Mini-Curso de C

Aula 05 - Vetores e Strings

Gustavo Henrique Aragão Silva

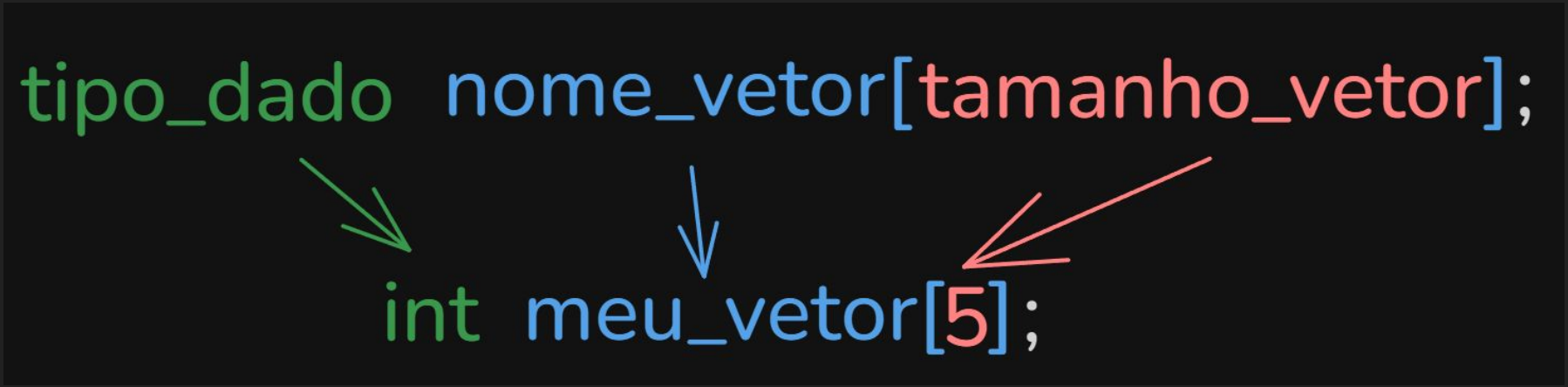
Agenda

- **Vetores**
  - Definição e Sintaxe
  - Acessando elementos de um vetor
  - Inicializando um vetor
  - Percorrendo vetores
  - Tamanho de um vetor
  - Exercícios propostos
- **Strings**
  - Conceito de String em C
  - Tabela ASCII
  - Declaração e Inicialização
  - Entrada e Saída de Strings
  - Funções da Biblioteca **<string.h>**
  - Exercícios propostos
- **Desafios** 🔥

Vetores

Vetores: Definição e Sintaxe

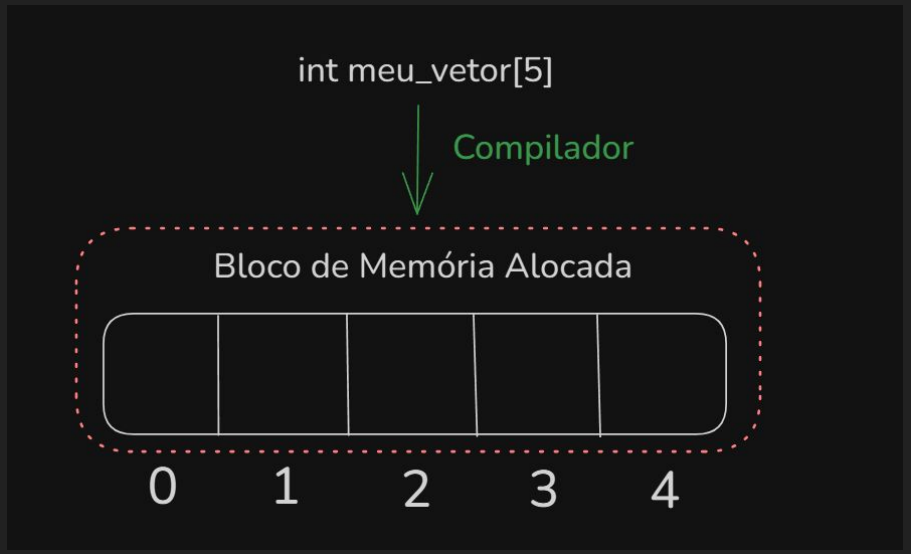
Um **vetor** ou **array** consiste em uma **estrutura de dados linear** que armazena uma sequência de elementos de **tamanho fixo**, todos do **mesmo tipo**, **em posições de memória contíguas**.



Vetores: Definição e Sintaxe

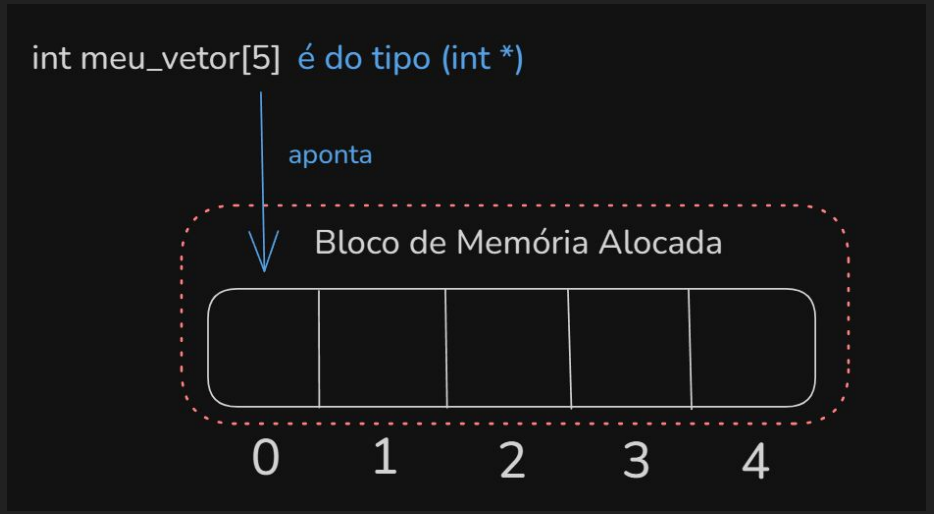
Quando declaramos um array em C, o **compilador aloca** um bloco de memória do tamanho **necessário** e **especificado** durante a declaração da variável.

Cada item de um vetor é **indexado** de **0** à **n - 1**, sendo n o tamanho do vetor.



Vetores: Definição e Sintaxe

A variável que acabamos de criar na realidade consiste em um **ponteiro** que aponta para o primeiro elemento do vetor.




```c
#include <stdio.h>

int main() {
    int meu_vetor[5];
    printf("%p\n", meu_vetor);
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
gustavo in ~/mini-curso-c-aula05
→  g++ exemplo.c && ./a.out
0x7ffc1d7a0a30
```

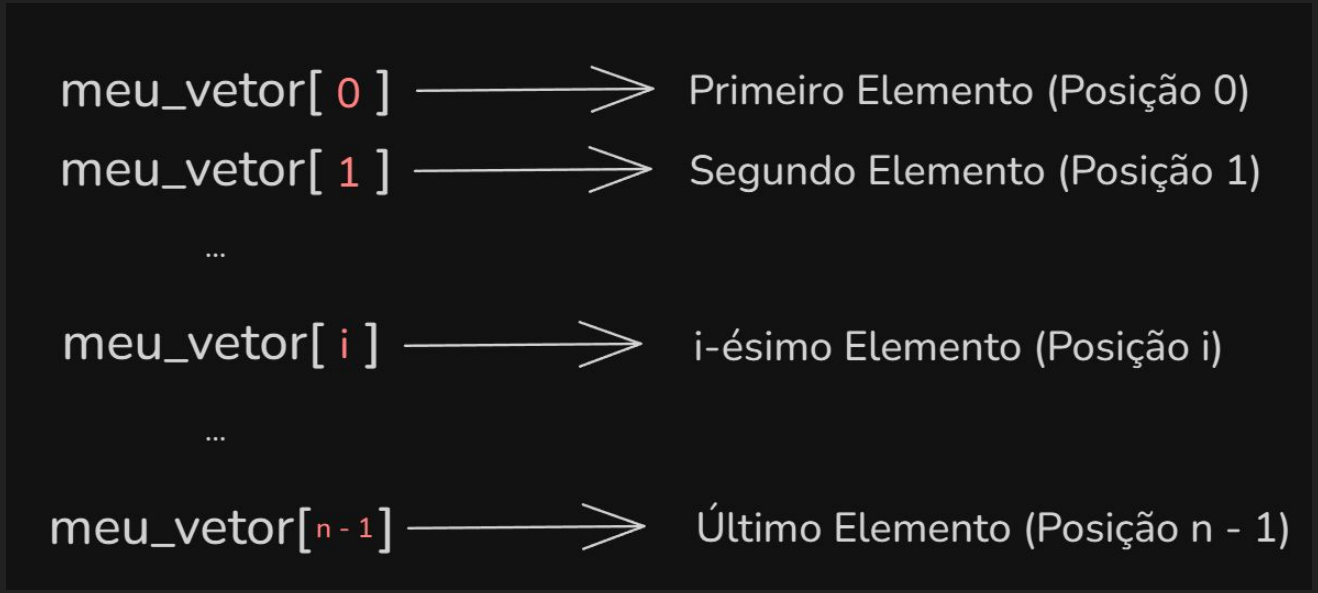
Vetores: Definição e Sintaxe (Prática)

Prática 01:

Crie um vetor e exiba o endereço que qual endereço foi alocado para essa variável.

Vetores: Acessando elementos de um vetor

Podemos acessar elementos de um vetor a partir da seguinte notação:



Também é possível fazer esse acesso a partir de **Aritmética de Ponteiros** 😎

Vetores: Acessando elementos de um vetor

Pergunta: Em um vetor de tamanho n, o que acontece se tentarmos acessar um elemento que está fora do intervalo [0, n - 1]?

Resposta Inocente: Vai “levantar” um **Index Out of Bounds**? **NÃO**

O programa vai tentar acessar uma **área de memória inválida** ou **não destinada ao vetor.**

Esse endereço acessado pode ser:

- ➔ De uma **outra variável do mesmo programa**.
- ➔ Uma **área protegida pelo sistema operacional** → Um **segmentation fault** pode ser levantado.

Vetores: Acessando elementos de um vetor

O valor lido ou escrito nessa posição será **imprevisível** (pode ser **lixo de memória**, **zero** ou até **sobrescrever dados importantes**).

```c
#include <stdio.h>

int main() {
    int meu_vetor[5];
    printf("%d ", meu_vetor[0]);
    printf("%d ", meu_vetor[1]);
    printf("%d ", meu_vetor[2]);
    printf("%d ", meu_vetor[3]);
    printf("%d ", meu_vetor[4]);
    printf("%d\n", meu_vetor[5]);
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
gustavo in ~/mini-curso-c-aula05
→  g++ exemplo.c && ./a.out
0 0 -1869587728 28946 840065552 32765
gustavo in ~/mini-curso-c-aula05
→  g++ exemplo.c && ./a.out
0 0 1974467312 30615 -1229745056 32764
```

Vetores: Acessando elementos de um vetor

Prática 02:

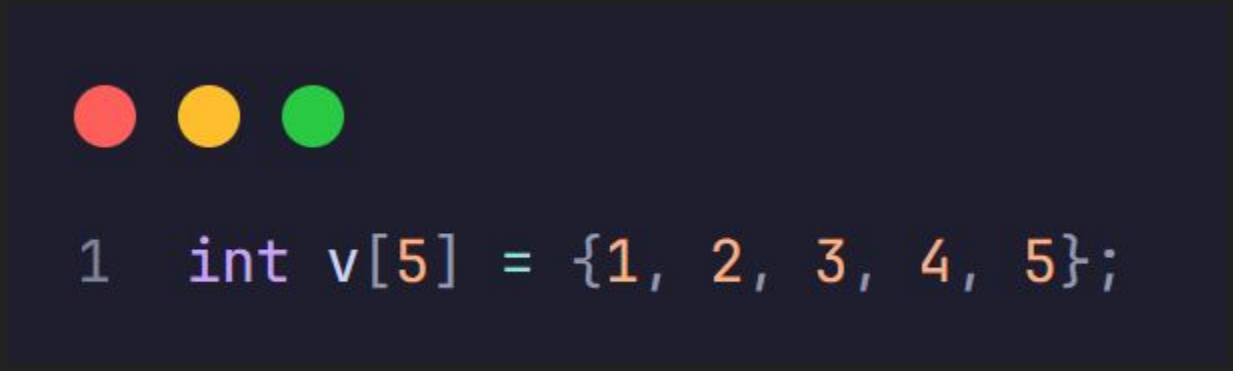
Como demonstrado no slide anterior, crie um vetor de tamanho 5 **sem inicializá-lo** e exiba os valores armazenados nas posições de 0 a 5 (inclusive).

Vetores: Inicializando um vetor

Como visto no experimento anterior, os elementos acessados do vetor tinham **valores aleatórios**.

Em C, há diversas formas de **inicializar** um vetor no instante de sua declaração, definindo os valores que ele armazenará.

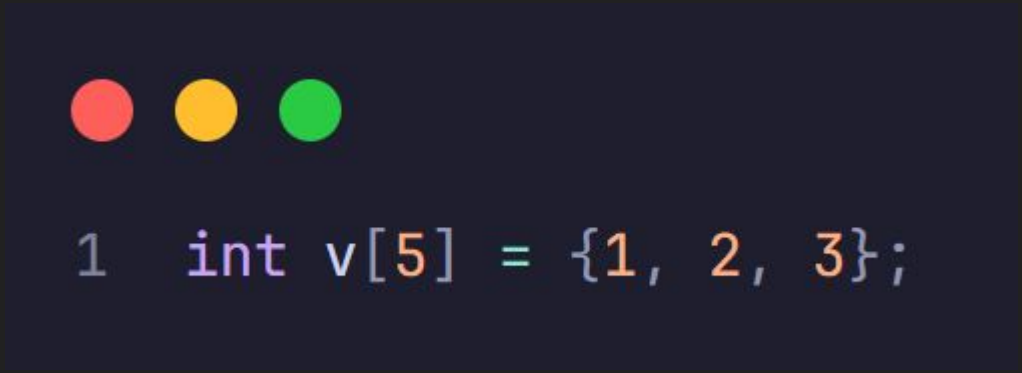
➔ **Inicialização Completa**



Vetores: Inicializando um vetor

➔ **Inicialização Parcial**

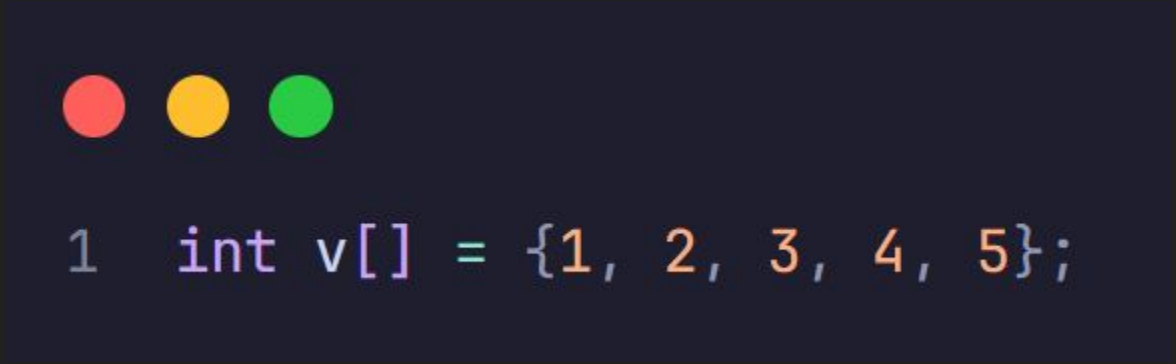
Se você fornecer menos valores do que o tamanho declarado, o compilador completa o restante com zeros



Vetores: Inicializando um vetor

➔ **Deixar o compilador calcular o tamanho**

Se você não especificar o tamanho, mas inicializar com valores, o compilador conta automaticamente.

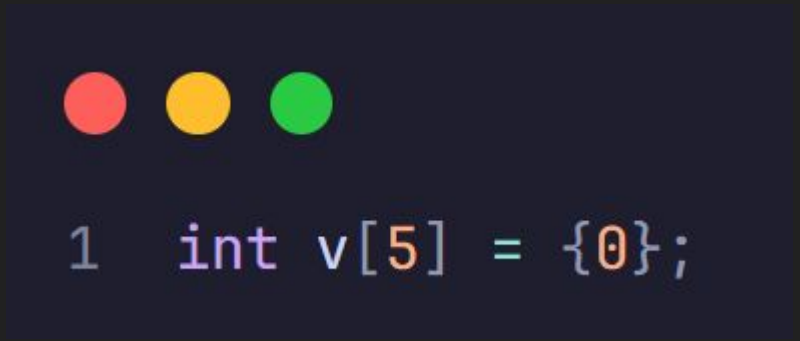


Vetores: Inicializando um vetor

➔ **Inicialização com atalho**

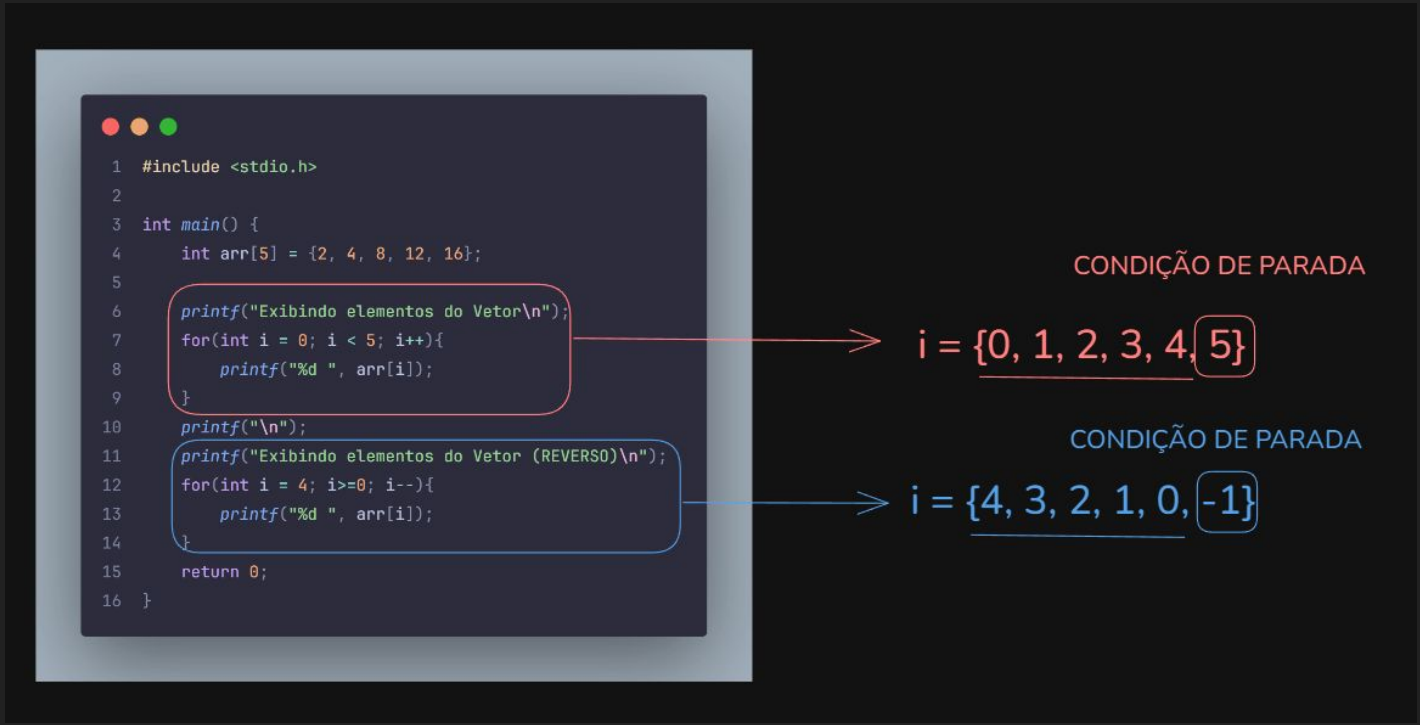
Caso você passe apenas {X}, todos os elementos do vetor será inicializar com o valor X.

Por exemplo, podemos inicializar um vetor com zeros rapidamente.



Vetores: Percorrendo um vetor

Para percorrer um vetor em C utilizamos o conceito de **loops (laços)**.



Vetores: Percorrendo um vetor

Note que ao criar nosso laço para percorrer podemos dividir o processo em **sub-tarefas**:

1. Enumerar uma **lista de índices** queremos acessar.

2. Identificar o índice **inicial** e **final**.

3. Ajustar a **condição de parada**.
   a. Lista de índices Crescente → **<** ou **<=**
   b. Lista de índices Decrescente → **>** ou **>=**

4. Definir o **Incremento/Decremento**.
   a. Lista de índices Crescente → **i++** ou **i += x**
   b. Lista de índices Decrescente → **i--** ou **i -= x**

```
for( inicialização ; condição de parada ; incremento decremento )
```

Vetores: Tamanho de um Vetor

O operador **sizeof()** em C é usada para descobrir **quantos bytes** um determinado tipo de dado ou variável ocupa na memória.

Esse operador é avaliado em **tempo de compilação**, ou seja, o valor é calculado antes mesmo do programa ser executado. **(Isso é um limitante!)**

O retorno é um **long int**, ou seja, é necessário usar a *flag* **%ld** no printf().

```
int tamanho_vetor = sizeof( meu_vetor ) / sizeof( tipo do dado )

ou

int tamanho_vetor = sizeof( meu_vetor ) / sizeof( meu_vetor[0] )
```

Vetores: Tamanho de um Vetor

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

    int tamanho = sizeof(arr) / sizeof(int);   // arr[0] também é válido no lugar de int
    printf("Tamanho do Vetor: %d\n", tamanho); // Tamanho do Vetor: 5

    printf("sizeof(arr) = %ld\n", sizeof(arr)); // sizeof(arr) = 20
    printf("sizeof(int) = %ld\n", sizeof(int)); // sizeof(arr) = 4

    return 0;
}
```

Vetores: Tamanho de um vetor

Prática 03:

Crie um vetor aleatório **sem especificar seu tamanho na declaração** e exiba o tamanho dele usando o **sizeof()**. Além disso, depois exiba a **quantidade de bytes** que a **variável do vetor** ocupa e o **tipo do vetor**.

Exercício 1: Operações com Vetor

Escreva um programa em C que realize as seguintes operações com um vetor de 5 números inteiros:

1. **Leia** os 5 valores inteiros digitados pelo usuário e armazene-os em um vetor.

2. **Exiba** todos os números digitados na mesma ordem em que foram inseridos.

3. Calcule e mostre a **soma e a média aritmética** dos elementos do vetor.

4. Solicite ao usuário que digite **um número x** e verifique se ele **existe** no vetor.
   a. Caso o número seja encontrado, informe a posição em que ele aparece.
   b. Caso contrário, exiba uma mensagem informando que o número não foi encontrado.

Solução 1: Operações com Vetor

```c
int main() {
    int arr[5];
    // Questão 1
    for (int i = 0; i < 5; i++) {
        printf("Digite arr[%d]: ", i);
        scanf("%d", &arr[i]);
    }
    // Questão 2
    printf("Exibir Vetor:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    // Questão 3
    long long int soma = 0LL;
    for (int i = 0; i < 5; i++) soma += arr[i];
    double media = (double) soma / 5;
    printf("Soma: %lld | Media: %.2lf\n", soma, media);
    // Questão 4
    int x, id = -1;
    printf("Digite o valor de x: ");
    scanf("%d", &x);
    for (int i = 0; i < 5; i++) {
        if (arr[i] == x) {
            id = i;
            break;
        }
    }
    if (id != -1) {
        printf("O valor %d esta na posicao %d\n", x, id);
    } else {
        printf("Valor não encontrado\n");
    }
    return 0;
}
```

Strings

Strings: Conceito de String em C

O termo ***string*** serve para identificar uma **sequência de caracteres**.

Na prática, as *strings* são usadas para representar **textos**, **palavras**, …

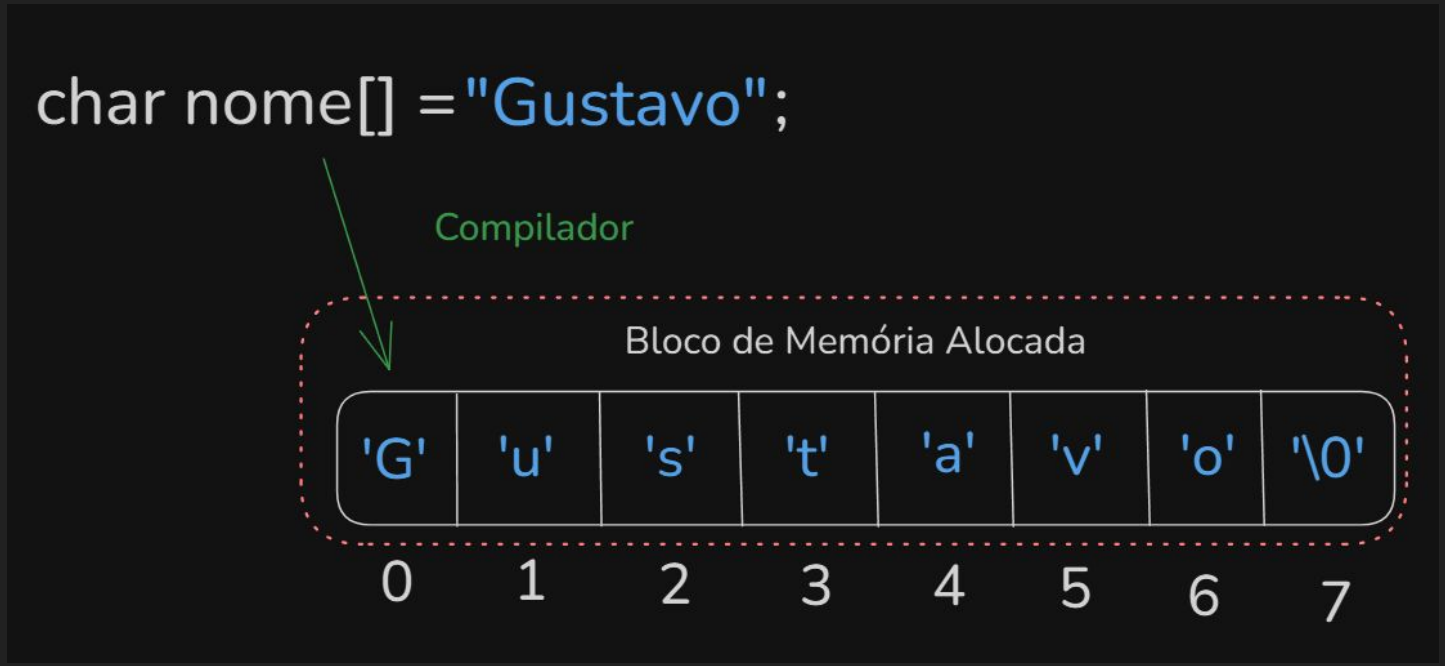
Em C, ao contrário de outras linguagens, **não** existe um tipo de dados nativo para representação uma *string*.

Para representar uma *string* em C, devemos criar um vetor de caracteres, ou seja, um vetor do tipo ***char***.

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

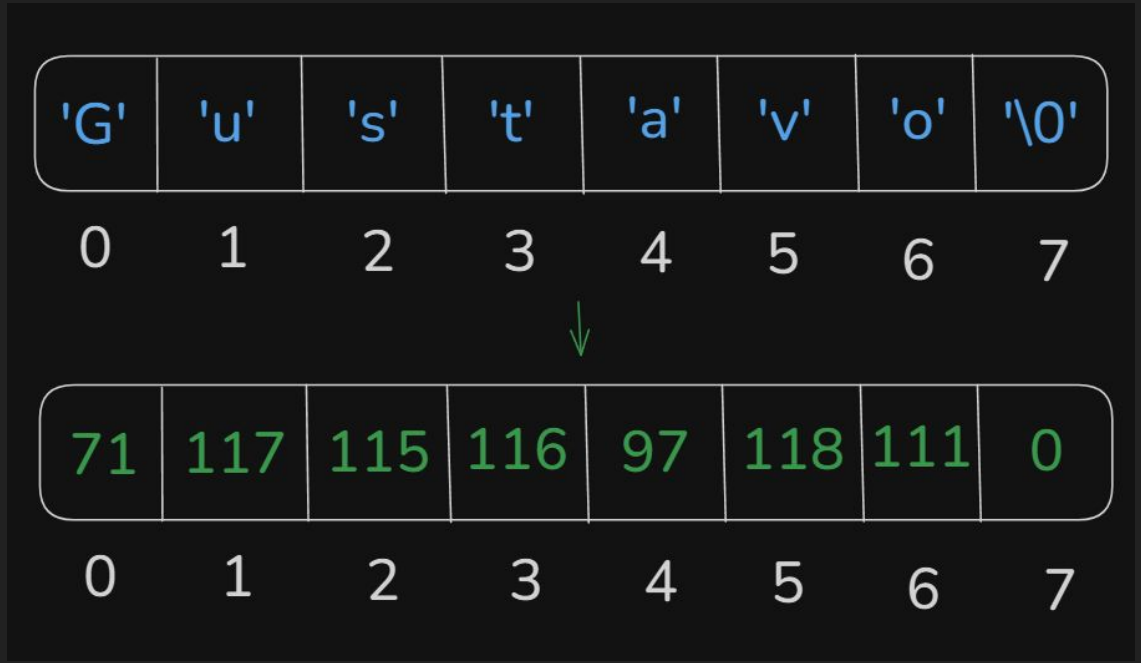
Strings: Conceito de String em C

Ocorre que o último caracter de uma string deve ser sempre o caracter nulo **‘\0’** que serve para indicar o final da *string*.



Strings: Conceito de String em C

Na memória do computador, **não** é armazenado o caractere em si, mas sim um **número** que o representa. Esse número é definido por padrões como a **tabela ASCII**, que associa cada símbolo a um valor numérico específico.



Strings: Tabela ASCII

ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Strings: Declaração e Inicialização

Declaração

A declaração de uma *string* é um simples **vetor unidimensional** do tipo ***char***.

```
char minha_string[tamanho_string]
```

Como nos vetores, também podemos omitir o tamanho na declaração.

```
char minha_string[ ]
```

Strings: Declaração e Inicialização

Inicialização

```c
#include <stdio.h>

int main() {
    // Forma 1: Usando string literal
    char nome1[30] = "gustavo";
    char nome2[] = "gustavo";
    // Forma 2: Usando lista de caracteres
    char nome3[30] = {'g', 'u', 's', 't', 'a', 'v', 'o'};
    return 0;
}
```

Strings: Entrada e Saída de Strings

Função *scanf*

A função *scanf* permite fazer leitura de *strings* usando **%s**.

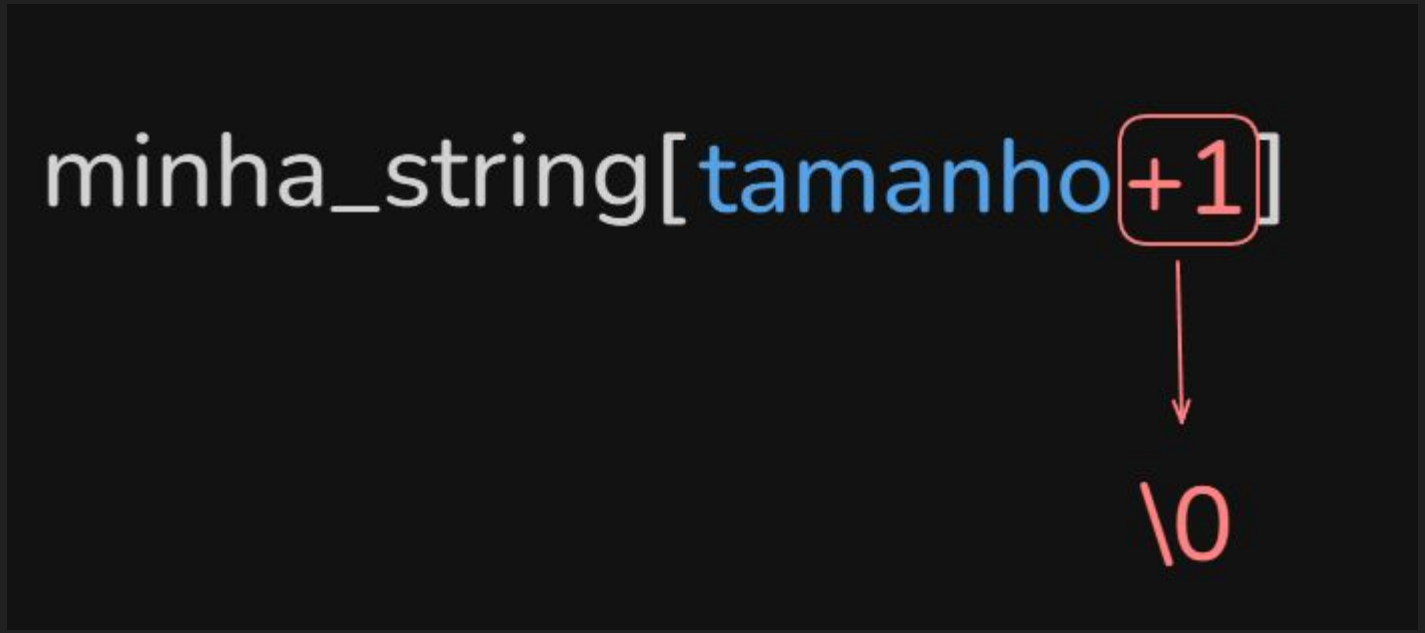
Em relação ao uso de *scanf* para armazenar string devemos observar duas coisas:

- A função scanf realiza a leitura **até encontrar um espaço**, depois encerra a leitura e coloca o caracter terminador **\0**.

- A variável que vai armazenar a string **não necessita ser precedida por &**.

Por conta do \0, sempre precisamos reservar mais do que o tamanho necessário.

Strings: Entrada e Saída de Strings



Ou seja, se o **tamanho máximo** de uma string é **100**, precisamos reserva um espaço de **100 + 1 = 101**.

Strings: Entrada e Saída de Strings

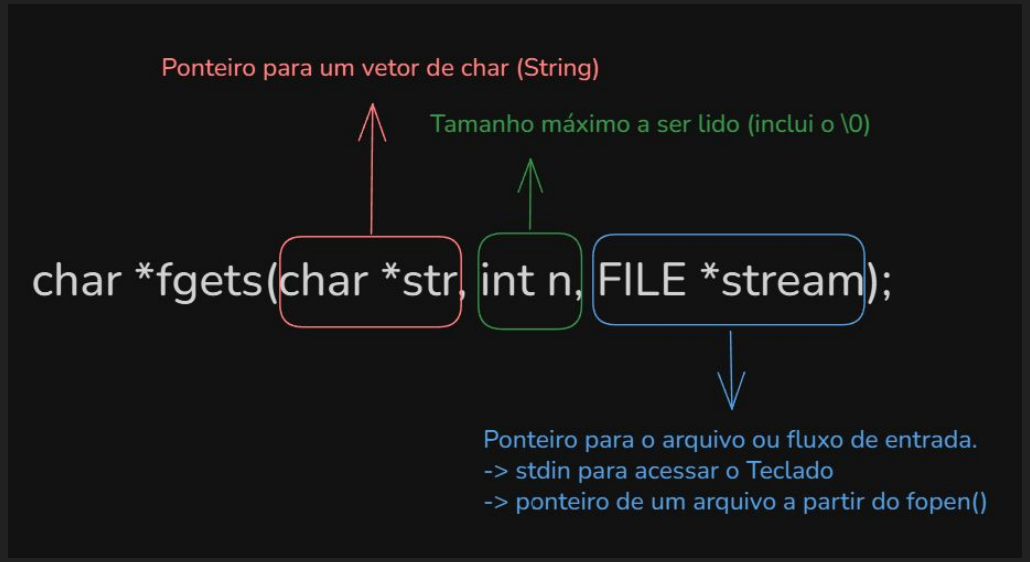
Função *scanf*

```c
#include <stdio.h>

int main() {
    char nome[100];
    printf("Digite o seu nome: ");
    scanf("%s", nome);

    printf("Oi, %s\n", nome);
    return 0;
}
```

Strings: Entrada e Saída de Strings

Função *fgets*

Esta função armazena tudo que foi digitado, inclusive os espaços, até que a tecla **ENTER** seja pressionada. Diferente do *scanf,* o *fgets* consegue ler a **linha inteira**.



Strings: Entrada e Saída de Strings

Função *fgets*

```c
#include <stdio.h>

int main() {
    char nome[100];
    printf("Digite o seu nome: ");
    fgets(nome, 100, stdin);

    printf("Oi, %s\n", nome);
    return 0;
}
```

Strings: Entrada e Saída de Strings

Função *printf*

```c
#include <stdio.h>

int main() {
    char texto[] = "A aula está muito massa!";
    printf("Uma verdade absoluta: %s\n", texto);
    return 0;
}
```

Strings: Funções da Biblioteca **<string.h>**

A biblioteca **<string.h>** da linguagem C, contém uma série de funções para manipular strings.

Nesta seção veremos como:

- **Copiar** strings usando **strcpy()**;
- **Concatenar** strings usando **strcat()**;
- **Descobrir** o tamanho de uma string **strlen()**;
- **Comparar** strings em C usando **strcmp()**;

Há diversas outras funções disponíveis na biblioteca… Fica a sugestão pesquisar mais sobre 🤓

Strings: Funções da Biblioteca **<string.h>**

Em outras linguagens, operações de **cópia/atribuição**, **concatenação** e **comparação** de strings são operações simples em outras linguagens.

Em C, elas **não** são possíveis de serem realizadas diretamente, ou seja, é necessário utilizar as funções mencionadas no *slide* anterior.

Errar isso é muito comum :(

Strings: Funções da Biblioteca **<string.h>**

```c
#include <stdio.h>

int main() {
    char nome[] = "Gustavo";
    char sobrenome[] = "Aragão";
    // Não podemos atribuir/copiar strings dessa forma
    nome = "Henrique";
    // Operador + não suporta concatenação
    char nome_concatenado[] = nome + sobrenome;
    // Os operadores >, <, >=, <= e == não suporta comparação de strings
    if (nome > sobrenome)
        printf("nome > sobrenome\n");
    else
        printf("nome <= sobrenome\n");
    return 0;
}
```

- **strcpy(string_destino, string_origem)**

Realiza a **cópia** do conteúdo da **string_origem** para a variável **string_destino**.

O parâmetro **string_origem** pode ser uma **outra variável** ou um **String Literal**.

```c
int main() {
    char nome[100] = "Gustavo";

    // String Literal
    nome = "Henrique";          // Inválido!
    strcpy(nome, "Henrique");   // Válido!

    // Outra variável
    char outro_nome[] = "Henrique";
    nome = outro_nome;          // Inválido!
    strcpy(nome, outro_nome);   // Válido!

    return 0;
}
```

- **strcat(string_destino, string_origem)**

Realiza a **concatenação** do conteúdo da **string_origem** com o conteúdo da variável **string_destino**.

O parâmetro **string_origem** pode ser uma **outra variável** ou um **String Literal**.

```c
int main() {
    char nome[100] = "Gustavo";
    char outro_nome[100] = "Henrique";

    // String Literal
    nome = nome + "Henrique";   // Inválido!
    strcat(nome, "Henrique");   // Válido!


    // // Outra variável
    nome = nome + outro_nome;   // Inválido!
    strcat(nome, outro_nome);   // Válido!

    return 0;
}
```

- **strcmp(string1, string2)**

Realiza a **comparação** do conteúdo de duas *strings*. Caso o retorno seja:

- **== 0** → O conteúdo das duas strings são **IGUAIS**.
- **< 0** → O conteúdo de **string1** é **MENOR LEXICOGRAFICAMENTE** que o de **string2**.
- **> 0** → O conteúdo de **string1** é **MAIOR LEXICOGRAFICAMENTE** que o de **string2**.

Os parâmetros **string1** e **string2** podem ser uma **variável** ou um **String Literal**.

⚠️ **Lexicograficamente** = **Ordem alfabética**, ou seja, como as palavras aparecem no **dicionário**.

```c
int main() {
    char nome[100] = "Gustavo";
    char outro_nome[100] = "Henrique";

    // Válido
    if (nome == outro_nome) {
        printf("nome == outro_nome\n");
    }
    // Inválido
    if (strcmp(nome, outro_nome)) {
        printf("nome == outro_nome\n");
    }

    return 0;
}
```

```c
int main() {
    char nome[100] = "Gustavo";
    char outro_nome[100] = "Henrique";

    // Literal
    if (strcmp(nome, "Henrique") == 0) {
        printf("nome == Henrique\n");
    } else if (strcmp(nome, "Henrique") < 0) {
        printf("nome < Henrique\n");
    } else if (strcmp(nome, "Henrique") > 0) {
        printf("nome > Henrique\n");
    }

    // Variável
    if (strcmp(nome, outro_nome) == 0) {
        printf("nome == outro_nome\n");
    } else if (strcmp(nome, outro_nome) < 0) {
        printf("nome < outro_nome\n");
    } else if (strcmp(nome, outro_nome) > 0) {
        printf("nome > outro_nome\n");
    }

    return 0;
}
```

- **strlen(string1)**

Descobre o **tamanho** do conteúdo da **string1**. A contagem é feita até encontrar o primeiro **\0**.

O parâmetro **string1** pode ser uma **variável** ou um **String Literal**.

```c
int main() {
    char nome[100] = "Gustavo";

    // Variável
    printf("%d\n", strlen(nome));
    // String Literal
    printf("%d\n", strlen("Gustavo"));

    return 0;
}
```

Exercício 2: Operações com String

Escreva um programa em C que realize as seguintes operações com *strings*:

1. Ler **o nome e o último nome** de uma pessoa (usando **fgets**).

2. Mostrar o **tamanho** do nome usando **strlen**.

3. **Concatenar** uma mensagem personalizada (ex: "Olá, <nome>!").

4. **Comparar** se o nome digitado é igual a **"Gustavo Aragão"**, mostrando o resultado da comparação.

Exercício 2: Operações com String

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

int main() {
    char nome[100];

    // Questão 1
    printf("Digite o nome e o último nome: ");
    fgets(nome, 100, stdin);
    nome[strcspn(nome, "\n")] = '\0';
    printf("Nome = %s\n", nome);
    // Questão 2
    int tamanho_nome = strlen(nome);
    printf("Tamanho = %d\n", tamanho_nome);
```

```c
    // Questão 3
    char mensagem[110] = "Olá, ";
    strcat(mensagem, nome);
    strcat(mensagem, "!");
    printf("Mensagem: %s\n", mensagem);
    // Questão 4
    if (strcmp(nome, "Gustavo Aragão") == 0) {
        printf("nome == 'Gustavo Aragão'\n");
    } else if (strcmp(nome, "Gustavo Aragão") < 0) {
        printf("nome < 'Gustavo Aragão'\n");
    } else if (strcmp(nome, "Gustavo Aragão") > 0) {
        printf("nome > 'Gustavo Aragão'\n");
    }
    return 0;
}
```

Desafios

Desafio 1: Contagem de Letras

Dada uma palavra composta apenas por letras minúsculas (de a até z), escreva um programa em C que conte quantas vezes cada letra aparece.

⚠️ **Não é permitido criar uma variável para cada letra. Você deve usar um vetor para contar as letras.**

- **Entrada**

Uma string (palavra) com até **100 caracteres**.

- **Saída**

Para cada letra que aparecer na palavra, imprimir no formato:

<letra>: quantidade

A saída deve estar em **ORDEM ALFABÉTICA**.

Desafio 1: Contagem de Letras

- **Exemplo:**
  - **Entrada:**

banana

  - **Saída:**

a: 3

b: 1

n: 2

Desafio 1: Contagem de Letras

LETRAS	Vetor de Contagem
'a'	cnt['a'] = 0
'b'	cnt['b'] = 0
'c'	cnt['c'] = 0
'd'	cnt['d'] = 0
...	...
'z'	cnt['z'] = 0

Desafio 1: Contagem de Letras

banana

cnt['a'] = 0

cnt['b'] = 0

cnt['n'] = 0

Desafio 1: Contagem de Letras

banana

cnt['a'] = 0

cnt['b'] = 1

cnt['n'] = 0

Desafio 1: Contagem de Letras

banana

cnt['a'] = 1

cnt['b'] = 1

cnt['n'] = 0

Desafio 1: Contagem de Letras

banana

cnt['a'] = 1

cnt['b'] = 1

cnt['n'] = 0

Desafio 1: Contagem de Letras

banana

cnt['a'] = 1

cnt['b'] = 1

cnt['n'] = 1

Desafio 1: Contagem de Letras

banana

cnt['a'] = 2

cnt['b'] = 1

cnt['n'] = 1

Desafio 1: Contagem de Letras

banana

cnt['a'] = 3

cnt['b'] = 1

cnt['n'] = 2

Desafio 1: Contagem de Letras

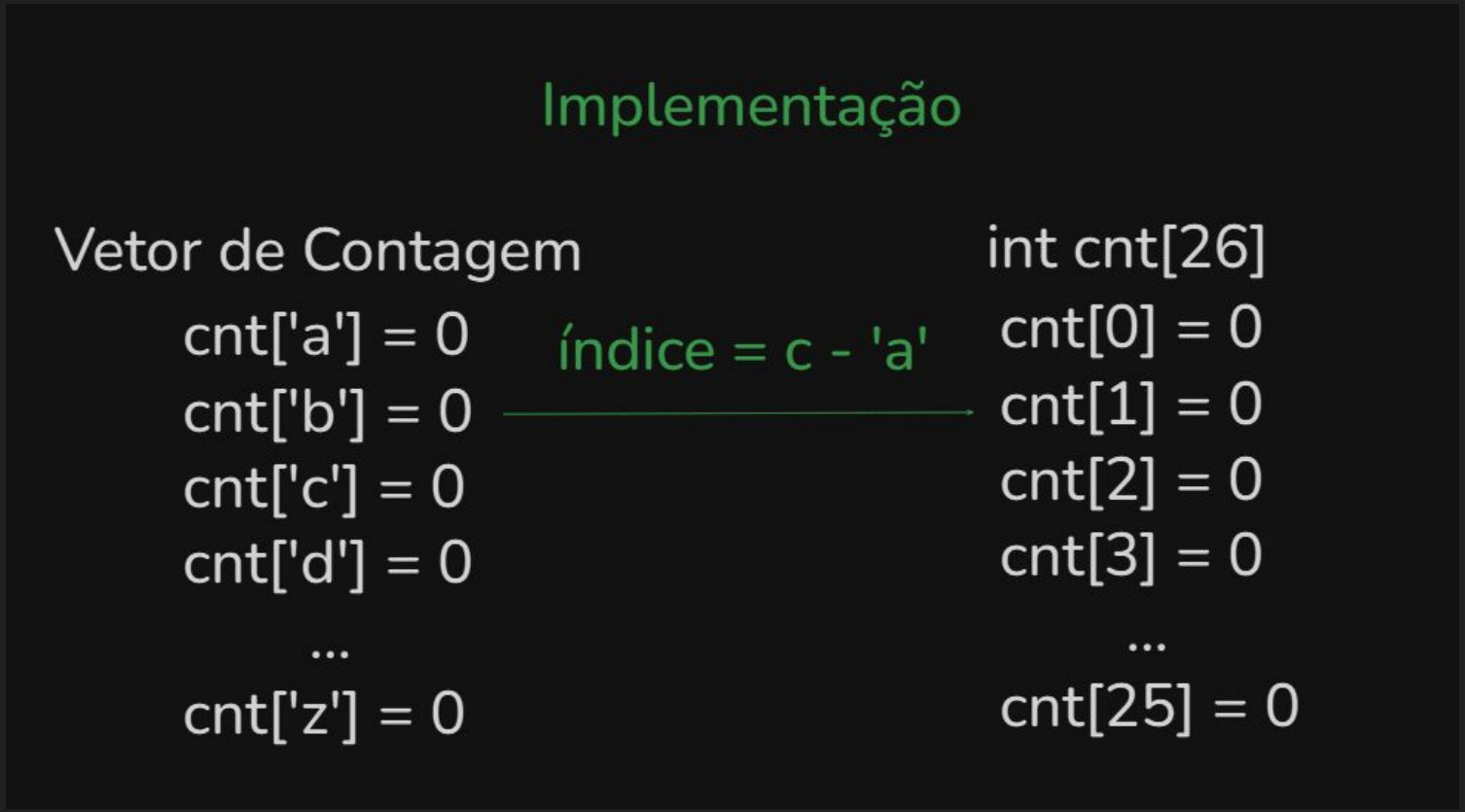
banana

cnt['a'] = 3

cnt['b'] = 1

cnt['n'] = 2

Desafio 1: Contagem de Letras



Desafio 2: Rotação de um Vetor

Dado um **vetor de n números inteiros** e **um valor k**, você deve deslocar os elementos para a direita k vezes, ou seja, uma rotação no sentido horário.

- **Entrada**

A primeira linha contém um inteiro n — tamanho do vetor (1 ≤ n ≤ 100).

A segunda linha contém n inteiros — os elementos do vetor.

A terceira linha contém um inteiro k — número de rotações.

- **Saída**

O vetor resultante após as rotações, apresentado no formato de lista: **[a1, a2, ..., an]**

Desafio 2: Rotação de um Vetor

- **Exemplo:**
  - **Entrada:**

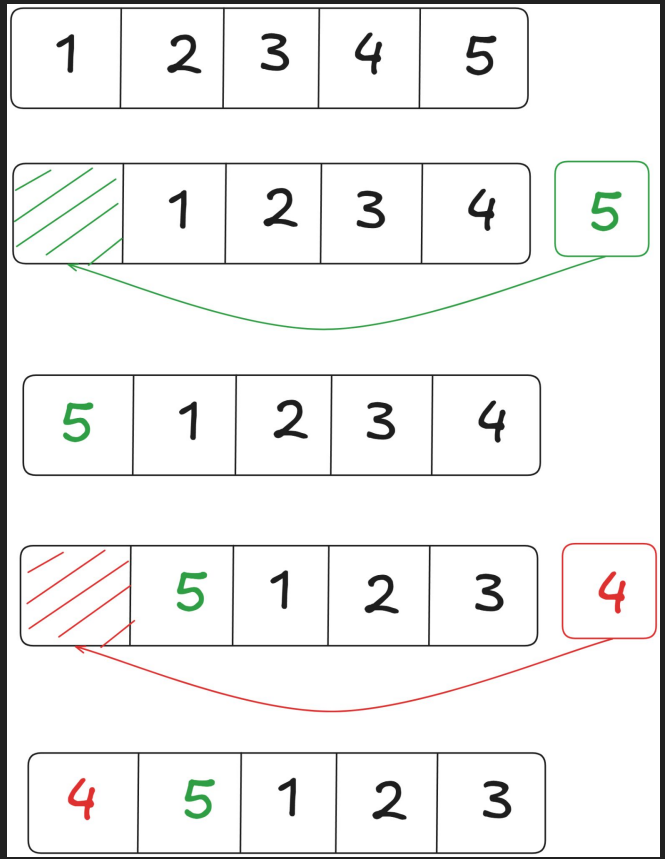
5

1 2 3 4 5

2

  - **Saída:**

[4, 5, 1, 2, 3]



Desafio 3: Rotação de um Vetor (Melhorado)

Melhorando o desafio…

Dado um vetor, determine **o número mínimo de rotações circulares** necessárias para que ele fique em ordem lexicográfica. Às vezes, não é possível fazer essa ordenação.

- **Entrada**

A primeira linha contém um inteiro n — tamanho do vetor (1 ≤ n ≤ 100).

A segunda linha contém n inteiros — os elementos do vetor.

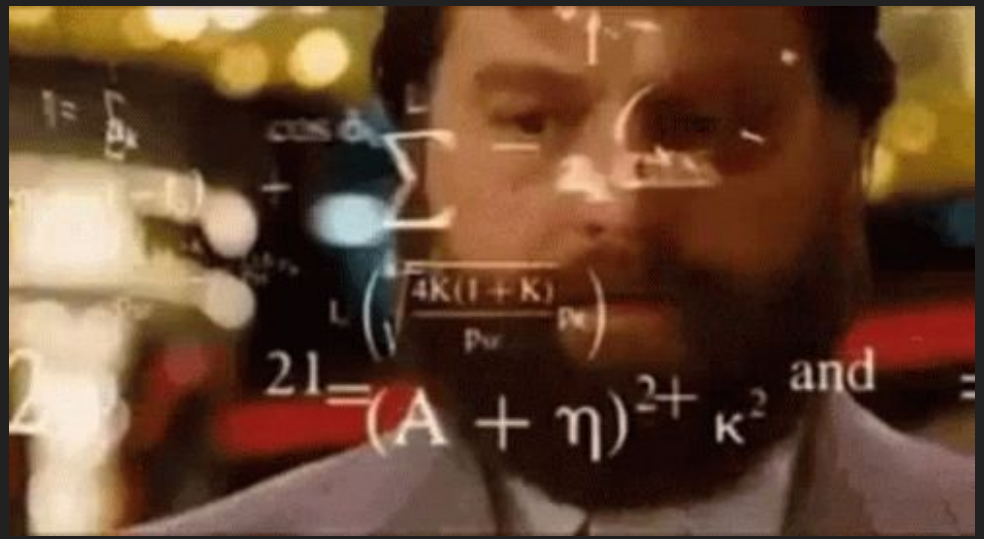
- **Saída**

Caso seja possível, retorne o número mínimo de rotações circulares. Caso contrário, o programa deve retornar **-1**.

Desafio 3: Rotação de um Vetor (Melhorado)

Desafio 3: Rotação de um Vetor (Melhorado)

Quando há solução?



Desafio 3: Rotação de um Vetor (Melhorado)

Condição 01:

O vetor deve ser **“quase” ordenado**, ou seja, ele deve ser **não decrescente**, **exceto por uma única quebra de ordem**.

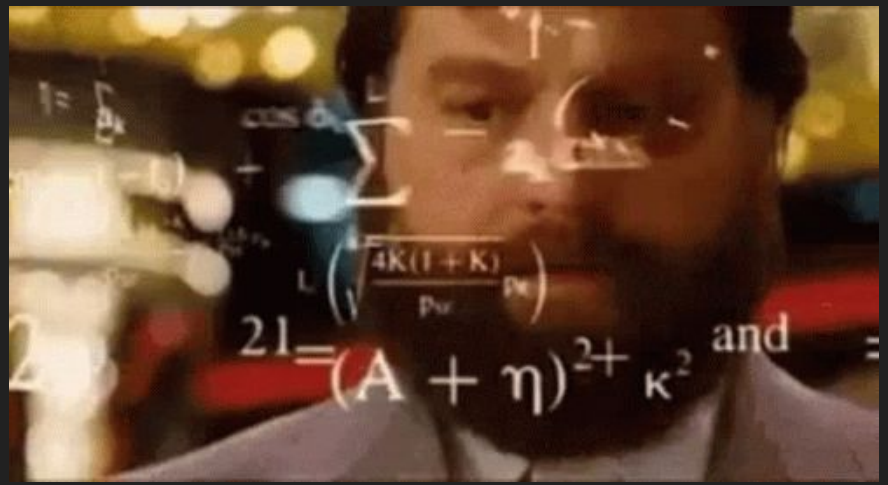
Desafio 3: Rotação de um Vetor (Melhorado)

Condição 02:

A rotação só gera um vetor ordenado se o primeiro elemento do **primeiro bloco (prefixo)** for **maior ou igual** ao último elemento do **segundo bloco (sufixo)**.

Desafio 3: Rotação de um Vetor (Melhorado)

Como calcular a quantidade de rotações?



Desafio 3: Rotação de um Vetor (Melhorado)

Basta calcular a **distância da quebra de ordem** para o **início do vetor**, no caso de rotações no **sentido anti-horário**, ou **final do vetor**, no caso de rotações no **sentido horário**.

Desafio 3: Rotação de um Vetor (Melhorado)

AGORA É COM VOCÊS!
“BASTA” IMPLEMENTAR :)

Dúvidas e Feedbacks?

Obrigado pela atenção