

Mini-Curso de C

Aula 05 - Vetores e Strings

Gustavo Henrique Aragão Silva



Quem sou eu?

- Estudante de **Ciência da Computação** (6º Período)
- **Monitor** de Programação Imperativa
- Pesquisador **PIBIC** com o **Prof. Dr. André Britto**
 - Tema: “Análise de Desempenho de MOEAs-Surrogates baseado no MOEA/D”
- Membro Efetivo na **Liga Acadêmica de Desenvolvimento Web (LAWD)**
- Viciado em **Programação Competitiva** 🎈



GitHub: @gustaaragao



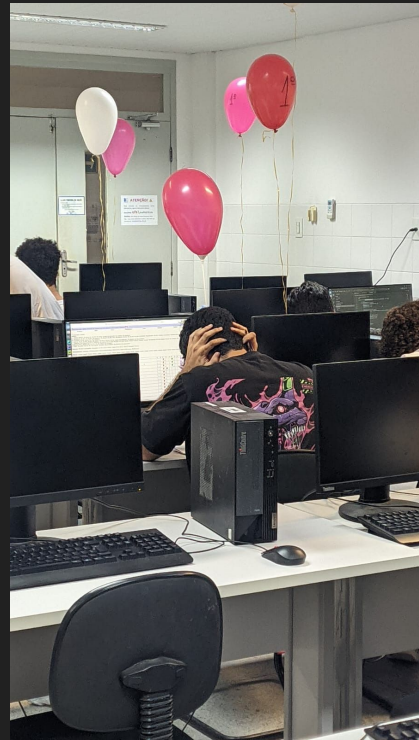
Linkedin: gustavoaragao



E-mail: gustavo.aragao@dcomp.ufs.br



Discord: gustaaragao



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**.

The diagram illustrates the syntax of a vector declaration. It shows a general form at the top and a specific example at the bottom, with arrows indicating the correspondence between the two.

```
tipo_dado nome_vetor[tamanho_vetor];
```

↓ ↓ ↓

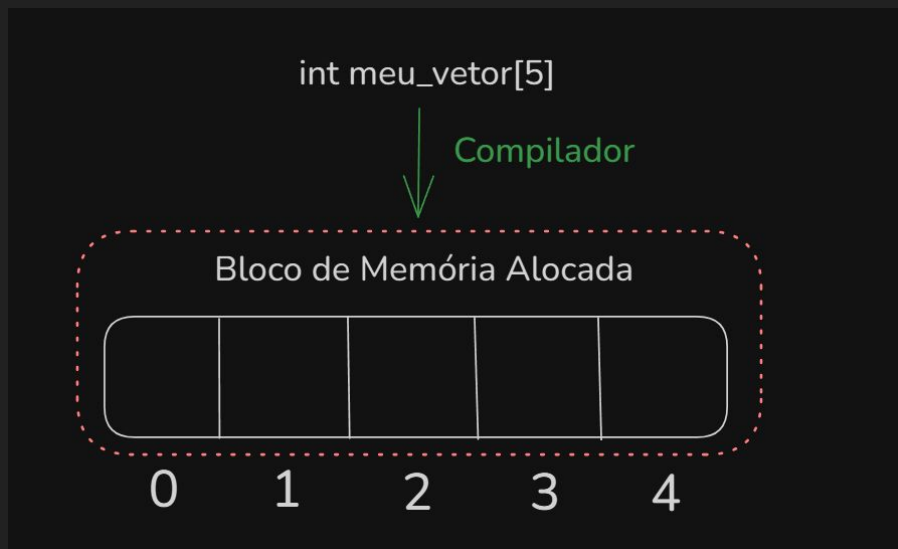
```
int meu_vetor[5];
```

The arrows show that 'tipo_dado' maps to 'int', 'nome_vetor' maps to 'meu_vetor', and 'tamanho_vetor' maps to '5'.

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.



```
1  #include <stdio.h>
2
3  int main() {
4      int meu_vetor[5];
5      printf("%p\n", meu_vetor);
6  }
7
```

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 foi reservado para essa variável.



Vetores: Acessando elementos de um vetor

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

<code>meu_vetor[0]</code>	→	Primeiro Elemento (Posição 0)
<code>meu_vetor[1]</code>	→	Segundo Elemento (Posição 1)
...		
<code>meu_vetor[i]</code>	→	i-ésimo Elemento (Posição i)
...		
<code>meu_vetor[n - 1]</code>	→	Último Elemento (Posição n - 1)

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 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**).

```
1  #include <stdio.h>
2
3  int main() {
4      int meu_vetor[5];
5      printf("%d ", meu_vetor[0]);
6      printf("%d ", meu_vetor[1]);
7      printf("%d ", meu_vetor[2]);
8      printf("%d ", meu_vetor[3]);
9      printf("%d ", meu_vetor[4]);
10     printf("%d\n", meu_vetor[5]);
11 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

gustavo in ~/mini-curso-c-aula05

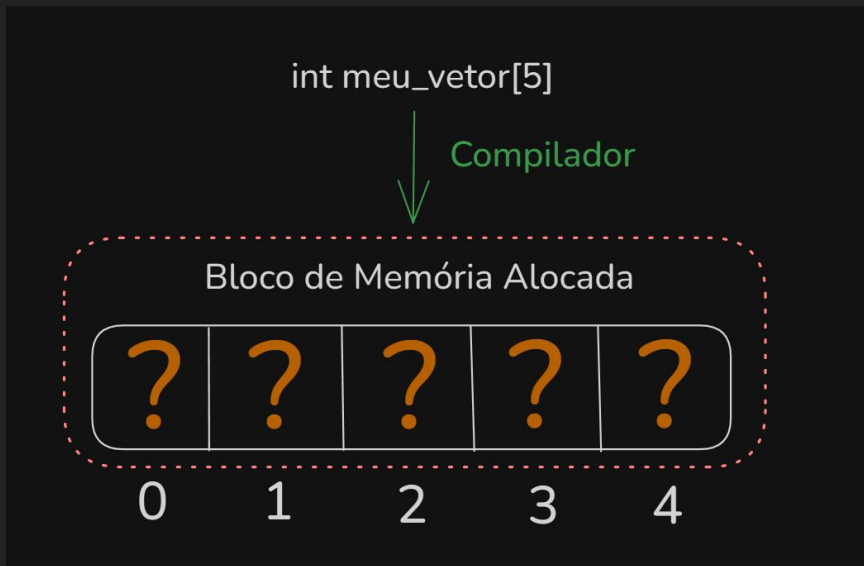
→ g++ exemplo.c && ./a.out

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



```
1  int v[5] = {1, 2, 3, 4, 5};
```

Vetores: Inicializando um vetor

→ Inicialização Parcial

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



```
1  int v[5] = {1, 2, 3};
```

Vetores: Inicializando um vetor

→ Deixar o compilador calcular o tamanho

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



```
1  int v[] = {1, 2, 3, 4, 5};
```

Vetores: Inicializando um vetor

→ Inicialização com atalho

Caso você passe apenas `{X}`, todos os elementos do vetor serão inicializados com o valor `X`.

Por exemplo, podemos inicializar um vetor com zeros rapidamente.



```
1  int v[5] = {0};
```

Vetores: Percorrendo um vetor

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

```
1 #include <stdio.h>
2
3 int main() {
4     int arr[5] = {2, 4, 8, 12, 16};
5
6     printf("Exibindo elementos do Vetor\n");
7     for(int i = 0; i < 5; i++){
8         printf("%d ", arr[i]);
9     }
10    printf("\n");
11    printf("Exibindo elementos do Vetor (REVERSO)\n");
12    for(int i = 4; i >= 0; i--){
13        printf("%d ", arr[i]);
14    }
15    return 0;
16 }
```

CONDIÇÃO DE PARADA

$i = \{0, 1, 2, 3, 4, \boxed{5}\}$

CONDIÇÃO DE PARADA

$i = \{4, 3, 2, 1, 0, \boxed{-1}\}$

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 $\rightarrow <$ ou \leq
 - b. Lista de índices Decrescente $\rightarrow >$ ou \geq
4. Definir o **Incremento/Decremento**.
 - a. Lista de índices Crescente $\rightarrow i++$ ou $i += x$
 - b. Lista de índices Decrescente $\rightarrow i--$ ou $i -= x$

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

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

```
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";
```

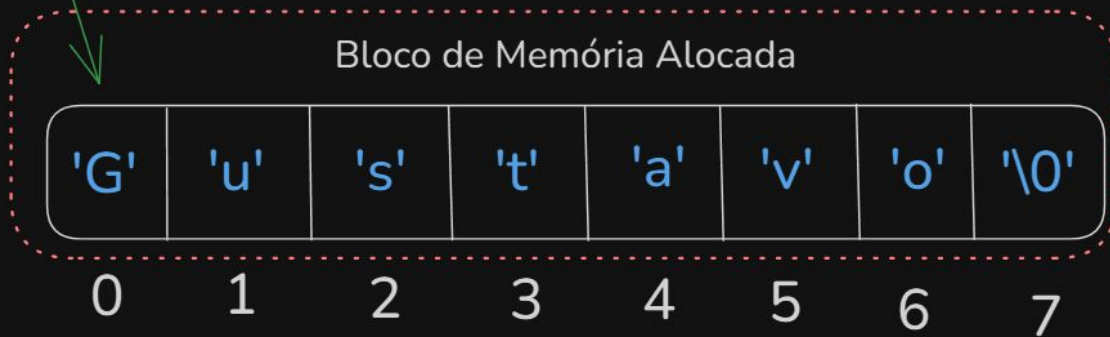
Strings: Conceito de String em C

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

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

Compilador

Bloco de Memória Alocada



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.

'G'	'u'	's'	't'	'a'	'v'	'o'	'\0'
0	1	2	3	4	5	6	7
↓							
71	117	115	116	97	118	111	0
0	1	2	3	4	5	6	7

Strings: Tabela ASCII

ASCII control characters			ASCII printable characters					
00	NULL	(Null character)	32	space	64	@	96	`
01	SOH	(Start of Header)	33	!	65	A	97	a
02	STX	(Start of Text)	34	"	66	B	98	b
03	ETX	(End of Text)	35	#	67	C	99	c
04	EOT	(End of Trans.)	36	\$	68	D	100	d
05	ENQ	(Enquiry)	37	%	69	E	101	e
06	ACK	(Acknowledgement)	38	&	70	F	102	f
07	BEL	(Bell)	39	'	71	G	103	g
08	BS	(Backspace)	40	(72	H	104	h
09	HT	(Horizontal Tab)	41)	73	I	105	i
10	LF	(Line feed)	42	*	74	J	106	j
11	VT	(Vertical Tab)	43	+	75	K	107	k
12	FF	(Form feed)	44	,	76	L	108	l
13	CR	(Carriage return)	45	-	77	M	109	m
14	SO	(Shift Out)	46	.	78	N	110	n
15	SI	(Shift In)	47	/	79	O	111	o
16	DLE	(Data link escape)	48	0	80	P	112	p
17	DC1	(Device control 1)	49	1	81	Q	113	q
18	DC2	(Device control 2)	50	2	82	R	114	r
19	DC3	(Device control 3)	51	3	83	S	115	s
20	DC4	(Device control 4)	52	4	84	T	116	t
21	NAK	(Negative acknowl.)	53	5	85	U	117	u
22	SYN	(Synchronous idle)	54	6	86	V	118	v
23	ETB	(End of trans. block)	55	7	87	W	119	w
24	CAN	(Cancel)	56	8	88	X	120	x
25	EM	(End of medium)	57	9	89	Y	121	y
26	SUB	(Substitute)	58	:	90	Z	122	z
27	ESC	(Escape)	59	;	91	[123	{
28	FS	(File separator)	60	<	92	\	124	
29	GS	(Group separator)	61	=	93]	125	}
30	RS	(Record separator)	62	>	94	^	126	~
31	US	(Unit separator)	63	?	95	_		
127	DEL	(Delete)						

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



```
1  #include <stdio.h>
2
3  int main() {
4      // Forma 1: Usando string literal
5      char nome1[30] = "gustavo";
6      char nome2[] = "gustavo";
7      // Forma 2: Usando lista de caracteres
8      char nome3[30] = {'g', 'u', 's', 't', 'a', 'v', 'o'};
9      return 0;
10 }
```

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



The diagram shows the code `minha_string[tamanho+1]` where `tamanho+1` is highlighted in blue and enclosed in a red rounded rectangle. A red arrow points from this rectangle down to the text `\0`, which is also in red. This illustrates that the character at the index immediately following the current string length is the null terminator.


```
minha_string[tamanho+1]
```

`\0`

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

Strings: Entrada e Saída de Strings

Função *scanf*



```
1  #include <stdio.h>
2
3  int main() {
4      char nome[100];
5      printf("Digite o seu nome: ");
6      scanf("%s", nome);
7
8      printf("Oi, %s\n", nome);
9      return 0;
10 }
```

Strings: Entrada e Saída de Strings

Função *printf*



```
1  #include <stdio.h>
2
3  int main() {
4      char texto[] = "A aula está muito massa!";
5      printf("Uma verdade absoluta: %s\n", texto);
6      return 0;
7  }
```

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>



```
1  #include <stdio.h>
2
3  int main() {
4      char nome[] = "Gustavo";
5      char sobrenome[] = "Aragão";
6      // Não podemos atribuir/copiar strings dessa forma
7      nome = "Henrique";
8      // Operador + não suporta concatenação
9      char nome_concatenado[] = nome + sobrenome;
10     // Os operadores >, <, >=, <= e == não suporta comparação de strings
11     if (nome > sobrenome)
12         printf("nome > sobrenome\n");
13     else
14         printf("nome <= sobrenome\n");
15     return 0;
16 }
```

- `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**.



```
1  int main() {
2      char nome[100] = "Gustavo";
3
4      // String Literal
5      nome = "Henrique";           // Inválido!
6      strcpy(nome, "Henrique");    // Válido!
7
8      // Outra variável
9      char outro_nome[] = "Henrique";
10     nome = outro_nome;           // Inválido!
11     strcpy(nome, outro_nome);    // Válido!
12
13     return 0;
14 }
```

- **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**.

```
1 int main() {
2     char nome[100] = "Gustavo";
3     char outro_nome[100] = "Henrique";
4
5     // String Literal
6     nome = nome + "Henrique"; // Inválido!
7     strcat(nome, "Henrique"); // Válido!
8
9
10    // // Outra variável
11    nome = nome + outro_nome; // Inválido!
12    strcat(nome, outro_nome); // Válido!
13
14    return 0;
15 }
```

- `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**.


```
1 int main() {
2     char nome[100] = "Gustavo";
3     char outro_nome[100] = "Henrique";
4
5     // Válido
6     if (nome == outro_nome) {
7         printf("nome == outro_nome\n");
8     }
9
10    // Inválido
11    if (strcmp(nome, outro_nome)) {
12        printf("nome == outro_nome\n");
13    }
14
15    return 0;
16 }
```

```
1 int main() {
2     char nome[100] = "Gustavo";
3     char outro_nome[100] = "Henrique";
4
5     // Literal
6     if (strcmp(nome, "Henrique") == 0) {
7         printf("nome == Henrique\n");
8     } else if (strcmp(nome, "Henrique") < 0) {
9         printf("nome < Henrique\n");
10    } else if (strcmp(nome, "Henrique") > 0) {
11        printf("nome > Henrique\n");
12    }
13
14    // Variável
15    if (strcmp(nome, outro_nome) == 0) {
16        printf("nome == outro_nome\n");
17    } else if (strcmp(nome, outro_nome) < 0) {
18        printf("nome < outro_nome\n");
19    } else if (strcmp(nome, outro_nome) > 0) {
20        printf("nome > outro_nome\n");
21    }
22
23    return 0;
24 }
```

- **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**.




```
1  int main() {
2      char nome[100] = "Gustavo";
3
4      // Variável
5      printf("%d\n", strlen(nome));
6      // String Literal
7      printf("%d\n", strlen("Gustavo"));
8
9      return 0;
10 }
```

Exercício 2: Operações com String

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

1. Ler o **primeiro nome** de uma pessoa (usando **scanf**).
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**", mostrando o resultado da comparação.

Exercício 2: Operações com String



```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char nome[100];
6
7      // Questão 1
8      printf("Digite o primeiro nome: ");
9      scanf("%s", nome);
10     printf("Nome = %s\n", nome);
11     // Questão 2
12     int tamanho_nome = strlen(nome);
13     printf("Tamanho = %d\n", tamanho_nome);
```



```
1      // Questão 3
2      char mensagem[110] = "Olá, ";
3      strcat(mensagem, nome);
4      strcat(mensagem, "!");
5      printf("Mensagem: %s\n", mensagem);
6      // Questão 4
7      if (strcmp(nome, "Gustavo") == 0) {
8          printf("nome == 'Gustavo'\n");
9      } else if (strcmp(nome, "Gustavo") < 0) {
10         printf("nome < 'Gustavo'\n");
11     } else if (strcmp(nome, "Gustavo") > 0) {
12         printf("nome > 'Gustavo'\n");
13     }
14     return 0;
15 }
```

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

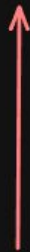
`cnt['a'] = 0`

`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

Implementação

Vetor de Contagem

cnt['a'] = 0

cnt['b'] = 0

cnt['c'] = 0

cnt['d'] = 0

...

cnt['z'] = 0

índice = c - 'a'

int cnt[26]

cnt[0] = 0

cnt[1] = 0

cnt[2] = 0

cnt[3] = 0

...

cnt[25] = 0

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 ($1 \leq n \leq 100$).

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:

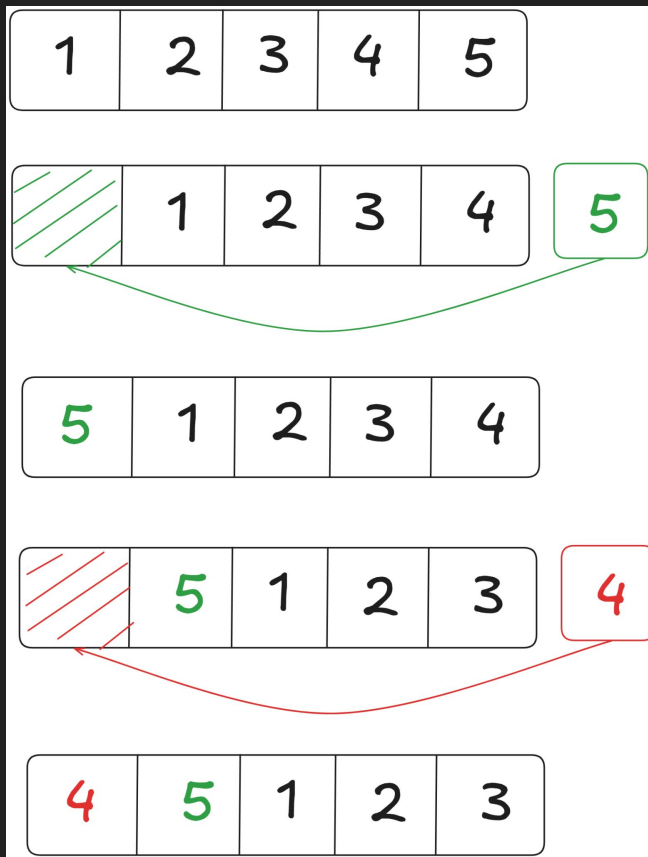
5

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 ($1 \leq n \leq 100$).

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)

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

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

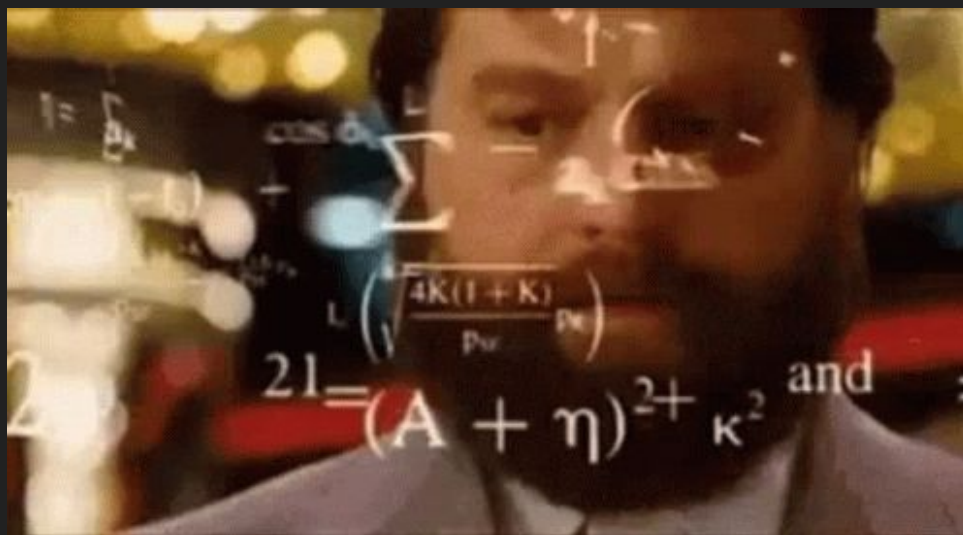
2	3	4	5	1
---	---	---	---	---

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

Resposta = 3

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)

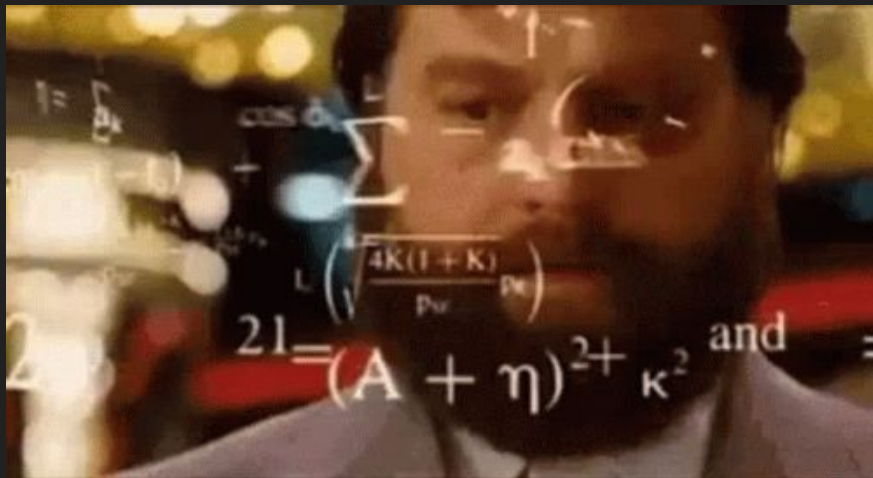
Condição 02:

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**.

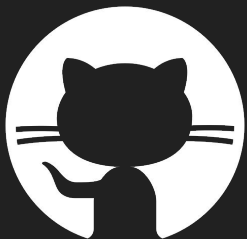


Desafio 3: Rotação de um Vetor (Melhorado)

AGORA É COM VOCÊS!
“BASTA” IMPLEMENTAR :)



Solução dos Desafios



[Link do Repositório
com as Soluções](#)



[Classroom do
Mini-Curso](#)

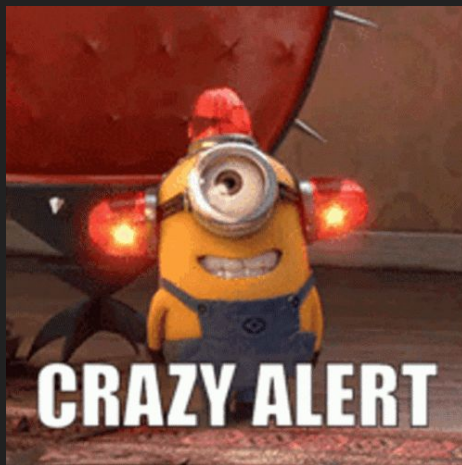


Dúvidas no Discord!

Vamos aprender juntos!



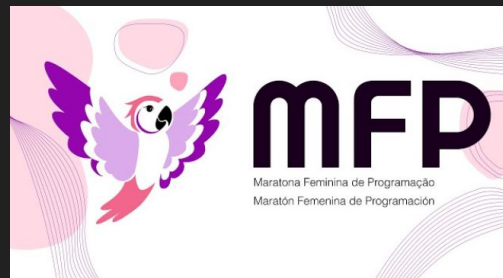
Uma mensagem do **Grupo de Programação Competitiva** 🎈



**Gostou dos
Desafios?**

E quer mais?

we need you



E outros...



Temos encontros **TODAS** as **SEXTAS-FEIRAS** das **15:00 – 20:30** no
Laboratório C03
(No CCET, ao lado da LADATA)



Grupo programação competitiva

Grupo do WhatsApp



Leia o QR code com a câmera do
WhatsApp para entrar neste grupo

Ou pelo LINK

**Dúvidas e
Feedbacks?**

Obrigado pela atenção

