

5954001 - Introdução à Computação I (Optativa para Química)

Strings

Prof. Dr. Denis M. L. Martins

DCM | FFCLRP | USP



Objetivos de aprendizagem

Ao final da aula, você deverá ser capaz de:

- Explicar a representação de strings em memória e o papel de `\0`.
- Diferenciar **capacidade do array**, **comprimento da string** e **bytes utilizados**.
- Ler, percorrer, comparar, copiar e concatenar strings com segurança.
- Relacionar arrays de `char` e ponteiros; distinguir texto de bytes.
- Implementar algoritmos de strings e testar seus casos-limite.

O que é uma string em C?

- Uma **string C** é uma sequência de **char** terminada pelo primeiro byte de valor **zero**, **\0**.
- Não existe um tipo fundamental **string** na linguagem C.
- A string pode estar armazenada em um array de **char** ou ser acessada por um ponteiro para **char**.
- Funções como **printf("%s", ...)** e **strlen()** dependem do terminador **\0**.

```
char nome[] = "Joao"; /* J o a o \0 */
```

Sem **\0** acessível nos limites válidos, funções que procuram o terminador podem ler memória indevida:
comportamento indefinido.

Modelo de memória: cinco bytes

```
char palavra[] = "Casa";
```

Índice	0	1	2	3	4
palavra	'C'	'a'	's'	'a'	'\0'

- Comprimento do texto: **4** bytes antes de **\0**.
- Tamanho do array: **5** bytes.
- O terminador participa do armazenamento, mas **não** da contagem de **strlen()**.

Declaração e inicialização

```
char a[] = "Carlos";           // 7 bytes
char b[10] = "Carlos";         // 10 bytes
char c[] = {'C', 'a', 'r', 'l', 'o', 's', '\\0'}; // 7 bytes
char d[7] = "Carlos";          // 7 bytes
```

- Em arrays com capacidade excedente e inicializador, os bytes restantes são inicializados com zero.
- `char e[6] = "Carlos";` ocupa 6 bytes, **mas não inclui `\\0`**: não forma uma string C válida.
- **Regra geral:** para armazenar **`n`** bytes de texto, reserve pelo menos **`n + 1`** bytes.

sizeof × **strlen**: duas perguntas diferentes

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

int main(void) {
    char texto[20] = "USP";
    printf("sizeof = %zu\n", sizeof texto);    // 20
    printf("strlen = %zu\n", strlen(texto));  // 3
    return 0;
}
```

- **sizeof texto** é o tamanho **do array em bytes** neste escopo.
- **strlen(texto)** conta bytes **até \0**, sem incluí-lo.
- O retorno de **strlen** é **size_t**: utilize **%zu** ao imprimir.

Capacidade não é comprimento

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

- **Capacidade:** 20 bytes (`sizeof nome`).
- **Comprimento:** 3 bytes (`strlen(nome)`).
- **Ocupação:** 4 bytes, incluindo `\0`.
- **Espaço disponível após a string:** 16 bytes.

```
'A' 'n' 'a' '\0' 0 0 0 ... 0  
<--- 4 -----> <--- 16 ---->
```

A quantidade de bytes livres pode ser relevante para concatenação; o array ainda tem capacidade fixa de 20 bytes.

String vazia e arrays não terminados

```
char vazia[8] = "";           // strlen(vazia) == 0
char ok[3] = {'0', 'K'};      // último elemento vira \0
char invalida[2] = {'0', 'K'}; // NÃO contém \0
```

- **ok** é uma string válida porque a inicialização parcial zera o terceiro byte.
- **invalida** é um array de caracteres, mas **não** uma string C válida.
- **Não execute** `strlen(invalida)` ou `printf("%s", invalida)`: podem acessar bytes fora do array.

Imprimindo strings e caracteres

```
#include <stdio.h>

int main(void) {
    char nome[] = "Ana";
    printf("Nome: %s\n", nome);
    printf("Primeira letra: %c\n", nome[0]);
    return 0;
}
```

- **%s** espera endereço de uma string corretamente terminada.
- **%c** imprime **um caractere** a partir de um valor inteiro.
- **puts(nome)** também imprime a string e acrescenta uma quebra de linha.

Leitura com `scanf`: limites e espaços

```
char nome[20];  
if (scanf("%19s", nome) == 1) {  
    printf("%s\n", nome);  
}
```

- `%s` para no primeiro caractere de espaço em branco.
- A largura `19` limita a leitura a 19 bytes de texto; o 20º byte recebe `\0`.
- Sem largura (`%s`), uma entrada longa pode **ultrapassar os limites** do array.
- `scanf` não é a melhor opção para ler nomes completos e frases.

fgets(): leitura de linhas

```
#include <stdio.h>

int main(void) {
    char nome[50];
    if (fgets(nome, sizeof nome, stdin) != NULL) {
        printf("Lido: %s", nome);
    }
    return 0;
}
```

- Lê espaços e, no máximo, **sizeof nome - 1** bytes.
- Quando cabe, preserva o **\n** que encerra a linha.
- Verifique o retorno: **NULL** sinaliza falha ou fim de arquivo sem bytes lidos.
- Se a linha ultrapassar a capacidade, **o restante permanece na entrada**.

Removendo a quebra de linha

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

int main(void) {
    char nome[50];
    if (fgets(nome, sizeof nome, stdin) != NULL) {
        nome[strcspn(nome, "\n")] = '\0';
        printf("Nome: [%s]\n", nome);
    }
    return 0;
}
```

- `strcspn(nome, "\n")` encontra o índice do primeiro `\n` ou o comprimento da string.
- Se não houver `\n`, a atribuição reescreve o próprio terminador `\0`.

Atenção: ausência de `\n` pode indicar **linha truncada**, não apenas fim de arquivo.

Quando a linha é maior que o buffer

```
char linha[8];
if (fgets(linha, sizeof linha, stdin) != NULL) {
    size_t n = strcspn(linha, "\n");
    if (linha[n] == '\n') {
        linha[n] = '\0';
    } else if (!feof(stdin)) {
        /* Pode haver caracteres pendentes na entrada. */
        /* Tratar truncamento explicitamente na aplicação. */
    }
}
```

- Não presuma que **fgets** sempre lê uma linha inteira.
- Uma solução completa pode verificar se há mais bytes e descartar o restante da linha.
- Para atividades iniciais, **defina a capacidade máxima de entrada.**

Percorrendo: **for**

```
#include <stdio.h>

int main(void) {
    char texto[] = "Radiacao";
    for (size_t i = 0; texto[i] != '\0'; i++) {
        printf("%c\n", texto[i]);
    }
    return 0;
}
```

- A condição **texto[i] != '\0'** define o ponto de parada.
- O índice começa em zero.
- O exemplo usa apenas ASCII, para que cada byte corresponda a um caractere.

Percorrendo: **while** e contagem manual

```
#include <stdio.h>

int main(void) {
    char texto[] = "Tomografia";
    size_t i = 0;
    while (texto[i] != '\0') {
        i++;
    }
    printf("Comprimento: %zu\n", i); // 10
    return 0;
}
```

A função **strlen()** executa conceitualmente uma busca pelo primeiro terminador; não confunda comprimento com capacidade.

Exercícios

Crie um programa que leia uma frase do teclado e:

- **Problema 1:** calcula e exibe a quantidade de caracteres na frase.
- **Problema 2:** conte quantas vogais existem na frase.
- **Problema 3:** Exibe a frase ao contrário (do final para o início).

<string.h>: visão geral

Função	Finalidade
<code>strlen(s)</code>	Comprimento até <code>\0</code>
<code>strcmp(a,b)</code>	Comparação lexicográfica
<code>strcpy(dest,orig)</code>	Copiar incluindo <code>\0</code>
<code>strcat(dest,orig)</code>	Concatenar incluindo <code>\0</code>
<code>strchr(s,c)</code>	Encontrar um caractere
<code>strstr(s,sub)</code>	Encontrar uma substring
<code>strcspn(s,rej)</code>	Comprimento até o primeiro byte rejeitado

As funções não conhecem, automaticamente, o tamanho de seu buffer de destino.

strlen(): contrato e complexidade

```
char texto[] = "Tomografia";  
printf("%zu\n", strlen(texto)); // 10
```

- Retorna a quantidade de bytes anteriores ao primeiro `\0`.
- Recebe uma string C válida.
- Percorre até achar o terminador: custo **$O(n)$** no comprimento.
- Em UTF-8, quantidade de bytes pode diferir da quantidade de caracteres percebidos.

strcmp(): três possibilidades

```
strcmp("Ana", "Ana") == 0;    // iguais  
strcmp("Ana", "Bia") < 0;    // ordem anterior  
strcmp("Bia", "Ana") > 0;    // ordem posterior
```

- O resultado **não precisa ser exatamente -1 ou 1**.
- Compara sequências de **bytes**; não é ordenação linguística sensível a acentos.
- Para verificar igualdade, use **strcmp(a, b) == 0**.
- **a == b** compara endereços (após conversões usuais), **não** o conteúdo textual.

Implementando `meu_strcmp()`

```
int meu_strcmp(const char a[], const char b[]) {  
    size_t i = 0;  
    while (a[i] != '\0' && a[i] == b[i]) {  
        i++;  
    }  
    unsigned char ca = (unsigned char)a[i];  
    unsigned char cb = (unsigned char)b[i];  
    return (ca > cb) - (ca < cb);  
}
```

- Se todos os bytes coincidirem até os terminadores, retorna zero.
- Na primeira diferença, compara **valores de `unsigned char`**.
- O contrato garante o **sinal**, não um valor absoluto específico.

strcpy(): cópia e risco

```
char origem[] = "MRI";  
char destino[20];  
strcpy(destino, origem);  
printf("%s\n", destino); // MRI
```

- Copia todos os bytes, **incluindo** `\0`.
- Requer **`capacidade(destino) >= strlen(origem) + 1`**.
- Se o destino for pequeno, ocorre escrita fora do array: **comportamento indefinido**.
- Fonte e destino não podem ter sobreposição de memória para **`strcpy`**.

strcat(): concatenar exige espaço

```
char destino[20] = "USP";  
char origem[] = "Ribeirao";  
  
if (strlen(destino) + strlen(origem) + 1 <= sizeof destino) {  
    strcat(destino, origem);  
}
```

- Espaço requerido: **strlen(destino) + strlen(origem) + 1**.
- Inclua o byte de **\0** na conta.
- Esse teste utiliza **sizeof destino** porque **destino é array neste escopo**.
- Em uma função que recebe **char destino[]**, é necessário passar a capacidade separadamente.

Erros frequentes e comportamento indefinido

```
char nome[5];  
// scanf("%s", nome); // sem limite: inseguro  
  
char s[2] = {'0', 'K'};  
// printf("%s", s); // sem terminador: inválido  
  
char dest[4] = "ABC";  
// strcat(dest, "D"); // não cabe o D e o \0
```

- **Buffer overflow:** leitura/escrita fora da capacidade.
- **String sem terminador:** leitura potencialmente fora dos limites.
- **Literal modificado:** comportamento indefinido.
- Corrigir o problema depende de contratos claros de memória e tamanho.

A Tabela ASCII

ASCII (American Standard Code for Information Interchange): Padrão para mapear caracteres em valores inteiros decimais.

Exemplos:

- O caractere 'A' é sempre representada pelo número 65 decimal (ou **01000001** binário).
- O caractere 'a' é 97.

ASCII Table

Dec	Hex	Char	Name	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NUL	null	32	20	Space	64	40	@	96	60	`
1	1	SOH	start of heading	33	21	!	65	41	A	97	61	a
2	2	STX	start of text	34	22	"	66	42	B	98	62	b
3	3	ETX	end of text	35	23	#	67	43	C	99	63	c
4	4	EOT	end of transmission	36	24	\$	68	44	D	100	64	d
5	5	ENQ	enquiry	37	25	%	69	45	E	101	65	e
6	6	ACK	acknowledge	38	26	&	70	46	F	102	66	f
7	7	BEL	bell	39	27	'	71	47	G	103	67	g
8	8	BS	backspace	40	28	(	72	48	H	104	68	h
9	9	HT	horizontal tab	41	29	)	73	49	I	105	69	i
10	A	LF	line feed	42	2A	*	74	4A	J	106	6A	j
11	B	VT	vertical tab	43	2B	+	75	4B	K	107	6B	k
12	C	FF	form feed	44	2C	,	76	4C	L	108	6C	l
13	D	CR	carriage return	45	2D	-	77	4D	M	109	6D	m
14	E	SO	shift out	46	2E	.	78	4E	N	110	6E	n
15	F	SI	shift in	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	data link escape	48	30	0	80	50	P	112	70	p
17	11	DC1	device control 1	49	31	1	81	51	Q	113	71	q
18	12	DC2	device control 2	50	32	2	82	52	R	114	72	r
19	13	DC3	device control 3	51	33	3	83	53	S	115	73	s
20	14	DC4	device control 4	52	34	4	84	54	T	116	74	t
21	15	NAK	negative acknowledge	53	35	5	85	55	U	117	75	u
22	16	SYN	synchronous idle	54	36	6	86	56	V	118	76	v
23	17	ETB	end of transmission block	55	37	7	87	57	W	119	77	w
24	18	CAN	cancel	56	38	8	88	58	X	120	78	x
25	19	EM	end of medium	57	39	9	89	59	Y	121	79	y
26	1A	SUB	substitute	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	escape	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	file separator	60	3C	<	92	5C	\	124	7C	
29	1D	GS	group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	US	unit separator	63	3F	?	95	5F	_	127	7F	DEL

From <https://ajsmith.org/tools/ascii-table/>

Caracteres e ASCII

- `char` é um **tipo inteiro distinto** de `int`.
- Em várias operações aritméticas, ocorre **promoção inteira** de `char` para `int`.
- No conjunto ASCII, `'A'` vale 65 e `'a'` vale 97.

```
char letra = 'B';  
printf("%d\n", (int)letra); // em ambiente ASCII: 66  
printf("%c\n", letra + 3); // em ambiente ASCII: E
```

A linguagem C não exige que todos os sistemas utilizem ASCII; a aula adota ASCII para exemplos introdutórios.

<ctype.h>: classificação de bytes

```
#include <ctype.h>
```

```
int isalpha(int c); // letra segundo o locale atual
int isdigit(int c); // dígito decimal
int isspace(int c); // espaço em branco
int tolower(int c); // converte maiúsculas
int toupper(int c); // converte minúsculas
```

```
char c = 'A';
if (isalpha((unsigned char)c)) {
    c = (char)tolower((unsigned char)c);
}
```

Para **char** potencialmente negativo, converta para **unsigned char** ao chamar estas funções. **tolower()** não realiza normalização Unicode completa.

Exercício: Palíndromo

Crie um programa que leia uma palavra/frase ASCII do teclado e verifique se ela é um palíndromo.

Versão 1: palavra ASCII:

- Ignorar apenas diferença entre maiúsculas/minúsculas.
- Comparar o primeiro e o último byte; aproximar os índices.
- Exemplos: **Ana**, **arara**, **Radar**.

Versão 2: frase ASCII:

- Ignorar espaços e pontuação (**isalnum**).
- Ignorar caixa (**tolower**).
- Exemplo: **O lobo ama o bolo**.

Extensão: textos com **é** / **ã** requerem regras e bibliotecas apropriadas para Unicode.

Vetores de strings: array bidimensional

```
#include <stdio.h>

int main(void) {
    char nomes[3][20] = {"Ana", "Bob", "Charlie"};
    printf("Total: %zu bytes\n", sizeof nomes); // 60
    for (size_t i = 0; i < 3; i++) {
        printf("%s\n", nomes[i]);
    }
    return 0;
}
```

- Cada linha tem capacidade de 20 bytes.
- Os dados pertencem a um bloco contíguo de memória.
- Cada **nomes[i]** precisa conter uma string C válida para **%s**.

Estratégia de testes

Teste cada programa com:

- String vazia, apenas Enter, um único caractere.
- Comprimento **exatamente igual** ao máximo admitido.
- Entrada maior que o buffer; restante da linha pendente.
- Espaços iniciais/finais e múltiplos espaços internos.
- Números, pontuação e diferença entre caixa alta/baixa.
- Texto acentuado: observar limitação de algoritmos ASCII.

Programas corretos dependem de **pré-condições**, **limites** e **casos extremos** explícitos.

Conclusão

- **String C:** bytes terminados em `\0`.
- **Comprimento $\neq$ capacidade:** `strlen()` versus `sizeof` (no array original).
- **Entrada:** prefira `fgets` e trate `\n`, retorno e truncamento.
- **Operações:** defina contratos de espaço antes de copiar/concatenar.
- **Ponteiros:** arrays frequentemente convertem-se em ponteiros, mas são tipos conceitualmente distintos.
- **Texto:** bytes ASCII são simples; UTF-8 exige atenção especial.
- **Qualidade:** testes de fronteira fazem parte do algoritmo.

Referências e estudo adicional

- ISO/IEC 9899 — Linguagem C (biblioteca padrão e strings).
- [cppreference — C string handling](#) ↗
- [cppreference — C character classification](#) ↗
- [Material do Prof. Carlos Maziero \(UFPR\)](#) ↗

Para compilar: `gcc -std=c11 -Wall -Wextra -Wpedantic arquivo.c -o programa`.

Dúvidas e Discussão

Exercício e Questões

Questões teóricas

1. Qual é a função do `\0`? Ele faz parte do resultado de `strlen()`?
2. Para `char s[12] = "ABC";`, determine `sizeof s`, `strlen(s)` e o espaço livre após o terminador.
3. Por que `char s[3] = {'O','K'};` é válido como string, enquanto `char t[2] = {'O','K'};` não é?
4. Qual a diferença entre `scanf("%19s", s)` e `fgets(s, 20, stdin)`?
5. Por que `a == b` não substitui `strcmp(a,b) == 0` para conteúdo textual?

Exercícios práticos

1. Leia uma linha e imprima cada byte ASCII em uma linha, sem imprimir `\n`.
2. Implemente `size_t meu_strlen(const char s[])`, sem usar `strlen()`.
3. Conte `a`, `A`, vogais, dígitos e espaços; especifique se `\n` entra na contagem.
4. Remova `\n` com `strcspn`; exiba primeira e última letra **se a string não estiver vazia**.
5. Leia uma frase ASCII e descubra a palavra mais longa, tratando sequências de espaços.