

Linguagens de Programação 1

```
printf("\aAula #%d\n", 3 + ++4 / 2);
```

Message of the Day

"Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin."

— John von Neumann

John von Neumann (1903–1957)

Inventor da arquitectura Von-Neumann que se tornou standard em processadores modernos. Tinha apenas uma memória tanto para dados como para código.

Contribuiu para a teoria dos jogos.

Teve um papel importante no projecto Manhattan (bomba atómica)

Conteúdo

O que vamos ver hoje?

Vectores: Como guardar vários valores do mesmo tipo

Strings: Cadeias de caracteres terminadas em `\0`

Matrizes: Tabelas de valores (vectores de vectores)

🎯 Vectores (Arrays)

Um vector é um conjunto de elementos do **mesmo tipo**.

Ocupam **posições contíguas** na memória.

Índices começam em **0!** 🚧

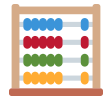
```
char socos[5] = {100, 120, 90, 130, 110};
printf("Soco mais forte: %d\n", socos[3]); // 130
```

📌 A tabela exemplifica o modelo de memória. Os Endereços são definidos no momento em que o programa corre.

Modelo da memória RAM:

| Endereço | Conteúdo | Identificador |
|----------|----------|---------------|
| 1024 | 100 | socos[0] |
| 1025 | 120 | socos[1] |
| 1026 | 90 | socos[2] |
| 1027 | 130 | socos[3] |
| 1028 | 110 | socos[4] |
| | 1024 | socos |
| ... | | |

📌 a variável `socos` contém o endereço de memória do elemento `socos[0]` (primeiro elemento do vector) 🤔🤔🤔🤔




Vectores (Arrays)



Declaração de vectores

```
tipo nome_do_vector[nr_de_elementos];
```

 o tipo pode ser qualquer um dos tipos que já conhecemos: `char`, `int`, `float`, `double`, `enum`.

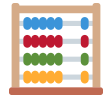
 pode ainda ser precedido dos qualificadores: `long`, `short`, `signed`, `unsigned`.

 Também podem ser de tipos criados por nós (vamos ver mais à frente)

Exemplos:

```
int players[50];  
double energy[50];
```

```
#define DIM 100  
char letters[DIM];  
unsigned short ids[2*DIM+2];
```



Inicialização automática de vectores

Apenas pode ser feita no momento da **declaração**. É possível inicializar automaticamente todos os elementos de um vector

```
int var[5] = {10, 15, 18, 20, 25};
```

Apenas no momento da declaração, o número de elementos pode ser omitido:

```
int var[] = {10, 15, 18, 20, 25};
```

O compilador percebe qual o tamanho necessário para o vector, neste caso 5.



Inicialização automática de vectores



Se indicarmos o **número de elementos**, mas **não inicializarmos todos**, os restantes são inicializados automaticamente com o valor **0**.

```
int var[5] = {10, 15};
```

| indice | 0 | 1 | 2 | 3 | 4 |
|--------|----|----|---|---|---|
| var | 10 | 15 | 0 | 0 | 0 |



Dica: Para inicializar tudo com **0**, faz `int golpes[10] = {0};`

Percorrer um Vector

```
#define DIM 5
int golpes[DIM] = {50, 100, 20, 80, 15};
```

| índice | conteúdo | variável |
|--------|----------|-----------|
| 0 | 50 | golpes[0] |
| 1 | 100 | golpes[1] |
| 2 | 20 | golpes[2] |
| 3 | 80 | golpes[3] |
| 4 | 15 | golpes[4] |

São indexados a partir da posição 0, até à posição $n-1$ (sendo n o número de elementos).

Usamos um **ciclo for** para iterar pelos elementos:

```
for (int i = 0; i < DIM; i++)
{
    printf("Golpes[%d]: %d\n", i, golpes[i]);
}
```


Funções com Vectors

📌 Em C não interessa a **dimensão** do vector que é passado como argumento de uma função. Apenas o seu tipo de dados.

```
void mostrar(int golpes[10]) {...}
```

equivale a:

```
void mostrar(int golpes[20]) {...}
```

equivale a:

```
void mostrar(int golpes[]) {...}
```

🤔🤔 Então, como é que sabemos o tamanho do vector dentro da função?

R: temos de passar um argumento extra com o tamanho do vector:

```
void mostrar(int golpes[], int tamanho) {...}
```

🚫 **Problema:** `sizeof(golpes)` dentro da função NÃO devolve o tamanho correto! 🤯

Funções com Vectors (exemplo)


```
#include <stdio.h>
#define DIM 5

void mostrar(int golpes[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        printf("Golpe %d: %d\n", i, golpes[i]);
    }
}
```

```
int main()
{
    int golpes[DIM] = {80, 95, 110, 120, 100};
    mostrar(golpes, DIM);
    return 0;
}
```

O Operador `sizeof`

 Usado para obter o tamanho (em bytes) de um tipo ou variável.

 Retorna um valor do tipo `size_t` . (basicamente é um inteiro)

 Sintaxe:

```
sizeof(tipo)  
sizeof(variável)
```

O Operador `sizeof`: Como Funciona?

```
int a;  
printf("Tamanho de int: %lu bytes\n", sizeof(int));  
printf("Tamanho de a: %lu bytes\n", sizeof(a));
```

✓ Ambas as chamadas retornam o tamanho de um `int`, mas a primeira usa o nome do tipo e a segunda usa uma variável.

Tamanhos Comuns de Tipos Primitivos

| Tipo | Tamanho (pode variar) |
|--------|-----------------------|
| char | 1 byte |
| int | 4 bytes |
| float | 4 bytes |
| double | 8 bytes |
| long | 4 ou 8 bytes |
| short | 2 bytes |

 Os tamanhos podem variar dependendo do sistema e compilador!

sizeof com Vetores

```
int vector[10];

printf("Tamanho do vector: %lu bytes\n", sizeof(vector));

printf("Tamanho de um elemento: %lu bytes\n", sizeof(vector[0]));

printf("Número de elementos: %lu\n", sizeof(vector) / sizeof(vector[0]));
```

- ✓ `sizeof(vector)` retorna o tamanho total do array em bytes.
- ✓ Para obter o número de elementos, dividimos pelo tamanho de um único elemento.

⚠️ `sizeof` em Vetores dentro de Funções

📌 Não é possível obter o tamanho real do vetor passado como argumento para uma função. ❌

```
void tamanhoArray(int v[]) {  
    printf("Tamanho dentro da função: %lu bytes\n", sizeof(v));  
}  
  
int main() {  
    int arr[10];  
    printf("Tamanho no main: %lu bytes\n", sizeof(arr));  
    tamanhoArray(arr);  
    return 0;  
}
```

✅ No `main()`, `sizeof(arr)` retorna o tamanho correto.

❌ Dentro da função, `sizeof(arr)` retorna o tamanho de um ponteiro, **não do vector** 🤖🤖🤖.

Vetores como Parâmetros de Função

- 📌 Em C, os vetores são passados por referência para funções.
- 📌 Isso significa que qualquer alteração dentro da função afeta o vetor original.
- ✅ Exemplo: Modificando um vetor dentro de uma função

```
#include <stdio.h>

void modificar(int v[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        v[i] *= 2; // Dobra cada elemento
    }
}
```

```
int main() {
    int valores[] = {1, 2, 3, 4, 5};
    int n = 5;

    modificar(valores, n);

    for (int i = 0; i < n; i++) {
        printf("%d ", valores[i]); // Imprime: 2 4 6 8 10
    }
    return 0;
}
```

- ✅ O vetor `valores` foi modificado dentro da função `modificar()` !

Vetores vs. Variáveis Comuns


 Comparação entre variáveis comuns e vetores passados para funções:

✓ Variável comum (cópia, não afeta a original):

```
void alterar(int x) {  
    x = 10;  
}  
  
int main() {  
    int a = 5;  
    alterar(a);  
    printf("%d\n", a); // Imprime 5 (valor não alterado)  
}
```

✓ Vetor (modifica o original, pois é passado por referência):

```
void alterarVetor(int v[]) {  
    v[0] = 99;  
}  
  
int main() {  
    int arr[] = {1, 2, 3};  
    alterarVetor(arr);  
    printf("%d\n", arr[0]); // Imprime 99 (valor foi alterado)  
}
```

 Vetores são sempre passados como referência, enquanto variáveis comuns são passadas por valor.

VLAs (Variable Length Arrays) em C

São vectores com tamanho variável determinado em tempo de execução.

Introduzidos no C99, mas removidos no C++ e desaconselhados no C11.

 A sua utilização é proibida nesta disciplina


 Exemplo de um VLA:

```
#include <stdio.h>

void criarVetor(int n) {
    int v[n]; // VLA (tamanho definido em tempo de execução)
    for (int i = 0; i < n; i++) {
        v[i] = i * 2;
        printf("%d ", v[i]);
    }
}
```

```
int main() {
    int tamanho;
    printf("Digite um tamanho: ");
    scanf("%d", &tamanho);

    criarVetor(tamanho);
    return 0;
}
```

 O tamanho do vetor `v[n]` só é conhecido em tempo de execução.

✗ Por que evitar VLAs?

- ✗ **Sem alocação dinâmica eficiente:** Usa **stack**, o que pode causar **stack overflow**.
- ✗ **Baixa portabilidade:** Não é suportado por todos os compiladores C.
- ✗ **C11 tornou o suporte opcional:** Muitos compiladores como **MSVC** não aceitam VLAs.
- ✗ **Impossível verificar o tamanho em tempo de compilação.**
- ✓ Veremos **Alternativas** mais à frente.

! Retorno de Vectores em Funções !

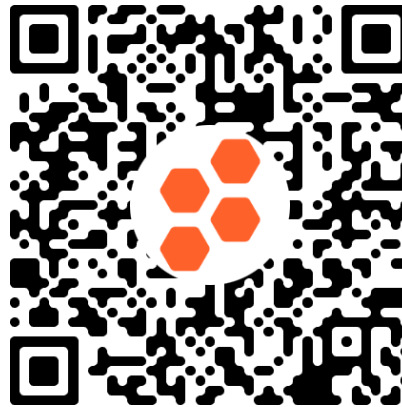
📌 Vectores criados dentro de uma função não podem ser retornados.

📌 A memória do vector local é desalocada ao sair da função!

```
int* criarArray()  
{  
    int asneira_suprema[5] = {1, 2, 3, 4, 5};  
  
    return asneira_suprema; // ✗ ERRO: Retorna memória inválida  
}
```

✅ **Solução:** existe, mas vamos ver mais à frente...

? Quizz - Vectores



No campo nome devem colocar o número de aluno 2XXXXXXX.



Strings - cadeias de caracteres

Cadeias de Caracteres (Strings) em C

Strings em C são **vetores de caracteres** terminados pelo caractere especial `\0` (caractere nulo).

Devemos sempre reservar espaço para o caractere `\0` ao declarar uma string.

```
char nome[10]; // Permite até 9 caracteres + '\0'
```

Declaração e Inicialização de Strings

```
char nome[20] = "oscar";
```

```
char nome[20] = {'o','s','c','a','r', '\0'};
```

```
char nome[] = "oscar";
```

// 0 compilador define o tamanho automaticamente. Incluindo espaço para o \0

```
char *nome = "oscar";
```

⚠ O `\0` deve ser sempre considerado, pois indica o fim da string.

✗ O que **não** podemos fazer com strings em C

✗ Atribuição:

```
char nome[10];  
nome = "Alberto Caeiro"; // ✗ ERRO! Strings não podem ser atribuídas diretamente
```

✗ Comparação:

```
if (nome == "Alberto Caeiro") { // ✗ ERRO! Não se pode comparar strings com ==  
    puts("asneira suprema");  
}
```

✗ Retorno:

```
char * asneira(void) {  
    char s1[] = "think twice...";  
    return s1; // ✗ ERRO! Não se pode retornar vectores locais  
}
```

Operações com Strings (`<string.h>`)

Cópia de Strings

```
strcpy(nome, "Alberto");
```

Concatenação de Strings

```
strcat(nome, " Caeiro");
```

Comparação de Strings

```
if (strcmp(nome, "Alberto Caeiro") == 0)  
    puts("sou um guardador de rebanhos");
```

`strcpy` retorna 0 se as strings forem iguais

Comprimento de uma String

```
int n = strlen(nome);  
printf("A string tem %d caracteres", n);
```

 `strlen()` retorna apenas o número de caracteres **antes** do `\0`.

Implementação de `strcat`

```
char *strcat(char s1[], char s2[]) {  
    int ls1 = strlen(s1);  
    int ls2 = strlen(s2);  
  
    for (int i = ls1, j = 0; j <= ls2; i++, j++)  
        s1[i] = s2[j];  
  
    return s1;  
}
```

- ✓ Concatena `s2` ao final de `s1`.
- ✓ O `\0` da `s2` é copiado para `s1`.

Exercícios - Válido ou Inválido?

```
char palavra[100];  
palavra = "TRUE"; // ✗ Inválido! Use strcpy()  
  
if (palavra == "TRUE") // ✗ Inválido! Use strcmp()  
    puts("It is TRUE!");
```

Correção:

```
strcpy(palavra, "TRUE");  
  
if (strcmp(palavra, "TRUE") == 0)  
    puts("It is TRUE!");
```

`printf()` com Strings

```
printf("%s\n", var); // Imprime a string normalmente  
printf("%10s\n", var); // Alinha à direita com espaço mínimo de 10 caracteres  
printf("%-10s\n", var); // Alinha à esquerda  
puts(var); // Similar ao printf("%s\n", var)
```


✓ `puts()` sempre adiciona `\n` no final.

`scanf()` com Strings

```
scanf("%s", nome); // Lê até encontrar espaço ou \n
scanf("%5s", nome); // Lê até 5 caracteres

scanf("%[^\\n]s", nome); // Lê até \n

fgets(nome, 128, stdin); // Alternativa segura
```

 `scanf("%s", nome);` **não usa** `&`



Converter para Maiúsculas

```
#include <ctype.h>
void str_to_upper(char str[]) {
    for (int i = 0 ; str[i] != '\0' ; i++)
        str[i] = toupper(str[i]);
}

int main() {
    char fish[100] = "halibut";
    str_to_upper(fish);
    printf("%s\n", fish); // Imprime "HALIBUT"
}
```

✓ Usa `<ctype.h>` para manipulação de caracteres.

Implementação de `strcpy`

```
void strcpy(char dest[], char src[]) {  
    int i = 0;  
    do {  
        dest[i] = src[i];  
    } while (src[i++] != '\0');  
}
```

✓ Alternativa compacta:

```
void strcpy(char dest[], char src[]) {  
    int i = 0;  
    while ((dest[i] = src[i++]) != '\0');  
}
```

Biblioteca `<string.h>`

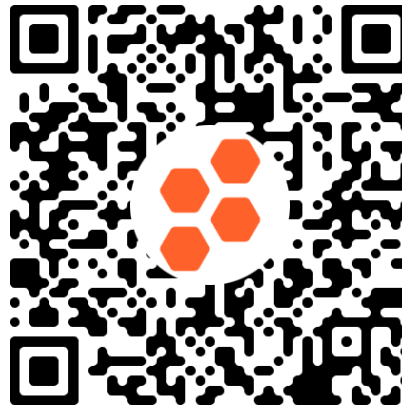
Funções úteis para Strings

```
strcpy(char *dest, char *src); // Copia strings
strcat(char *s1, char *s2);    // Concatena strings (resultado em s1)
strlen(char * s);              // Retorna o tamanho
strcmp(char * s1, char * s2);   // Compara strings
strcasecmp(char * s1, char * s2); // Compara strings ignorando o 'case'
strncmp(char * s1, char * s2, int n) // comparar apenas `n` caracteres!
_stricmp(char *dest, char *src) // compara strings – compiladores para Windows
```

Resumo

- ✓ Strings em C são vetores terminados em `\0`.
- ✓ Use `<string.h>` para manipular strings corretamente.
- ✓ **Não** compare strings com `==`, use `strcmp()`.
- ✓ `printf()` e `scanf()` têm formatos especiais para strings.
- ✓ **Cuidado com buffer overflow** ao lidar com strings!

? Quizz - Strings



No campo nome devem colocar o número de aluno 2XXXXXXX.

Exercício - O que será impresso?

```
#include <stdio.h>
#define MAX 64
void func(char s[], int n) {
    puts(s);
    s[n] = '\\0';
    puts(s);
}

int main(void) {
    char s[MAX] = "Use_the_force_Luke";
    func(s, 11);
    return 0;
}
```

- (A) Use_the_force_Luke\\n Use_the_for\\n
- (B) Use_the_force_Luke\\n Use_the_fo\\n
- (C) Use_the_force_Luke\\n Use_the_forc\\n
- (D) Use_the_force_Luke\\n rce_Luke\\n
- (E) Nenhuma das anteriores

Nota: a função `puts()` imprime a string recebida como parâmetro seguida de um `\\n`

Resposta

A saída é **b) Use_the_for** 

Explicação:

`s[11] = '\0';` corta a string após **Use_the_for**.

O resto da memória **ainda contém os caracteres antigos**, mas a string termina no **\0** !



Matrizes

Matrizes: Vetores de Vetores

Uma **matriz** é um **vetor multidimensional**, i.e. **vector de vetores**

São declaradas com **duas dimensões ou mais**.

Em C, declaramos assim:

```
int socos[2][3] = {  
    {100, 120, 110}, // Linha 0  
    {90, 130, 105}  // Linha 1  
};
```

 **Dica:** A primeira dimensão é **linhas**, a segunda é **colunas**.

Estrutura de Matrizes

Primeira dimensão → Número de linhas

Segunda dimensão → Número de colunas

```
tipo nome_matriz[num_linhas][num_colunas];
```

✓ Exemplo:

```
char Galo[3][3]; // Matriz 3x3
Galo[0][0] = 'X';
Galo[0][2] = '0';
Galo[1][1] = 'X';
Galo[2][2] = '0';
```

✓ **Galo[2][2]** armazena **'0'**.

Acesso a Elementos

Podemos **acessar e modificar** elementos da matriz:

```
char soup[5][5];  
soup[0][0] = 'e';  
soup[0][1] = 'e';  
soup[0][2] = 'u';  
soup[0][3] = 'l';  
soup[1][0] = 'u';
```

✓ Cada elemento é referenciado como `matriz[linha][coluna]`.


Erro Comum: Dimensão Inválida

 O seguinte código **não compila**:

```
int scores[3][] = {  
    {'1', '2', '3'},  
    {'4', '5', '6'},  
    {'7', '8', '9'}  
};
```

 ERRO: Deve especificar o número de colunas

Exercício: Encontra 4 erros no código abaixo?

 Objectivo é que o vector `acc` guarde em cada elemento a soma das linhas da matriz.

I.e. `acc[0] = scores[0][0] + scores[0][1] + scores[0][2]`, `acc[1] = scores[1][0] + scores[1][1] + scores[1][2]` ...

```
int main() {
    int n = 3, i, j;
    int scores[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int acc[n]; // Vector para armazenar a soma das linhas

    for (i = 0; i < 3; i++) {
        for (j = 0; j <= 3; j++) {
            acc[i] += scores[j][i];
        }
    }
    return 0;
}
```

Exercício: Encontra 4 erros no código abaixo?

 O código abaixo não funciona corretamente. Por quê?

```
int main() {
    int n = 3, i, j;
    int scores[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int acc[n]; // ✗ 2 ERROS: 1 - VLA, 2 - Falta inicializar

    for (i = 0; i < 3; i++) {
        for (j = 0; j <= 3; j++) { // ✗ ERRO: iteracao vai até 3 inclusive
            acc[i] += scores[j][i]; // ✗ ERRO: troca linhas por colunas
        }
    }
    return 0;
}
```

Exercício: Encontra 4 erros no código abaixo?

Correção

```
#define DIM 3

int main() {
    int i, j;
    int scores[][DIM] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int acc[DIM] = {0}; // inicializa todos os elementos a 0

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            acc[i] += scores[i][j];
        }
    }
    return 0;
}
```

Passagem de Matrizes para Funções

 **Dica:** TEMOS de especificar o número de colunas ao passar matrizes para funções! 

```
#define DIM 5
void inic(char s[][DIM]) {
    s[0][0] = 'e';
    s[0][1] = 'e';
    s[0][2] = 'u';
    s[0][3] = 'l';
}
```

```
int main(void) {
    char soup[DIM][DIM];
    inic(soup);
    return 0;
}
```

A função que recebe a Matriz, recebe-a por **referência**, e por isso ela pode alterar o conteúdo da matriz original.

Ou seja, no exemplo, o `main()` vai ficar com a matriz alterada pela função `inic()`

Retorno de Matrizes de Funções

 **Dica:** TEMOS de especificar o número de colunas ao passar matrizes para funções! 

✗ ERRO GRAVE: Não podemos retornar uma matriz local. Ela vai "desaparecer" da memória e a função que está a invocar não vai conseguir aceder à matriz.

```
#define DIM 3
double ** eye() {
    double eye[][DIM] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
    return eye;
}
```

✓ Solução: Receber a matriz por parâmetro e retornar a mesma matriz.

```
#define DIM 3
double ** eye(double eye[][DIM]) {
    eye[0][0] = eye[1][1] = eye[2][2] = 1;
    eye[0][1] = eye[0][2] = 0;
    eye[1][0] = eye[1][2] = 0;
    eye[2][0] = eye[2][1] = 0;
    return eye;
}
```


Percorrendo uma Matriz

✅ Para percorrer uma matriz, usamos dois loops aninhados:

💡 **Dica:** percorre linha a linha!

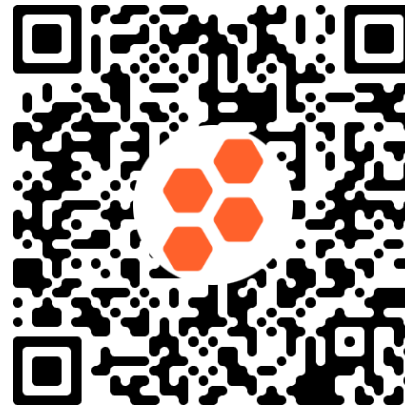
```
for (i = 0; i < 4; i++) {  
    for (j = 0; j < 4; j++) {  
        printf("%d ", matriz[i][j]);  
    }  
    printf("\n");  
}
```

- ◆ **Primeiro loop** percorre as linhas.
- ◆ **Segundo loop** percorre as colunas.

Resumo

- ✓ **Matrizes são vetores multidimensionais.**
- ✓ **Devemos especificar o número de colunas ao declarar uma matriz.**
- ✓ **Podemos inicializar uma matriz na declaração.**
- ✓ **Para percorrer uma matriz, usamos dois loops aninhados.**
- ✓ **Sempre defina o número de colunas ao passar uma matriz para uma função.**

? Quizz - Matrizes



No campo nome devem colocar o número de aluno 2XXXXXXX.

? Q&A

 **Dúvidas?**