

## Aula 19 – O tipo char e cadeias de caracteres (strings)

### Caracteres

Caracteres ocupam 1 byte (8 bits) na memória e são declarados com o tipo `char`.

### Declaração:

```
char a, b;  
char x = 'a';  
char y[12];  
char z[26] = {"abcdefghijklmnopqrstuvwxy"};  
char w[5] = {'a', 'e', 'i', 'o', 'u'};
```

### Usos:

Uma variável do tipo `char` ocupa 1 byte (8 bits) e pode ser usada para armazenar um valor inteiro sem sinal (`unsigned char`) entre 0 e 255 ou um valor com sinal (`char`) entre -128 a 127. Pode ser usada tanto como um número ou como caractere.

Exemplos:

```
char a, b[26];  
:  
:  
/* constante do tipo char fica entre apóstrofes */  
a = '.'; b[3] = '&';  
/* pode atribuir um valor numérico a variável tipo char */  
a = 32; b[8] = 20  
/* branquear a vetor b usando a como contador */  
for (a = 0; a < 20; a++) b[a] = ' ';  
/* colocar em b as letras maiúsculas */  
/* em ASCII 65=A, 66 = B, . . . , 90 = Z */  
for (a = 0; a < 26; a++) b[a] = a + 65;  
/* imprimir b com formato numérico e caractere */  
for (a = 0; a < 26; a++)  
    printf("b[%2d] = %2d e b[%2d] = %1c", a, b[a], a, b[a]);
```

### Codificação ASCII (American Standard Code for Information Interchange)

Caracteres são armazenados internamente como números. Assim é necessária uma codificação, isto é, uma associação entre um caractere e um valor numérico. Existem várias tipos de codificação (EBCDIC, BCD, BAUDOT, etc.), mas a mais utilizada é o ASCII.

O ASCII é um código de 7 bits, portanto possui 128 possíveis combinações, que hoje em dia é a codificação básica usada em informática, e que admite várias extensões,

principalmente para possibilitar a representação de caracteres especiais em várias línguas (letras acentuadas, por exemplo, alfabeto russo, japonês, etc.). As extensões estão associadas normalmente aos valores de 129 a 255, usando, portanto todos os 8 bits de um byte.

O programa abaixo imprime todos os caracteres, com os correspondentes valores decimais e hexadecimais desde 32 até 254. Antes de 32, estão alguns caracteres gráficos, bem como os caracteres especiais que tem alguma função especial quando enviados aos dispositivos de entrada e saída, por exemplo:

CR (13) – carriage return ( retorna para o início da linha corrente)

LF (10) – line feed ( pula para a próxima linha)

FF (14) – form feed ( pula para a próxima página)

BEL (07) – beep (aciona o dispositivo sonoro)

Etc.

Veja um trecho da saída também abaixo.

```
#include <stdio.h>
/* Imprime a tabela com os caracteres ASCII de 32 a 254 */
int main() {
    unsigned char i;
    for (i = 32; i < 255; i++)
        printf("\ndecimal = %3d * hexadecimal = %2x * caracter =
%1c", i, i, i);
}
```

```
decimal = 32 * hexadecimal = 20 * caracter =
decimal = 33 * hexadecimal = 21 * caracter = !
decimal = 34 * hexadecimal = 22 * caracter = "
decimal = 35 * hexadecimal = 23 * caracter = #
decimal = 36 * hexadecimal = 24 * caracter = $
decimal = 37 * hexadecimal = 25 * caracter = %
decimal = 38 * hexadecimal = 26 * caracter = &
decimal = 39 * hexadecimal = 27 * caracter = '
decimal = 40 * hexadecimal = 28 * caracter = (
decimal = 41 * hexadecimal = 29 * caracter = )
decimal = 42 * hexadecimal = 2a * caracter = *
decimal = 43 * hexadecimal = 2b * caracter = +
decimal = 44 * hexadecimal = 2c * caracter = ,
decimal = 45 * hexadecimal = 2d * caracter = -
decimal = 46 * hexadecimal = 2e * caracter = .
decimal = 47 * hexadecimal = 2f * caracter = /
decimal = 48 * hexadecimal = 30 * caracter = 0
decimal = 49 * hexadecimal = 31 * caracter = 1
decimal = 50 * hexadecimal = 32 * caracter = 2
decimal = 51 * hexadecimal = 33 * caracter = 3
decimal = 52 * hexadecimal = 34 * caracter = 4
decimal = 53 * hexadecimal = 35 * caracter = 5
decimal = 54 * hexadecimal = 36 * caracter = 6
decimal = 55 * hexadecimal = 37 * caracter = 7
decimal = 56 * hexadecimal = 38 * caracter = 8
decimal = 57 * hexadecimal = 39 * caracter = 9
```

```
decimal = 58 * hexadecimal = 3a * caracter = :  
decimal = 59 * hexadecimal = 3b * caracter = ;  
decimal = 60 * hexadecimal = 3c * caracter = <  
.....  
.....  
.....
```

Observe agora que é equivalente atribuir um caractere ou o valor numérico correspondente.

```
char a;  
:  
:  
/* são equivalentes */  
a = 32;  
a = '\ ';
```

### Entrada e saída de caracteres

O formato %c é usado para ler ou imprimir caracteres. Na saída, se especificado um comprimento, por exemplo, %3c, são colocados brancos à esquerda.

O programa abaixo lê um vetor de 20 caracteres e imprime os caracteres lidos intercalando-os com um branco.

```
include <stdio.h>  
int main() {  
    char a[100];  
    int i;  
    for (i = 0; i < 20; i++)  
        scanf("%c", &a[i]);  
  
    for (i = 0; i < 20; i++)  
        printf("%2c", a[i]);  
}
```

Outra forma é o uso das funções getchar e putchar:

int getchar () – devolve o próximo caractere digitado.  
void putchar (char x) – imprime o caractere x

Veja o exemplo abaixo que também lê e imprime uma seqüência de caracteres.

```
#include <stdio.h>  
#define fim '#'  
int main() {  
    char c;  
    /* Le um conjunto de caracteres ate encontrar '#'  
       O controle só volta ao programa após o enter.  
       Portanto, só imprime após o enter */  
    c = getchar();  
    while (c != fim) {  
        putchar(c);  
        c = getchar();  
    }  
}
```

```
/* outra forma */  
while ((c = getchar()) != fim) putchar (c);  
}
```

## Char e unsigned char

Conforme vimos nos exemplos acima, uma variável do tipo `char`, pode ser usada como uma variável `int` ou `short`. A diferença é que **char** tem apenas 8 bits enquanto que o **short** tem 16 bits e **int** tem 32 bits.

Quando se declara `char` (ou `short` ou `int`), o bit mais à esquerda é o bit de sinal na notação complemento de 2. Se não há necessidade do sinal declara-se `unsigned char` (ou `unsigned short` ou `unsigned int`).

Tipo	Valores
Char	-128 a +127
unsigned char	0 a 255
Short	$-2^{15}$ a $+2^{15}-1$
unsigned short	0 a $2^{16}-1$
Int	$-2^{31}$ a $+2^{31}-1$
unsigned int	0 a $2^{32}-1$

Quando usamos uma variável do tipo `char` para conter apenas caracteres, a configuração do caractere armazenado pode corresponder a um número negativo. Algumas conversões indesejáveis podem ocorrer quando se misturam `char` e `int`.

Para evitar inconsistências, é conveniente sempre usar-se `unsigned char` quando se usa o valor numérico de uma variável do tipo `char`.

## Cadeias de caracteres ou strings

A manipulação de seqüências ou cadeias de caracteres (strings) é uma operação muito comum em processamento não numérico. Imagine os programas editores de texto que precisam fazer operações do tipo:

- Procurar uma cadeia de caracteres no texto
- Eliminar uma cadeia do texto
- Substituir uma cadeia por outra
- Etc.

A manipulação de caracteres merece uma especial atenção das linguagens de programação e não é diferente em C.

Cadeias de caracteres são seqüências de caracteres terminadas pelo caractere especial zero binário, ou `'\0'`. São armazenadas em vetores de caracteres.

## Entrada e saída de cadeias de caracteres

Usando o comando `printf` e o formato `%s`. Veja o exemplo abaixo:

```
#include <stdio.h>
int main() {
    char a[100];

    /* Le todos os caracteres digitados e coloca em a
       insere um \0 no final */
    scanf("%s", a);
    /* imprime todos os caracteres até o '\0' */
    printf("%s", a);
}
```

Supondo **char x**, ambos os comandos abaixo atribuem zero binário à variável **x**:

```
x = '\0';
x = 0;
```

Como nas demais entradas com formato, o caractere **espaço**, quando digitado, é um delimitador de campo. Portanto, mesmo quando se usa o formato `%s`, a cadeia de caracteres lidos irá somente até o primeiro espaço.

Exemplo: se no programa acima for digitado **"MARIA DA SILVA"**, somente será lido **"MARIA"**.

Usando as funções `gets` e `puts`:

```
char *gets(char *s);
```

Lê os caracteres digitados e os coloca no vetor `s` até que seja digitado `<enter>`. O `<enter>` é descartado e é inserido um `\0` no final.

```
int puts(const char *s);
```

Imprime os caracteres do vetor `s` até encontrar um `\0`. Adiciona um caractere `\n` no final, ou seja, muda de linha.

Veja o exemplo abaixo:

```
#include <stdio.h>
int main() {
    char a[100];
    /* Le e imprime um string */
    gets (a);
    puts (a);
}
```

## Algumas funções usando strings

```
/* move string a para b */
void move ( char a[], char b[]) {
    int i = 0;
    while (a[i] != '\0') {b[i] = a[i]; i++;}
    b[i] = a[i];
}

/* devolve o tamanho do string a */
int tamanho (char a[]) {
    int i = 0;
    while (a[i] != '\0') i++;
    return i
}

/* idem a move usando a função tamanho */
void move ( char a[], char b[]) {
    int i, k;
    k = tamanho (a);
    for (i = 0; i <= k; i++) b[i] = a[i];
}

/* outra versão da move */
void copia (char s[],char t[]) {
    int i=0;
    while((s[i]=t[i])!='\0') i++;
}

/* concatena 2 strings */
void concatena (char s[], char t[]) {
    int i=0,j=0;
    while (s[i]!='\0')i++;
    while((s[i++]=t[j++])!= '\0');
}

```

## Comparação de strings:

A comparação entre dois strings é feita, levando em conta o código numérico da sua codificação ASCII de cada caractere. Lembre-se que a codificação ASCII leva em conta a ordem alfabética, isto é, 'a' < 'b' < 'c' < . . . 'z'. Assim:

```
"maria da silva" > "maria da selva"
"xico" > "francisco" (neste caso os complementos são diferentes)
"maria" < "mariana" (tamanhos diferentes – o de maior comprimento é o maior)
"antonio dos santos" < "antonio santos"
```

```
/* compara dois strings a e b e devolve 0 se a=b, 1 se a>b e -1 se a<b */
int compara (unsigned char a[], unsigned char b[]) {
    int i, k;
    k = tamanho (a);
    for (i = 0; i < k+1; i++)
        if (a[i] != b[i])
```

```
        if (a[i] > b[i]) return 1;
        else return -1;
    /* Se chegou aqui é porque todos eram iguais inclusive o (k+1)-ésimo
       que é o zero binário (fim do string).
       Note que é verdade mesmo se tamanho(a) diferente do tamanho(b) */
    return 0;
}

/* idem, outra solução */
int compara (unsigned char a[], unsigned char b[]) {
    int i = 0;
    while (a[i] != 0 && b[i] != 0)
        if (a[i] != b[i])
            if (a[i] > b[i]) return 1;
            else return -1;
    /* Se chegou aqui é porque um dos strings (ou os dois)
       chegou ao final. Em qualquer caso o i-esimo caracter
       decide se a==b, a>b ou a<b */
    if (a[i] == b[i]) return 0;
    else if (a[i] > b[i]) return 1;
    else return -1;
}

/* Idem devolvendo 0 se a=b, valor>0 se a>b e valor<0 se a<b
int compara (unsigned char a[], unsigned char b[]) {
    int i = 0;
    while (a[i] == b[i])
        if (a[i] == '\0') return 0;
        else i++;
    /* Se chegou aqui é porque são diferentes */
    return a[i] - b [i];
}
}
```

Outra solução para a comparação:

```
int comparar (unsigned char s[], unsigned char t[]) {
    int i;
    for(i=0;s[i]==t[i];i++) if(s[i]=='\0') return 0;
    return s[i]-t[i];
}
}
```

### **Letras maiúsculas, minúsculas e vogais acentuadas**

Como as letras maiúsculas são menores que as minúsculas, quando estas são misturadas numa string, a comparação fica confusa. Por exemplo:

“antonio” > “Antonio” e “antonio” > “anTonio”

Uma solução normalmente usada para evitar esse tipo de confusão é transformar todas as letras para maiúsculas antes de fazer a comparação.

```
/* Transforma as minúsculas em maiúsculas da string a */
```

```
void tudo_maiuscula(char a[]) {  
    int i = 0;  
    while (a[i] != 0 )  
        /* verifica se a[i] é minúscula */  
        if (a[i] >= 'a' && a[i] <= 'z')  
            /* Transforma em maiúscula */  
            a[i] = a[i] - 'a' + 'A';  
}
```

Agora vamos refazer a **compara** usando a função **tudo\_maiuscula**:

```
int novacompara (char a[], char b[]) {  
    int i = 0;  
    /* transforma a e b */  
    tudo_maiuscula(a);  
    tudo_maiuscula(b);  
    /* retorna o mesmo que compara (a, b) */  
    return compara(a, b);  
}
```

Um outro problema na comparação alfabética ocorre com as vogais acentuadas: á, ã, â, é, ê, í, ó, õ, ô, ú.

Cada uma delas tem o seu código numérico correspondente e a comparação pode ficar confusa.

A melhor solução para uma comparação alfabética mais limpa seria trocar todas as vogais acentuadas pelas não acentuadas, transformar para maiúsculas (ou minúsculas) e comparar.

Situação análoga ocorre com o ç (cedilha). O melhor é transformá-lo na letra c.

## Exercícios

P79) Escreva uma função `compacta (char a[], char b[])` que recebe o string a e devolve o string b eliminando os brancos de a.

P80) Idem `compacta (char a[])` que devolve no próprio a. Faça sem usar vetores auxiliares, isto é, o algoritmo deve eliminar os brancos movendo os caracteres para novas posições do próprio a.

P81) Idem `compacta (char a[])`, substituindo cadeias de mais de um branco, por um só branco.

P82) Escreva a função `int contapal (char a[])`, que devolve como resultado o número de palavras do string a. Uma palavra é uma seqüência de caracteres não brancos, precedida e seguida por brancos.

P82a) Escreva a função `int substring (char a[], char b[])`, que verifica se o string a é sub-string de b devolvendo 1 (sim) ou 0 (não).

Vamos explorar um pouco mais os exercícios P82 e P82a.



## Algoritmos de Busca de Palavras em Texto (“pattern matching”)

A busca de padrões dentro de um conjunto de informações tem uma grande aplicação em computação. São muitas as variações deste problema, desde procurar determinadas palavras ou sentenças em um texto até procurar um determinado objeto dentro de uma seqüência de bits que representam uma imagem. Todos eles se resumem a procurar certa seqüência de bits ou bytes dentro de uma seqüência maior de bits ou bytes.

Este é o algoritmo usado pelos programas buscadores da internet (ex: Google).

Vamos considerar a versão de procurar uma seqüência de bytes dentro de outra, ou ainda, procurar uma **palavra** dentro de um texto. **Palavra** deve ser entendida como uma seqüência qualquer de caracteres. Assim, a formulação do problema fica:

Dada uma seqüência  $a$  de  $m > 0$  bytes ( $a[1], \dots, a[m]$  ou  $a[1..m]$ ) **verificar quantas vezes** ela ocorre em uma seqüência  $b$  de  $n$  elementos ( $b[1], \dots, b[n]$  ou  $b[1..n]$ ).

### O algoritmo tradicional

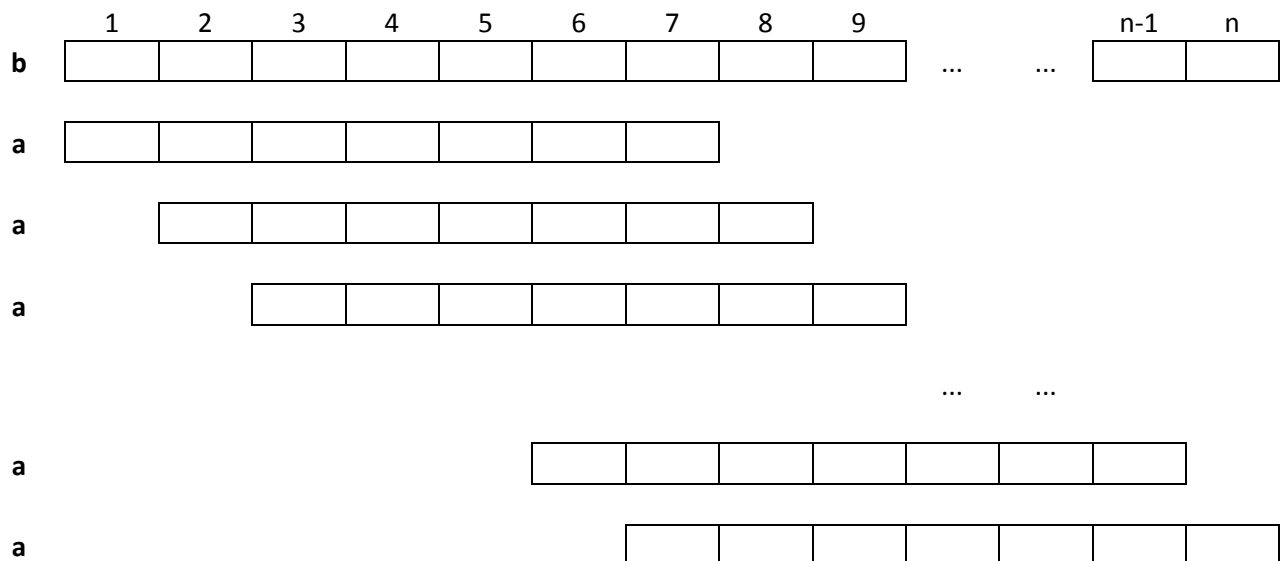
A solução mais trivial deste problema consiste então em comparar:

$a[1]$  com  $b[1]$ ;  $a[2]$  com  $b[2]$ ; ...  $a[m]$  com  $b[m]$

$a[1]$  com  $b[2]$ ;  $a[2]$  com  $b[3]$ ; ...  $a[m]$  com  $b[m+1]$

...

$a[1]$  com  $b[n-m+1]$ ;  $a[2]$  com  $b[n-m+2]$ ; ...  $a[m]$  com  $b[n]$



Exemplos:

**b - O alinhamento do pensamento provoca casamento**

**a - mento** – Ocorre 3 vezes

**a - n** – Ocorre 5 vezes

**a - casa** – Ocorre 1 vez

**a - ovo** – Ocorre 1 vez

**a - prova** – Ocorre 0 vezes

**b - ababababa**

**a - bab** – Ocorre 3 vezes

**a - abab** – Ocorre 3 vezes

**a - bababa** – Ocorre 2 vezes

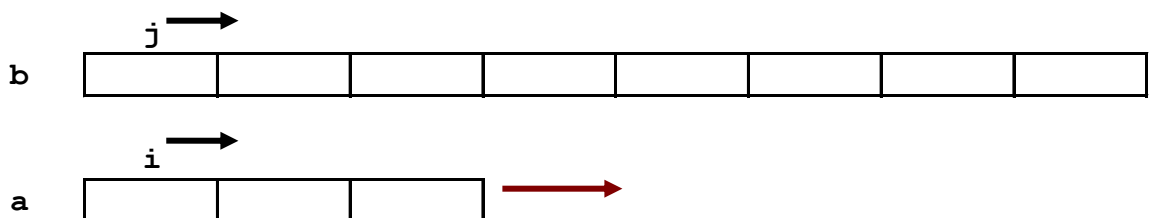
Nos algoritmos abaixo vamos considerar os índices começando do 1 e não do 0. Os elementos `a[0]` e `b[0]` serão ignorados.

Abaixo esta primeira solução:

```
int casapadrao1(char a[], int m, char b[], int n) {  
    int i, j, k, conta=0;  
    if (m<=0) return 0;  
    for (k=1; k<=n-m+1; k++) {  
        for (j=k, i=1; i<=m; j++, i++)  
            if (a[i] != b[j]) break;  
        if (i>m) conta++;  
    }  
    return conta;  
}
```

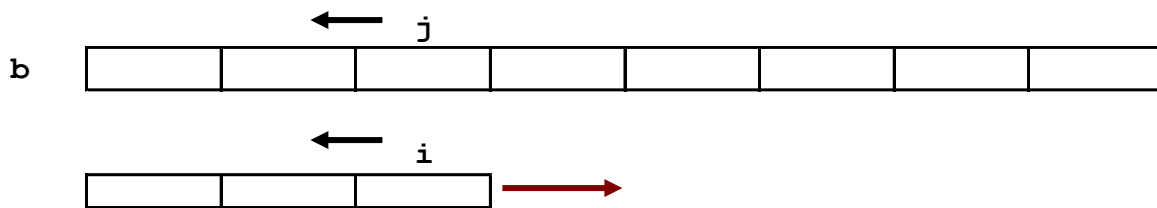
Na solução acima, `i` e `j` caminham da esquerda para a direita.

Também `a` se desloca da esquerda para a direita a cada nova tentativa.

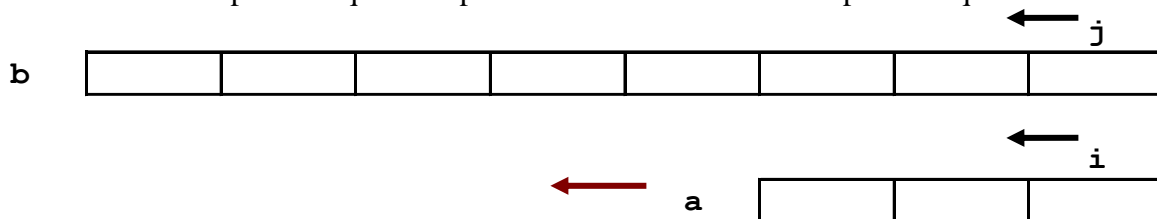


Podemos ter algumas variações, todas equivalentes:

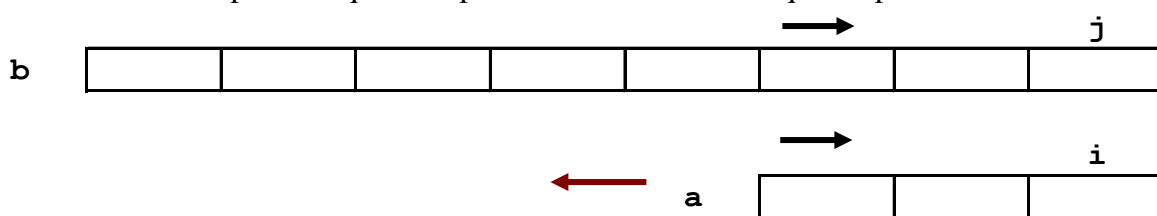
Procurando `a` em `b` da direita para a esquerda:



Varrendo b da direita para a esquerda e procurando a em b da direita para a esquerda:



Varrendo b da direita para a esquerda e procurando a em b da esquerda para a direita:



### Exercícios:

- 1) Adapte o algoritmo acima, para as 3 versões citadas.
- 2) Quantas comparações são feitas no mínimo e no máximo? Encontre seqüências a e b onde o mínimo e o máximo ocorrem.
- 3) Explique porque as versões acima são todas equivalentes.

### Outros algoritmos de busca de palavras em texto

O algoritmo consome um tempo proporcional a  $m \cdot n$ . Existem outros algoritmos mais eficientes que não veremos neste curso. Todos eles fazem uso de ocorrências anteriores de sub-cadeias para decidir a partir de qual caracter a busca será retomada.

- 1) Algoritmo de Boyer-Moore (1971)  
Também é proporcional a  $m \cdot n$ , porém no caso geral (médio) é mais eficiente.
- 2) Algoritmo de Knuth-Morris-Pratt [KNUTH D.E., MORRIS (Jr) J.H., PRATT V.R., 1977, Fast pattern matching in strings, *SIAM Journal on Computing* 6(1):323-350].  
O mais conhecido. É proporcional a  $m+n$ , portanto melhor que os anteriores.

### Funções pré-definidas com strings

Existem várias funções pré-definidas que mexem com cadeias de caracteres. Para usá-las é necessário um `#include <string.h>`. Veja algumas abaixo:

`char *strcpy(s, r)` – copia string `r` para a cadeia `s`. Devolve ponteiro para `s`.

`char *strncpy(s, r, n)` – copia no máximo `n` caracteres de `r` para `s`. Retorna ponteiro para `s`. Se `s` tiver menos que `n` caracteres preenche `s` com `'\0'`.

`char *strcat(s, r)` – concatena cadeia `r` no final da cadeia `s`. Retorna ponteiro para `s`.

`char *strncat(s, r, n)` – concatena no máximo `n` caracteres da cadeia `r` para a cadeia `s`. termina `s` com `'\0'`. Retorna ponteiro para `s`.

`int strcmp(s, r)` – compara a cadeia `s` com a cadeia `r`. Retorna `<0` se `s < r`, `0` se `s = r` e `>0` se `s > r`.

`int strncmp(s, r, n)` – compara no máximo `n` caracteres da cadeia `s` com a cadeia `r`. Retorna `<0` se `s < r`, `0` se `s = r` e `>0` se `s > r`.

`char *strchr(s, c)` – retorna ponteiro para a primeira ocorrência do caractere `c` na cadeia `s` ou `NULL` se não estiver presente.

`char *strrchr(s, c)` – retorna ponteiro para a última ocorrência do caractere `c` na cadeia `s` ou `NULL` se não estiver presente.

`int strspn(s, r)` – retorna tamanho do prefixo em `s` que coincide com `r`.

`int strcspn(s, r)` – retorna tamanho do prefixo em `s` que não coincide com `r`.

`char *strpbrk(s, r)` – retorna ponteiro para a primeira ocorrência na cadeia `s` de qualquer caractere na cadeia `r`, ou `NULL` se não achar.

`char *strstr(s, r)` – retorna ponteiro para a primeira ocorrência da cadeia `r` na cadeia `s`, ou `NULL` se não achar.

`int strlen(cs)` – retorna tamanho de `s`.

Embora pré-definidas, as funções acima tem uma implementação simples. Por exemplo:

`strlen(s)` é a função `tamanho(s)` definida acima.

`strncmp(s, r, n)` é a função `compara(s, r, n)` definida acima.

`strcpy(s, r)` é a função `move(s, r)` definida acima.

Quando se usa tais funções, é conveniente se levar em conta a eficiência dos algoritmos. O exemplo abaixo conta o número de brancos de uma string:

```
c = 0;  
for (i=0;i<strlen(a);i++)  
    if (a[i] == ' ') c++;
```

Melhor seria:

```
c = 0;  
k = strlen(a);  
for (i=0;i<k;i++)  
    if (a[i] == ' ') c++;
```

## Funções de comparação de caracteres

O argumento das funções abaixo é sempre um int. Portanto podemos passar um char como parâmetro. As funções devolvem um valor diferente de zero se forem verdadeiras, ou zero se forem falsas.

```
isupper ( c ) – se c é letra maiúscula.  
islower ( c ) – se c é letra minúscula.  
isalpha ( c ) – equivalente a isupper ( c ) || islower ( c )  
isdigit ( c ) – caractere entre '0' e '9'.  
isalnum ( c ) – equivalente a isalpha ( c ) || isdigit ( c )  
isspace ( c ) – se c é igual a branco.
```

## Tabela de códigos ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	\$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: [www.LookupTables.com](http://www.LookupTables.com)

## Tabela estendida de códigos ASCII

128	Ç	144	É	161	í	177	⌘	193	⌞	209	⌠	225	β	241	⊕
129	ü	145	æ	162	ó	178	⌘	194	⌟	210	⌡	226	Γ	242	⊖
130	é	146	Æ	163	ú	179		195	⌠	211	⌢	227	π	243	⊗
131	â	147	ô	164	ñ	180	†	196	–	212	⌣	228	Σ	244	∫
132	ä	148	ö	165	Ñ	181	‡	197	‡	213	⌤	229	σ	245	∫
133	à	149	ò	166	ª	182	‡	198	‡	214	⌥	230	μ	246	+
134	â	150	û	167	º	183	π	199	‡	215	⌦	231	τ	247	±
135	ç	151	ù	168	¿	184	π	200	⌧	216	⌧	232	Φ	248	°
136	ê	152	–	169	–	185	‡	201	⌨	217	⌨	233	⊙	249	·
137	ë	153	Ö	170	¬	186	‡	202	〈	218	〈	234	Ω	250	·
138	è	154	Û	171	½	187	π	203	〉	219	■	235	δ	251	√
139	ï	156	£	172	¼	188	‡	204	⌫	220	■	236	∞	252	–
140	î	157	¥	173	ı	189	‡	205	=	221	■	237	φ	253	²
141	ì	158	–	174	«	190	‡	206	≠	222	■	238	ε	254	■
142	Ä	159	f	175	»	191	γ	207	≠	223	■	239	∩	255	
143	Å	160	á	176	⌘	192	L	208	⌘	224	α	240	≡		

Source: [www.LookupTables.com](http://www.LookupTables.com)