diversas formas de percurso que podemos realizar em uma matriz, podemos, por exemplo, consultar os elementos da *diagonal principal*, que correspondem aos elementos M_{11}, M_{22} e M_{33}, cujos os índices da linha e da coluna são iguais:

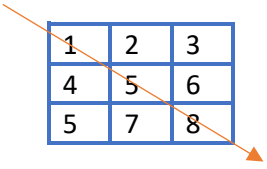
```
#define L 3
#define C 3
int main(){

	int M[L][C] = {1,2,3,4,5,6,5,7,8};
	int i;

		for(i=0; i<L; i++)
			printf("Elemento %d, linha %d, coluna %d \n", M[i][i], i, i);
}
```

Como saída teremos:

```
Elemento 1, linha 0, coluna 0
Elemento 5, linha 1, coluna 1
Elemento 9, linha 2, coluna 2
```



Bom, como sugestão de exercício prático, sugerimos a implementação dos percursos em uma matriz:

- Linha a linha, da direita para a esquerda
- A impressão da diagonal secundária
- A impressão das colunas da direita para a esquerda

A *atribuição de valores* para uma matriz é também de forma análoga às outras variáveis simples, via *scanf*:

```
#define L 3
#define C 3
int main(){

	int M[L][C] = {};
	int linha;
	int coluna;

	for(linha=0; linha<L; linha++)
		for(coluna=0; coluna<L; coluna++){
			printf("Informe um elemento para a matriz: ");
			scanf("%d", &M[linha][coluna]);
		}
}
```

Ou por meio de atribuição direta:

```
M[linha][coluna] = x
```

As *operações lógicas* também podem ser realizadas tranquilamente em elementos da matriz:

```
#define L 3
#define C 3
int main(){

	int M[L][C] = {};
	int linha;
	int coluna;

	for(linha=0; linha<L; linha++)
		for(coluna=0; coluna<L; coluna++){
			if (M[linha][coluna] % 2 == 0)
				printf("Eh par");
			else
				printf("Eh impar");
		}
}
```

Uso de Matrizes em Funções

Assim como os vetores, a passagem de uma matriz como atributo de uma função é realizada por *referência*, já que *matrizes são vetores multidimensionais*, ou seja, são endereços para espaços de memória. Então, as assinaturas de funções que recebem como parâmetros matrizes são dadas de três formas.

A seguir apresentamos a representação de assinaturas (protótipos) de funções que realizam a soma dos elementos das linhas de uma matriz:

	Assinaturas	Representação
1	int somaLinhas(int matriz[L][C])	Indicação explícita das dimensões da matriz
2	int somaLinhas(int matriz[][C])	Indicação explícita apenas da última dimensão da matriz
3	int somaLinhas(int (*matriz)[C])	Indicação da matriz por ponteiros

A seguir apresentamos o mesmo corpo para as assinaturas (protótipos) das funções vistas na tabela anterior:

1	int somaLinhas(int matriz[L][C]){
	int l, j, soma = 0;
	for(l=0; l<L; l++)
	for(j=0; j<C; j++)
	soma += matriz[l][j];
	return soma;
	}

2	int somaLinhas(int matriz[][C]){
	int l, j, soma = 0;
	for(l=0; l<L; l++)
	for(j=0; j<C; j++)
	soma += matriz[l][j];
	return soma;
	}

3	int somaLinhas(int (*matriz)[C]){
	int l, j, soma = 0;
	for(l=0; l<L; l++)
	for(j=0; j<C; j++)
	soma += matriz[l][j];
	return soma;
	}

O uso das funções, com a passagem de uma matriz, também é usual, nada de novo. Segue:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define L 3
#define C 3
int somaLinhas(int matriz[L][C]){
        int l, j, soma = 0;
        for(l=0; l<L; l++)
                for(j=0; j<C; j++)
                        soma += matriz[l][j];
        return soma;
}
int main(){
        int linha, coluna;
        srand(time(NULL));

        for(linha=0; linha<L; linha++)
                for(coluna=0; coluna<C; coluna++)
                        M[linha][coluna] = rand() % 10;

        printf("A soma dos elementos sao = %d", somaLinhas(M));
}
```

Definição de constantes

Indicação de uma matriz como parâmetro

Como matrizes são endereços de memórias homogêneas, não é preciso passar &M[0][0] para indicar o endereço do primeiro elemento da matriz, basta indicar seu nome

Super legal, né? Então, vamos praticar, exercitar e trocar várias ideias no nosso fórum.