

STRINGS

(vetores de caracteres)

Introdução à Programação

Profª. Giorgia Mattos – giorgia@ci.ufpb.com

Strings

- **Cadeias/conjuntos de caracteres**

- Uma cadeia de caracteres, mais conhecida como string, é uma sequência de símbolos/caracteres
 - Letras: A, a, B, g, ...
 - Números: 0, 4, 139,
 - Caracteres: espaços em branco, !, ?, *, +, ', "....
- A linguagem C não possui um tipo de dado **string**, para isso utiliza vetores de elementos do tipo char onde cada posição do vetor armazena um caracter
- O fim de uma string é representada pelo caracter terminador '**\0**'

Strings

- Exemplos:

- “Linguagem C”
- “Aula de programação”
- “Centro de Informática - João Pessoa - PB - Brasil”
- “CEP: 58051-900”
- “Fone: +55 (83) 3216-7200”

Representação da string str “Linguagem C”

L	i	n	g	u	a	g	e	m		C	\0
0	1	2	3	4	5	6	7	8	9	10	11
str											

L	i	n	g	u	a	g	e	m		C	\0
0	1	2	3	4	5	6	7	8	9	10	11
str											

Nem sempre o '\0' está na última posição do vetor

Mas a string sempre termina na primeira ocorrência do '\0'

s	o	r	v	e	t	e	\0	t	p	i	k
0	1	2	3	4	5	6	7	8	9	10	11
sobremesa											

s	e	x	t	a	\0	t	e	r	ç	a	\0
0	1	2	3	4	5	6	7	8	9	10	11
diaSemana											

L	i	n	g	u	a	g	e	m		C	\0
0	1	2	3	4	5	6	7	8	9	10	11
str											

String armazenada no vetor *str*: "Linguagem C"

s	o	r	v	e	t	e	\0	t	p	i	k
0	1	2	3	4	5	6	7	8	9	10	11
sobremesa											

Caracteres após o '\0' fazem parte do vetor mas não da string

String armazenada no vetor *sobremesa*: "sorvete"

s	e	x	t	a	\0	t	e	r	ç	a	\0
0	1	2	3	4	5	6	7	8	9	10	11
diaSemana											

Mesmo que apareçam outros '\0' a string termina no primeiro

String armazenada no vetor *diaSemana*: "sexta"

Com a necessidade do **'\0'** para delimitar a **string**, o tamanho dela *sempre* será *menor* do que o tamanho do vetor que a armazena.

L	i	n	g	u	a	g	e	m		C	\0
0	1	2	3	4	5	6	7	8	9	10	11
str											

Tamanho do vetor: 12
Tamanho da string: 11

String armazenada no vetor **str**: **"Linguagem C"**

s	o	r	v	e	t	e	\0	t	p	i	k
0	1	2	3	4	5	6	7	8	9	10	11
sobremesa											

Tamanho do vetor: 12
Tamanho da string: 7

String armazenada no vetor **sobremesa**: **"sorvete"**

s	e	x	t	a	\0	t	e	r	ç	a	\0
0	1	2	3	4	5	6	7	8	9	10	11
diaSemana											

Tamanho do vetor: 12
Tamanho da string: 5

String armazenada no vetor **diaSemana**: **"sexta"**

Outros exemplos:

\0	i	w	a	u	m	r	e	m	u	C	x
0	1	2	3	4	5	6	7	8	9	10	11
str1											

String armazenada: ""
Tamanho da string: 0

String **vazia**, o '\0' está na primeira posição do vetor.

s	o	r	v	e	t	e	c	r	e	m	e
0	1	2	3	4	5	6	7	8	9	10	11
sobremesa1											

Não é uma String! O vetor não tem o '\0', mas continua sendo um vetor de caracteres.

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

```
int main(void) {
```

```
puts("Meu programa em C!");
```

```
return 0;
```

}

Desde o primeiro programa trabalhamos com strings - Toda sequência de caracteres que aparece entre aspas no código é uma **string**.

Ao executar esse programa, em algum local da memória será alocado um vetor armazenando essa **string**.

M	e	u		p	r	o	g	r	a	m	a		e	m		C	!	!	\0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
(nome desconhecido)																			


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

```
int main(void){  
    double raio, area;
```

```
    raio = 5.0;
```

```
    area = 3.14 * raio*raio;
```

```
    printf("R: %lf\n", raio);
```

```
    printf("A: %lf\n", area);
```

```
    return 0;
```

```
}
```

Outro exemplo:

O programa tem duas **strings**. Elas são **constantes strings**.

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

```
int main(void){
```

```
    double raio, area;
```

```
    raio = 5.0;
```

```
    area = 3.14 * raio * raio;
```

```
    printf("R: %lf\n", raio);
```

```
    printf("A: %lf\n", area);
```

```
    return 0;
```

```
}
```

Esses dois valores também são constantes, são **constantes reais**.

```
#include <stdio.h>

int main(void){
    double raio, area;

    raio = 5.0;
    area = 3.14 * raio*raio;
    printf("R: %lf\n", raio);
    printf("A: %lf\n", area);
    return 0;
}
```

Mas além de serem *constantes strings*, as duas **strings** do código-fonte também são **strings constantes**.

Ou seja, elas são armazenadas em uma área especial da memória que não permite alterações nos valores ali armazenados.

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

```
int main(void){
```

```
    double raio, area;
```

```
    raio = 5.0;
```

```
    area = 3.14 * raio*raio;
```

```
    printf("R: %lf\n", raio);
```

```
    printf("A: %lf\n", area);
```

```
    return 0;
```

```
}
```

Durante a execução de um programa, a área de memória ocupada por ele é dividida em quatro partes com funções e comportamentos distintos.

Memória

stack

heap

constantes

código / instruções

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

```
int main(void){
```

```
    double raio, area;
```

```
    raio = 5.0;
```

```
    area = 3.14 * raio*raio;
```

```
    printf("R: %lf\n", raio);
```

```
    printf("A: %lf\n", area);
```

```
    return 0;
```

```
}
```

Durante a execução de um programa, a área de memória ocupada por ele é dividida em quatro partes com funções e comportamentos distintos.

Vamos analisar 2 delas.

Memória

stack

constantes

```
int main(void){  
    double raio, area;  
  
    raio = 5.0;  
    area = 3.14 * raio*raio;  
    printf("R: %lf\n", raio);  
    printf("A: %lf\n", area);  
    return 0;  
}
```

As *constantes strings* alocam *strings constantes* na região constante.

A **área** constante é **protegida** e qualquer tentativa de **alteração** nos seus valores é uma **operação inválida** que causará o **encerramento do programa**.

stack

5.0
raio

78.5
area

constantes

R	:		%	1	f	\n	\0
0	1	2	3	4	5	6	7
(nome desconhecido)							

A	:		%	1	f	\n	\0
0	1	2	3	4	5	6	7
(nome desconhecido)							

```
int main(void){
    double raio, area;

    raio = 5.0;
    area = 3.14 * raio*raio;
    printf("R: %lf\n", raio);
    printf("A: %lf\n", area);
    return 0;
}
```

Valor **3.14** em **double**

constantes

A	:		%	1	f	\n	\0
0	1	2	3	4	5	6	7
(nome desconhecido)							

```
int main(void){  
    double raio, area;  
  
    raio = 5.0;  
    area = 3.14 * raio*raio;  
    printf("R: %lf\n", raio);  
    printf("A: %lf\n", area);  
    return 0;  
}
```

O valor de uma **constante string** é o valor do endereço de memória onde está armazenado o primeiro caractere da **string constante**.

Memória

stack

constantes

5.0
raio

78.5
area

R	:		%	1	f	\n	\0
0	1	2	3	4	5	6	7
(nome desconhecido)							

A	:		%	1	f	\n	\0
0	1	2	3	4	5	6	7
(nome desconhecido)							


```
int main(void){
    double raio, area;

    raio = 5.0;
    area = 3.14 * raio*raio;
    printf("R: %lf\n", raio);
    printf("A: %lf\n", area);
    return 0;
}
```

Com isso, podemos deduzir que no primeiro parâmetro o ***printf*** recebe o endereço de memória de um caractere, ou seja, um ***ponteiro para char***.

Memória

stack

constantes

R	:	%	1	f	\n	\0
0	1	2	3	4	5	6
(nome desconhecido)						

A	:		%	1	f	\n	\0
0	1	2	3	4	5	6	7
(nome desconhecido)							

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

```
int main(void){  
    char *str;
```

```
    return 0;
```

```
}
```

Esse programa aloca uma variável *ponteiro para char* chamada ***str***.

Memória

(lixo)

str

stack

constantes


```
#include <stdio.h>

int main(void){
    char *str;

    str = "AloMundo";

    return 0;
}
```

```
#include <stdio.h>

int main(void){
    char *str;

    str = "AloMundo";

    return 0;
}
```

Podemos atribuir esse valor a um ponteiro para caracteres.

Memória

(lixo)
str

(lixo)
str

The diagram shows a single stack frame for a function named 'lixo'. The frame is represented as a box with a light blue background and a thin black border. Inside the box, the text '(lixo)' is written in a dark blue, monospaced font. Below this, the text 'str' is written in a smaller, dark blue, monospaced font. The frame is positioned on the left side of the diagram, with a large, empty white area to its right. The word 'stack' is written in green text at the bottom left of the diagram.

constantes

A	l	o	M	u	n	d	o	\0
0	1	2	3	4	5	6	7	8
(nome desconhecido)								

A	l	o	M	u	n	d	o	\0
0	1	2	3	4	5	6	7	8
(nome desconhecido)								

```
#include <stdio.h>

int main(void){
    char *str;

    str = "AloMundo";

    return 0;
}
```

```
#include <stdio.h>

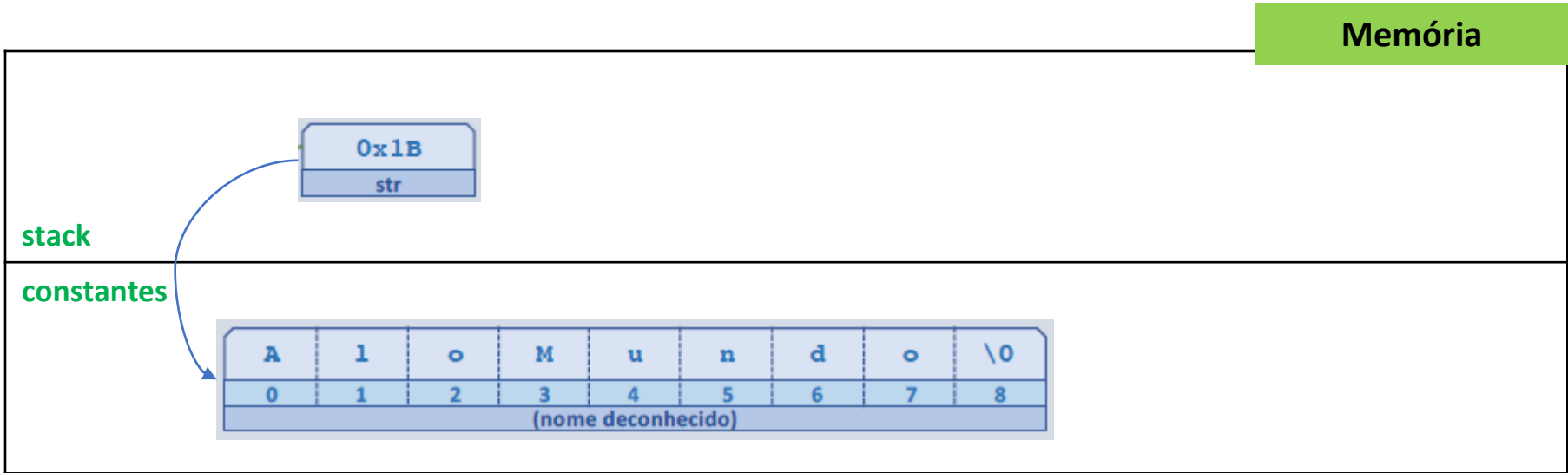
int main(void){
    char *str;

    str = "AloMundo";

    return 0;
}
```

Podemos atribuir esse valor a um ponteiro para caracteres.

Fazendo com que o a variável ***str*** aponte para essa ***string constante***. (O valor do endereço é só um exemplo)



```
#include <stdio.h>

int main(void){
    char *str;

    str = "AloMundo";
    puts(str);
    return 0;
}
```

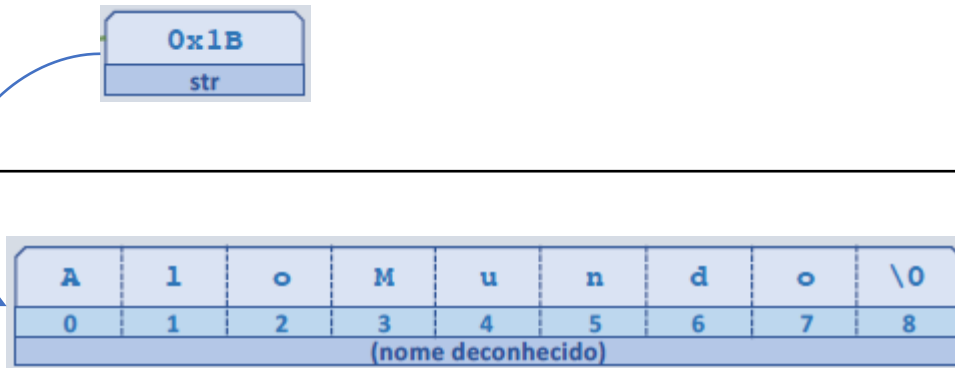
Então, já que **str** guarda o endereço de uma **string**, podemos passar esse endereço como parâmetro para funções que recebem **strings constantes**, como nesse **puts**.

Observe que seria o mesmo que passar a própria **constante string**, já que o seu valor seria também o endereço onde estaria armazenada.

Memória

stack

constantes



Strings Constantes

- É uma sequência de caracteres entre aspas que aparece no código.
 - Com uma exceção que veremos daqui a pouco
- Uma *string* constante é representada pelo endereço do seu primeiro caractere em memória
 - Portanto, pode ser atribuído a um ponteiro para **char**
- Pode ser usada assim:

```
char *str = "bolo";
```

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

```
int main(void){
```

```
    char *str = "bolo";
```

```
    puts(str);
```

```
    return 0;
```

```
}
```

A memória desse programa será organizada como ilustrado:

O ***puts*** imprime "***bolo***" e pula linha

Memória

stack

0x75

str

constantes

b	o	l	o	\0
0	1	2	3	4
(nome desconhecido)				


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

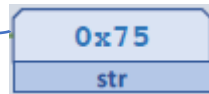
```
int main(void){  
    char *str = "bolo";  
    puts(str);  
  
    str[3] = 'a';  
    puts(str);  
  
    return 0;  
}
```

Fazendo uso da relação de equivalência entre ponteiros e vetores, podemos tentar atualizar a *string*.

O objetivo é alterar o caractere no índice **3** e transformar a *string* em **"bola"**.

Memória

stack



constantes

b	o	l	o	\0
0	1	2	3	4
(nome desconhecido)				

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

```
int main(void){
```

```
    char *str = "bolo";
```

```
    puts(str);
```

```
    str[3] = 'a';
```

```
    puts(str);
```

```
    return 0;
```

```
}
```

Essa operação é inválida! A atribuição causará uma ***falha de segmentação*** e o programa será ***imediatamente abortado***.

Memória

stack

0x75

str

constantes

b	o	l	o	\0
0	1	2	3	4
(nome desconhecido)				

```
#include <stdio.h>

int main(void){
    const char *str = "bolo";

    puts(str);

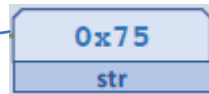
    return 0;
}
```

Portanto, ao trabalhar com ***strings constantes***, o ideal é definir o ponteiro com o ***const***.

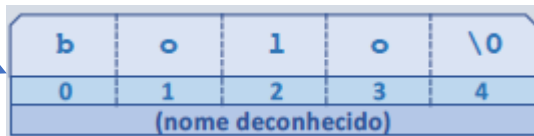
Dessa forma estamos instruindo o ponteiro a apontar para uma região de memória que não pode ser alterada, evitando que sejam adicionadas instruções que tentem fazer essa alteração.

Memória

stack



constantes



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

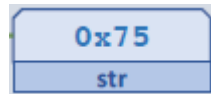
```
int main(void){  
    const char *str = "bolo";  
    puts(str);  
  
    str = "bola";  
    puts(str);  
  
    return 0;  
}
```

Para fazer **str** representar outra *string*, temos que fazer o ponteiro apontar para outra **string constante**.

A *string* anterior torna-se *inacessível*, já que não existe mais nada referenciando seu endereço. Ela ocupará seu espaço na memória até o encerramento do programa.

Memória

stack



constantes

b	o	l	o	\0
0	1	2	3	4
(nome desconhecido)				

b	o	l	a	\0
0	1	2	3	4
(nome desconhecido)				

Armazenamento de Strings em Vetores

- É possível armazenar uma **string** em um **vetor** de caracteres
- Exemplos:

```
char ar1[] = "bolo";
```

```
char nomePessoa[] = "Fulano de Tal";
```

Isso **não é uma string constante**, é apenas uma facilidade oferecida pela linguagem para iniciar um vetor de caracteres com uma *string*.

É a exceção comentada anteriormente. Qualquer outra ocorrência de caracteres entre aspas em um código-fonte, desde que não seja na iniciação de um vetor (como aqui), será uma *string constante*.

Armazenamento de Strings em Vetores

- É possível armazenar uma **string** em um **vetor** de caracteres
- Exemplos:

```
char ar1[] = "bolo";
```

Isso **não** é uma **string constante**, é apenas uma facilidade oferecida pela linguagem para iniciar um vetor de caracteres com uma *string*.

```
char ar1[] = {'b', 'o', 'l', 'o', '\0'};
```

Se não fosse o facilitador, a *string* teria que ser iniciada assim.

As duas definições são equivalentes. Observe que usando o facilitador, o `'\0'` é inserido automaticamente.

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

```
int main(void){
```

```
    char str[] = "bolo";
```

```
    puts(str);
```

```
    return 0;
```

```
}
```

A organização da memória para este exemplo fica assim.

Ou seja, a definição de **str** aloca, na pilha, um vetor de caracteres com **5** elementos contendo a *string* **"bolo"**.

Memória

The diagram shows a memory stack with a table representing the 'str' array. The table has two rows: the first row contains the characters 'b', 'o', 'l', 'o', and a null terminator '\0'; the second row contains the indices 0, 1, 2, 3, and 4. The label 'str' is centered below the table. An arrow points from the 'str' array definition in the code to this table.

b	o	l	o	\0
0	1	2	3	4

str

stack

constantes

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

```
int main(void){
```

```
    char str[] = "bolo";
```

```
    puts(str);
```

```
    str[3] = 'a';
```

```
    puts(str);
```

```
    return 0;
```

```
}
```

Como **str** está alocado na pilha, ele pode ser alterado.

Memória

b	o	l	o	\0
0	1	2	3	4
str				

stack

constantes


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

```
int main(void){
```

```
    char str[] = "bolo";
```

```
    puts(str);
```

```
    str[3] = 'a';
```

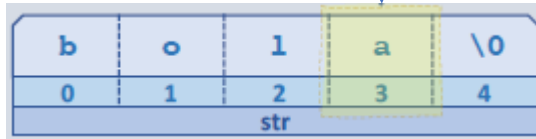
```
    puts(str);
```

```
    return 0;
```

```
}
```

Como **str** está alocado na pilha, ele pode ser alterado.

Memória



b	o	l	a	\0
0	1	2	3	4
str				

stack

constantes

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

```
int main(void){
```

```
    char str[] = "bolo";
```

```
    puts(str);
```

```
    str[3] = 'a';
```

```
    puts(str);
```

```
    return 0;
```

```
}
```

Como **str** está alocado na pilha, ele pode ser alterado.

O segundo **puts** imprime a *string* atualizada: **"bola"**.

Memória

b	o	l	a	\0
0	1	2	3	4
str				

stack

constantes

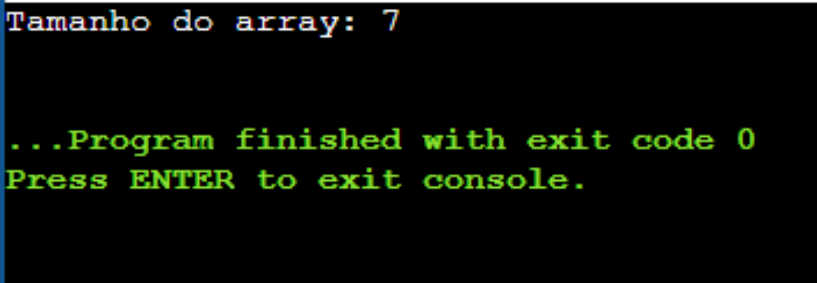
Armazenamento de Strings

O que será impresso?

```
int main(void) {  
    char str[] = "camelo";  
    int tamanhoStr, tamanhoElem;  
  
    tamanhoStr = sizeof(str);  
    tamanhoElem = sizeof(str[0]);  
    printf("Tamanho do array: %d\n", tamanhoStr/tamanhoElem);  
    return 0;  
}
```

Quando não definimos explicitamente o tamanho do *vetor*, ele terá o tamanho da quantidade de elementos iniciados.

Aqui são iniciados **7** caracteres, contando com o **'\0'** que foi automaticamente adicionado.



```
Tamanho do array: 7  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Armazenamento de Strings

```
char ar1[4] = "bolo";
```

Não é uma *string*!

Quando iniciamos um *vetor* com mais elementos que o tamanho declarado, os elementos a mais são ignorados. O **vetor de caracteres *ar1*** não armazena uma ***string*** porque não possui o elemento ***'\0'***.

Armazenamento de Strings

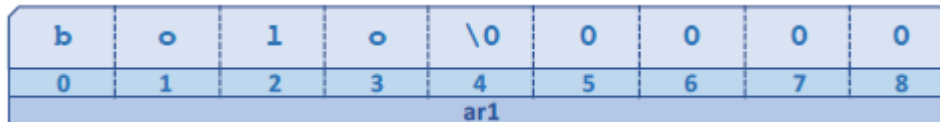
```
char ar1[5] = "bolo";
```

Atenção ao tamanho dos vetores:
Lembre-se que eles devem ter
espaço para o `'\0'`.

Armazenamento de Strings

Como ***ar1*** será alocado na memória?
Os elementos não iniciados em um *vetor* são completados com zeros.

```
char ar1[9] = "bolo";
```



b	o	l	o	\0	0	0	0	0
0	1	2	3	4	5	6	7	8

ar1

Comparando Ponteiros, Strings e Caracteres

O que será impresso?

```
#include <stdio.h>
int main(void) {
    size_t tamanhoChar = sizeof((char) 'A');
    size_t tamanhoStr = sizeof("A");
    printf("tamanho char = %d\n tamanho str = %d\n", tamanhoChar, tamanhoStr);
    return 0;
}
```

```
tamanho char = 1
tamanho str = 2

...Program finished with exit code 0
Press ENTER to exit console.
```

Comparando Ponteiros, Strings e Caracteres

	CARACTER CONSTANTE	STRING CONSTANTE COM UM CARACTER
Espaço ocupado	1 byte	2 bytes
Tipo	char	char *
Exemplo	'A'	"A"


```
int main() {  
    char *p = "string"; /* O asterisco aqui eh declarador */  
    char c;  
  
    p = &c;  
    *p = 'a';           /* Legal e normal */  
    *p = "a";           /* Legal mas não eh normal */  
    p = "a";            /* Legal e normal */  
    p = 'a';            /* Problema a vista! */  
    *p = "string"; /* O asterisco aqui eh operador de */  
                      /* indireção */  
    return 0;  
}
```

```
int main() {  
    char *p = "string"; /* O asterisco aqui eh declarador */  
    char c;  
  
    p = &c;  
    *p = 'a';           /* Legal e normal */  
    *p = "a";           /* Legal mas não eh normal */  
    p = "a";            /* Legal e normal */  
    p = 'a';            /* Problema a vista! */  
    *p = "string"; /* O asterisco aqui eh operador de */  
                        /* indireção */  
  
    return 0;  
}
```

As constantes marcadas com **verde** são ***strings constantes***. Seus valores são os endereços de memória onde estão armazenadas.

```
int main() {  
    char *p = "string"; /* O asterisco aqui eh declarador */  
    char c;  
  
    p = &c;  
    *p = 'a';           /* Legal e normal */  
    *p = "a";           /* Legal mas não eh normal */  
    p = "a";           /* Legal e normal */  
    p = 'a';           /* Problema a vista! */  
    *p = "string"; /* O asterisco aqui eh operador de */  
                        /* indireção */  
  
    return 0;  
}
```

As constantes marcadas com **rosa** são **constantes caractere**. Seus valores são os inteiros de 1 byte com o código do caractere.

```

int main() {
1   char *p = "string"; /* O asterisco aqui eh declarador */
2   char c;
3
4   p = &c;
5   *p = 'a';           /* Legal e normal */
6   *p = "a";           /* Legal mas não eh normal */
7   p = "a";            /* Legal e normal */
8   p = 'a';            /* Problema a vista! */
9   *p = "string";      /* O asterisco aqui eh operador de */
10                          /* indireção */
11   return 0;
}

```

1. O ponteiro **p** aponta para o primeiro caractere da **string constante "string"**;
2. Agora **p** aponta para o (único) caractere **c**. A **string constante "string"** ficou perdida, ocupa espaço mas não pode ser usada porque ninguém mais conhece o seu endereço;
5. Como **p** está apontando para **c**, essa indireção atribui o valor do caractere **'a'** à variável **c**.

```

int main() {
1   char *p = "string"; /* O asterisco aqui eh declarador */
2   char c;
3
4   p = &c;
5   *p = 'a';           /* Legal e normal */
6   *p = "a";           /* Legal mas não eh normal */
7   p = "a";            /* Legal e normal */
8   p = 'a';            /* Problema a vista! */
9   *p = "string";      /* O asterisco aqui eh operador de */
10                          /* indireção */
11   return 0;
}

```

6. O compilador aceita essa instrução mas ela é problemática: o valor da **string constante** "a" é o endereço de memória onde essa **string** está armazenada. Esse valor, inteiro, será convertido em **1 byte**, ocorrerá **overflow** e a variável **c** receberá o código de um caractere que não pode ser previsto. Mas receberá um valor.

```

int main() {
1   char *p = "string"; /* O asterisco aqui eh declarador */
2   char c;
3
4   p = &c;
5   *p = 'a';           /* Legal e normal */
6   *p = "a";           /* Legal mas não eh normal */
7   p = "a";            /* Legal e normal */
8   p = 'a';            /* Problema a vista! */
9   *p = "string"; /* O asterisco aqui eh operador de */
10                          /* indireção */
11   return 0;
}

```

7. O ponteiro ***p*** agora aponta para essa ***outra string constante "a"***, distinta daquela da linha anterior.
8. O ponteiro ***p*** vai apontar para o endereço de memória cujo valor é o código do caracter ***'a'***. O que tem lá?

```

int main() {
1   char *p = "string"; /* O asterisco aqui eh declarador */
2   char c;
3
4   p = &c;
5   *p = 'a';           /* Legal e normal */
6   *p = "a";           /* Legal mas não eh normal */
7   p = "a";           /* Legal e normal */
8   p = 'a';           /* Problema a vista! */
9   *p = "string";      /* O asterisco aqui eh operador de */
10                          /* indireção */
11   return 0;
}

```

9. Mesma situação que a linha 6 – mas aqui tem o agravante de tentar alterar uma região de memória que não deveria, já que a linha anterior fez ***p*** apontar para uma região misteriosa: falha de segmentação.

Comparando Ponteiros, Strings e Caracteres

EXEMPLO	NORMAL?	JUSTIFICATIVA
<code>*p = 'a'</code>	Sim	Os dois lados da atribuição são do tipo char
<code>*p = "a"</code>	Não	O lado esquerdo da atribuição é do tipo char e o lado direito é do tipo char *
<code>p = "a"</code>	Sim	Os dois lados da atribuição são do tipo char *
<code>p = 'a'</code>	Não	O lado esquerdo da atribuição é do tipo char * e o lado direito é do tipo char


```

int main(void) {
1      char ar[10];
2      char *ptr = "10 espaços";
3
4      ar = "errado"; /* ILEGAL */
5      ar[2] = 'a';
6      ptr[5] = 'b'; /* Programa eh abortado com sinal SIGSEGV */
7      *(ptr + 5) = 'b'; /* Programa eh abortado com sinal SIGSEGV */
8      ptr = "OK";
9      *ptr = "ilegal?"; /* Programa eh abortado com sinal SIGSEGV */
10     printf("%c", 3["Estranho"]);
11     return 0;
}

```

1. O **vetor** *ar* é alocado com 10 elementos.
2. O ponteiro *ptr* é alocado e uma **string constante** com 11 elementos. O ponteiro aponta para essa string constante.

```

1 int main(void) {
2     char ar[10];
3     char *ptr = "10 espaços";
4
5     ar = "errado"; /* ILEGAL */
6     ar[2] = 'a';
7     ptr[5] = 'b'; /* Programa eh abortado com sinal SIGSEGV */
8     *(ptr + 5) = 'b'; /* Programa eh abortado com sinal SIGSEGV */
9     ptr = "OK";
10    *ptr = "ilegal?"; /* Programa eh abortado com sinal SIGSEGV */
11    printf("%c", 3["Estranho"]);
12    return 0;
13 }

```

4. O nome do vetor ***ar*** tem o valor do endereço de memória onde ele está alocado. Porém, apenas para leitura, não é possível modificá-lo.

5. O *terceiro elemento* do vetor ***ar*** recebe o valor ***'a'***.

```

int main(void) {
1      char ar[10];
2      char *ptr = "10 espaços";
3
4      ar = "errado"; /* ILEGAL */
5      ar[2] = 'a';
6      ptr[5] = 'b'; /* Programa eh abortado com sinal SIGSEGV */
7      *(ptr + 5) = 'b'; /* Programa eh abortado com sinal SIGSEGV */
8      ptr = "OK";
9      *ptr = "ilegal?"; /* Programa eh abortado com sinal SIGSEGV */
10     printf("%c", 3["Estranho"]);
11     return 0;
}

```

6. A tentativa de alterar o *sexto elemento* da *string constante* apontada por ***ptr*** causará o encerramento do programa, pois ela está armazenada em uma região de memória que não pode ser alterada.
7. O mesmo caso da linha 6, já que é uma instrução equivalente à anterior.

```

int main(void) {
1      char ar[10];
2      char *ptr = "10 espaços";
3
4      ar = "errado"; /* ILEGAL */
5      ar[2] = 'a';
6      ptr[5] = 'b'; /* Programa eh abortado com sinal SIGSEGV */
7      *(ptr + 5) = 'b'; /* Programa eh abortado com sinal SIGSEGV */
8      ptr = "OK";
9      *ptr = "ilegal?"; /* Programa eh abortado com sinal SIGSEGV */
10     printf("%c", 3["Estranho"]);
11     return 0;
}

```

8. O ponteiro ***ptr*** agora aponta para a ***string constante*** "OK".
9. Essa indireção *alteraria* o primeiro caractere da *string* apontada por ***ptr***. Mas ***ptr*** está apontando para uma ***string constante*** e a alteração é protegida.
10. Imprime 'r', o quarto elemento da ***string constante*** (vetor) "Estranho".


Strings como Parâmetro de Funções

Para definir um parâmetro como *string*, fazemos igual definimos um parâmetro vetor. Afinal uma *string* é um vetor de caracteres.

```
void Funcao1(const char str[]);
```

```
void Funcao2(const char *str);
```

Mas normalmente essa é a forma preferida.



Strings como Parâmetro de Funções

Diferentemente de um parâmetro que é um vetor comum, com uma string não é preciso do parâmetro adicional com o seu tamanho pois a própria *string* já traz essa informação: ela acaba no caractere `'\0'`.

É seguro acessar seus elementos até a posição em que o caractere terminador seja encontrado.

```
void Funcao2(const char *str);
```

Leitura de Strings: scanf()

- Quando um programa precisa ler uma string do teclado, essa string precisa ser armazenada em um vetor.

```
char str[31];
```

```
printf("Digite a string (max de 30 caracteres): ");
```

- Podemos ler uma string caracter a caracter, como faríamos com qualquer outro vetor, mas é mais simples ler a string inteira:

```
scanf("%s", str);
```

- A leitura a partir do teclado utilizando o comando scanf() lê somente até o primeiro espaço, ou seja, armazena somente uma palavra. Para ler uma cadeia de caracteres até encontrar um <enter> (fim da string), use:

```
scanf("%[^\n]s", str);
```

Leitura de Strings: gets()

```
char str[31];
```

```
printf("Digite a string (max de 30 caracteres): ");
```

- Para contornar as deficiências do scanf() para armazenar strings, podemos utilizar a função **gets()**, que faz a leitura e o armazenamento de caracteres até encontrar o caracter de fim de linha ('\n' e '\0')

```
gets(str);
```

Permite a leitura de frases inteiras, com várias palavras, passando o endereço do vetor que armazenará a *string* como parâmetro. Além do '\0', o **gets** também incluirá o '\n' digitado na *string*.

Leitura de Strings: gets()

```
char str[31];  
printf("Digite a string (max de 30 caracteres): ");  
gets(str);
```

Devido a problemas de corrupção de memória quando o usuário digita mais caracteres que a capacidade do vetor, a função **gets** é considerada **perigosa** e foi **descontinuada**.

Ao compilar um programa que faz chamada à função **gets**, o compilador emite uma mensagem parecida com essa:

warning: 'gets' is deprecated [-Wdeprecated-declarations]

warning: the 'gets' function is dangerous and should not be used.

Leitura de Strings: fgets()

- Problema: tanto o scanf() quanto o gets() podem ler e tentar armazenar mais caracteres do que aqueles que foram definidos no momento da declaração do vetor de caracteres, provocando erros (corrupção de memória)
- Para a leitura segura de strings devemos usar a função **fgets()**

```
char str[31];
```

```
printf("Digite seu nome (max de 30 caracteres): ");
```

```
fgets(str, 31, stdin);
```

A função fará a leitura de *no máximo* 30 caracteres, já que ela também insere o `'\0'` no final da *string*. Quando houver espaço, o `'\n'` também será incluído na *string* lida.

Escrita de Strings na tela: printf() e puts()

```
char str[31];  
  
printf("Digite sua string (max de 30 caracteres): ");  
scanf("%s", str);  
  
printf("A string digitada:\n%s\n", str);  
puts("A string digitada:");  
puts(str);
```

Para imprimir *strings*, basta passarmos o endereço do primeiro caractere delas, tanto no ***puts*** quanto no primeiro parâmetro ou com uso do formatador ***%s*** no ***printf***.

Leitura e Escrita de Caracteres

- É possível ler através do teclado apenas um caracter com as funções **scanf ()** e **getchar ()**
 - scanf () – usar o formatador %c
 - getchar () – lê um caracter do teclado, apresenta-o na tela e aguarda que a tecla <enter> seja pressionada

```
char a;
```

```
scanf("%c", &a);
```

```
a = getchar();
```

Leitura e Escrita de Caracteres

- Podemos escrever um caracter na tela utilizando o comando `printf()`, com o mesmo controle usado na leitura (`%c`)

```
printf("%c\n", a);
```

- De forma análoga ao `getchar()`, temos o **`putchar()`** que escreve apenas 1 caracter

```
putchar(a);
```

Exemplos:

1. Quantos caracteres tem uma string, qual o tamanho da string?

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

```
int tamanhoString (const char *str){
```

```
    int i;
```

```
    for (i=0; str[i]!='\0';i++);
```

```
    return i;
```

```
}
```

```
int main(){
```

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

```
    printf("Tamanho da string '%s' = %d\n", palavra, tamanhoString(palavra));
```

```
    return 0;
```

```
}
```

Exemplos:

2. Copiar uma string para outra string, como uma atribuição.

```
#include <stdio.h>

void copiaString (char *copiaS, const char *str){
    int i;

    for (i=0; str[i]!='\0';i++)
        copiaS[i] = str[i];

    copiaS[i] = '\0';
}

int main(){
    char palavra[] = "paralelo", copia[9];

    copiaString (copia, palavra);
    printf("String copiada = %s\n", copia);
    return 0;
}
```

Exemplos:

3. Comparar duas strings e responder se elas são iguais ou não.

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

```
int comparaString (const char *str1, const char *str2){
```

```
    int i, iguais = 0;
```

```
    for (i=0; str1[i] == str2[i];i++)
```

```
        if (str1[i] == '\0')
```

```
            iguais = 1;
```

```
    return iguais; }
```

```
int main(){
```

```
    char palavra1[] = "bola", palavra2[] = "bola";
```

```
    if (comparaString (palavra1, palavra2))
```

```
        printf("As palavras sao iguais!");
```

```
    else
```

```
        printf ("As palavras sao diferentes!");
```

```
    return 0; }
```


A Biblioteca string.h

- **Manipulando cadeias de caracteres**

- As cadeias de caracteres são tão importantes que existe uma biblioteca de funções implementadas só para manipular strings, a biblioteca **string.h**.
- Entre as diversas funcionalidades oferecidas por esta biblioteca, podemos destacar:
 - **strlen (str)** — Retorna o tamanho da string str em número de caracteres

```
int x;  
x = strlen (str);
```

A Biblioteca string.h

- **Manipulando cadeias de caracteres**

- **strcpy (destino, fonte)** — Copia a string fonte para a string destino

strcpy (strd, strf);

- **strcat (destino, fonte)** — Concatena/junta a string fonte no fim da string destino

strcat (strd, strf);

A Biblioteca string.h

• Manipulando cadeias de caracteres

- strcmp (str1, str2) — Compara duas cadeias de caracteres e retorna um valor
 - = 0, se str1 e str2 forem iguais
 - negativo, se str1 for menor que str2
 - positivo, se str1 for maior que str2

```
x = strcmp (str1, str2);
```

```
if (strcmp(str1,str2) == 0)  
    printf ("As strings são iguais!");
```

Maior e menor aqui não tem a ver com o comprimento das *strings*, mas com a forma como elas são comparadas pela função, seguindo a **Ordem de Colação** (posição dos caracteres na tabela ASCII) .

Strings – Exercícios práticos

Todos os exercícios devem usar funções

1) Fazer um programa para ler uma string e um caracter qualquer. Calcular o número de ocorrências desse caracter na string. Exemplo: Seja a string "maracatu" e o caracter 'a', então o número de ocorrências de 'a' é 3.

2) Ler uma frase e contar o número de palavras dessa frase. Considere que as palavras estão separadas por espaços.

3) Faça um programa que calcule e mostre o número de vogais de uma string. Feito isso, o programa deve criptografar uma frase dada pelo usuário (a criptografia troca as vogais da frase por *).

Frase: eu estou na escola

*Saída: ** *st** n* *sc*/**

4) Escreva um programa que receba uma frase com caracteres em minúsculo e transforme o primeiro caractere de cada palavra da frase em maiúsculo.

Strings – Exercícios práticos

Todos os exercícios devem usar funções

5) Escreva um programa que receba uma string (palavra), determine se ela é palíndromo. Um palíndromo é uma palavra que tenha a propriedade de poder ser lida tanto da direita para a esquerda como da esquerda para a direita. Ex.: arara, ovo, anilina, salas.

6) Fazer um programa para ler uma string e um caracter qualquer. Construir um vetor (OC) contendo as posições (índices) de onde ocorre o caracter na string. Exemplo: Seja a string "abracadabra!!!" e o caracter 'a', então o vetor de índices OC deverá conter os seguintes valores: { 0 3 5 7 10 -1}. O valor -1 indica final de vetor, ou seja, que não existem mais ocorrências. Caso, não exista nenhuma ocorrência, deve ser armazenado o valor -1 no vetor.

7) Escreva um programa que recebe uma string e transforma alguns dos caracteres em maiúsculos e outros em minúsculos. Faça sorteios com [a função rand\(\) para gerar números aleatórios em C](#), que serão usados para escolher os índices dos caracteres que serão alterados. Use [a função toupper\(\) para converter o caracter para maiúsculo](#).

Strings – Exercícios práticos

Todos os exercícios devem usar funções

8) Crie um programa que receba uma string e um caractere, e apague todas as ocorrências desse caractere na string.

9) Escreva um programa que receba o nome completo de um usuário e gere o seu login. O login será composto pela primeira letra de cada nome em letras maiúsculas e as mesmas letras minúsculas. **O login deve estar em uma nova string.**

<i>Nome: Pedro Hansdorf</i>	<i>>> Login: PHph</i>
<i>Nome: Robson Soares Soares</i>	<i>>> Login: RSSrss</i>
<i>Nome: Olívia Silva Santos Marques</i>	<i>>> Login: OSSMossm</i>

10) Implemente uma função que junta/concatena duas strings, assim como a função strcat().

Strings – Exercícios práticos

11) PESQUISA:

Pesquise a função, o funcionamento e a utilização das seguintes funções para manipulação de strings em C:

- strstr();
- strchr() ;
- strrchr();
- atoi();
- strtod();