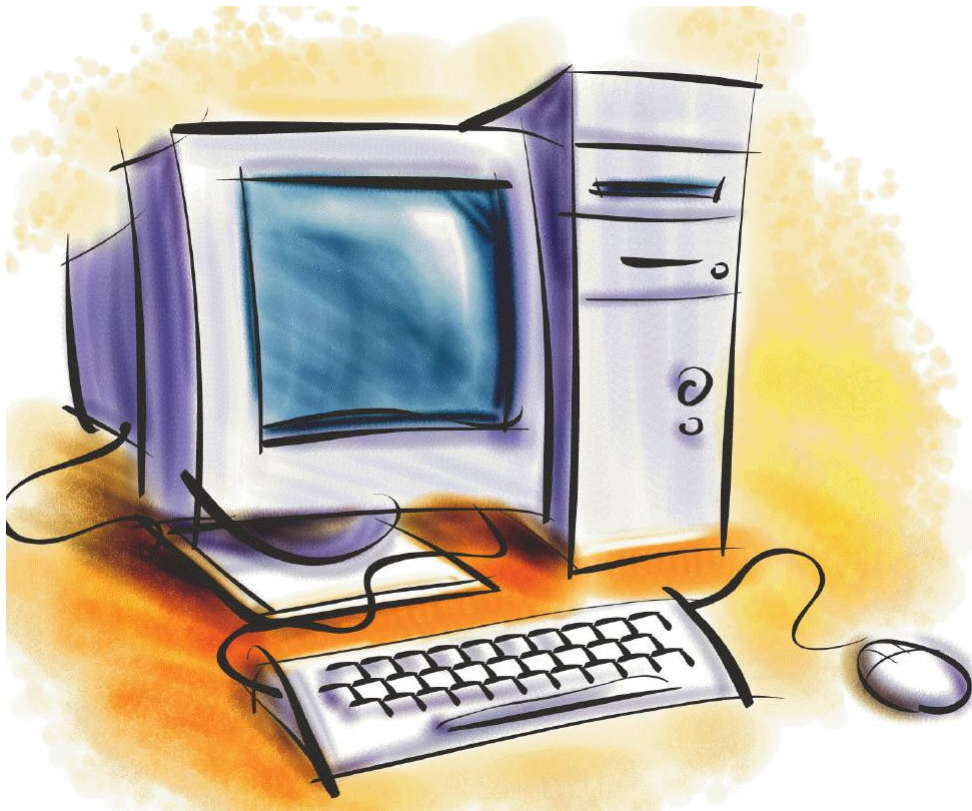


ALGORITMOS E ESTRUTURAS DE DADOS I



UNIDADE 3

ARRANJOS

PROF. NAÍSSÉS ZÓIA LIMA

Vetores

Introdução

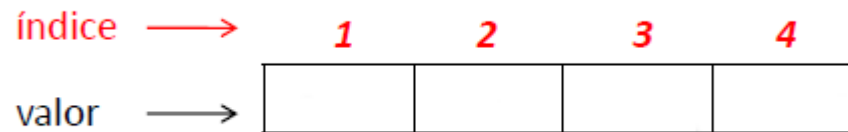
- Até agora vimos variáveis de tipos de dados básicos capazes de armazenar um dado de cada vez.
- Existem casos onde é necessário armazenar uma grande quantidade de dados ao mesmo tempo.
 - Entretanto, a criação de uma quantidade grande de variáveis é inviável. Como resolver?
 - Solução: **vetores**!
- Um **vetor** (também chamado de **array**) é uma estrutura de dados composta por uma quantidade determinada de elementos de um mesmo tipo primitivo (inteiro, real, caractere, lógico).
- Um **vetor** é um agrupamento de dados de mesmo tipo representado por uma única variável.

Introdução

- Por ter como característica armazenar dados que seguem um mesmo tipo básico, diz-se que os **vetores** são **estruturas de dados homogêneas**.
- Sabe-se que a variável funciona como um espaço na memória do computador, capaz de armazenar apenas um único dado.
- Analogamente, um **vetor** pode ser considerado um conjunto de variáveis agrupadas (que ocupam vários de espaços na memória) referenciadas por uma única variável.
- Mas como isso é possível?

Introdução

- Indica-se a quantidade de dados que um **vetor** deve armazenar no momento da sua declaração
- Isso indica quanto espaço em memória deve ser reservado para o **vetor**.



- Para acessar os valores de um **vetor**, deve-se indicar o nome da variável que representa o **vetor** seguido da posição (**índice**) referente ao dado que se deseja acessar.
- Note que num **vetor** os dados são organizados **sequencialmente**.

Vetores

Declaração

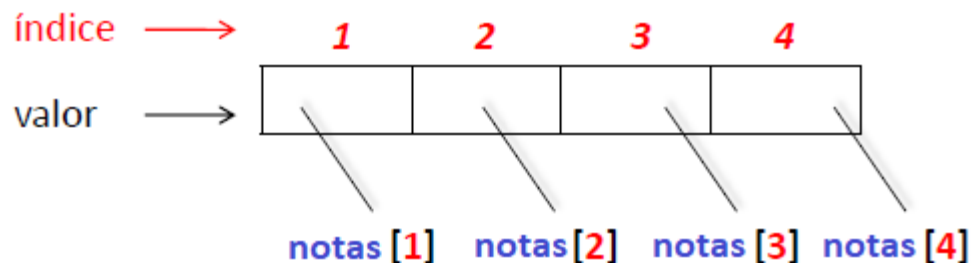
- Declaração de vetor:

DECLARE **nome**[**tamanho**] **tipo**

- **nome** é o identificador do vetor;
- **tamanho** é a quantidade de elementos que o vetor possuirá;
- **tipo** é o tipo dos elementos do vetor

Exemplo: **DECLARE** notas[4] REAL

O vetor chamado **notas** possui **4** elementos do tipo **real**



Obs: O vetor **notas** é somente uma variável, mas é capaz de armazenar 4 valores do tipo REAL !

Vetores

Atribuição de Valores

- Atribuição de valores:
 - Como os vetores possuem várias posições, nas atribuições é necessário informar em qual delas o valor será armazenado.
 - Exemplo:
 - Atribuir o valor 5 na posição de índice 4 do vetor notas: **notas[4] ← 5**
 - Atribuir o valor 0 na posição de índice 2 do vetor notas: **notas[2] ← 0**

notas		0		5
-------	--	---	--	---

- Cuidado para não fazer atribuições em índices maiores que o tamanho do vetor. Exemplo: **notas[10] ← 50**

Vetores

Atribuição de Valores

- Preenchendo um vetor:

- Significa atribuir valores a todas as posições. Assim, é necessário utilizar algum mecanismo que controle o valor do índice

PARA $i \leftarrow 1$ ATÉ 4 FAÇA

INICIO

 ESCREVA “Digite uma nota”

 LEIA notas[i]


FIM

- Utilizou-se a estrutura de repetição **PARA**, pois podemos garantir que variável i assume todos os possíveis valores para o índice do vetor. Em cada execução do laço de repetição, uma posição do vetor é preenchida.

Vetores

Atribuição de Valores

- Simulação:

Memória			Tela 								
i = 1	nota	<table border="1"><tr><td>95</td><td></td><td></td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	95				1	2	3	4	Digite uma nota: 95
95											
1	2	3	4								
i = 2	nota	<table border="1"><tr><td>95</td><td>13</td><td></td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	95	13			1	2	3	4	Digite uma nota: 13
95	13										
1	2	3	4								
i = 3	nota	<table border="1"><tr><td>95</td><td>13</td><td>47</td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	95	13	47		1	2	3	4	Digite uma nota: 47
95	13	47									
1	2	3	4								
i = 4	nota	<table border="1"><tr><td>95</td><td>13</td><td>47</td><td>-25</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	95	13	47	-25	1	2	3	4	Digite uma nota: -25
95	13	47	-25								
1	2	3	4								

Vetores

Acesso aos Valores


- Mostrando os valores de um vetor:
 - Significa exibir os valores de todas as posições. Assim, é necessário utilizar algum mecanismo que controle o valor do índice.

PARA $i \leftarrow 1$ ATÉ 4 FAÇA

INICIO

 ESCREVA “A nota “, i , “ é: “, $\text{notas}[i]$

FIM

Memória					Tela								
i = 1	nota	<table><tr><td>95</td><td>13</td><td>47</td><td>-25</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	95	13	47	-25	1	2	3	4	A nota 1 é: 95		
95	13	47	-25										
1	2	3	4										
i = 2	nota	<table><tr><td>95</td><td>13</td><td>47</td><td>-25</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	95	13	47	-25	1	2	3	4	A nota 2 é: 13		
95	13	47	-25										
1	2	3	4										
i = 3	nota	<table><tr><td>95</td><td>13</td><td>47</td><td>-25</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	95	13	47	-25	1	2	3	4	A nota 3 é: 47		
95	13	47	-25										
1	2	3	4										
i = 4	nota	<table><tr><td>95</td><td>13</td><td>47</td><td>-25</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	95	13	47	-25	1	2	3	4	A nota 4 é: -25		
95	13	47	-25										
1	2	3	4										

Vetores em C

- Uma diferença básica entre vetores em pseudocódigo e C é que em C os índices para identificar as posições dos vetores começam sempre de zero (0) e vão até o tamanho do vetor menos uma unidade.

- Exemplo:

`int nota[4];`



Vetor de inteiros

`nota[0], nota[1], nota[2], nota[3]`

índice	→	0	1	2	3
valor	→	20	12	3	13

Obs.: tamanho m -> índice 0 a $(m-1)$

Vetores em C

Declaração

- Declaração de vetores em C:

tipo nome[tamanho];

- **tipo:** indica o tipo de cada elemento do vetor
- **nome:** é o identificador da variável
- **tamanho:** indica o tamanho do vetor, onde o menor valor é 1

- Exemplos:

```
int peso[10];
```

```
float nota[41];
```

```
char nome[80];
```

Vetores em C

Atribuição de Valores

- Atribuição de valores às posições do vetor:
 - Como os vetores possuem várias posições, nas atribuições é necessário informar em qual delas o valor será armazenado. A posição inicia pelo índice zero.
- Exemplo: **float peso[5];**
 - **peso[0] = -4;** (atribui o valor -4 à primeira posição do vetor)
 - **peso[1] = 4;** (atribui o valor 4 à segunda posição do vetor)
 - **peso[3] = 90;** (atribui o valor 90 à quarta posição do vetor)
- **Cuidado!**
 - **peso[5] = 15;** Tenta atribuir o valor 15 ao sexto elemento do vetor, mas o vetor só possui 5 posições

Vetores em C

Atribuição de Valores

- Preenchendo um vetor:

- Significa atribuir valores a todas as posições. Assim, é necessário utilizar algum mecanismo que controle o valor do índice.

```
float peso[5];  
for ( int i = 0 ; i < 5 ; i++ ) {  
    printf("Digite o valor da posicao %d:", i);  
    scanf("%f", &peso[i]);  
}
```

- Utilizou-se a estrutura de repetição **for**. Assim podemos garantir que variável **i** assume todos os possíveis valores para o índice do vetor. Em cada execução do laço de repetição, uma posição do vetor será preenchida.
- Lembre-se que o vetor inicia da posição zero!

Vetores em C

Acesso aos Valores

- Acessando valores do vetor:
 - De forma semelhante ao pseudocódigo, para acessar os elementos do vetor, devemos informar o índice para o qual se deseja obter o valor, juntamente com o nome da variável.
- Exemplo: **float peso [5];**
 - **peso[0]**: primeiro elemento do vetor
 - **peso[1]**: segundo elemento do vetor
 - **peso[2]**: terceiro elemento do vetor
 - **peso[3]**: quarto elemento do vetor
 - **peso[4]**: quinto elemento do vetor
- **Cuidado!**
 - **peso[5]**: tenta acessar o sexto elemento do vetor, mas o vetor só possui 5 posições

Vetores em C

Acesso aos Valores

- Imprimindo os valores de um vetor:
 - Significa exibir os valores das posições do vetor. Assim, é necessário utilizar algum mecanismo que controle o valor do índice.
 - Utilizando o mesmo exemplo do vetor **float peso[5]**, o código abaixo exibe os valores de cada posição do vetor.

```
for ( int i = 0 ; i < 5 ; i++ ) {  
    printf("\n O valor da posicao %d e %f", i, peso[i]);  
}
```


Vetores em C

Inicialização Direta

- Usa-se chaves para inicializar o vetor no momento da sua declaração.

```
int vet[5] = {10, 20, 30, 40, 50};    // [10, 20, 30, 40 , 50]
```

```
int vet[5] = {10, 20};    // [10, 20, 0, 0 , 0]
```

```
int vet[5] = { };    // [0, 0, 0, 0 , 0]
```

```
int vet[5] = {10, 20, 30, 40, 50, 60};    // ERRO – [10, 20, 30, 40 , 50]
```

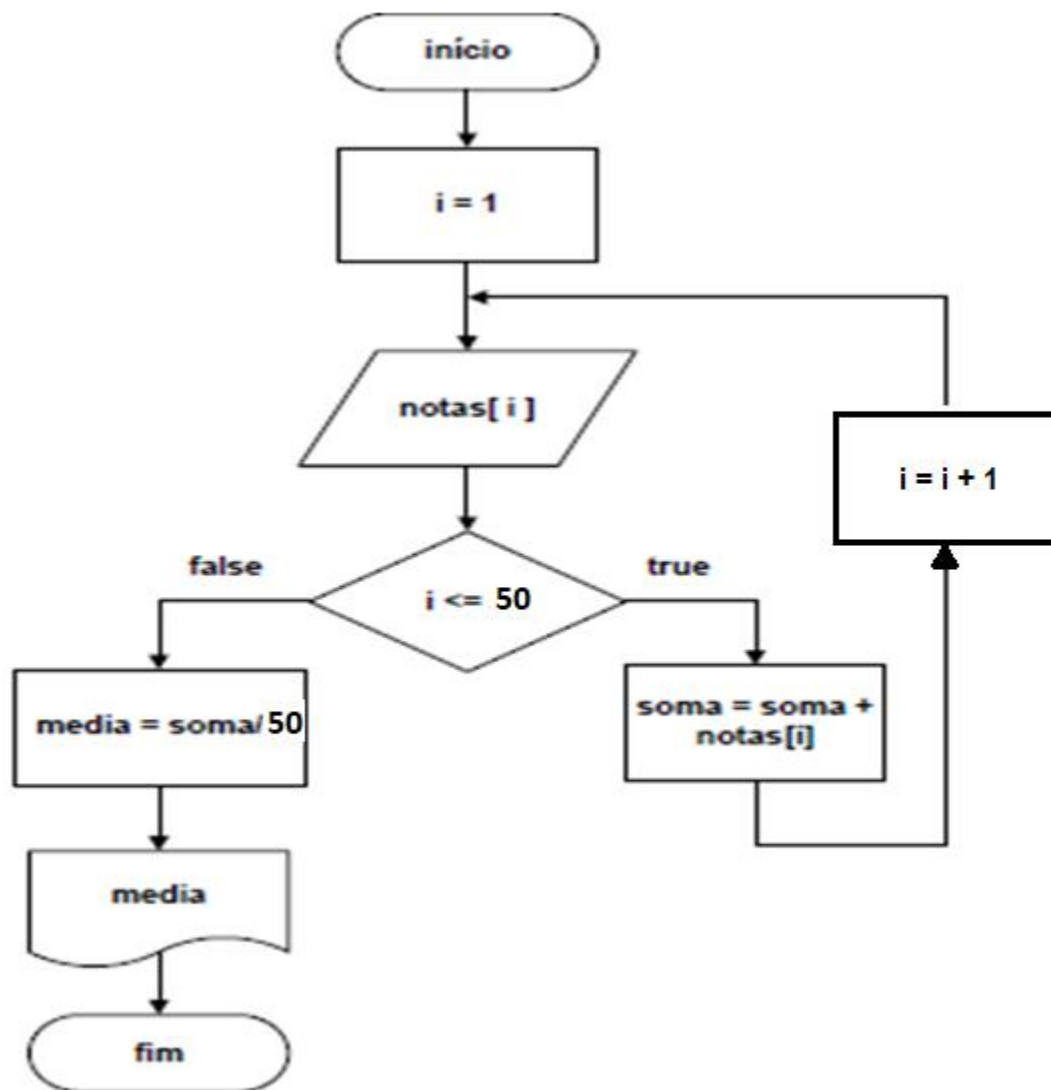
```
int vet[5] = { [2] = 30 };    // [0, 0, 30, 0 , 0]
```

```
int vet[ ] = { 10, 20, 30 };    // [10, 20, 30]
```

Exemplo

- ❑ Fazer um algoritmo em fluxograma, pseudocódigo e C para calcular a média de 50 notas fornecidas pelo usuário usando vetor.
- ❑ Crie um outra versão do algoritmo em 2 passos: no primeiro, preencher o vetor com a notas; no segundo, calcular a média.

Exemplo – Fluxograma



Exemplo – Pseudocódigo

ALGORITMO “Media_notas”

DECLARE notas[50], soma, media, i NUMÉRICO

soma \leftarrow 0

PARA i \leftarrow 1 ATÉ 50 FAÇA

INÍCIO

 ESCREVA “Digite a nota ”, i

 LEIA notas[i]

 soma \leftarrow soma + notas[i]

FIM

media \leftarrow soma / 50

ESCREVA “A media das notas e: ”, media

FIM_ALGORITMO

Exemplo – C

```
//Bibliotecas
```

```
int main( )
```

```
{
```

```
    float notas[50], soma = 0, media =0;
```

```
    for (int i = 0 ; i < 50 ; i++)
```

```
    {
```

```
        printf("Digite a nota %d:", i+1); // i+1 porque começa de zero
```

```
        scanf("%d", &notas[i]);
```

```
        soma = soma + notas[i];
```

```
    }
```

```
    media = soma/50;
```

```
    printf("A media das notas e %.1f", media);
```

```
}
```

Exemplo – Resultado na tela

Digite a nota 1: 10

Digite a nota 2: 7

Digite a nota 3: 8

Digite a nota 4: 9

Digite a nota 5: 6

A media das notas e: 8

Exercício

□ Seja o vetor abaixo:

$v = [2 \ 6 \ 8 \ 3 \ 10 \ 9 \ 1 \ 21 \ 33 \ 14]$

Considerando $x = 2$ e $y = 4$, escreva o valor referente à solicitação. Indique, inclusive, se a posição existe.

- a) $v[x]$
- b) $v[x + 1]$
- c) $v[x + 10]$
- d) $v[y + 3]$
- e) $v[y - 3]$
- f) $v[x + y]$
- g) $v[x - y]$
- h) $v[y - x]$

Matrizes

Introdução

- Vetores são variáveis **compostas unidimensionais**, isto é, **arranjos unidimensionais**
- Porém existem situações em que a natureza dos dados nos indica que forma de armazenamento possui mais de uma dimensão
 - Nesses casos, utilizamos **variáveis compostas multidimensionais**
 - De forma semelhante aos vetores, matrizes são identificadas por um nome e um tipo
 - Os elementos na matriz são diferenciadas pela especificação da posição dentro da estrutura

Introdução

- Uma **matriz** é uma variável composta bidimensional.
- Assim como um vetor, uma **matriz** armazena dados que seguem um mesmo tipo básico, sendo assim chamada de **estrutura de dados homogênea (bidimensional)**.
- A variável referente a uma **matriz** atua como uma grade de linhas e colunas, onde a interseção entre uma linha e uma coluna armazena um dado.

— Exemplo: MAT[3][4]

A diagram of a 3x4 matrix grid. The columns are indexed 1 to 4 from left to right, and the rows are indexed 1 to 3 from top to bottom. The grid is labeled 'MAT' at the bottom center.

	1	2	3	4
1				
2				
3				
	MAT			

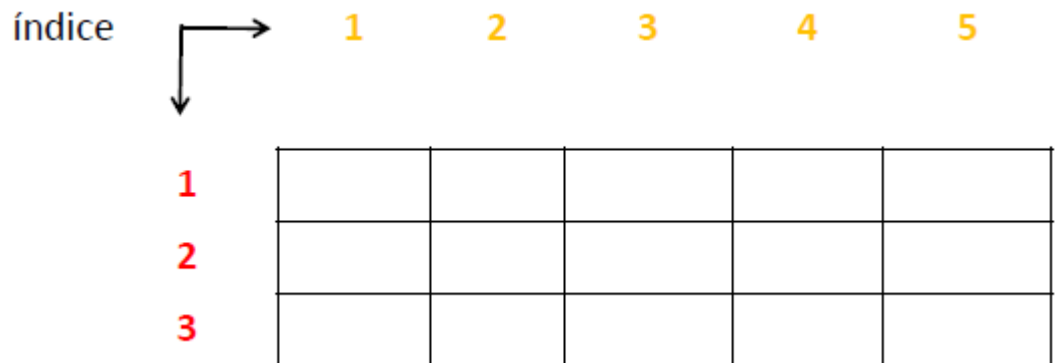
Matrizes

Declaração

- Declaração de uma matriz:

DECLARE nome[dimensão 1, dimensão 2] Tipo

- **nome:** é o identificador da matriz (nome dado à mesma)
- **[dimensão1, dimensão 2]:** são as possíveis dimensões da matriz
- **Tipo:** é o tipo de dados que pode ser armazenado na matriz (inteiro, real, caractere ou lógico)
- Exemplo: criar uma matriz de 3 linhas e 5 colunas
 - **DECLARE matX[3, 5] INTEIRO**



Matrizes

Atribuição de Valores

- Atribuição de valores:

- $\text{matX}[1,4] \leftarrow 5$

Atribui 5 à posição correspondente à **linha 1** e à **coluna 4** da matriz

The diagram shows a 2x6 matrix. Above the matrix, column indices 1 through 6 are labeled in orange. To the left of the matrix, row indices 1 and 2 are labeled in red. An arrow labeled 'índice' points to the row indices. The value 5 is assigned to the cell at row 1, column 4, indicated by a blue '5' in that cell.

índice	1	2	3	4	5	6
1				5		
2						

Matrizes

Atribuição de Valores


- Preenchendo uma matriz:
 - Para preencher uma matriz, é necessário identificar todas as suas posições. Como possui duas dimensões, é preciso a utilização de um **índice** para cada dimensão da matriz.
 - Exemplo:

```
PARA i ← 1 ATÉ 2 FAÇA
  INÍCIO
    PARA j ← 1 ATÉ 3 FAÇA
      INÍCIO
        ESCREVA "Digite uma valor: "
        LEIA matX[i,j]
      FIM
    FIM
  FIM
```

- O exemplo acima preenche uma matriz com 2 linhas e 3 colunas. Observe que o índice *i* varia com número de linhas e o índice *j* varia com número de colunas. Cabe ressaltar que, para cada valor de *i*, a variável *j* varia de 1 a 3, ou seja, as 3 colunas que cada linha possui.

Matrizes

Atribuição de Valores

Memória		Tela 	Digitado
i	j		
1	1	Digite uma valor:	12
	2	Digite uma valor:	9
	3	Digite uma valor:	3
2	1	Digite uma valor:	-23
	2	Digite uma valor:	4
	3	Digite uma valor:	2

índice

	1	2	3
1	12	9	3
2	-23	4	2

Matrizes

Acesso aos Valores

- Exibindo valores da matriz:
 - Para mostrar os elementos de uma matriz, é preciso identificar suas posições. Assim, exige-se a utilização de um **índice** para cada dimensão

```
PARA i ← 1 ATÉ 2 FAÇA
  INÍCIO
    PARA j ← 1 ATÉ 3 FAÇA
      INÍCIO
        ESCREVA matX[i,j]
      FIM
    FIM
  FIM
```

Matrizes em C

Declaração

- Declaração de arranjos multidimensionais em C:
 - **tipo nome[dimensão1] [dimensão2] ... [dimensão n]**
 - **tipo**: é o tipo de dados que a matriz pode armazenar
 - **nome**: é o identificador da matriz (nome dado à mesma)
 - **[dimensão1][dimensão2]...[dimensão n]**: representam as possíveis dimensões da matriz
 - Exemplo:
 - `int materia[4][10]`
matriz **materia** de **inteiros** que possui **4 linhas** e **10 colunas**
 - `char mat[4][5]`
matriz **mat** de **caracteres** que possui **4 linhas** e **5 colunas**

Matrizes em C

Declaração

- De forma semelhante como ocorre em vetores, os índices começam sempre de **0 (zero)** em C.
- Assim, com a declaração anterior criou-se uma variável chamada **materia** contendo 4 linhas (0 a 3) com 10 colunas (0 a 9), capazes de armazenar números inteiros

```
int materia[4][10]
```

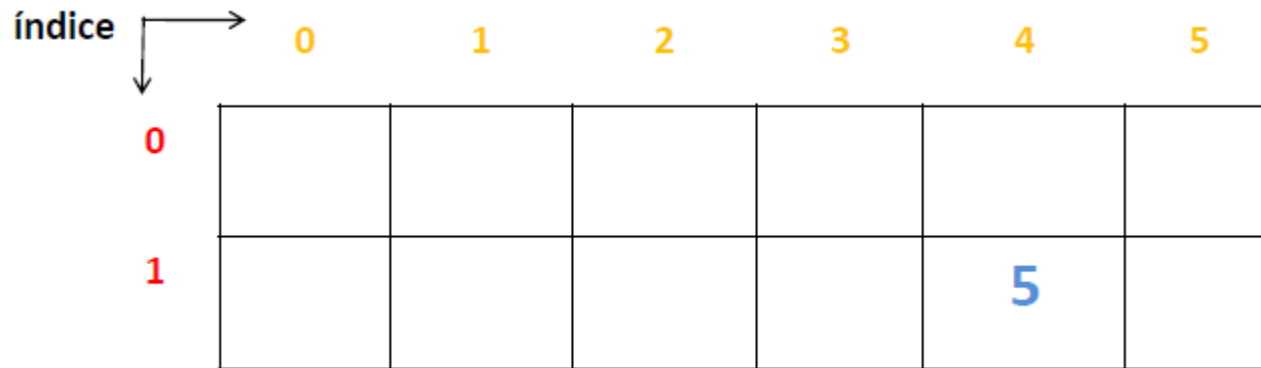
índice

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										

Matrizes em C

Atribuição de Valores

- Atribuição de valores:
 - Exemplo: `materia [1][4] = 5;`
Atribui o valor 5 à segunda linha (índice 1) quinta coluna (índice 4) da matriz **materia**



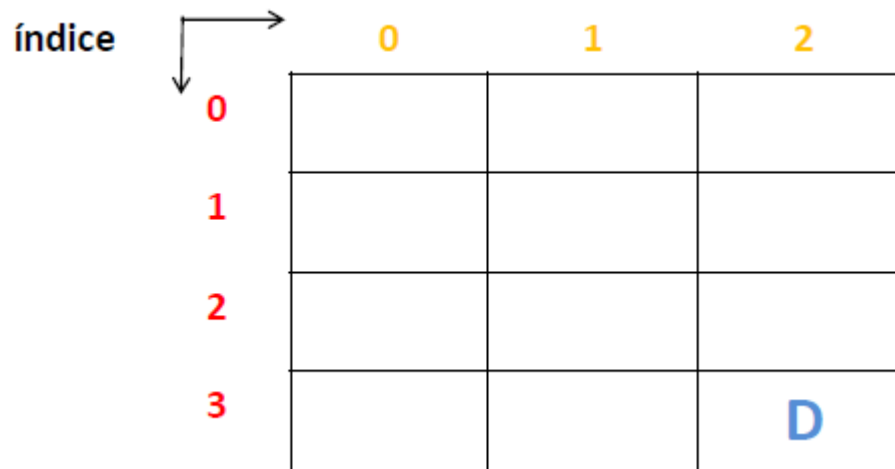
The diagram illustrates a 2x6 matrix. To the left of the matrix, a vertical arrow points downwards from the word 'índice' to the row indices '0' and '1', which are written in red. Above the matrix, a horizontal arrow points to the right from the word 'índice' to the column indices '0', '1', '2', '3', '4', and '5', which are written in orange. The matrix itself is a grid of 12 cells. The cell at row index 1 and column index 4 contains the value '5' in blue.

índice	0	1	2	3	4	5
0						
1					5	

Matrizes em C

Atribuição de Valores

- Atribuição de valores
 - Exemplo: `mat[3][2] = 'D';`
Atribui a letra 'D' à quarta linha (índice 3) terceira coluna (índice 2) da matriz **mat**



	0	1	2
0			
1			
2			
3			D

Matrizes em C

Atribuição de Valores

- Preenchendo uma matriz:

```
int materia[4][10];
for ( int i = 0 ; i < 4 ; i++ ) { //linhas
    for ( int j = 0 ; j < 10 ; j++ ) { //colunas
        printf("Digite o valor da linha %d coluna %d:", i, j);
        scanf("%d", &materia[i][j]);
    }
}
```

❑ Obs.: Uso de duas estruturas de repetição para acessar todos os valores da matriz.

❑ O **for** externo (índice **i**) varia de 0 a 3 (percorrendo as 4 linhas)

❑ O **for** interno (índice **j**) varia de 0 a 9 (percorrendo as 10 colunas)

Matrizes em C

Acesso aos Valores

- Mostrando valores da matriz:

```
for ( int i = 0 ; i < 4 ; i++ ) { //linhas
    for ( int j = 0 ; j < 10 ; j++ ) { //colunas
        printf("%d ", materia[i][j]);
    }
    printf("\n");
}
```

- ❑ Obs.: Uso de duas estruturas de repetição para acessar todos os valores da matriz.
 - ❑ O **for** externo (índice **i**) varia de 0 a 3 (percorrendo as 4 linhas)
 - ❑ O **for** interno (índice **j**) varia de 0 a 9 (percorrendo as 10 colunas)

Matrizes em C

Inicialização Direta

- Usa-se lista de chaves para inicializar a matriz no momento da sua declaração.

```
int mat[2][3] = { {10, 20, 30}, {40, 50, 60} };  
    // [      10, 20, 30,  
    //      40, 50, 60 ]
```

```
int mat[2][3] = { {10, 20}, {40} };  
    // [      10, 20, 0,  
    //      40,  0, 0 ]
```

```
int mat[2][3] = { {10}, { } };  
    // [      10, 0, 0,  
    //      0, 0, 0  ]
```

```
int mat[2][3] = { };  
    // [      0, 0, 0,  
    //      0, 0, 0  ]
```

Matrizes em C

Inicialização Direta

- Usa-se lista de chaves para inicializar a matriz no momento da sua declaração.

```
int mat[2][3] = { {10, 20}, {[1]=40} };  
    // [      10, 20, 0,  
    //      0, 40, 0 ]
```

```
int mat[2][3] = { {10, 20}, {[1]=40}, {50} }; //ERRO  
    // [      10, 20, 0,  
    //      0, 40, 0 ]
```

```
int mat[2][3] = { [0][1] = 20, [1][2]=30 };  
    // [      0, 20, 0,  
    //      0,  0, 30]
```

```
int mat[2][3] = { 10, 20, 30, 40, 50, 60 };  
    // [      10, 20, 30,  
    //      40, 50, 60      ]
```

Matrizes em C

Inicialização Direta

- Usa-se lista de chaves para inicializar a matriz no momento da sua declaração.

```
int mat[ ][ ] = { {10, 20}, {30, 40} }; //ERRO
```

```
int mat[ ][3] = { {10, 20}, {30, 40} };  
    // [    10, 20, 0,  
    //    30, 40, 0    ]
```

```
int mat[2][3] = { {10, 20}, {[1]=40, 50} };  
    // [    10, 20, 0,  
    //    0, 40, 50    ]
```


Arranjos Multidimensionais em C

- Exemplo para 3 dimensões

```
int mat[4][5][10];  
for ( int i = 0 ; i < 4 ; i++ ) { //dimensão1  
    for ( int j = 0 ; j < 5 ; j++ ) { //dimensão2  
        for ( int k = 0 ; k < 10 ; k++ ) { //dimensão3  
            printf("Digite o valor da dimensão %d, %d, %d", i, j, k);  
            scanf("%d", &mat[i][j][k]);  
        }  
    }  
}
```

Exercícios

1. Quantos elementos existem nos arranjos abaixo?

a) `int matriz1 [2][3][4]`

b) `float matriz2 [10][5][10][6]`

c) `float matriz3 [1][1][1][1]`

Exercícios

2. Indique os valores para o arranjo **m** tridimensional:

DECLARE **m**[2][4][3] INTEIRO

		1	2	3	4			1	2	3	4			1	2	3	4
1	2	1	2	3	4	1	2	1	1	1	1	1	2	0	0	1	1
		5	-5	3	0			-3	2	0	0			-1	-1	-2	-2
		1						2						3			

- a) **m**[1][2][3]
- b) **m**[2][4][3]
- c) **m**[2][1][1]
- d) **m**[3][5][4]

Exercícios

3. Observe o programa a seguir. Qual a saída do programa?

```
int main() {
    int t, i, m[3][4];
    for (t = 0; t < 3; t++) {
        for (i = 0; i < 4; i++) {
            m[t][i] = (t*4)+i+1;
        }
    }
    for (t = 0; t < 3; t++) {
        printf("\n");
        for (i = 0; i < 4; i++) {
            printf("%4d", m[t][i]);
            i < 3 ? printf("|") : 0;
        }
    }
}
```

Exercícios

5. Faça um algoritmo que imprima a transposta de uma matriz 5x5.

Exercícios

6. Faça um algoritmo que imprima os elementos da diagonal principal de uma matriz 5x5.

Exercícios

7. Faça um algoritmo que imprima os elementos da diagonal secundária de uma matriz 5x5.

Exercícios

8. Faça um algoritmo que imprima os elementos abaixo da diagonal principal de uma matriz 5x5.

Exercícios

9. Faça um algoritmo que imprima os elementos abaixo da diagonal secundária de uma matriz 5x5.

Exercícios

10. Faça um algoritmo pseudocódigo, fluxograma e C para preencher uma matriz com 5 notas de 10 alunos. Após isso, calcule, imprima e armazene em um vetor a média das notas de cada aluno.

Observações

Constantes em C

```
const int nLin = 4;
const int nCol = 3;
int m[nLin][nCol];
for ( int i = 0 ; i < nLin ; i++ ) {
    for ( int j = 0 ; j < nCol ; j++ ) {
        printf("%d ", m[i][j]);
    }
    printf("\n");
}
```

- ❑ nLin e nCol são constantes, isto é, não podemos atribuir novos valores a essas variáveis.
- ❑ Note que nLin e nCol foram usadas para declarar a matriz e também para as estruturas de repetição. Boa prática!

Constantes em C

```
#define nLin 4
```

```
#define nCol 3
```

```
int m[nLin][nCol];  
for ( int i = 0 ; i < nLin ; i++ ) {  
    for ( int j = 0 ; j < nCol ; j++ ) {  
        printf("%d ", m[i][j]);  
    }  
    printf("\n");  
}
```

- ☐ nLin e nCol são constantes, mas em nível de pré-processador!
- ☐ nLin e nCol não possuem tipo.
- ☐ Se usarmos const, são variáveis e possuem tipo.
- ☐ Se usarmos define, há “apenas” substituição de texto.

Arranjo de tamanho variável

- Arranjos de tamanho variável (variable-length array - VLA) são arranjos cujos tamanhos são determinados em tempo de execução ao invés de tempo de compilação.
- Suponha que não se saiba, em tempo de compilação, qual o tamanho do arranjo e que essa informação será fornecida, por exemplo, pelo usuário no momento da execução do programa.
- Nesse caso, precisamos ser capazes de declarar o array com tamanho definido no momento da execução do programa, isto é, com tamanho variável.
- Para isso, usamos VLA.

Arranjo de tamanho variável

```
#include <stdio.h>
```

```
int main()
{
    int n;
    printf("Digite o tamanho do vetor: ");
    scanf("%d", &n);
    int v[n];
    for(int i=0; i<n; i++) {
        printf("Digite um numero: ");
        scanf("%d", &v[i]);
    }
    printf("Imprimindo o vetor: \n");
    for(int i=0; i<n; i++) {
        printf("%d ", v[i]);
    }
}
```

Arranjo de tamanho variável

```
#include <stdio.h>
```

```
int main()
{
    int lin, col;
    printf("Digite a quantidade de linhas e colunas da matriz M: ");
    scanf("%d %d", &lin, &col);
    int mat[lin][col];
    printf("Preenchendo a matriz:\n");
    for(int i=0; i<lin; i++) {
        for(int j=0; j < col; j++){
            printf("M(%d,%d)=", i+1, j+1);
            scanf("%d", &mat[i][j]);
        }

    }
    printf("Imprimindo a matriz:\n");
    for(int i=0; i<lin; i++) {
        for(int j=0; j < col; j++){
            printf("%5d", mat[i][j]);
        }
        printf("\n");
    }
}
```


Ponteiros

Endereço de Memória

- Podemos obter o endereço de memória de uma variável usando o operador &:

```
int myAge = 43;
```

```
printf("%d", myAge); // Imprime o valor de myAge (43)
```

```
printf("%p", &myAge); // Imprime o endereço de memória de myAge (0x7ffe5367e044)
```

- Você já usou isso no scanf!!!

Ponteiros

- Ponteiro é uma variável que armazena o endereço de memória de outra variável.
- Uma variável do tipo ponteiro define o tipo de dado para o qual aponta (ex. int).
- O ponteiro é criado com o operador *.
- O endereço da variável apontada é atribuído ao ponteiro.

```
int myAge = 43;
int* ptr = &myAge; // ptr é um ponteiro para int, que guarda o endereço de myAge

// Imprime o valor de myAge (43)
printf("%d\n", myAge);

// Imprime o endereço de memória de myAge (0x7ffe5367e044)
printf("%p\n", &myAge);

// Imprime o endereço de memória apontado por ptr, que corresponde ao endereço de
myAge (0x7ffe5367e044)
printf("%p\n", ptr);
```

Ponteiros

Algumas razões para o uso de ponteiros:

- Manipular elementos de arranjos;
- Receber argumentos em funções que necessitem modificar o argumento original;
- Criar estruturas de dados complexas, como listas encadeadas e árvores binárias, em que um item deve conter referências a outro;
- Alocar e desalocar memória do sistema;
- Passar para uma função o endereço de outra.
- Manipular arquivos.

Cuidado! Ponteiros devem ser tratados com atenção, uma vez que é possível corromper dados armazenados em outros endereços de memória.

Ponteiros

- Por meio do ponteiro, é possível acessar o conteúdo da variável apontada, usando o operador de indireção (dereferência) *.

```
int myAge = 43;
int* ptr = &myAge;

// Imprime o endereço de memória de myAge (0x7ffe5367e044)
printf("%p\n", ptr);

// Imprime o valor armazenado por myAge (43) usando dereferência ou indireção
printf("%d\n", *ptr);

// Altera o valor de myAge usando o ponteiro por dereferência ou indireção
*ptr = 51;

// Imprime o o valor armazenado em myAge (51)
printf("%d\n", myAge);
```

Ponteiros

- Ponteiros devem ser inicializados quando são definidos ou então em uma instrução de atribuição.
- Ponteiros podem ser inicializados com NULL, zero, ou um endereço.
- NULL: não aponta para nada, é uma constante simbólica.
- Inicializar um ponteiro com zero é o mesmo que inicializar com NULL.
- As expressões abaixo são equivalentes.

```
int* ptr = NULL;
```

```
int *ptr = 0;
```

Operações com Ponteiros

- Por meio do ponteiro, é possível acessar o conteúdo da variável apontada, usando o operador de indireção (dereferência) *.

```
int x1 = 43, x2 = 51;  
int *ptr1 = &x1, *ptr2 = &x2;
```

```
// ptr1 passa a apontar para x2  
ptr1 = ptr2;
```

```
// Imprime o valor armazenado por x2 (51)  
printf("%d\n", *ptr1);
```

Operações com Ponteiros

```
int myAge = 43;
int* ptr = &myAge;

// Imprime o endereço de memória de myAge (0x7ffe5367e044)
printf("%p\n", ptr);

// Imprime o tamanho em memória de myAge - int - em bytes (4)
printf("%d\n", sizeof(myAge));

// Incrementa o ponteiro - vai para o próximo endereço de memória com referência a
int - avança 4 bytes
ptr++;

// Imprime o endereço de memória apontado por ptr (0x7ffe5367e048).
printf("%p\n", ptr);

// Imprime o endereço de memória correspondente ao avanço de 8 bytes (2 * 4) a
partir de ptr (0x7ffe5367e050). ptr não foi alterado
printf("%p\n", ptr+2);

// Imprime o endereço de memória apontado por ptr (0x7ffe5367e048).
printf("%p\n", ptr);
```


Operações com Ponteiros

- Por meio do ponteiro, é possível acessar o conteúdo da variável apontada, usando o operador de indireção (dereferência) *.

```
int x1 = 43, x2 = 51;  
int *ptr1 = &x1, *ptr2 = &x2;
```

```
// Imprime o endereço de memória apontado por ptr1 (0x7ffe5367e044).  
printf("%d\n", ptr1);
```

```
// Imprime o endereço de memória apontado por ptr2 (0x7ffe5367e048).  
printf("%d\n", ptr2);
```

```
// Imprime a quantidade de avanços de ptr1 para ptr2 (1).  
printf("%d\n", ptr2 - ptr1);
```

```
ptr2++;
```

```
// Imprime a quantidade de avanços de ptr1 para ptr2 (2).  
printf("%d\n", ptr2 - ptr1);
```

Ponteiros x Arranjos

- Ponteiros também podem ser usados para acessar arranjos.
- Considere o seguinte arranjo de inteiros:

```
int myNumbers[4] = {25, 50, 75, 100};           //Imprime:  
                                                25  
for (int i = 0; i < 4; i++) {                 50  
    printf("%d\n", myNumbers[i]);             75  
}                                              100
```

```
int myNumbers[4] = {25, 50, 75, 100};           //Imprime:  
                                                0x7ffe70f9d8f0  
for (int i = 0; i < 4; i++) {                 0x7ffe70f9d8f4  
    printf("%p\n", &myNumbers[i]);           0x7ffe70f9d8f8  
}                                              0x7ffe70f9d8fc
```

- Logo, podemos imprimir (percorrer) o arranjo com ponteiro!

Ponteiros x Arranjos

- Em C, a variável que define arranjo é, na verdade, um ponteiro para o primeiro elemento do arranjo.
- O endereço do primeiro elemento do arranjo é o mesmo da variável do arranjo.
- Considere o seguinte arranjo de inteiros:

```
int myNumbers[4] = {25, 50, 75, 100};
```

```
printf("%p\n", myNumbers);
```

```
printf("%p\n", &myNumbers[0])}
```

```
//Imprime:
```

```
0x7ffe70f9d8f0
```

```
0x7ffe70f9d8f0
```

Ponteiros x Arranjos

- Imprimindo o vetor com ponteiro:

```
int myNumbers[4] = {25, 50, 75, 100};           //Imprime:
int *ptr = myNumbers;                           25
                                                  50
for (int i = 0; i < 4; i++) {                   75
    printf("%d\n", *(ptr + i));                 100
}
```

```
int myNumbers[4] = {25, 50, 75, 100};           //Imprime:
int *ptr = myNumbers;                           25
                                                  50
for (int i = 0; i < 4; i++) {                   75
    printf("%d\n", ptr[i]);                     100
}
```

Ponteiros x Arranjos

- Preenchendo o vetor com ponteiro:

```
int myNumbers[4];
int *ptr = myNumbers;

for (int i = 0; i < 4; i++) {
    scanf("%d", ptr+i);
}
```

```
int myNumbers[4];
int *ptr = myNumbers;

for (int i = 0; i < 4; i++) {
    scanf("%d", &ptr[i]);
}
```

Ponteiros x Arranjos

- Preenchendo o vetor com ponteiro:

```
int myNumbers[4];  
int *ptr = myNumbers;
```

```
*ptr = 10;  
*(ptr+1) = 20;  
*(ptr+2) = 30;  
*(ptr+3) = 40;
```

```
int myNumbers[4];  
int *ptr = myNumbers;
```

```
ptr[0] = 10;  
ptr[1] = 20;  
ptr[2] = 30;  
ptr[3] = 40;
```

Ponteiros

- Exercício: O que será impresso pelo programa?

```
#include <stdio.h>

int main()
{
    int a=5, b=12, c;
    int *p;
    int *q;
    p = &a;
    q = &b;
    c = *p + *q;
    printf("c = %d", c);
}
```

Ponteiros

- Exercício: O que será impresso pelo programa?

```
#include <stdio.h>

int main()
{
    int a=5, b=12, c;
    int *p;
    int *q;
    p = &a;
    q = &b;
    c = *p + *q;
    printf("c = %d", c);
}
```

Resposta:

c = 17

Ponteiros

- Exercício: O que será impresso pelo programa?

```
#include <stdio.h>
```

```
int main()
{
    int x, y, *p;
    y = 0;
    p = &y;
    x = *p;
    x = 4;
    (*p)++;
    --x;
    (*p) += x;
    printf("x=%d y=%d *p=%d", x, y, *p);
}
```

Ponteiros

- Exercício: O que será impresso pelo programa?

```
#include <stdio.h>

int main()
{
    int x, y, *p;
    y = 0;
    p = &y;
    x = *p;
    x = 4;
    (*p)++;
    --x;
    (*p) += x;
    printf("x=%d y=%d *p=%d", x, y, *p);
}
```

Resposta:

x=3 y=4 *p=4

Ponteiros

- Exercício: O que será impresso pelo programa?

```
#include <stdio.h>
#include <math.h>

int main()
{
    const int TAM = 10;
    int vet[10] = {1,2,3};
    int *ptr = vet;
    for(int i = 0; i < TAM; i++) {
        *(ptr++) = pow(i+1, 2);
    }
    for(int i = 0; i < TAM; i++) {
        printf("%4d ", *(--ptr));
    }
}
```

Ponteiros

- Exercício: O que será impresso pelo programa?

```
#include <stdio.h>
#include <math.h>

int main()
{
    const int TAM = 10;
    int vet[10] = {1,2,3};
    int *ptr = vet;
    for(int i = 0; i < TAM; i++) {
        *(ptr++) = pow(i+1, 2);
    }
    for(int i = 0; i < TAM; i++) {
        printf("%4d ", *(--ptr));
    }
}
```

Resposta:

100 81 64 49 36 25 16 9 4 1

Ponteiros

- Exercício: Considere a matriz a seguir. Como preencher a matriz usando ponteiro? Como imprimir a matriz usando ponteiro?

```
#include <stdio.h>
#define NLIN 2
#define NCOL 3

int main()
{
    int m[NLIN][NCOL];
}
```

Alocação Dinâmica de Memória

- A área de alocação dinâmica é chamada heap. A área de alocação estática é chamada stack.
- A linguagem C oferece um conjunto de funções que permitem a alocação ou a liberação dinâmica de memória:
 - malloc(): aloca memória e não inicializa
 - calloc(): aloca memória e inicializa com bits zerados
 - realloc(): altera o tamanho da memória alocada
 - free(): libera a memória alocada
- As funções estão disponíveis na biblioteca “stdlib.h”.
- As funções trabalham com ponteiros.

Alocação Dinâmica de Memória

- malloc:
- a função “malloc” ou “alocação em memória” em C é usada para alocar dinamicamente um único bloco de memória com o tamanho especificado.

```
int *ptr;
```

```
int n = 5;
```

```
ptr = (int*) malloc(n * sizeof(int));
```

- Serão alocados 20 bytes por cada inteiro ter tamanho de 4 bytes.
- Se não houver espaço disponível, malloc retorna NULL.

Alocação Dinâmica de Memória

- calloc:
- a função “calloc” aloca memória dinamicamente, mas inicializa todos os bits com zero.

```
int *ptr;
```

```
int n = 5;
```

```
ptr = (int*) calloc(n, sizeof(int));
```

- Serão alocados 20 bytes por cada inteiro ter tamanho de 4 bytes.
- Se não houver espaço disponível, calloc retorna NULL.

Alocação Dinâmica de Memória

- realloc:
- a função de “realocação” em C realloc é usada para alterar dinamicamente uma alocação feita anteriormente com “malloc”.
- A realocação de memória mantém a informação já armazenada e os blocos extras alocados serão inicializados com “lixo”. Se não houver espaço, retorna NULL.

```
int *pi;
```

```
pi = (int *) malloc(sizeof(int));
```

```
pi = (int *) realloc(pi, 5*sizeof(int));
```

Alocação Dinâmica de Memória

- free:
- A função free em C é usada para desalocar dinamicamente a memória.
- A memória alocada com as funções malloc (), calloc() e realloc () não é desalocada automaticamente.
- A função free é ajuda a reduzir o desperdício de memória ao liberá-la

```
int *pi;  
pi = (int *) malloc(sizeof(int));  
pi = (int *) realloc(pi, 5*sizeof(int));  
free(pi);
```

Alocação Dinâmica de Memória

- Exemplo: Programa para calcular a soma de n números digitados pelo usuário.

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

int main() {
    int n, i, *ptr, sum = 0;

    printf("Digite a quantidade de números: ");
    scanf("%d", &n);

    //alocando memória
    ptr = (int*) malloc(n * sizeof(int));

    //se a memória não tiver sido alocada
    if(ptr == NULL) {
        printf("Error na alocação dinâmica");
        exit(0);
    }

    printf("Digite os números: ");
    for(i = 0; i < n; ++i) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Soma = %d", sum);

    //desalocando memória
    free(ptr);

    return 0;
}
```

Alocação Dinâmica de Memória

- Exemplo: Programa para calcular a soma de n números digitados pelo usuário.

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

int main() {
    int n, i, *ptr, sum = 0;

    printf("Digite a quantidade de números: ");
    scanf("%d", &n);

    //alocando memória
    ptr = (int*) malloc(n * sizeof(int));

    //se a memória não tiver sido alocada
    if(ptr == NULL) {
        printf("Error!          memory          not
allocated.");
        exit(0);
    }

    printf("Digite os números: ");
    for(i = 0; i < n; ++i) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Soma = %d", sum);

    //desalocando memória
    free(ptr);

    return 0;
}
```

```
Digite a quantidade de números: 3
Digite os números: 100
20
36
Soma = 156
```

Alocação Dinâmica de Memória

- Exemplo: Programa que usa realloc.

```
int main() {
    int *ptr, i , n1, n2;
    printf("Digite o tamanho: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Endereços de memória previamente alocados:\n");
    for(i = 0; i < n1; ++i)
        printf("%p\n", ptr + i);

    printf("\nDigite o novo tamanho: ");
    scanf("%d", &n2);

    // Realocando memória
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("Endereços de memória após realocação:\n");
    for(i = 0; i < n2; ++i)
        printf("%p\n", ptr + i);

    free(ptr);

    return 0;
}
```

Alocação Dinâmica de Memória

- Exemplo: Programa que usa realloc.

```
int main() {
    int *ptr, i, n1, n2;
    printf("Digite o tamanho: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Endereços de memória previamente
    alocados:\n");
    for(i = 0; i < n1; ++i)
        printf("%p\n", ptr + i);

    printf("\nDigite o novo tamanho: ");
    scanf("%d", &n2);

    // Realocando memória
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("Endereços de memória após
    realocação:\n");
    for(i = 0; i < n2; ++i)
        printf("%p\n", ptr + i);

    free(ptr);

    return 0;
}
```

Digite o tamanho: 2

Endereços de memória previamente
alocados:

26855472

26855476

Digite o novo tamanho: 4

Endereços de memória após realocação:

26855472

26855476

26855480

26855484

Alocação Dinâmica de Memória

- Exercício: Crie um programa que aloca dinamicamente uma matriz com NLIN linhas e NCOL colunas, onde NLIN e NCOL são definidos pelo usuário. O programa deve preencher a matriz e imprimir em seguida.

Alocação Dinâmica de Memória

- Solução 1 – Usando ponteiro simples

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

```
int main()
{
    int NLIN, NCOL, *p;
    printf("Digite o numero de linhas e colunas: ");
    scanf("%d %d", &NLIN, &NCOL);
    p = (int*) malloc(NLIN * NCOL * sizeof(int));
    for(int i = 0; i < NLIN * NCOL; i++) {
        printf("Digite um numero: ");
        scanf("%d", p+i); //scanf("%d", &p[i]);
    }
    for(int i = 0; i < NLIN; i++) {
        for(int j = 0; j < NCOL; j++) {
            printf("%5d", *(p + i*NCOL + j)); //printf("%5d", p[i*M + j]);
        }
        printf("\n");
    }
    free(p);
    return 0;
}
```


Alocação Dinâmica de Memória

- Solução 1 – Usando ponteiro simples

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

int main()
{
    int NLIN, NCOL, *p;
    printf("Digite o numero de linhas e colunas: ");
    scanf("%d %d", &NLIN, &NCOL);
    p = (int*) malloc(NLIN * NCOL * sizeof(int));
    for(int i = 0; i < NLIN * NCOL; i++) {
        printf("Digite um numero: ");
        scanf("%d", p+i); //scanf("%d", &p[i]);
    }
    for(int i = 0; i < NLIN * NCOL; i++) {
        if(i % NCOL == 0)
            printf("\n");
        printf("%5d", *(p + i)); //printf("%5d", p[i]);
    }
    free(p);
    return 0;
}
```

Alocação Dinâmica de Memória

- Solução 1 – Usando ponteiro simples

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

int main()
{
    int NLIN, NCOL, *p;
    printf("Digite o numero de linhas e colunas: ");
    scanf("%d %d", &NLIN, &NCOL);
    p = (int*) malloc(NLIN * NCOL * sizeof(int));
    for(int i = 0; i < NLIN * NCOL; i++) {
        printf("Digite um numero: ");
        scanf("%d", p+i); //scanf("%d", &p[i]);
    }
    for(int i = 0; i < NLIN * NCOL; i++) {
        printf("%5d", *(p + i)); //printf("%5d", p[i]);
        if((i + 1) % NCOL == 0)
            printf("\n");
    }
    free(p);
    return 0;
}
```

Alocação Dinâmica de Memória

- Solução 2 – Usando arranjo de ponteiro

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

int main()
{
    int NLIN, NCOL;
    printf("Digite o numero de linhas e colunas: ");
    scanf("%d %d", &NLIN, &NCOL);
    int *p[NLIN];
    for(int i = 0; i < NLIN; i++) {
        p[i] = (int*) malloc(NCOL * sizeof(int));
    }
    for(int i = 0; i < NLIN; i++) {
        for(int j = 0; j < NCOL; j++) {
            printf("Digite um numero: ");
            scanf("%d", &p[i][j]); //scanf("%d", p[i] + j);
        }
    }
    for(int i = 0; i < NLIN; i++) {
        for(int j = 0; j < NCOL; j++) {
            printf("%5d", p[i][j]); //printf("%5d", *(p[i] + j));
        }
        printf("\n");
    }
    for(int i = 0; i < NLIN; i++) {
        free(p[i]);
    }
    return 0;
}
```

Alocação Dinâmica de Memória

- Solução 3 – Usando ponteiro para ponteiro

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

int main()
{
    int NLIN, NCOL;
    printf("Digite o numero de linhas e colunas: ");
    scanf("%d %d", &NLIN, &NCOL);
    int **p = (int**) malloc(NLIN * sizeof(int*));
    for(int i = 0; i < NLIN; i++) {
        p[i] = (int*) malloc(NCOL * sizeof(int));
    }
    for(int i = 0; i < NLIN; i++) {
        for(int j = 0; j < NCOL; j++) {
            printf("Digite um numero: ");
            scanf("%d", &p[i][j]); //scanf("%d", *(p+i) + j);
        }
    }
    for(int i = 0; i < NLIN; i++) {
        for(int j = 0; j < NCOL; j++) {
            printf("%5d", p[i][j]); //printf("%5d", (*(p+i) + j));
        }
        printf("\n");
    }
    for(int i = 0; i < NLIN; i++) {
        free(p[i]);
    }
    free(p);
    return 0;
}
```

Referências bibliográficas

- **ASCENCIO, A. F. G. e CAMPOS, E. A. V. Fundamentos da Programação de Computadores – Algoritmos, Pascal e C/C++. São Paulo: Pearson Prentice Hall, 2007. 2ª Edição.**
- **SOUZA, A. Furlan; GOMES, Marcelo Marques; SOARES, Marcio Vieira e CONCILIO, Ricardo. Algoritmos e Lógica de Programação. 2ª ed. Ver. e ampl. São Paulo: Cengage Learning 2011.**