



Estruturas de Dados 1

03 – *Strings*: Texto e Caracteres

Antonio Angelo de Souza Tartaglia
angelot@ifsp.edu.br

Estruturas de Dados 1



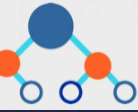
- I/O ou E/S pelo console
 - C é praticamente única no seu enfoque das operações de entrada/saída. A razão disso, é que a linguagem não define nenhuma palavra-chave que realize E/S;
 - Ao contrário, entrada de saída são efetuadas pelas funções disponíveis nas bibliotecas.
 - O sistema de E/S de C é uma parcela elegante da engenharia que oferece um flexível, porém, coeso mecanismo para transferir dados entre dispositivos. No entanto, o sistema de E/S de C é muito grande e envolve diversas funções diferentes.
 - Aqui veremos algumas; as mais utilizadas.

Estruturas de Dados 1



- I/O ou E/S pelo console
- Em C, temos E/S pelo console e por meio de arquivos. Tecnicamente, C faz pouca distinção entre entradas e saídas pelo console ou por arquivos. Contudo, conceitualmente elas são mundos muito diferentes;
- Neste momento cobriremos apenas E/S pelo console, definidas pelo padrão C ANSI - American National Standards Institute, como aquelas que realizam entradas pelo teclado (dispositivo *stdin* – *Standard input*), e a saída, ou impressão, para a tela
- Para conhecimento, entradas e saídas padrão podem, e muitas vezes o são, redirecionadas para outros dispositivos, como por exemplo, arquivos, impressoras e etc.
- <https://www.ansi.org>

Estruturas de Dados 1



- Ativando caracteres para o português com a biblioteca <locale.h>
 - Utilizamos a função `setlocale()`, para adaptar o compilador ao idioma local desejado. Esta função pertence a biblioteca <locale.h>, que trata de informações sobre a apresentação de alguns dados, tais como
 - Caracteres de acentuação específica;
 - Formato de números e valores monetários;
 - Formatação de data e hora.
 - `setlocale()` suporta dois argumentos, onde o primeiro, é definido o que se deseja alterar, e o segundo, para qual linguagem. Seu protótipo na biblioteca <locale.h> é:

```
char* setlocale(int categoria, const char *local);
```

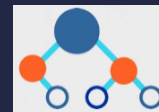


- Ativando caracteres para o português com a biblioteca <locale.h>
 - Para o argumento categoria, podemos utilizar:
 - LC_ALL – faz referência a todos os aspectos da localização (todos abaixo);
 - LC_COLLATE – especifica as regras de localidade para comparação de textos;
 - LC_CTYPE – especifica as regras para classificação e conversão de caracteres;
 - LC_MONETARY – especifica a notação monetária de uma localidade;
 - LC_NUMERIC – especifica a notação numérica da localidade;
 - LC_TIME – especifica a notação de data e tempo de uma localidade

```
char* setlocale(int categoria, const char *local);
```

```
#define LC_ALL      0
#define LC_COLLATE  1
#define LC_CTYPE    2
#define LC_MONETARY 3
#define LC_NUMERIC  4
#define LC_TIME     5
```

Estruturas de Dados 1



- Ativando caracteres para o português com a biblioteca <locale.h>
 - Se a localidade for uma *string* vazia “”, o nome do local será obtido a partir das variáveis de ambiente do sistema operacional, o que resolve o problema na maioria dos casos:

```
setlocale(LC_ALL, "");
```

- Para alterar tudo para o português explicitamente, podemos utilizar duas formas:

```
setlocale(LC_ALL, "pt_BR.UTF-8");
```

- ou:

```
setlocale(LC_ALL, "Portuguese");
```

- Ativando caracteres para o português com a biblioteca <locale.h>

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

```
int main() {
    setlocale(LC_ALL, "Portuguese");
    float valor = 2.45;
    char str[80] = "Agora podemos utilizar á, à, â, é, ç, ã, õ, e outros";
    printf("%s \n", str);
    printf("E exibir valores monetários com vírgula: %.2f\n\n\n", valor);
}
```

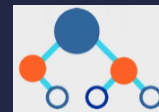
No código, continuamos a utilizar o separador "."

"C:\Users\angelot\Documents\Aulas\GRUEDA1\Aula 03 - Strings_texto e caracteres\material de apoio\locale.exe"

```
Agora podemos utilizar á, à, â, é, ç, ã, õ, e outros
E exibir valores monetários com vírgula: 2,45
```

```
Process returned 0 (0x0)   execution time : 0.060 s
Press any key to continue.
```

Mas na exibição, o operador ".", é automaticamente convertido para ",", que é o padrão pt - BR



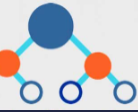
- Lendo e escrevendo caracteres

- A mais simples das funções E/S pelo console são **getchar()**, que lê um caractere do teclado, e **putchar()** que escreve um caractere na tela;
- A função **getchar()** espera até que uma tecla seja pressionada, e devolve o seu valor sem necessitar teclar *enter* para confirmação. A tecla pressionada é também automaticamente mostrada na tela.
- A função **putchar()** escreve seu argumento caractere na tela a partir da posição atual do cursor, abaixo os protótipos para **getchar()** e **putchar()**:

```
int getchar(void); //void nesse caso significa que nada recebe como parâmetro
```

```
int putchar(int <variável que tem o caractere>); //caractere é valor numérico
```

Todos os caracteres em C são armazenados como valores inteiros.
Consultar Tabela ASCII



- Lendo e escrevendo caracteres

- **getchar()** e **putchar()** pertencem à biblioteca **<stdio.h>**.

```
int getchar(void);
```

- Como podemos ver em seu protótipo, a função **getchar()** é declarada como retornando um inteiro. Contudo, seu retorno pode ser atribuído normalmente à uma variável do tipo **char**;

```
int putchar(int <variável que tem o caractere>);
```

- No caso de **putchar()**, apesar de ser declarada como recebendo um parâmetro do tipo inteiro, ela poderá ser chamada normalmente passando-se como argumento uma variável que contenha um caractere, ou seja, uma variável do tipo **char**.

Estruturas de Dados 1

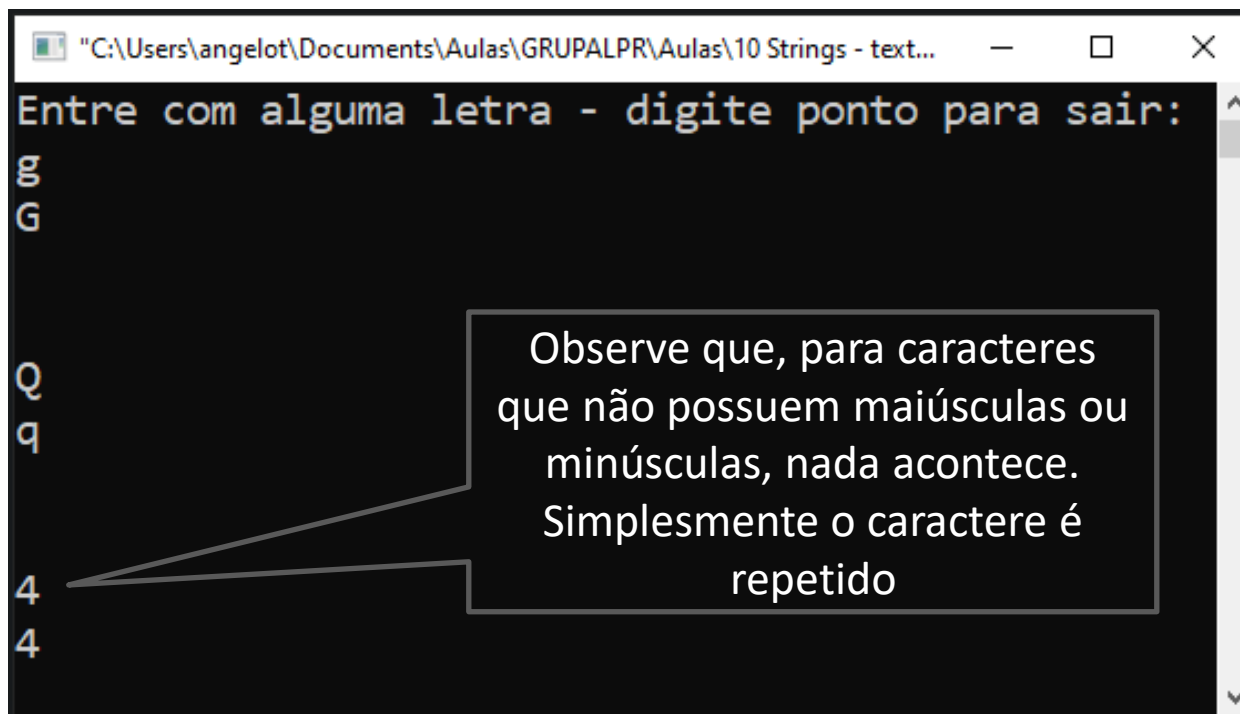


- Lendo e escrevendo caracteres

```
#include <stdio.h>
#include <ctype.h> //Contém funções e macros para manipulação de caracteres.

int main() {
    char ch;
    printf("Entre com alguma letra - digite ponto para sair: \n");
    do{
        ch = getchar();
        if(islower(ch)){
            ch = toupper(ch);
        }else{
            ch = tolower(ch);
        }
        putchar(ch);
    }while(ch != '.');
}
```

As funções `islower()`,
`toupper()` e `tolower()`,
pertencem à biblioteca
`ctype.h`



```
"C:\Users\angelot\Documents\Aulas\GRUPALPR\Aulas\10 Strings - text..."
Entre com alguma letra - digite ponto para sair:
g
G
Q
q
4
4
```

Observe que, para caracteres que não possuem maiúsculas ou minúsculas, nada acontece. Simplesmente o caractere é repetido

Estruturas de Dados 1



- A biblioteca **ctype.h** contém funções e macros para manipulação de caracteres.
- Utilizando as funções desta biblioteca podemos verificar se um caractere é numérico, ou se é maiúsculo, minúsculo, representa espaço em branco etc.
- Na listagem abaixo, podemos visualizar as principais funções de **ctype.h**.
 - **Funções para conversão de caracteres maiúsculos e minúsculos:**
 - **tolower()**
Converte o caractere em minúsculo
 - **toupper()**
Converte caractere minúsculo em maiúsculo.
 - **Funções para manipulação de caracteres**
 - **isalnum()**
Verifica se o caractere é alfanumérico
 - **isalpha()**
Verificar se o caractere é uma letra do alfabeto
 - **iscntrl()**
Verificar se o caractere é um caractere de controle
 - **isdigit()**
Verificar se o caractere é um dígito decimal
 - **isgraph()**
Verifica se o caractere tem representação gráfica
 - **islower()**
Verifica se o caractere é minúsculo
 - **isprint()**
Verifica se o caractere é imprimível.
 - **ispunct**
Verifica se o caractere é um ponto
 - **isspace()**
Verificar se o caractere é um espaço em branco
 - **isupper()**
Verifica se o caractere é uma letra maiúscula
 - **isxdigit()**
Verifica se o caractere é um dígito hexadecimal



- Lendo e escrevendo *Strings* – definição de *string*

- ***String*** é o nome que usamos para definir uma sequência de caracteres adjacentes na memória do computador. Essa sequência de caracteres, que pode ser uma palavra ou frase, é armazenada na memória do computador em forma de *array*, ou vetor, do tipo **char**;
- Por ser a *string* um *array* de caracteres, sua declaração segue as mesmas regras da declaração de um *array*, ou vetor convencional:

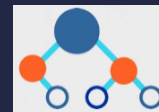
```
char str[6];
```

- Esta declaração cria na memória do computador uma *string* (*array* ou vetor de caracteres) de nome **str** e tamanho igual a 6. Apesar de ser um *array*, devemos ficar atentos para o fato de que as *strings* têm no elemento seguinte à última letra da palavra/frase armazenada, o caractere '**\0**';
- Isso ocorre porque podemos definir um vetor para armazenar uma *string* com o tamanho maior do que a palavra a ser armazenada.
- Este caractere, '**\0**', também chamado de **NULL**, indica o fim de uma sequência de caracteres.

- Lendo e escrevendo *Strings* – definição de *string*
 - Imagine uma *string* definida com o tamanho de 30 caracteres, mas utilizada apenas para armazenar um nome com 13 caracteres “**José da Silva**”. Nesse caso, teremos 17 posições não utilizadas e que estão preenchidas com “**lixo de memória**” (um valor qualquer). Obviamente não queremos que todo esse “**lixo**” seja impresso em tela quando a *string* for exibida.
 - Dessa forma, o caractere ‘ `\0` ’ indica o fim da sequência de caracteres e o início das posições restantes da *string* que não estão sendo utilizadas no momento:

J	o	s	é		d	a		S	i	l	v	a	\0	¥	#	*	£	¹	¬"	M	€	w	g	5	!	u	»	■	Å
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

- Portanto, quando uma *string* é definida, ou declarada, é necessário sempre se considerar o caractere de controle ‘ `\0` ’, ou seja, tamanho necessário para a *string* + 1.



- Lendo e escrevendo *Strings* – inicializando uma *string*

- Uma *string* pode ser lida do teclado ou já ser **definida** com um valor inicial. Para sua inicialização, pode-se usar o mesmo princípio definido na inicialização de vetores e matrizes:

```
char str[30] = {'J','o','s','é',' ',' ','d','a',' ',' ','S','i','l','v','a','\0'};
```

- Mas isso não é muito prático, por isso, a inicialização de *strings* também pode ser feita por meio de aspas duplas:

```
char str[30] = "Jose da Silva";
```

- Essa forma de inicialização possui a vantagem de já inserir o caractere ' \0 ' automaticamente no final da *string*;
- Também como nos vetores e matrizes, podemos acessar individualmente cada elemento da *string* utilizando seu índice, por exemplo, **str[5]** acessa o elemento ' d ' da *string* **str**.



- Lendo e escrevendo *Strings*
 - O próximo passo no tratamento de entradas e saídas através do console, são as funções **gets()** e **puts()**. Elas permitem ler e escrever *strings* de caracteres no console;
 - **gets()** lê uma *string* de caracteres inserida pelo teclado e coloca-a na variável passada como argumento. Pode-se digitar no teclado a *string* desejada, até que a tecla **enter** seja pressionada, lembrando que a tecla **enter** também é um caractere, '**\n**' – n1 ou nova linha.
 - O '**\n**' não se torna parte da *string*; em seu lugar é colocado o terminador nulo '**\0**', e então **gets()** retorna o conteúdo digitado;
 - É possível a correção de caracteres digitados erradamente (antes de teclar **enter**) utilizando a tecla **BACKSPACE**, uma vez que os caracteres ficam armazenados no *buffer* de teclado antes de serem armazenados definitivamente na variável.

- Lendo e escrevendo *Strings*
 - O protótipo de **gets()**, que pertence a biblioteca **<stdio.h>** é:

```
char *gets(char *str);
```

- Onde **str** é uma matriz de caracteres que recebe os caracteres enviados pelo usuário.

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

```
int main() {
```

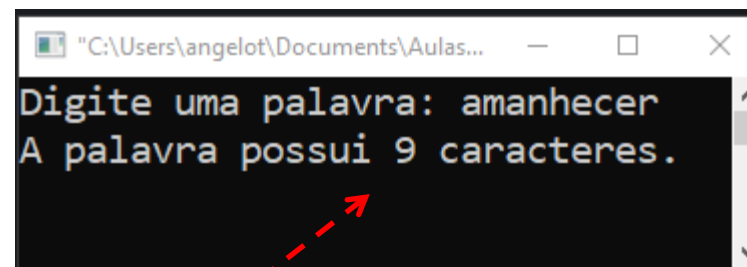
```
    char str[80];
```

```
    printf("Digite uma palavra: ");
```

```
    gets(str);
```

```
    printf("A palavra possui %d caracteres.\n\n", strlen(str));
```

```
}
```



- A função **strlen()**, que pertence à biblioteca **string.h**, devolve o tamanho em caracteres da *string* passada como argumento.



- Lendo e escrevendo *Strings*
 - Basicamente, para ler uma *string* do teclado utilizamos **gets()**. No entanto, existe uma outra função que utilizada de forma adequada, também permite a leitura de *strings* do teclado de forma mais segura. Essa função é **fgets()**. Também pertencendo a biblioteca **<stdio.h>**, seu protótipo é:

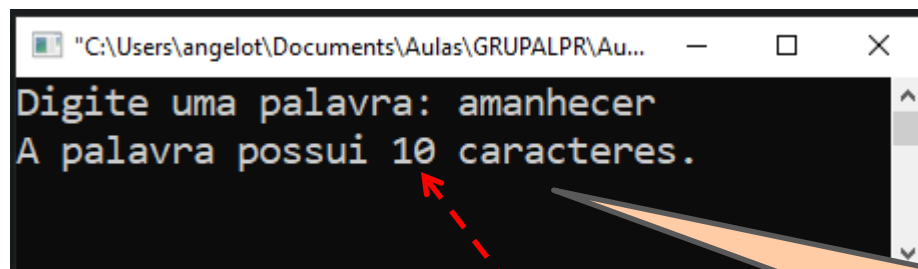
```
char *fgets(char *str, int tamanho, FILE *fp);
```
 - Ela recebe 3 parâmetros como entrada:
 - **str**: a variável vetor de caracteres que receberá a *string* lida do teclado;
 - **tamanho**: o limite máximo de caracteres a serem lidos, e;
 - **fp**: a variável que está associada ao local (arquivo ou teclado), de onde a *string* será lida.
 - Assim como **gets()**, **fgets()** lê a *string* do teclado até que um caractere de nova linha (**enter**) seja lido, sendo que este caractere ' \n ', **fará parte da nova string**, ao contrário de **gets()**, que não o incorpora.

- Lendo e escrevendo *Strings*
 - Diferente de **gets()**, **fgets()** especifica um tamanho máximo de entrada para a *string* e a lê até que um '**\n**' (**enter**) seja lido, ou **tamanho - 1** caracteres tenham sido lidos. Isso evita o estouro de *buffer*, que ocorre quando se tenta ler algo maior do que o que se pode ser armazenado na *string*;

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

```
int main() {
```

```
    char str[80];
    printf("Digite uma palavra: ");
    fgets(str, 79, stdin);
    printf("A palavra possui %d caracteres.\n\n", strlen(str));
}
```



Com **fgets()**, o caractere de nova linha ('**\n**') fará parte da *string*, o que não ocorria com o **gets()**.



- Lendo e escrevendo *Strings*

- A função **puts()** escreve seu argumento *string* na tela, seguido por uma nova linha, ou seja, um ' `\n` '. Também pertencendo a biblioteca `<stdio.h>`, seu protótipo é:

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

- **puts()** reconhece os mesmos códigos de barra invertida (`\`) aplicados à **printf()**, como por exemplo o ' `\t` ' para tabulação ou o ' `\n` ' para nova linha;
- Uma chamada a **puts()** requer bem menos tempo do que a mesma chamada a **printf()** porque **puts()** pode escrever **apenas strings** de caractere, não podendo escrever números ou efetuar conversões de formato. Portanto, **puts()** ocupa menos espaço, e é executada mais rapidamente que **printf()**. Por essa razão, a função **puts()** é frequentemente utilizada quando é importante ter um código altamente otimizado. Uma chamada passando a *string* "alo" como argumento:

```
puts("alo");
```

```
alo      //saída apresentada no console
```

- Entrada/saída formatada com **printf()** e **scanf()**

- As funções **printf()** e seu complemento **scanf()** realizam entrada e saída formatada, isto é, **printf()** escreve dados na tela (imprime no vídeo) e **scanf()** lê dados do teclado. As duas funções podem operar em qualquer dos tipos de dados intrínsecos, incluindo caracteres, *strings* e números;

- O protótipo de **printf()** na biblioteca **<stdio.h>** é:

```
int printf(const char *<string_de_controle>, ...);
```

printf() pode receber um número variável de argumentos. Essa quantidade é definida pelos comandos de controle. Um comando para cada argumento.

- **printf()** devolverá o número de caracteres escritos ou um número negativo em caso de erro;
- A *string_de_controle* consiste em dois tipos de itens. O primeiro tipo é formado por caracteres que serão impressos na tela. O segundo contém comandos de formatos que definem a maneira pela qual os argumentos subsequentes serão mostrados.

- Entrada/saída formatada com **printf()** e **scanf()**
 - Um comando de controle começa com um símbolo percentual (%) e é seguido pelo código do formato. Deve haver o mesmo número de argumentos e de comandos de formato, e estes são combinados na ordem da esquerda para a direita. Por exemplo, a seguinte chamada a **printf()**:

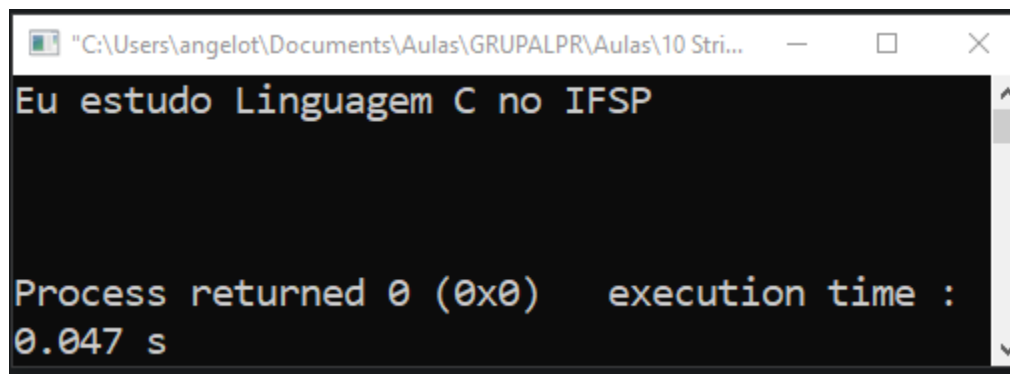
```
#include <stdio.h>

int main() {

    printf("Eu %s Linguagem %c no %s \n\n", "estudo", 'C', "IFSP");

}
```

- Gera a saída a seguir na tela:



```
"C:\Users\angelot\Documents\Aulas\GRUPALPR\Aulas\10 Stri...
Eu estudo Linguagem C no IFSP

Process returned 0 (0x0)   execution time :
0.047 s
```

Estruturas de Dados 1



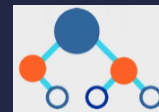
- Entrada/saída formatada com **printf()** e **scanf()**
 - Comandos de formato e sequencias de escape para **printf()**:

Código	Formato
%c	Caractere
%d	Inteiros decimais com sinal
%i	Inteiros decimais com sinal
%e	Notação científica (e minúsculo)
%E	Notação científica (E maiúsculo)
%f	Ponto flutuante decimal
%g	Usa %e ou %f, o que for mais curto
%G	Usa %E ou %F, o que for mais curto
%o	Octal sem sinal
%s	String de caracteres
%u	Inteiros decimais sem sinal
%x	Hexadecimal sem sinal (letras minúsculas)
%X	Hexadecimal sem sinal (letras maiúsculas)
%p	Apresenta um ponteiro (endereço de memória)

Sequência de escape	Representa
\a	Campainha (alerta)
\b	Backspace
\f	Avanço de página
\n	Nova linha (o mesmo que a tecla enter)
\r	Retorno de carro
\t	Tabulação horizontal
\v	Tabulação vertical
\'	Aspas simples
\"	Aspas duplas
\\	Barra invertida
%%	Caractere Porcentagem
\?	Ponto de interrogação literal
\ooo	Caractere ASCII em notação octal
\x hh	Caractere ASCII em notação hexadecimal



- Entrada/saída formatada com **printf()** e **scanf()**
 - Para escrever um caractere individual, é utilizado o comando de controle e o código de formatação '**%c**'. Isso faz com que o seu argumento associado seja escrito sem modificações na tela. Para se escrever uma *string*, utiliza-se '**%s**'.
 - Para se escrever valores numéricos, pode-se utilizar '**%d**' ou '**%i**' para indicar um número inteiro com sinal (+ ou -);
 - Para um valor sem sinal, é utilizado o '**%u**'; Os comandos '**%e**' e '**%E**', instruem a função **printf()** a exibir o argumento passado, que é do tipo **double**, em notação científica, e teremos como saída:
 - `1e+06` -> 10^6 `1e-03` -> 10^{-3}
 - `1E-02` -> 10^{-2} `1E+09` -> 10^9

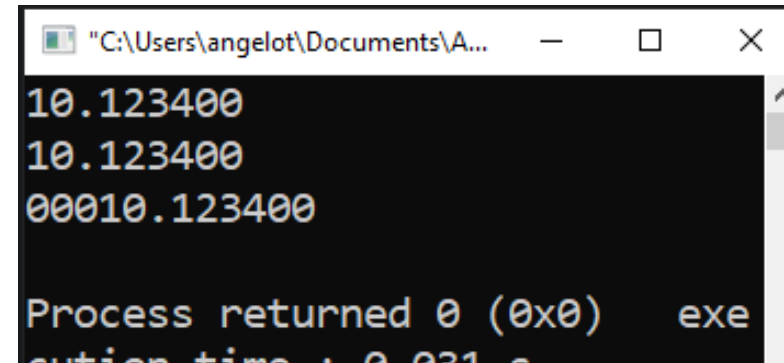


- Entrada/saída formatada com `printf()` e `scanf()`
 - Especificador de largura mínima de campo
 - Um número colocado entre o símbolo ‘%’ e o código de formato, age como um especificador de largura mínima de campo. Isso preenche a saída com espaços, para assegurar que ela atinja um certo comprimento mínimo;
 - Se a *string*, ou número, for maior do que o mínimo, ela será escrita por inteiro;
 - O preenchimento padrão é feito com espaços. Se for necessário o preenchimento com 0s (zeros), deve-se colocar o caractere ‘0’ antes do especificador de largura mínima de campo. Por exemplo, ‘%05d’ preencherá um número que tenha menos de 5 dígitos com 0s de forma que seu comprimento total seja de 5 caracteres.

- Entrada/saída formatada com `printf()` e `scanf()`

```
#include <stdio.h>

int main() {
    double item = 10.1234;
    printf("%f\n", item);
    printf("%3f\n", item);
    printf("%012f\n", item);
}
```



```
"C:\Users\angelot\Documents\A...  -  □  X
10.123400
10.123400
00010.123400

Process returned 0 (0x0)   exe
cutio n time : 0.031 s
```

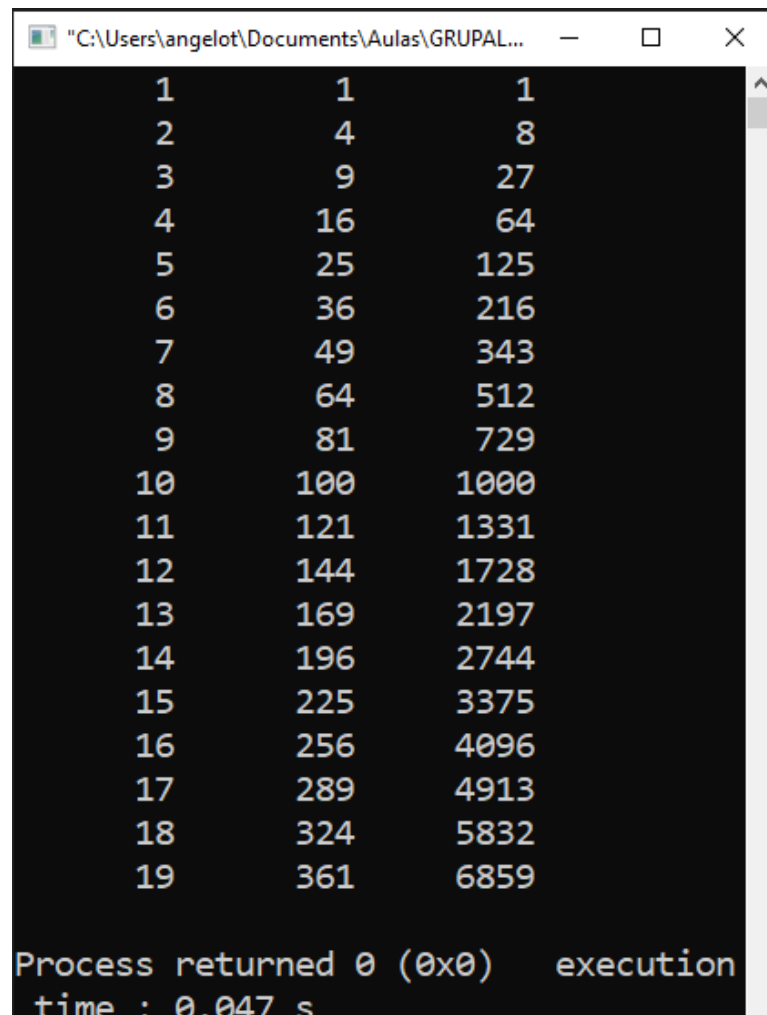
- O modificador de largura de campo, normalmente é utilizado para produzir tabelas em que as colunas são alinhadas. Por exemplo, o próximo programa produz uma tabela com os quadrados e os cubos dos números de 1 a 19.

Estruturas de Dados 1

- Entrada/saída formatada com `printf()` e `scanf()`

```
#include <stdio.h>

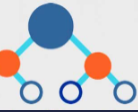
int main(){
    int i;
    //mostra tabela quadrados e cubos
    for(i = 1; i < 20; i++){
        printf("%8d %8d %8d\n", i, i*i, i*i*i);
    }
}
```



1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000
11	121	1331
12	144	1728
13	169	2197
14	196	2744
15	225	3375
16	256	4096
17	289	4913
18	324	5832
19	361	6859

Process returned 0 (0x0) execution time : 0.047 s



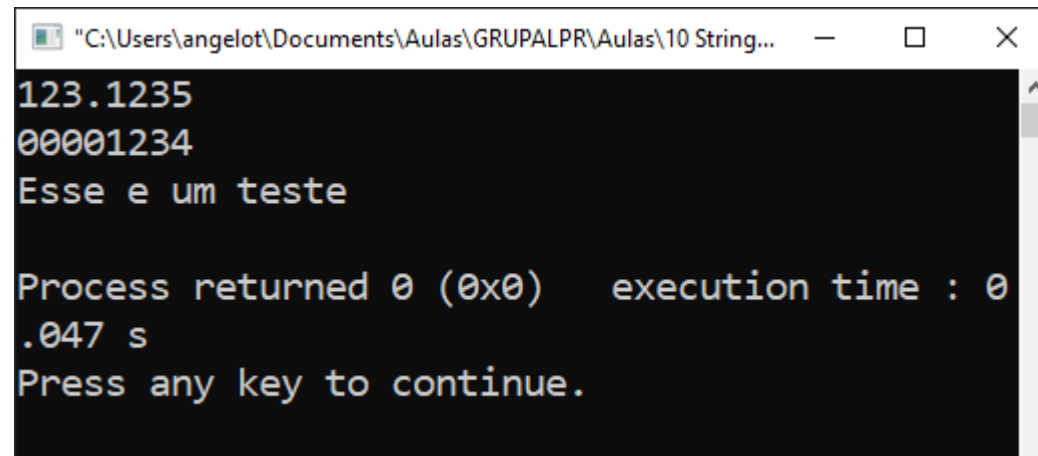


- Entrada/saída formatada com `printf()` e `scanf()`
 - O especificador de precisão segue o especificador de largura mínima de campo (se houver algum), consistindo em um ponto seguido de um número inteiro. O seu significado exato depende do tipo de dado a que está sendo aplicado;
 - Quando é aplicado a dados de ponto flutuante, ele determina o quantidade de casas decimais a serem exibidas. Por exemplo, ‘`%10.4f`’ exibe um número com pelo menos **10** caracteres que possui dentre esses **10** caracteres, **4** casas decimais.
 - Quando aplicado a *strings*, determina o comprimento máximo do campo. Por exemplo, ‘`%5.7s`’ exibe uma *string* de pelo menos **5** e no máximo **7** caracteres. Se a *string* for maior do que a largura máxima do campo, os caracteres finais serão truncados;
 - Quando aplicado a tipos inteiros, determina o número mínimo de dígitos que aparecerão para cada número. Zeros serão adicionados para completar o número de dígitos.

- Entrada/saída formatada com `printf()` e `scanf()`

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

```
int main() {  
    printf("%.4f\n", 123.1234567890);  
    printf("%.8d\n", 1234);  
    printf("%10.15s\n", "Esse e um teste simples");  
}
```

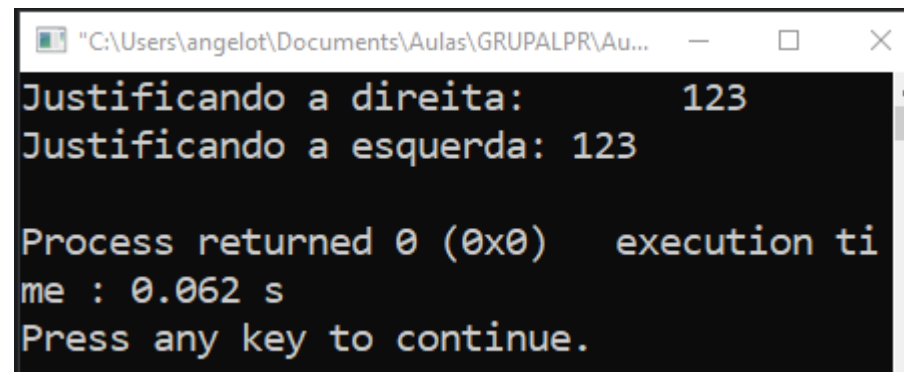


```
"C:\Users\angelot\Documents\Aulas\GRUPALPR\Aulas\10 String..."  
123.1235  
00001234  
Esse e um teste  
  
Process returned 0 (0x0) execution time : 0.047 s  
Press any key to continue.
```

- Entrada/saída formatada com `printf()` e `scanf()`
 - Justificando a saída
 - Por padrão, toda saída é justificada à direita. Isto é, se a largura do campo for maior que os dados escritos, os dados serão colocados na extremidade direita do campo. A saída pode ser justificada à esquerda colocando-se um sinal de subtração imediatamente após o '%'. Por exemplo, '%-10.2f' justifica à esquerda um número de ponto flutuante com duas casas decimais em um campo de 10 caracteres:

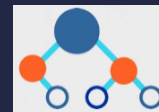
```
#include <stdio.h>

int main() {
    printf("Justificando a direita:  %8d\n", 123);
    printf("Justificando a esquerda: %-8d\n", 123);
}
```



```
"C:\Users\angelot\Documents\Aulas\GRUPALPR\Au...
Justificando a direita:      123
Justificando a esquerda: 123

Process returned 0 (0x0)   execution ti
me : 0.062 s
Press any key to continue.
```



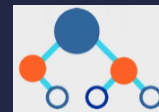
- Entrada/saída formatada com **printf()** e **scanf()**
 - **scanf()** é a rotina de entrada pelo console de uso geral. Ela pode ler todos os tipos de dados intrínsecos e converte automaticamente números ao formato interno apropriado. É muito parecida com o inverso de **printf()**. Faz parte da biblioteca **<stdio.h>**, e seu protótipo é:

```
int scanf(const char *string_de_controle, ...);
```
 - **scanf()** devolve o número de itens de dados que foi atribuído, com êxito, a um valor. A *string* de controle determina como os valores são lidos para as variáveis apontadas na lista de argumentos.
 - A *string* de controle consiste em 3 classificações de caracteres:
 - Especificadores de formato;
 - Caracteres de espaço em branco;
 - Caracteres de espaço não-branco.

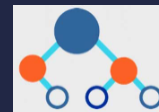


- Entrada/saída formatada com **printf()** e **scanf()**
 - Especificadores de formato – são precedidos por um sinal ‘%’ e informam ao **scanf()** que tipo de dado deve ser lido imediatamente após. Listados abaixo, estes especificadores coincidem , na ordem da esquerda para a direita, com os argumentos na lista de argumentos:

Código	Formato
%c	Lê um caractere
%d	Lê um Inteiro decimal
%i	Lê um Inteiro decimal
%e	Lê um número em ponto flutuante
%f	Lê um número em ponto flutuante
%g	Lê um número em ponto flutuante
%o	Lê um número octal
%s	Lê uma <i>string</i> de caracteres
%u	Lê um inteiro sem sinal
%x	Lê um número hexadecimal
%p	Lê um um ponteiro (endereço de memória)



- Entrada/saída formatada com **printf()** e **scanf()**
 - Para ler um número inteiro, deve-se utilizar os especificadores ' %d ' ou ' %i '. Esses especificadores fazem exatamente a mesma coisa, são incluídos para compatibilidade com versões mais antigas de C;
 - Para ler um número de ponto flutuante, representado em notação científica ou padrão, deve-se utilizar ' %e ', ' %f ' ou ' %g '. Novamente, eles fazem exatamente a mesma coisa, são incluídos por questões de compatibilidade com outras versões de C;
 - A função **scanf()** termina a leitura de um valor (número ou caractere), quando o **primeiro caractere não numérico** é encontrado;
 - Vimos anteriormente que é possível ler caracteres individuais utilizando **getchar()**. Mas **scanf()** também pode ser utilizada para ler um caractere, utilizando-se o especificador de formato ' %c '.



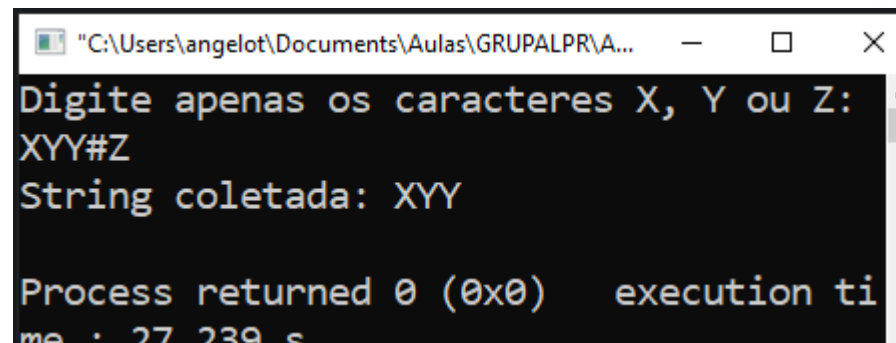
- Entrada/saída formatada com **printf()** e **scanf()**
 - Lendo *strings* – **scanf()** também pode ser utilizada para a leitura de uma *string* do *stream* (fluxo) de entrada, usando o especificador de formato ‘**%s**’. Este especificador faz com que **scanf()** leia caracteres até que seja encontrado um caractere de espaço em branco ou “**n1**” (*enter*). Os caracteres lidos são colocados então em uma matriz ou *array* de caracteres que foi passado à função **scanf()** como argumento. A *string* resultante, armazenada no argumento passado, terá o caractere de terminação ‘**\0**’.
 - Para **scanf()**, um caractere de espaço em branco é um espaço (*space* - 32 na tabela ASCII), um *enter* (**n1** - 10 na tabela ASCII) ou uma tabulação (**ht** - 9 na tabela ASCII);
 - Ao contrario de **gets()**, que lê uma *string* até que seja digitado um *enter*, **scanf()** lê a *string* até encontrar o primeiro não alfanumérico. Isso significa que **scanf()** não pode ser usada para ler uma *string* como “**isto é um teste**”, porque ao primeiro espaço encontrado (que não é alfanumérico), terminaria o processo de leitura, e seria coletada apenas a palavra “**isto**”.



- Entrada/saída formatada com **printf()** e **scanf()**
 - *scanset* – Um *scanset* define um conjunto de caracteres que pode ser lido por **scanf()**. É definido colocando-se uma *string* dos caracteres a serem procurados, entre colchetes. O colchete deve ser precedido por '**%**'. Por exemplo, o seguinte *scanset* informa ao **scanf()** para ler apenas os seguintes caracteres X, Y, Z:

```
#include <stdio.h>

int main() {
    char str[5];
    printf("Digite apenas os caracteres X, Y ou Z: \n");
    scanf(" %[XYZ]", &str);
    printf("String coletada: %s\n", str);
}
```



- Quando *scanset* é utilizado, **scanf()** continua a ler caracteres até que encontre um que não pertença ao *scanset*. Se o primeiro caractere do *scanset* for '**^**', instrui ao **scanf()** para aceitar qualquer símbolo que não esteja no *scanset*, invertendo o processo.

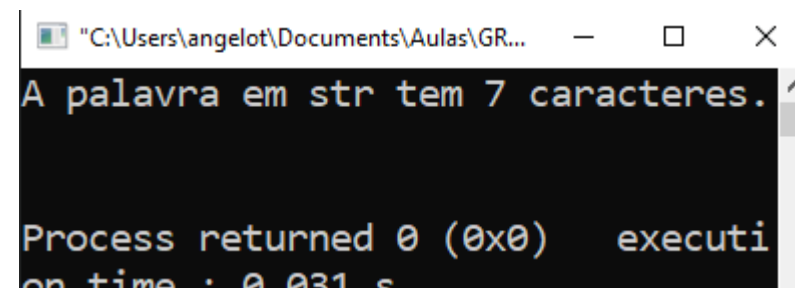


- Funções para manipulação de strings

- A biblioteca padrão da linguagem C possui funções especialmente desenvolvidas para manipulação de *strings* em seu componente **<string.h>**. Vejamos algumas:
- **strlen()** – retorna o número de caracteres que existem antes do ‘\0’, e não o tamanho do *array* ou matriz no qual a *string* está armazenada.

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

int main() {
    char str[15] = "tamanho"; //tamanho = 7 caracteres
    int qtd = strlen(str);
    printf("A palavra em str tem %d caracteres.\n", qtd);
}
```



- Basicamente, **strlen()** recebe como argumento uma *string* e retorna o número de caracteres armazenados nesta *string* passada como argumento.

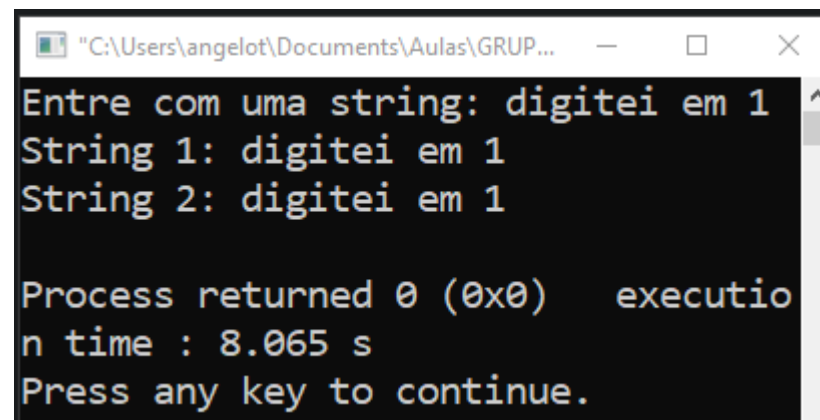
- Funções para manipulação de strings

- Copiando uma *string* – Uma *string*, é um *array*, ou vetor. A linguagem C não suporta a atribuição de um *array* para outro. Nesse sentido, a única maneira de atribuir o conteúdo de uma *string* a outra é a cópia de elemento por elemento, de uma *string* para outra. C possui uma função pronta para essa tarefa, **strcpy()**, que copia o segundo argumento para o primeiro argumento, e que pertence a biblioteca **<string.h>**:

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

O compilador entende a falta de **string.h** e a inclui automaticamente, porém, gera um *warning* solicitando sua inclusão.

```
int main() {  
    char str1[100], str2[100];  
    printf("Entre com uma string: ");  
    gets(str1);  
    strcpy(str2, str1);  
    printf("String 1: %s\n", str1);  
    printf("String 2: %s\n", str2);  
}
```



```
"C:\Users\angelot\Documents\Aulas\GRUP...  -  □  ×  
Entre com uma string: digitei em 1  
String 1: digitei em 1  
String 2: digitei em 1  
  
Process returned 0 (0x0)   executio  
n time : 8.065 s  
Press any key to continue.
```



- Funções para manipulação de *strings*

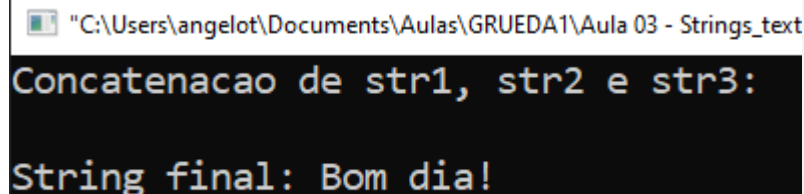
- Concatenando *strings* – Essa é uma tarefa bastante comum ao se trabalhar com *strings*. Basicamente, essa operação consiste em copiar uma *string* para o final da outra *string*, unindo as duas em uma só *string*. Para isso utilizamos a função **strcat()**, que pertence à biblioteca **<string.h>** e seu protótipo é:

```
char* strcat(char* destino, char* origem);
```

- **strcat()** copia a sequência de caracteres contida em **origem** para o final da *string* **destino**, e retorna a *string* **destino**, o primeiro caractere da *string* **origem**, sobrescreve o caractere '**\0**' de **destino**. Exemplo:

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

int main() {
    char str1[10] = "Bom", str2[10] = " ", str3[10] = "dia!";
    printf("Concatenacao de str1, str2 e str3: ");
    strcat(str1, str2);
    strcat(str1, str3);
    printf("\n\nString final: %s\n", str1);
}
```



```
"C:\Users\angelot\Documents\Aulas\GRUEDA1\Aula 03 - Strings_text
Concatenacao de str1, str2 e str3:
String final: Bom dia!
```

- Funções para manipulação de *strings*

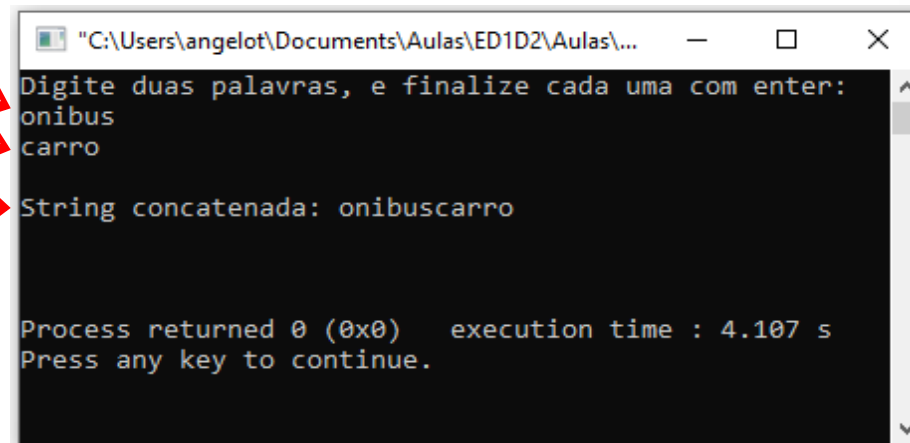
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main() {
    char string1[20], string2[20];
    printf("Digite duas palavras, e finalize cada uma com enter:\n");
    gets(string1);
    gets(string2);
    //Concatenando Strings
    strcat(string1, string2);
    printf("\nString concatenada: %s", string1);
    printf("\n\n\n");
}
```

O N I B U S \0

C A R R O \0

O N I B U S C A R R O \0



```
"C:\Users\angelot\Documents\Aulas\ED1D2\Aulas\..."
Digite duas palavras, e finalize cada uma com enter:
onibus
carro

String concatenada: onibuscarrro

Process returned 0 (0x0)   execution time : 4.107 s
Press any key to continue.
```

- Funções para manipulação de *strings*

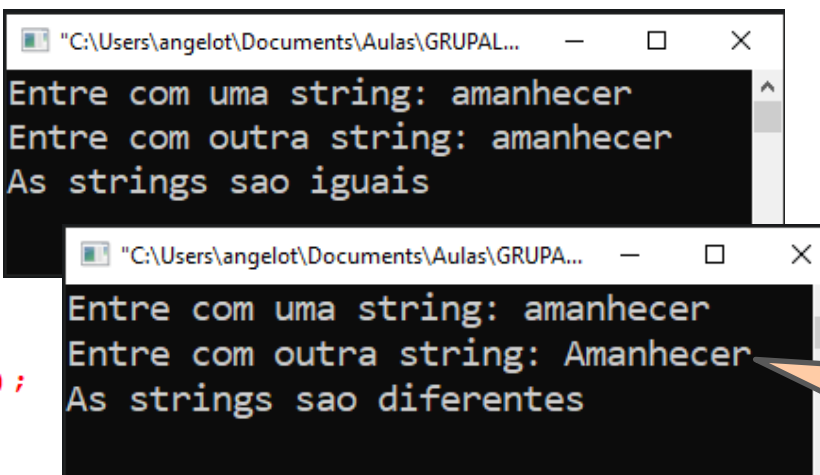
- Comparando duas *strings* – Da mesma forma que o operador de atribuição não funciona para *strings*, o mesmo ocorre com operadores relacionais usados para comparar duas *strings*. Assim para este tipo de comparação existe a função **strcmp()**, seu protótipo na biblioteca **<string.h>**:

```
int strcmp(char *str1, char *str2);
```

- strcmp()** compara posição a posição as duas *strings* (str1 e str2) e retorna um valor inteiro igual a zero, no caso de igualdade, e um valor diferente de zero em caso de diferença, como abaixo:

```
#include <stdio.h>

int main() {
    char str1[100], str2[100];
    printf("Entre com uma string: ");
    gets(str1);
    printf("Entre com outra string: ");
    gets(str2);
    if(!strcmp(str1, str2)){
        printf("As strings sao iguais\n");
    }else{
        printf("As strings sao diferentes\n");
    }
    system("pause");
}
```



```
strcmp("bb","aa") == 1
strcmp("aa","bb") == -1
strcmp("bb","bb") == 0
strcmp("aa","aa") == 0
```

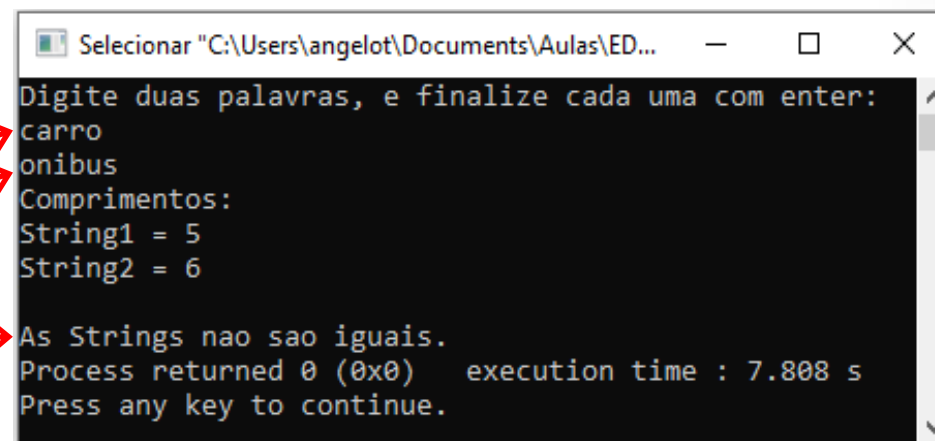
strcmp() é *case-sensitive*! Ela faz diferenciação entre maiúsculas e minúsculas!

- Funções para manipulação de *strings*

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

```
int main(){
    char string1[20], string2[20];
    int tamString1, tamString2;
    printf("Digite duas palavras, e finalize cada uma com enter:\n");
    gets(string1);
    gets(string2);
    //obtendo o tamanho das Strings
    tamString1 = strlen(string1);
    tamString2 = strlen(string2);

    printf("Comprimentos: \nString1 = %d\nString2 = %d", tamString1, tamString2);
    //compara as Strings
    if(!strcmp(string1, string2)){
        printf("\n\nAs Strings sao iguais!");
    }else{
        printf("\n\nAs Strings nao sao iguais.");
    }
}
```



```
Selecionar "C:\Users\angelot\Documents\Aulas\ED...
Digite duas palavras, e finalize cada uma com enter:
carro
onibus
Comprimentos:
String1 = 5
String2 = 6
As Strings nao sao iguais.
Process returned 0 (0x0) execution time : 7.808 s
Press any key to continue.
```

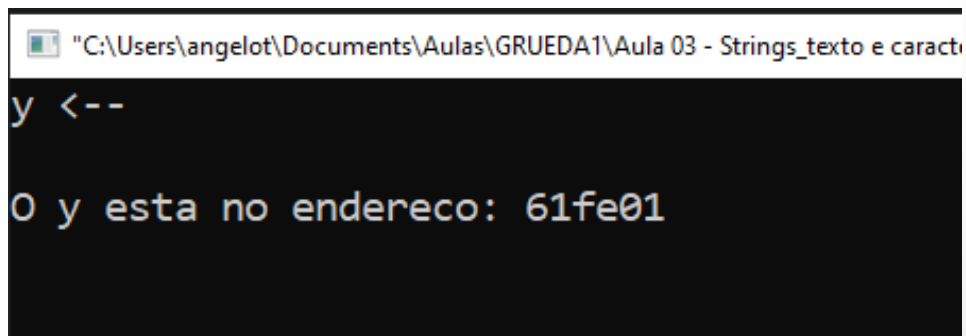

- Funções para manipulação de *strings*

- Busca dentro de *string* – A função **strchr()** devolve a localização (endereço de memória), ou ponteiro, para a primeira ocorrência do caractere buscado (**ch**) dentro da *string* alvo. Se não encontrada, **strchr()** devolve um valor nulo, que é igual a zero. Seu protótipo na biblioteca `<string.h>`:

```
char *strchr(const char *str, int ch);
```

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

int main() {
    char *p, letra = 'y';
    char str[30] = "procura-se o --> y <--";
    p = strchr(str, letra);
    printf(p); //imprime a partir da posição encontrada
    printf("\n\nO y esta no endereco: %x \n\n\n", p);
}
```



```
"C:\Users\angelot\Documents\Aulas\GRUEDA1\Aula 03 - Strings_texto e caract...
y <--
O y esta no endereco: 61fe01
```



- Funções para manipulação de *strings*

- Busca dentro de *string* – A função **strstr()** devolve a localização (endereço de memória), ou ponteiro, para a primeira ocorrência da *string* buscada dentro da *string* alvo. Se não encontrada, strstr() devolve um valor nulo, que é igual a zero. Seu protótipo na biblioteca <string.h>:

```
char* strstr(const char *alvo, const char *buscada);
```

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

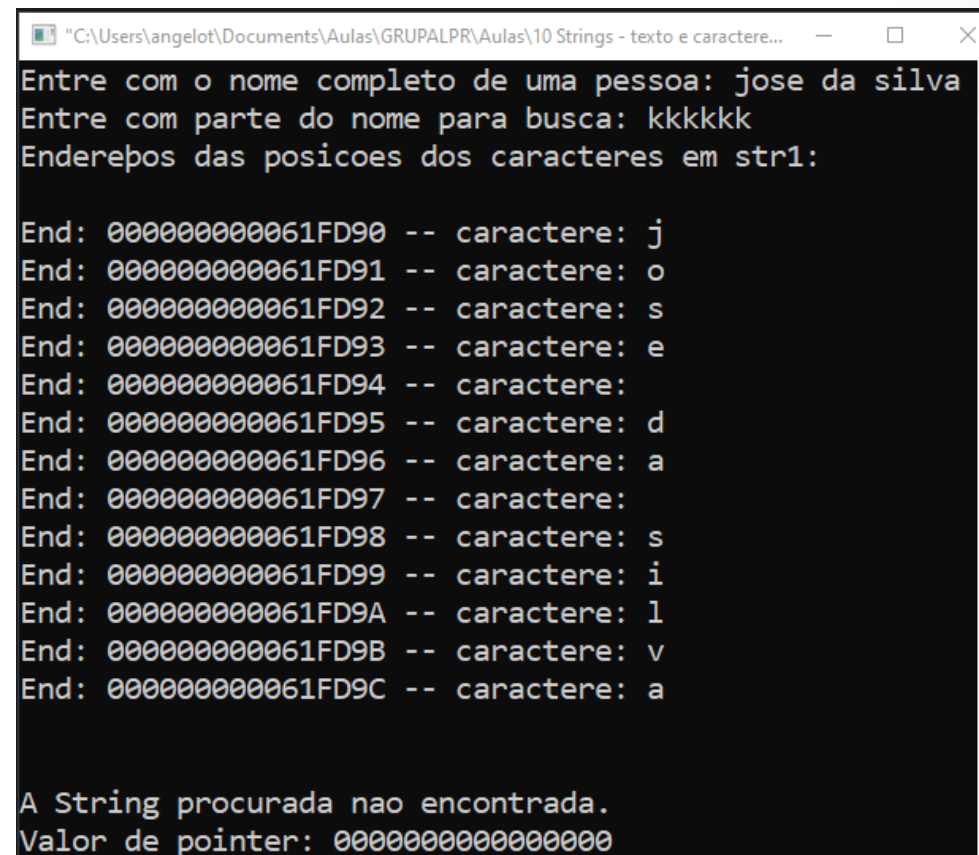
int main(){
    char str1[100], str2[100], *pointer;
    printf("Entre com o nome completo de uma pessoa: ");
    gets(str1);
    printf("Entre com parte do nome para busca: ");
    gets(str2);
    printf("Endereços das posicoes dos caracteres em str1: \n\n");
    for(int i = 0; i < strlen(str1); i++){
        printf("End: %p -- caractere: %c \n", &str1[i], str1[i]);
    }
    pointer = strstr(str1, str2);
    if(pointer != NULL){
        printf("\n\nString encontrada na posicao: %p", pointer);
    }else{
        printf("\n\nA String procurada nao encontrada.\n");
        printf("Valor de pointer: %p", pointer);
    }
}
```

Estruturas de Dados 1

- Funções para manipulação de *strings*
 - Busca dentro de *string*

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

int main(){
    char str1[100], str2[100], *pointer;
    printf("Entre com o nome completo de uma pessoa: ");
    gets(str1);
    printf("Entre com parte do nome para busca: ");
    gets(str2);
    printf("Endereços das posicoes dos caracteres em str1: \n\n");
    for(int i = 0; i < strlen(str1); i++){
        printf("End: %p -- caractere: %c \n", &str1[i], str1[i]);
    }
    pointer = strstr(str1, str2);
    if(pointer != NULL){
        printf("\n\nString encontrada na posicao: %p", pointer);
    }else{
        printf("\n\nA String procurada nao encontrada.\n");
        printf("Valor de pointer: %p", pointer);
    }
}
```

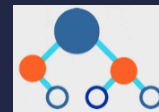


```
"C:\Users\angelot\Documents\Aulas\GRUPALPR\Aulas\10 Strings - texto e caractere...
Entre com o nome completo de uma pessoa: jose da silva
Entre com parte do nome para busca: kkkkkk
Enderepos das posicoes dos caracteres em str1:

End: 000000000061FD90 -- caractere: j
End: 000000000061FD91 -- caractere: o
End: 000000000061FD92 -- caractere: s
End: 000000000061FD93 -- caractere: e
End: 000000000061FD94 -- caractere:
End: 000000000061FD95 -- caractere: d
End: 000000000061FD96 -- caractere: a
End: 000000000061FD97 -- caractere:
End: 000000000061FD98 -- caractere: s
End: 000000000061FD99 -- caractere: i
End: 000000000061FD9A -- caractere: l
End: 000000000061FD9B -- caractere: v
End: 000000000061FD9C -- caractere: a

A String procurada nao encontrada.
Valor de pointer: 0000000000000000
```

J	o	s	é		d	a		S	i	l	v	a	\0	¥	#	*	£	¹	¬	"	M	Ç	w	g	5	!	u	»	■	Å
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	



- Funções para manipulação de *strings*
 - Busca dentro de *string*

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

int main(){
    char str1[100], str2[100], *pointer;
    printf("Entre com o nome completo de uma pessoa: ");
    gets(str1);
    printf("Entre com parte do nome para busca: ");
    gets(str2);
    printf("Endereços das posicoes dos caracteres em str1: \n\n");
    for(int i = 0; i < strlen(str1); i++){
        printf("End: %p -- caractere: %c \n", &str1[i], str1[i]);
    }
    pointer = strstr(str1, str2);
    if(pointer != NULL){
        printf("\n\nString encontrada na posicao: %p", pointer);
    }else{
        printf("\n\nA String procurada nao encontrada.\n");
        printf("Valor de pointer: %p", pointer);
    }
}
```

```
"C:\Users\angelot\Documents\Aulas\GRUPALPR\Aulas\10 Strings - texto e caractere..."
Entre com o nome completo de uma pessoa: jose da silva
Entre com parte do nome para busca: silva
Enderepos das posicoes dos caracteres em str1:

End: 000000000061FD90 -- caractere: j
End: 000000000061FD91 -- caractere: o
End: 000000000061FD92 -- caractere: s
End: 000000000061FD93 -- caractere: e
End: 000000000061FD94 -- caractere:
End: 000000000061FD95 -- caractere: d
End: 000000000061FD96 -- caractere: a
End: 000000000061FD97 -- caractere:
End: 000000000061FD98 -- caractere: s
End: 000000000061FD99 -- caractere: i
End: 000000000061FD9A -- caractere: l
End: 000000000061FD9B -- caractere: v
End: 000000000061FD9C -- caractere: a

String encontrada na posicao: 000000000061FD98
```

J	o	s	é		d	a		S	i	l	v	a	\0	¥	#	*	£	¹	¬	"	M	Ç	w	g	5	!	u	»	■	Å
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	

Estruturas de Dados 1



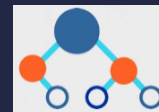
- Funções para manipulação de *strings*
 - Busca dentro de *string*

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

int main() {
    char string1[20] = "Brasileiro";
    char string2[20] = "Brasil";
    char letra = 's';
    //Comparando Strings
    if(strchr(string1, "s")){//strchr devolve valor diferente de 0
        printf("\nO caracter \"%c\" esta na string \"%s\"", letra, string1);
    }
    if(strstr(string1, string2)){//strstr devolve valor diferente de 0
        printf("\n\nA String \"%s\" esta na string \"%s\"", string2, string1);
    }

    printf("\n\n\n");
}
```

A screenshot of a terminal window with a black background and white text. The window title bar shows the file path "C:\Users\angelot\Documents\Aulas\ED1D2\Aulas\Aul...". The terminal output consists of two lines of text: "O caracter \"s\" esta na string \"Brasileiro\"" and "A String \"Brasil\" esta na string \"Brasileiro\"". Below this, it shows "Process returned 0 (0x0) execution time : 0.110 s" and "Press any key to continue.".



- Funções para manipulação de *strings*
 - Busca dentro de *string* – A função **strstr()**, é capaz de encontrar trechos de uma *string* dentro de outra. Porém, há um detalhe a ser observado: *C* é *case sensitive*.
 - Portanto, no momento de realizar a busca por uma *string*, se faz necessário que as duas *strings* envolvidas na busca estejam armazenadas da mesma forma: todos os caracteres em letras minúsculas ou todos em letras maiúsculas.
 - Isso pode ser facilmente resolvido utilizando-se as funções **strupr()** ou **strlwr()**.
 - **strupr()** – converte todos os caracteres da *string* passada como argumento, para letras maiúsculas;
 - **strlwr()** – converte todos os caracteres da *string* passada como argumento, para letras minúsculas.
 - Estas funções **não fazem parte da biblioteca padrão C**. Dessa forma, elas estão disponíveis somente nas versões Windows do compilador.

- Funções para manipulação de *strings*

- Caso seja necessário efetuar esse tipo de conversão em outro ambiente, Linux por exemplo, é necessária a conversão caractere a caractere utilizando as funções **toupper()** ou **tolower()** da biblioteca **<ctype>**, varrendo todo o vetor de caracteres onde está armazenada a *string*:

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

char *strupr(char *str);
char *strlwr(char *str);

int main(){
    char palavra[50] = "Trabalhando com linguagem C no IFSP!";
    printf("\nAntes: %s\n", palavra);
    strupr(palavra);
    printf("\nMaiusculas: %s\n", palavra);
    strlwr(palavra);
    printf("\nMinusculas: %s\n", palavra);
    return 0;
}
```

```
char *strupr(char *str){
    while(*str){
        *str = toupper(*str);
        str++;
    }
    return str;
}
```

```
char *strlwr(char *str){
    while(*str){
        *str = tolower(*str);
        str++;
    }
    return str;
}
```



- Funções para manipulação de *strings*

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

char *strupr(char *str);
char *strlwr(char *str);

int main(){
    char palavra[50] = "Trabalhando com linguagem C no IFSP!";
    printf("\nAntes: %s\n", palavra);
    strupr(palavra);
    printf("\nMaiusculas: %s\n", palavra);
    strlwr(palavra);
    printf("\nMinusculas: %s\n", palavra);
    return 0;
}
```

```
char *strupr(char *str){
    while(*str){
        *str = toupper(*str);
        str++;
    }
    return str;
}
```

```
char *strlwr(char *str){
    while(*str){
        *str = tolower(*str);
        str++;
    }
    return str;
}
```

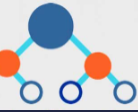
C:\Users\angelot\Documents\Aulas\GRUEDA1\Aula 03 - Strings_texto e caracteres\material de a

```
Antes: Trabalhando com linguagem C no IFSP!

Maiusculas: TRABALHANDO COM LINGUAGEM C NO IFSP!

Minusculas: trabalhando com linguagem c no ifsp!
```


Estruturas de Dados 1



- Atividades – todas devem ser realizadas utilizando Linguagem C
 - 1 – Faça um programa que leia o nome de um trabalhador, sua idade e seu salário. Imprima em tela todos os seus dados, utilizando a formatação onde for necessário.
 - 2 – Escreva um programa que receba três *strings* para o nome de uma pessoa (nome e sobrenomes), concatene formatando estas *strings* para apresentação do nome em tela.
 - 3 – Leia uma *string* do teclado e conte quantas vogais (a, e, i, o, u) ela possui no total (não individual). Entre com uma consoante (escolhida via teclado), e substitua todas as vogais da *string* coletada, por essa consoante. Por fim imprima em tela esta nova *string*.
 - 4 – Construa um programa que leia duas *strings* do teclado. Imprima uma mensagem informando se a segunda *string* está contida na primeira.

Estruturas de Dados 1



- Atividades – todas devem ser realizadas utilizando Linguagem C
 - 5 – Escreva um programa que leia uma *string* do teclado e converta todos os seus caracteres em maiúscula. Não utilize nenhuma função da biblioteca `<ctype.h>`, subtraia o valor 32 dos caracteres. Consulte a tabela ASCII para isso.
 - 6 – Utilizando agora as funções disponíveis na biblioteca `<ctype.h>`, leia do teclado uma *string* e inverta a caixa dos seus caracteres. Se o caractere for maiúsculo, transforme-o em minúsculo. Se for minúsculo, modifique-o para maiúsculo. No mesmo programa implemente as funções (escreva o código delas) `strupr()` e `strlwr()`, apresentadas no final dos slides e efetue a mesma operação de inversão maiúsculas/minúsculas.