



## **Aula 22: Revisão para Prova 02**

### **Introdução a Programação**

---

**Túlio Toffolo & Puca Huachi**  
<http://www.toffolo.com.br>

BCC202 — 2019/2  
Departamento de Computação — UFOP

# Avaliação

- 3 Provas (60% da nota):
  - Prova 01: 15% da nota (24/09/2019).
  - **Prova 02: 20% da nota (29/10/2019).**
  - Prova 03: 25% da nota (03/12/2019).
- Exercícios em aula práticas (10% da nota):
  - Atividades em todas as aulas serão entregues via moodle.
- Trabalho(s) prático(s) (30% da nota):
  - Entrega 01: 10% da nota.
  - Entrega 02: 20% da nota.
  - Código e documentação serão entregues via moodle.
  - Apresentação para o(s) professor(es) da disciplina no final do semestre.
- Ponto extra: frequência e exercícios nas aulas teóricas

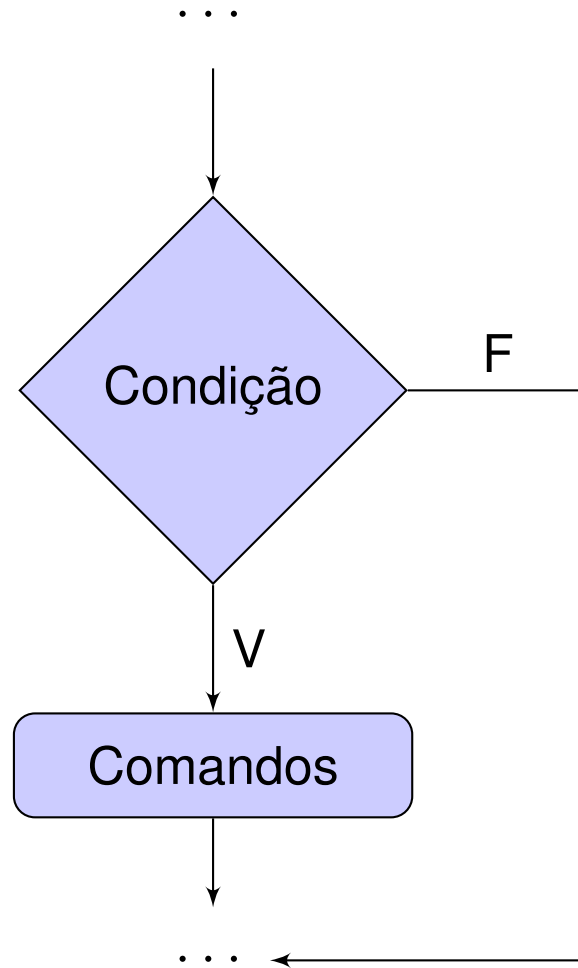
# Revisão

- A seguir alguns slides foram selecionados das aulas anteriores para uma breve revisão do conteúdo.
- A aula de hoje está organizada em 4 partes:
  1. Comandos de repetição
  2. Vetores (*arrays*)
  3. Matrizes
  4. Cadeia de caracteres

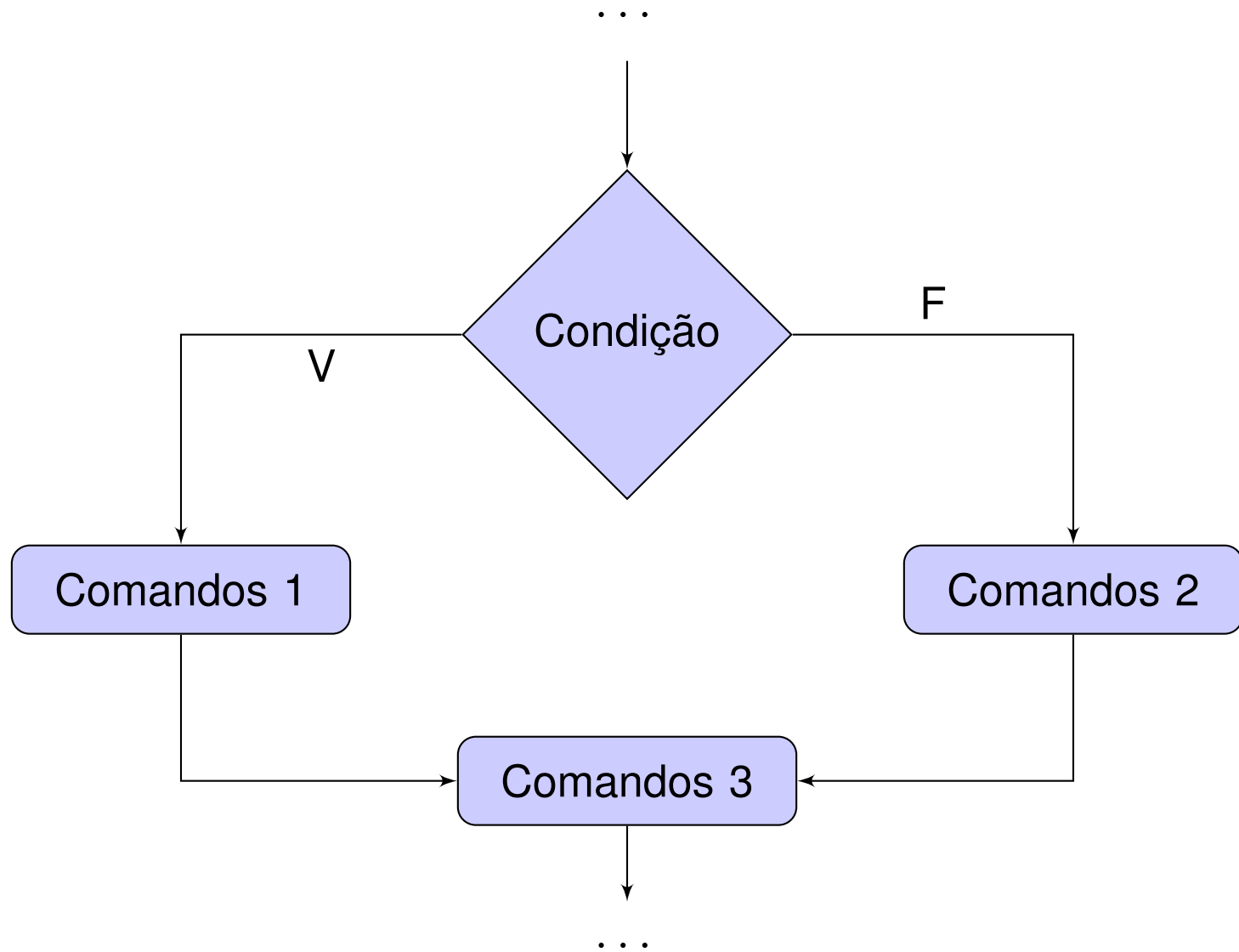
# 1

## Comandos de repetição

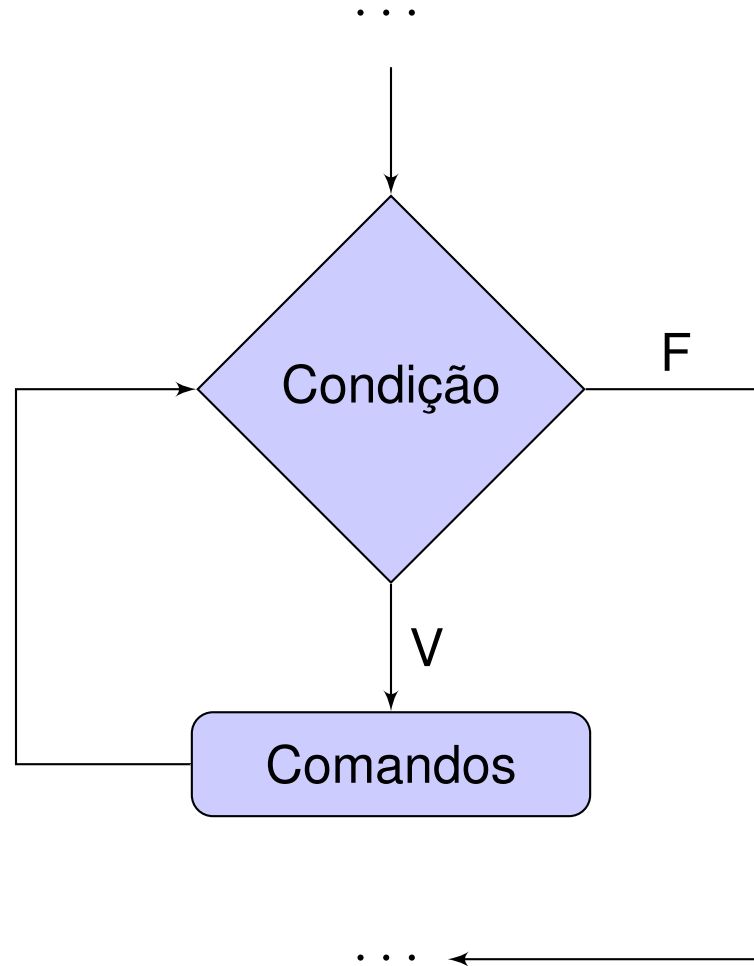
# Tomada de decisão



# Tomada de decisão



# Laço while



# Laço while

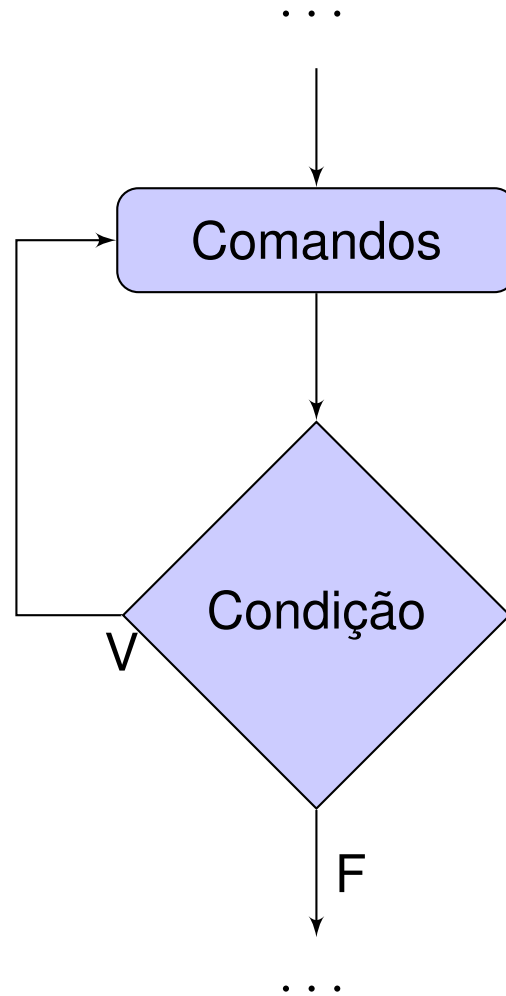
- Sintaxe:

```
1 while (<expressão>
2 {
3     <comando_1>;
4     ...
5     <comando_n>;
6 }
```

- As linhas 2–6 representam o corpo do laço.



# Laço do-while



## Comando do-while

```
1 do {  
2     <comando_1>;  
3     ...  
4     <comando_n>;  
5 } while (<expressão>;
```

No comando **do-while**, ao contrário do comando `while`, o teste do laço/loop está no final, isso significa que os <comandos> serão executados **no mínimo uma vez**.

# Laços for

Em um laço **controlado por contador**, os comandos (corpo do laço) são repetidos um **número predeterminado de vezes**.

Sintaxe:

```
1  for (<inicialização>; <condição>; <incremento>)
2  {
3      <comando_1>;
4      ...
5      <comando_n>;
6  }
```

# Exercícios da última aula

## Exercício 2

Suponha que exista 50 alunos em uma sala. Faça um programa que determina quantos desses alunos tem idade maior que 20 anos. O usuário (coitado) deve digitar a matrícula e idade de todos os 50 alunos.

**Qual é o melhor comando de repetição para resolver o exercício?**

```
1  int main()
2  {
3      int matricula, idade;
4      int contador = 0;
5      int i = 0;
6      while (i < 50) {
7          printf("Digite a matrícula e idade do aluno %d: ", i+1);
8          scanf("%d %lf", &matricula, &idade);
9          if (idade > 20) contador++;
10         i++;
11     }
12     printf("\n%d alunos tem mais de 20 anos!\n", contador);
13     return 0;
14 }
```

```
1  int main()
2  {
3      int matricula, idade;
4      int contador = 0;
5      for (int i = 0; i < 50; i++) {
6          printf("Digite a matrícula e idade do aluno %d: ", i+1);
7          scanf("%d %lf", &matricula, &idade);
8          if (idade > 20) contador++;
9      }
10     printf("\n%d alunos tem mais de 20 anos!\n", contador);
11     return 0;
12 }
```

# Laços Aninhados

|   |           |
|---|-----------|
| 1 | 1         |
| 2 | 22        |
| 3 | 333       |
| 4 | 4444      |
| 5 | 55555     |
| 6 | 666666    |
| 7 | 7777777   |
| 8 | 88888888  |
| 9 | 999999999 |

**Repetição 1:** temos nove repetições de linhas ( $1 \leq n \leq 9$ ).

**Repetição 2:** temos, em cada linha, a repetição de  $n$  caracteres que identificam a própria linha, sendo  $1 \leq n \leq 9$ . Assim, temos  $n = 1$  na linha 1,  $n = 2$  na linha 2, e assim sucessivamente, até a linha 9.

- Para obter a saída acima, realizamos a **Repetição 2** dentro da **Repetição 1**.

# Laços Aninhados

```
1  #include <stdio.h>
2
3  int main()
4  {
5      // Repetição 1
6      for (int linha = 1; linha <= 9; linha++) {
7
8          // Repetição 2
9          for (int coluna = 1; coluna <= linha; coluna++) {
10             printf("%d", linha);
11         }
12
13         printf("\n");
14     }
15
16     return 0;
17 }
```

## Exemplo 1

Faça um programa que imprime a tabuada de  $x$  até  $y$  (valores de  $x$  e  $y$  devem ser digitados pelo usuário).

```
1  Digite os valores para x e y: 5 15
2
3  Tabuada de multiplicação!
4
5      |   5   6   7   8   9  10  11  12  13  14  15
6  -----
7  5 |  25  30  35  40  45  50  55  60  65  70  75
8  6 |  30  36  42  48  54  60  66  72  78  84  90
9  7 |  35  42  49  56  63  70  77  84  91  98 105
10 8 |  40  48  56  64  72  80  88  96 104 112 120
11 9 |  45  54  63  72  81  90  99 108 117 126 135
12 10 | 50  60  70  80  90 100 110 120 130 140 150
13 11 | 55  66  77  88  99 110 121 132 143 154 165
14 12 | 60  72  84  96 108 120 132 144 156 168 180
15 13 | 65  78  91 104 117 130 143 156 169 182 195
16 14 | 70  84  98 112 126 140 154 168 182 196 210
17 15 | 75  90 105 120 135 150 165 180 195 210 225
```



```

1  int main()
2  {
3      int x, y;
4      printf("Digite os valores para x e y: ");
5      scanf("%d %d", &x, &y);
6
7      // imprimindo o cabeçalho
8      printf("\nTabuada de multiplicação!\n\n");
9      printf("    | ");
10     for (int j = x; j <= y; j++)
11         printf("%3d ", j);
12     printf("\n----");
13     for (int j = x; j <= y; j++)
14         printf("----");
15     printf("\n");
16
17     // calculando (e imprimindo) a tabuada
18     for (int i = x; i <= y; i++) {
19         printf("%2d | ", i);
20         for (int j = x; j <= y; j++)
21             printf("%3d ", i*j);
22         printf("\n");
23     }
24     return 0;
25 }

```

# Comando `continue`

Exemplo de uso em laço `while`:

```
1 while (<condição>) {  
2     <comando_1>;  
3     ...  
4     continue;  
5     ...  
6     <comando_n>;  
7 }
```

# Comando break

Exemplo de uso em laço **for**:

```
1  for (<inicialização>; <condição>; <incremento/decremento>) {  
2      <comando_1>;  
3      ...  
4      break;  
5      ...  
6      <comando_n>;  
7  }
```

2

**Vetores (*arrays*)**

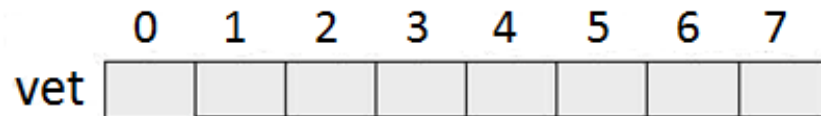
# Introdução a vetores

- Um vetor é:
  - uma variável composta **homogênea unidimensional**,
  - formada por uma sequência de valores, todos do mesmo **tipo de dado**,
  - alocados sequencialmente na memória,
  - que são acessados usando um único identificador (nome).

# Introdução a vetores

O que distingue os diferentes valores armazenados em um vetor é um **índice**.

● Exemplo:



# Declaração de um vetor

<tipo> identificador [<número de posições>];

- Tipo: int, float, double, etc.
- Identificador: é o nome da variável que identifica o vetor.
- Número de posições: é o tamanho do vetor!

## Exemplos:

```
1 int vetor[5];
```

```
1 double notas[50];
```

```
1 char palavra[20];
```

# Declaração de um vetor

Diferentes forma de declarar um vetor:

```
1 // declaração sem inicializar os valores do vetor (eles terão 'lixo')
2 int v1[3];
3
4 // declaração inicializando os valores do vetor
5 int v2[3] = {0, 2, 5};
6
7 // declaração alternativa inicializando os valores do vetor
8 int v3[] = {0, 2, 5};
```



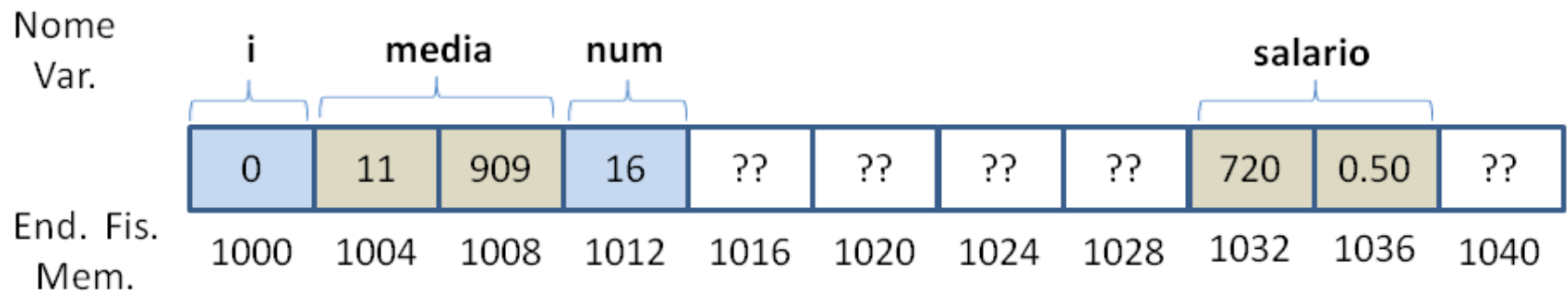
# Uso de constantes em vetores

```
1  ...
2  #define TAM_MAX 10
3
4  int main()
5  {
6      double vetor[TAM_MAX];
7
8      // coloca os valores {TAM_MAX, TAM_MAX-1, ..., 1} no vetor
9      for (int i = 0; i < TAM_MAX; i++) {
10         vetor[i] = TAM_MAX - i;
11     }
12     ...
13     return 0;
14 }
```

# Alocação de memória para vetores

Exemplo (assuma que toda a memória disponível está ilustrada abaixo):

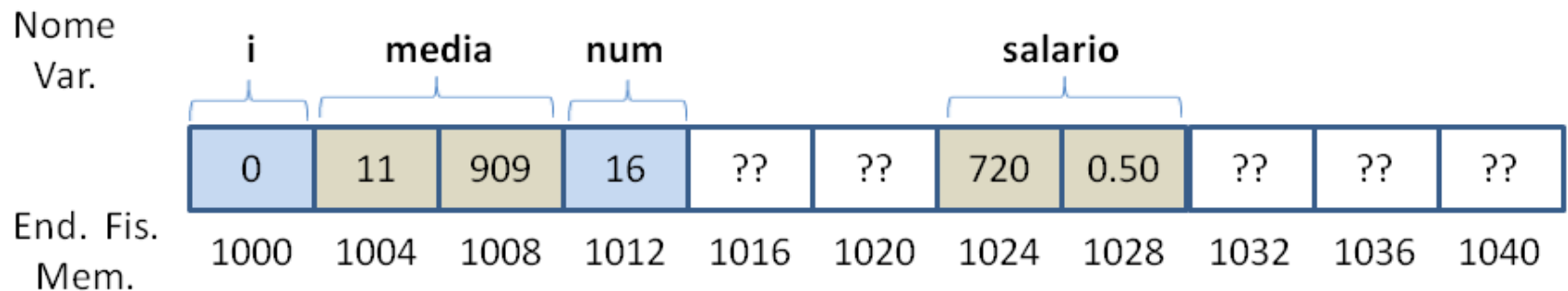
```
float salario;  
int i = 0;  
float media = 11909;  
int num;  
...  
num = 16;  
salario = 7200.50;  
int vet[4]; // ok, existe espaço
```



# Alocação de memória para vetores

Exemplo (assuma que toda a memória disponível está ilustrada abaixo):

```
float salario;  
int i = 0;  
float media = 11909;  
int num;  
...  
num = 16;  
salario = 7200.50;  
int vet[4]; // não existe espaço
```



# Vetores são ponteiros?

Então, a variável `vetor` do código abaixo armazena um **endereço de memória**:

```
1  int vetor[5];
```

- No entanto, `vetor` tem uma característica especial: é “**read-only**”.
- Portanto o **endereço de memória** para o qual `vetor` aponta não pode ser alterado.

# Conteúdo de vetores

E o código abaixo?

```
1  int vetor[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
2  printf("vetor[0] = %d\n", vetor[0]);
```

- Vai imprimir o **conteúdo** do inteiro na posição 0 de **vetor**. Ou seja, o **conteúdo** do endereço de memória **vetor+0**.

```
1  vetor[0] = 1
```

# Conteúdo de vetores

E o código abaixo?

```
1  int vetor[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
2  printf("vetor[5] = %d\n", vetor[5]);
```

- Vai imprimir o **conteúdo** do inteiro na posição 5 de **vetor**. Ou seja, o **conteúdo** do endereço de memória **vetor+5**.
- Note que ao somar 5 em **vetor**, será somado o valor  $20 = 5 \times 4$  ao endereço de memória de **vetor**, onde:
  - 5 é o número de elementos;
  - 4 é o tamanho de cada elemento (no caso, de um `int`).

```
1  vetor[5] = 6
```

# Conteúdo de vetores

E o código abaixo?

```
1  int nro = 10;  
2  int *p = &nro;  
3  printf("*p = %d\n", *p);  
4  printf("p[0] = %d\n", p[0]);
```

- Vai imprimir o **conteúdo** do ponteiro **p** e, em seguida, o **conteúdo** do endereço de memória **p+0**.

```
1  *p = 10  
2  p[0] = 10
```

# Vetores e funções

E... se quisermos criar uma função que busca um elemento?

- O que a função retornaria?
- Quais seriam os parâmetros da função?
  - Vamos precisar saber qual **vetor** e qual o **tamanho**.
- Assim, qual seria um possível protótipo para a função?

```
1  /* Função que busca um número em um vetor de inteiros e retorna a
2   * posição em que o número está; caso o número não seja encontrado,
3   * a função retorna -1.
4   */
5  int buscaLinear(int vetor[], int tamanho, int valor);
```



# Ponteiros e vetores

A variável que representa um vetor pode ser vista como um ponteiro.

- Mas... o que o código a seguir vai imprimir?

```
1  int v[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
2  printf("*v = %d", *v);
```

```
1  *v = 1
```

- `*v` imprime o conteúdo (do tipo `int`) que está no endereço de memória `v`, que é exatamente igual a `v[0]`.

# Ponteiros e vetores

Portanto:

- $v[0]$  é equivalente a  $*v$ , que é equivalente a  $*(v+0)$
- $v[4]$  é equivalente a  $*(v+4)$

Aritmética de ponteiros:

- Ao somar 4 em um ponteiro do tipo `int*`, estamos “pulando” 4 inteiros.
- Assim, podemos utilizar **indexação** ( $v[4]$ ) ou aritmética de ponteiros ( $*(v+4)$ ) para ler/escrever na memória.

# Ponteiros e vetores

Qual a diferença prática das funções a seguir?

```
1 void imprimeVetor1(int v[], int n) {
2     for (int i = 0; i < n; i++)
3         printf("%d ", v[i]);
4     printf("\n");
5 }
6
7 void imprimeVetor2(int *v, int n) {
8     for (int i = 0; i < n; i++)
9         printf("%d ", v[i]);
10    printf("\n");
11 }
12
13 void imprimeVetor3(int *v, int n) {
14     for (int i = 0; i < n; i++)
15         printf("%d ", *(v+i));
16    printf("\n");
17 }
```

3

**Matrizes**

# Matrizes: variáveis compostas homogêneas

- As variáveis compostas homogêneas correspondem a um conjunto de elementos de mesmo tipo e que compartilham um mesmo nome;
- Cada um dos elementos é unicamente identificado por um número inteiro (índice) que especifica a sua localização dentro da estrutura;
- Estas variáveis podem ser **unidimensionais (vetores)** ou **multidimensionais (matrizes)**;

# Matriz bi-dimensional

Por exemplo, uma matriz bi-dimensional pode ser vista como uma tabela de  $m$  linhas e  $n$  colunas.

The diagram illustrates a 3x4 matrix. The rows are indexed 0, 1, and 2, and the columns are indexed 0, 1, 2, and 3. A bracket on the left indicates the number of rows is  $m$ , and a bracket on top indicates the number of columns is  $n$ .

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 |   |   |   |   |
| 1 |   |   |   |   |
| 2 |   |   |   |   |

# Declaração de matrizes

`<tipo> <identificador> [<linhas>] [<colunas>];`

- `<tipo>`: tipo dos dados que serão armazenados no vetor (int, char, float, etc);
- `<identificador>`: nome dado à variável;
- `<linhas>`: número de elementos da primeira dimensão;
- `<colunas>`: número de elementos da segunda dimensão;
- As linhas e colunas são numeradas de 0 até *tamanho* – 1.

# Observação

- C/C++ não verifica o limite das dimensões das variáveis compostas;
- Se uma instrução for feita com índices além do limite, é possível que não ocorra um erro de execução do programa e outros valores sejam sobrepostos na memória;
- É responsabilidade do programador providenciar a verificação dos limites das dimensões das variáveis compostas;



# Inicialização de Matrizes II

Inicializando na declaração. Processo semelhante à inicialização de vetores.

```
1  int matriz[3][4] = { {10, 20, 30, 40},  
2                        {50, 60, 70, 80},  
3                        {90, 11, 22, 33} };
```

Mas podemos fazer também:

```
1  int matriz[3][4] = { 10, 20, 30, 40,  
2                        50, 60, 70, 80,  
3                        90, 11, 22, 33 };
```

Ou ainda:

```
1  int matriz[3][4] = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 11, 22, 33 };
```

## Passando matrizes por parâmetro

Em C/C++ você precisa indicar o tamanho de todas as dimensões de uma matriz passada por parâmetro, exceto a dimensão mais à esquerda.

```
1 void imprimirMatriz2(int matriz[][10], int n, int m)
2 {
3     for (int i = 0; i < n; ++i) {
4         for (int j = 0; j < m; ++j)
5             printf("%d ", matriz[i][j]);
6         printf("\n");
7     }
8 }
```

```
1 int main()
2 {
3     int A[10][10];
4     ...
5     imprimirMatriz2(A, 10, 10);
6     ...
7     return 0;
8 }
```

# Passando matrizes por parâmetro

Mas... porquê todas as dimensões menos a mais à esquerda??

- Por conta da forma como matrizes são representadas na memória!
- As linhas são colocadas sequencialmente em um “**vetorzão**”.
- Exemplo: seja a matriz  $3 \times 3$  a seguir

```
1  matriz[3][3] = { { 10, 20, 30 },  
2                    { 40, 50, 60 },  
3                    { 70, 80, 90 } };
```

- Ela será representada na memória como um vetor de tamanho 9:

```
1  i          --->    0   0   0   1   1   1   2   2   2  
2  j          --->    0   1   2   0   1   2   0   1   2  
3  M[i][j]    ---> { 10, 20, 30, 40, 50, 60, 70, 80, 90 }
```

# Passando matrizes por parâmetro

Logo, quando acessamos o campo  $[i][j]$  de uma matriz  $4 \times 5$ :

- C/C++ acessa o campo  $5 \times i + j$  do “vetorzão” (em que 5 é o número de colunas).
- Para tal, o compilador deve saber quantas colunas há em cada linha
  - Ou seria impossível multiplicar por **5** neste exemplo.

Se você não quiser definir as dimensões da sua matriz em tempo de compilação, há algumas alternativas...

- que aprenderemos em breve...  
(quando falarmos sobre **alocação dinâmica**)

4

# Cadeias de caracteres

# Tabela ASCII

| Decimal | Hex | Char                   | Decimal | Hex | Char    | Decimal | Hex | Char | Decimal | Hex | Char  |
|---------|-----|------------------------|---------|-----|---------|---------|-----|------|---------|-----|-------|
| 0       | 0   | [NULL]                 | 32      | 20  | [SPACE] | 64      | 40  | @    | 96      | 60  | `     |
| 1       | 1   | [START OF HEADING]     | 33      | 21  | !       | 65      | 41  | A    | 97      | 61  | a     |
| 2       | 2   | [START OF TEXT]        | 34      | 22  | "       | 66      | 42  | B    | 98      | 62  | b     |
| 3       | 3   | [END OF TEXT]          | 35      | 23  | #       | 67      | 43  | C    | 99      | 63  | c     |
| 4       | 4   | [END OF TRANSMISSION]  | 36      | 24  | \$      | 68      | 44  | D    | 100     | 64  | d     |
| 5       | 5   | [ENQUIRY]              | 37      | 25  | %       | 69      | 45  | E    | 101     | 65  | e     |
| 6       | 6   | [ACKNOWLEDGE]          | 38      | 26  | &       | 70      | 46  | F    | 102     | 66  | f     |
| 7       | 7   | [BELL]                 | 39      | 27  | '       | 71      | 47  | G    | 103     | 67  | g     |
| 8       | 8   | [BACKSPACE]            | 40      | 28  | (       | 72      | 48  | H    | 104     | 68  | h     |
| 9       | 9   | [HORIZONTAL TAB]       | 41      | 29  | )       | 73      | 49  | I    | 105     | 69  | i     |
| 10      | A   | [LINE FEED]            | 42      | 2A  | *       | 74      | 4A  | J    | 106     | 6A  | j     |
| 11      | B   | [VERTICAL TAB]         | 43      | 2B  | +       | 75      | 4B  | K    | 107     | 6B  | k     |
| 12      | C   | [FORM FEED]            | 44      | 2C  | ,       | 76      | 4C  | L    | 108     | 6C  | l     |
| 13      | D   | [CARRIAGE RETURN]      | 45      | 2D  | -       | 77      | 4D  | M    | 109     | 6D  | m     |
| 14      | E   | [SHIFT OUT]            | 46      | 2E  | .       | 78      | 4E  | N    | 110     | 6E  | n     |
| 15      | F   | [SHIFT IN]             | 47      | 2F  | /       | 79      | 4F  | O    | 111     | 6F  | o     |
| 16      | 10  | [DATA LINK ESCAPE]     | 48      | 30  | 0       | 80      | 50  | P    | 112     | 70  | p     |
| 17      | 11  | [DEVICE CONTROL 1]     | 49      | 31  | 1       | 81      | 51  | Q    | 113     | 71  | q     |
| 18      | 12  | [DEVICE CONTROL 2]     | 50      | 32  | 2       | 82      | 52  | R    | 114     | 72  | r     |
| 19      | 13  | [DEVICE CONTROL 3]     | 51      | 33  | 3       | 83      | 53  | S    | 115     | 73  | s     |
| 20      | 14  | [DEVICE CONTROL 4]     | 52      | 34  | 4       | 84      | 54  | T    | 116     | 74  | t     |
| 21      | 15  | [NEGATIVE ACKNOWLEDGE] | 53      | 35  | 5       | 85      | 55  | U    | 117     | 75  | u     |
| 22      | 16  | [SYNCHRONOUS IDLE]     | 54      | 36  | 6       | 86      | 56  | V    | 118     | 76  | v     |
| 23      | 17  | [ENG OF TRANS. BLOCK]  | 55      | 37  | 7       | 87      | 57  | W    | 119     | 77  | w     |
| 24      | 18  | [CANCEL]               | 56      | 38  | 8       | 88      | 58  | X    | 120     | 78  | x     |
| 25      | 19  | [END OF MEDIUM]        | 57      | 39  | 9       | 89      | 59  | Y    | 121     | 79  | y     |
| 26      | 1A  | [SUBSTITUTE]           | 58      | 3A  | :       | 90      | 5A  | Z    | 122     | 7A  | z     |
| 27      | 1B  | [ESCAPE]               | 59      | 3B  | ;       | 91      | 5B  | [    | 123     | 7B  | {     |
| 28      | 1C  | [FILE SEPARATOR]       | 60      | 3C  | <       | 92      | 5C  | \    | 124     | 7C  |       |
| 29      | 1D  | [GROUP SEPARATOR]      | 61      | 3D  | =       | 93      | 5D  | ]    | 125     | 7D  | }     |
| 30      | 1E  | [RECORD SEPARATOR]     | 62      | 3E  | >       | 94      | 5E  | ^    | 126     | 7E  | ~     |
| 31      | 1F  | [UNIT SEPARATOR]       | 63      | 3F  | ?       | 95      | 5F  | _    | 127     | 7F  | [DEL] |

# Cadeia de caracteres

Cadeias de caracteres são simplesmente arrays/vetores de caracteres que terminam com o caractere `'\0'`:

- O caractere especial `'\0'` indica o final da cadeia de caracteres
- Note que para armazenar 10 caracteres precisamos de 11 posições
  - Uma posição adicional para o caractere `'\0'`
- Estas cadeias são também chamadas de *strings*

# Exemplos

Suponha um array de 15 caracteres

- `char nome[15]:`

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

- `scanf("%s", nome);`  
(suponha que o usuário digitou Puca)

|   |   |   |   |    |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|----|---|---|---|---|---|---|---|---|---|---|
| P | u | c | a | \0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
|---|---|---|---|----|---|---|---|---|---|---|---|---|---|---|



# Operações em cadeia de caracteres

A função `strlen()` (abreviação de *string length*) é utilizada para calcular o tamanho de uma *string*.

- Sim, **calcular** o tamanho: a função percorrerá o array de caracteres em busca do caractere `'\0'`.
- Lembre-se, portanto, que há um custo elevado em chamar essa função várias vezes.

# Operações em cadeia de caracteres

Exemplo:

```
1 char nome[15];  
2 scanf("%s", nome); // le uma 'string' do usuário  
3  
4 int tamanho = strlen(nome); // calcula o tamanho da 'string'  
5 printf("%d\n", tamanho);
```

Se o usuário digitar 'Toffolo', qual será o conteúdo do vetor?

|   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|
| T | o | f | f | o | l | o | \0 | ? | ? | ? | ? | ? | ? | ? |
|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|

O tamanho da cadeia de caracteres (ou seja, a 'posição' do \0):

1 7

## Biblioteca <string.h>

A biblioteca <string.h> contém algumas funções úteis:

- `char *strcpy(char x[], char y[]);`  
copia a string x (inclusive o byte \0 final) no espaço alocado para y. Cabe ao usuário garantir que o espaço alocado a y tem pelo menos `strlen(x) + 1` bytes. Retorna o endereço de x.
- `char *strcat(char x[], char y[]);`  
concatena as strings x e y, isto é, acrescenta y ao final de x. Retorna o endereço da string resultante, ou seja, o endereço de x. Cabe ao usuário garantir que o espaço alocado a x é suficiente para comportar `strlen(y)` bytes adicionais.
- `int strcmp(char x[], char y[]);`  
compara lexicograficamente as strings x e y. Retorna um número estritamente negativo se x vem antes de y, 0 se x é igual a y e um número estritamente positivo se x vem depois de y.

# Manipulação de strings

Podemos utilizar `scanf` para ler strings.

- No entanto, `scanf` finalizará a leitura quando encontrar um espaço.
- Exemplo:

```
1 char nomeCompleto[100];  
2 printf("Digite o nome completo: ");  
3 scanf("%s", nomeCompleto);  
4 printf("Nome: %s", nomeCompleto);
```

- Exemplo de execução do código acima:

```
1 Digite o nome completo: Tulio Angelo Machado Toffolo  
2  
3 Nome: Tulio
```

# Manipulação de strings

Uma alternativa é utilizar a função `gets` ou `fgets`:

- `gets(x)`: lerá da entrada o que for digitado pelo usuário até uma quebra de linha (`'\n'`) ser detectada e armazenará na *string* `x`.
- `fgets(x, n, stdin)`: lerá da entrada o que for digitado pelo usuário até uma quebra de linha (`'\n'`) ser detectada ou o limite máximo de `n` caracteres ser atingido, e armazenará na *string* `x`.
  - `stdin`: constante que indica a entrada padrão (entrada do usuário). É uma abreviação para *standard input*.

Importante: `fgets` **incluirá o caractere** `'\n'` em `x`.



Perguntas?