Estrutura de Dados I

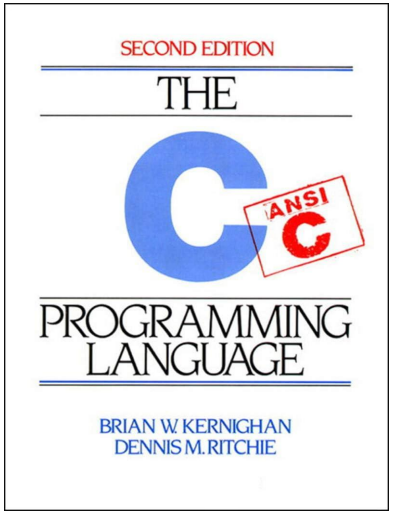
Capítulo 3: Bibliotecas e Interfaces

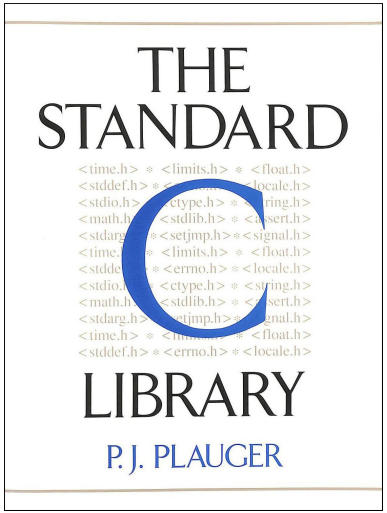
2025/1

Prof. Abrantes Araújo Silva Filho

Por que aprender sobre bibliotecas e interfaces?

- Em algumas situações, 90% ou mais de um programa consiste, atualmente, de códigos de bibliotecas
- Aprender uma linguagem é, de fato, aprender no mínimo 2 coisas:
  - A **linguagem em si**
  - A **biblioteca padrão da linguagem**



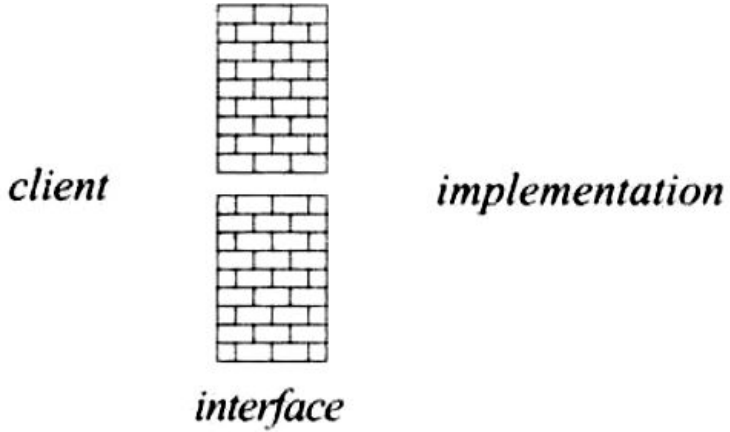


Objetivo deste capítulo

- Fazer com que você aprende sobre bibliotecas de modo a **enfatizar a diferença entre a biblioteca por si mesma e os demais programas que uso dela**. Faremos isso através do foco na fronteira entre a biblioteca e os programas que a utilizam: a **interface**.

- Termos iniciais:
  - **Biblioteca**: código escrito por outras pessoas (ou você mesmo) que você utiliza nos seus próprios programas clientes
  - **Cliente**: é o seu programa, o código que utiliza as bibliotecas
  - **Interface**: a fronteira que separa uma biblioteca de seus clientes: fornece um canal de comunicação entre o cliente e a biblioteca, e uma barreira evita que detalhes complexos de um lado afetem o outro lado.

O conceito de interface



- **Interface é uma fronteira entre duas coisas**
  - Uma cerca entre dois terrenos
  - Uma porta entre o apartamento e o corredor

- Nos programas, **a interface é uma fronteira conceitual, uma separação abstrata entre duas coisas: a IMPLEMENTAÇÃO DA BIBLIOTECA e os CLIENTES QUE USAM A BIBLIOTECA**:
  - a **IMPLEMENTAÇÃO é o código da biblioteca**
  - e **os CLIENTES são os programas que usam essa biblioteca**

- Objetivo:
  - fornecer clientes informações suficientes para que os **clientes possam usar a biblioteca sem conhecer os detalhes internos de implementação da biblioteca**
  - cria um **canal de comunicação** e também uma **barreira** de separação
  - é fundamental para o **gerenciamento de complexidade**

O conceito de interface

Acima da barreira da abstração: o usuário não se importa em como a distância Euclidiana é calculada, assume que o cálculo é correto. Só precisa saber como usar a função, p. ex.: quais devem ser as entradas?

Interface (barreira da abstração)

Abaixo da barreira da abstração: o programador cria a função de modo a garantir que ela funcionará corretamente, conforme o esperado, desde que utilizada do modo correto (entradas corretas). O programador não se importa em como e para que o usuário utilizará a função.

O conceito de interface

A interface (barreira da abstração) é o "contrato" entre quem utilizará a função e quem criou a função, e é essa barreira que esconde os detalhes internos. O contrato que a interface cria estabelece que a função funcionará de modo correto, sem que o usuário precise se preocupar com os detalhes internos, desde que esse usuário utilize a função de acordo com especificações (entradas, valores, etc.)

Interface (barreira da abstração)

Exemplos de interfaces (contratos):

- **Blocos no Snap!**
- **Pedais no carro**
- **Contratar uma construtora para uma casa**
  - **Contratar mestre de obras**
    - **Contratar pedreiros**
      - **Contratar ajudantes**

Interfaces e implementações em C

- Na **Ciência da Computação**, a interface é um conceito abstrato, um contrato abstrato, que vincula e separa o cliente da implementação de uma biblioteca.

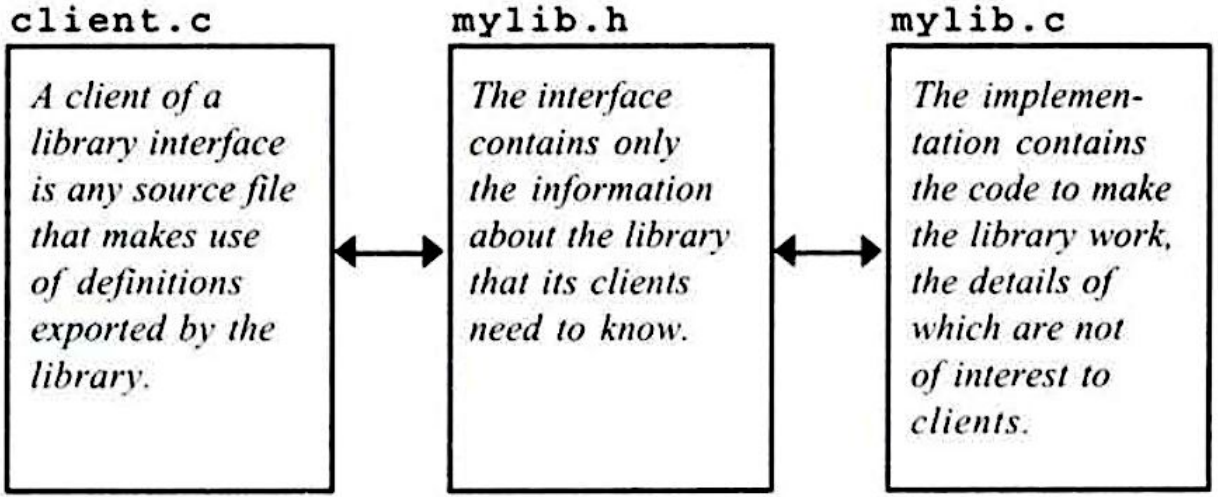
- E na **Programação**? Como representar uma interface abstrata como algo concreto? Como tornar esses conceitos concretos na linguagem C?
  - **Interface = HEADER FILE (.h) da biblioteca**
  - **Implementação = CÓDIGO (.c) e o OBJECT FILE (.o) da biblioteca**

- Conteúdo da interface:
  - declarações de tipos
  - declarações de variáveis
  - protótipos de subprogramas
  - etc.

Os protótipos costumam ser a maior parte a interface, e fazem com que as funções fiquem disponíveis para os clientes utilizarem. Isso é dito **exportar** as funções ou definições. Uma definição exportada é chamada de **elemento da interface**. Tudo que foi exportado pela interface fica disponível para uso pelos clientes.

Interfaces e implementações em C

- Atenção para um coisa importante: qual a **diferença** entre uma **INTERFACE** e uma **HEADER FILE**?
  - A **interface é um conceito abstrato**, tal como um algoritmo (ex.: busca binária)
  - A **header file é a uma realização concreta da interface**, tal como o código do algoritmo (ex.: uma busca binária pode ser codificado de modo iterativo ou recursivo)



Packages (pacotes) e abstrações

- Um termo comum (e confuso!) é o termo **pacote**. Entenderemos isso como:
  - **O conjunto formado pela interface (.h) e a implementação (.c e .o) de uma biblioteca**

- Para entender uma biblioteca temos que olhar "além do software" e **entender a base conceitual, a abstração que a biblioteca fornece**. Diferentes bibliotecas podem trabalhar com a mesma coisa, mas oferecer abstrações completamente diferentes:
  - stdio.h `printf, scanf`
  - CRpaic.h `get_int, get_float, get_double, get_string, get_char`
  - simpio.h `GetInteger, GetReal, GetLine`

- stdio.h fornece abstraçẽos poderosas e flexíveis; CRpaic.h e simpio.h fornecem abstrações menos flexíveis mas mais fáceis para o iniciante
- Os **pacotes concretizam essas abstrações** tornando-as "reais"

Princípios de bom projeto de interfaces

- Programar é difícil pois os **programas refletem a complexidade dos problemas** que tentam resolver, e envolve:
  - Quantidade enorme de complexidade
  - Considerar casos especiais
  - Considerar requisitos dos usuários
  - Considerar inúmeros detalhes

Princípios de bom projeto de interfaces

- Para isso precisamos **reduzir e controlar a complexidade**. Isso pode ser feito de diversas formas:
  - Decompor problemas
  - Reconhecer padrões
  - Criar algoritmos
  - Criar estruturas de dados adequadas
  - Criar abstrações adequadas

  } **Pensamento Computacional**

  - Dar acesso a um conjunto de definições e funções que implementam a programação de uma abstração. Ex.:
    stdio.h
    CRpaic.h

  } **Interface**
  (reduz a complexidade em mais alto nível)

Princípios de bom projeto de interfaces

- Em geral tentamos criar interfaces que sejam:
  - **Unificadas**: uma interface deve definir uma **abstração consistente** que trata de **um único tema unificador** (ex.: não misturar tratamento de strings com números)
  - **Simples**: a interface deve ser o mais simples possível e **esconder o maior número de detalhes** possíveis do cliente (ex.: usar tipos abstratos de dados)
  - **Suficientes**: se o cliente precisar de uma abstração, a interface deve **fornecer toda a funcionalidade necessária** para o cliente (ex.: deixar de fornecer um "`get_int`" em uma interface de obtenção de dados faz com sua interface não seja suficiente para o cliente, que precisará buscar outra solução mais poderosa)

Princípios de bom projeto de interfaces

- Em geral tentamos criar interfaces que sejam:
  - **Gerais**: deve ser flexível o suficiente para **atender múltiplos clientes** (ex.: se a interface `minhas_funcoes.h` tem uma função `get_int` que só suporta inteiros menores do que 100, ela não atenderá às necessidades de múltiplos clientes, não será útil para outras pessoas)
  - **Estáveis**: os subprogramas e demais objetos definidos na interface devem **continuar com a mesma estrutura e efeito**, mesmo se a implementação subjacente for alterada (ex.: alterar o comportamento obriga os clientes a serem reescritos, o que compromete o valor da interface)

Princípios de bom projeto de interfaces

- O projetista de uma interface precisa ter extremo cuidado pois, depois que a interface é publicada e começa a ser utilizada, fazer mudanças pode ser extremamente custoso ou, até mesmo, impossível.
    - **Alterar a interface**: significa alterar a implementação interna de funcionalidades já públicas de modo que alterações nos clientes são necessárias. Isso é muito custoso ou impossível de fazer, pois muitos clientes já dependem da interface do jeito que está atualmente. Ex.: `scanf`
    - **Estender a interface**: significa adicionar funcionalidades à interface ou alterá-la de modo que nenhuma alteração nos clientes é necessária. Ex.: adicionar uma função totalmente nova

Como criar interfaces?

- Vamos aprender com um exemplo simples. Nosso problema é o seguinte:
  - Precisamos criar uma biblioteca que facilita que programas clientes simulem processos aleatórios como, por exemplo, jogar uma moeda ou um dado.

- Usar um computador, que é determinístico, para obter um comportamento que simula um comportamento aleatório é complexo. Precisamos esconder essa complexidade dos clientes.

- Nosso trabalho é então criar a interface "**`aleatorio.h`**" que facilita a simulação de processos aleatórios.

- Os comentários devem fornecer todas as informações que o cliente precisa para saber como usar a interface, sem ter que estudar a implementação!

Criando uma interface: **aleatorio.h**

```
1  /**
2   * Arquivo: aleatorio.h
3   * --------------------
4   * Esta interface fornece diversos subprogramas para auxiliar a geração de
5   * números pseudo-aleatórios.
6   */
7
8  #ifndef _ALEATORIO_H
9  #define _ALEATORIO_H
58
59 #endif
```

interface boilerplate:
evita que o compilador leia a mesma interface várias vezes durante o processo de compilação

Criando uma interface: **aleatorio.h**

```
1  /**
2   * Arquivo: aleatorio.h
3   * --------------------
4   * Esta interface fornece diversos subprogramas para auxiliar a geração de
5   * números pseudo-aleatórios.
6   */
7
8  #ifndef _ALEATORIO_H
9  #define _ALEATORIO_H
10
11 #include <stdbool.h>
12
```

A sua interface pode precisar de outras interfaces. Aqui precisamos do tipo de dados “bool”.

Criando uma interface: **aleatorio.h**

```
13 /**
14  * Função: inteiro_aleatorio
15  * Uso: n = inteiro_aleatorio(min, max);
16  * ------------------------------------
17  * Esta função retorna um número inteiro aleatório no intervalo FECHADO
18  * [min, max], significando que o resultado é sempre maior do que ou igual à
19  * "min" e menor do que ou igual à "max".
20  */
21
22 int inteiro_aleatorio (int min, int max);
23
```

Criando uma interface: **aleatorio.h**

```
24 /**
25  * Função: double_aleatorio
26  * Uso: d = double_aleatorio(min, max);
27  * ------------------------------------
28  * Esta função retorna um número double aleatório no intervalo SEMI-ABERTO À
29  * DIREITA [min, max), significando que o resultado seja sempre maior do que ou
30  * iqual à "min", mas estritamente menor do que "max".
31  */
32
33 double double_aleatorio (double min, double max);
34
```

Criando uma interface: **aleatorio.h**

```
35 /**
36  * Predicado: ao_acaso
37  * Uso: if (ao_acaso(p)) ...
38  * ------------------------
39  * Este predicado tem o objetivo de simular eventos aleatórios que ocorrem com
40  * uma determinada probabilidade fixa. Ele retorna "true" com probabilidade
41  * indicada por "p", que deve ser um double entre 0.0 (nunca) e 1.0 (sempre).
42  * Por exemplo: chamar o predicado ao_acaso(0.30) returna "true" em 30% das
43  * vezes.
44  */
45
46 bool ao_acaso (double p);
47
```

Criando uma interface: **aleatorio.h**

```
48 /**
49  * Procedimento: randomizar
50  * Uso: randomizar( );
51  * ------------------
52  * Este procedimento inicializa o gerador de números pseudo-aleatórios para que
53  * seus resultados sejam imprevisíveis. Se este procedimento não for chamado, os
54  * outros subprogramas retornarão sempre os mesmos valores.
55  */
56
57 void randomizar (void);
58
```

Computadores são determinísticos, até mesmo o cálculo de números “aleatórios”. Se você não tomar nenhuma atitude o computador sempre calculará o MESMO número aleatório (bom para debug). Como os números não são realmente aleatórios, são chamados de pseudo-aleatórios. É possível “aleatorizar” esse cálculo com **seeds**.

Documentar a interface: **aleatorio.h**

- A documentação da interface tem como objetivo o programador que irá utilizar a interface em seu programa cliente, e **deve conter todas as informações que o cliente precisa**. Se a documentação da interface está bem escrita, os clientes só precisam confiar na interface, não precisam ler o código da implementação para saber como a biblioteca funciona.

- Documentar não é escrever muitos comentários misturados com o código! Uma boa documentação não suja o código, mas limita-se a

  explicar o que cada subprograma faz

- A documentação **não deve perder tempo tentando explicar como cada subprograma faz o que faz**.

Documentar a interface: **aleatorio.h**

- A documentação da interface é um **mini manual** que dá informações completas sobre o uso correto de cada subprograma, tipo de dado ou outra coisa que a interface define.

- Esse mini manual deve dizer coisas como:
  - **para que servem todos** os tipos de dados, estruturas de dados e variáveis
  - o que cada subprograma **recebe** e o que **devolve**
  - os **efeitos** que os subprogramas produzem
  - os **efeitos colaterais** (se pertinentes)

- Para maiores detalhes:
  - https://www.ime.usp.br/~pf/algoritmos/aulas/docu.html

PERGUNTA CHAVE 1

- Agora que temos a interface (`aleatorio.h`) e apenas a interface, já é possível escrever o programa cliente? Ou temos que implementar a interface primeiro (`aleatorio.c` e `aleatorio.o`)?

PERGUNTA CHAVE 1

- Agora que temos a interface (`aleatorio.h`) e apenas a interface, já é possível escrever o programa cliente? Ou temos que implementar a interface primeiro (`aleatorio.c` e `aleatorio.o`)?

- A interface contém tudo o que o cliente precisa saber sobre a biblioteca (variáveis, tipos, funções e como chamar as funções). Então **é possível escrever o cliente apenas com a interface**!

Criando o cliente: **craps.c**

```
 1 /**
 2  * Arquivo: craps.c
 3  * ----------------
 4  * Este programa simula uma parte do jogo de cassino chamado "Craps", que é
 5  * jogado com um par de dados. No início do jogo você joga os dados e calcula o
 6  * total. Se o resultado dessa primeira jogada for 7 ou 11, você ganha o jogo
 7  * com o resultado que os jogadores chamam de "natural". Se o resultado dessa
 8  * primeira jogada for 2, 3 ou 12, você perde o jogo com o resultado que os
 9  * jogadores chamam de "craps". Em qualquer outra situação o total obtido
10  * torna-se a "pontuação" e você deve continuar a jogar os dados novamente até
11  * que você obtenha novamente a "pontuação" (e ganhar o jogo) ou até que você
12  * obtenha um 7 (e perder o jogo); outros totais, incluindo o 2, 3, 11 e 12 não
13  * têm nenhum efeito sobre essa fase do jogo.
14  */
15
16 #include "aleatorio.h"
17 #include <stdbool.h>
18 #include <stdio.h>
19
```

Criando o cliente: **craps.c**

```
20 /* Protótipos */
21
22 /**
23  * Função: jogar_dados
24  * Uso: total = jogar_dados( );
25  * ---------------------------
26  * Esta função joga dois dados independentes e retorna a soma dos resultados.
27  * Como efeito colateral o resultado é exibido na tela.
28  */
29
30 static int jogar_dados (void);
31
32 /**
33  * Predicado: obter_pontuacao
34  * Uso: flag = obter_pontuacao(pontuacao);
35  * ---------------------------------------
36  * Este predicado é responsável pela parte do jogo onde você joga os dados
37  * repetidas vezes até que você obtenha sua pontuação ou um 7. O predicado
38  * retorna TRUE se você obteve a pontuação, e FALSE se obteve 7, o que ocorrer
39  * primeiro. Outros resultados não têm efeito nenhum.
40  */
41
42 static bool obter_pontuacao (int pontuacao);
43
```

Criando o cliente: **craps.c**

```
44 /* Programa principal */
45
46 int main (void)
47 {
48     int pontuacao = 0;
49
50     randomizar();
51
52     printf("Este programa joga uma partida de \"Craps\".\n");
53     pontuacao = jogar_dados();
54
55     switch (pontuacao)
56     {
57     case 7: case 11:
58         printf("Você obteve %d, um natural, e você ganhou!\n", pontuacao);
59         break;
60     case 2: case 3: case 12:
61         printf("Você obteve %d, um craps, e você perdeu.\n", pontuacao);
62         break;
63     default:
64         printf("Sua pontuação é %d.\n", pontuacao);
65         if (obter_pontuacao(pontuacao))
66             printf("Você obteve a pontuação, você ganhou!\n");
67         else
68             printf("Você obteve um 7, você perdeu.\n");
69         break;
70     }
71
72     return 0;
73 }
```

Criando o cliente: **craps.c**

```
74
75 /* Definições de subprogramas */
76
77 /**
78  * Função: jogar_dados
79  * Uso: total = jogar_dados( );
80  * ---------------------------
81  */
82
83 static int jogar_dados (void)
84 {
85     int d1, d2, total;
86
87     printf("Jogando os dados . . .\n");
88     d1 = inteiro_aleatorio(1, 6);
89     d2 = inteiro_aleatorio(1, 6);
90     total = d1 + d2;
91     printf("Os dados foram %d e %d, com total de %d.\n", d1, d2, total);
92     return total;
93 }
94
```

Criando o cliente: **craps.c**

```
 95 /**
 96  * Predicado: obter_pontuacao
 97  * Uso: flag = obter_pontuacao(pontuacao);
 98  * ----------------------------------------
 99  */
100
101 static bool obter_pontuacao (int pontuacao)
102 {
103     int resultado = 0;
104     while (true)
105     {
106         resultado = jogar_dados();
107         if (resultado == pontuacao) return true;
108         if (resultado == 7) return false;
109     }
110 }
```

PERGUNTA CHAVE 2

- Agora que temos a interface (`aleatorio.h`) e o cliente (`craps.c`), é possível compilar o cliente? Ou temos que implementar a interface primeiro (`aleatorio.c` e `aleatorio.o`)?

PERGUNTA CHAVE 2

- **SIM, é possível compilar o cliente até o arquivo objeto** (`craps.o`)!
- **NÃO é possível fazer a linkagem e gerar o executável**, pois aí precisamos da implementação (`aleatorio.c` e `aleatorio.o`)

```
[abrantesasf@ideapad ~/ed1]$ gcc -c -std=c17 -Wall -Wpedantic -o craps.o craps.c

[abrantesasf@ideapad ~/ed1]$ ls aleatorio* craps*
aleatorio.h  craps.c  craps.o
```

Implementando a interface: **aleatorio.c**

- Como algumas outras bibliotecas do padrão C geram números pseudo-aleatórios? As funções básicas para isso estão em **`stdlib.h`**.
  - **`int rand (void);`**
    - retorna um número **pseudo-aleatório não negativo**, **entre 0 e RAND_MAX** (uma constante definida em stdlib.h que depende de sua máquina/compilador — exatamente por isso não é adequada para todos os clientes, além de ser mais “espartana”)
    - como os números são pseudo-aleatórios, existe um padrão na geração dos números, mas é difícil descobrir… do ponto de vista do programador os números PARECEM aleatórios, mas não são: são uma seqüência que, com o tempo, se repetem
    - **funciona aplicando uma série de cálculos matemáticos ao último valor que foi produzido** de tal forma que:
      - os números esteja uniformemente distribuídos entre 0 e RAND_MAX
      - a seqüência continua por muito tempo antes de se repetir

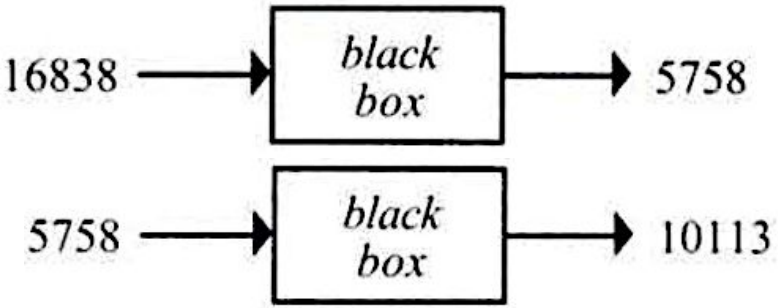
Implementando a interface: **aleatorio.c**

```c
/**
 * Arquivo: rand_max.c
 * Versão : 1.0
 * Data   : 2025-03-09 09:57
 * ------------------------
 * Este programa ilustra o comportamento da função "rand", definida na
 * biblioteca stdlib.h.
 */

/*** Includes ***/

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

/*** Constantes Simbólicas ***/

#define CHAMADAS 10

/*** Função Main: ***/

int main (void)
{
    printf("Neste computador RAND_MAX é %d.\n", RAND_MAX);
    printf("As primeiras %d chamadas à função \"rand\" são:\n", CHAMADAS);
    for (size_t i = 0; i < CHAMADAS; i++)
        printf("%10d\n", rand());
}
```

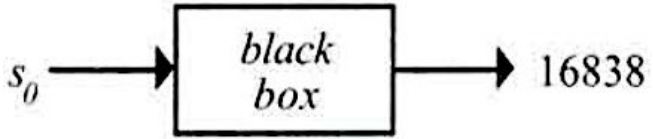
```
[abrantesasf@ideapad ~/ed1/cap03]$ ./rand_max
Neste computador RAND_MAX é 2147483647.
As primeiras 10 chamadas à função "rand" são:
1804289383
 846930886
1681692777
1714636915
1957747793
 424238335
 719885386
1649760492
 596516649
1189641421
```

Implementando a interface: **aleatorio.c**

- Se a primeira chamada à rand produziu o número 16838, as próximas serão, por exemplo:



- Mas qual é o valor da entrada para a 1ª chamada de rand, que gerou 16838?



- Esse valor s_0 é a **seed** (semente), uma constante a partir da qual os números pseudo-aleatórios são gerados. Como é a mesma para todos os programas, a mesma seqüência de números pseudo-aleatórios é gerada.

Implementando a interface: **aleatorio.c**

- Para que seu programa utilize OUTRA seqüência de números pseudo-aleatórios, temos que alterar a seed com:
  - `srand(seed)`
    (seed é um número inteiro)

- Uma estratégia comum é utilizar o valor do relógio interno do sistema como seed, pois isso garante que a cada vez que seu programa iniciar, receberá uma nova seed e a seqüência de números pseudo-aleatórios será diferente. Para obter a hora usamos a biblioteca `time.h`. e fazemos o cast para um número inteiro:
  - `srand((int) time(NULL));`

Implementando a interface: **aleatorio.c**

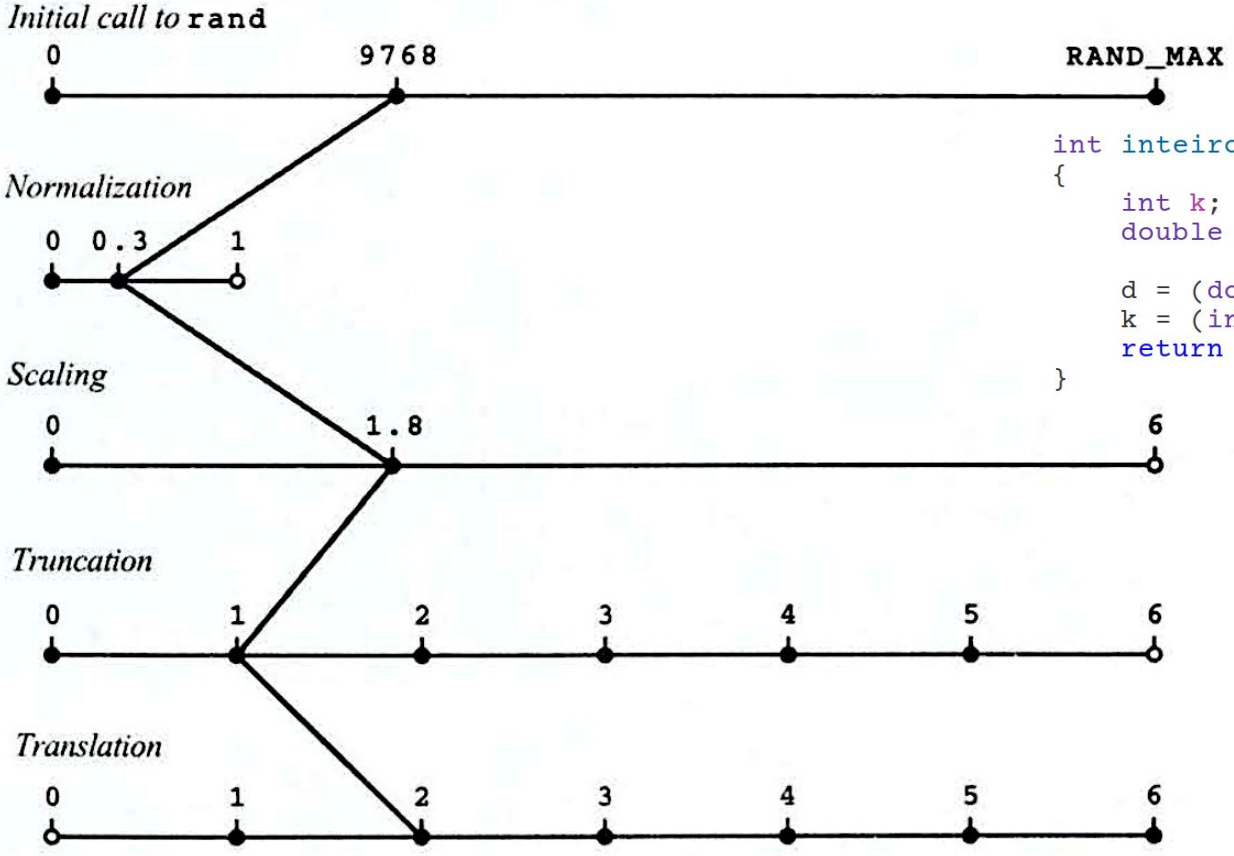
```
 1 /**
 2  * Arquivo: aleatorio.c
 3  * --------------------
 4  * Este arquivo implementa a interface aleatorio.h.
 5  */
 6
 7 #include "aleatorio.h"
 8 #include <stdbool.h>
 9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <time.h>
12
```

A implementação deve incluir a própria interface!

Implementando a interface: **aleatorio.c**

```c
/**
 * Função: inteiro_aleatorio
 * Uso: n = inteiro_aleatorio(min, max);
 * ------------------------------------
 * Esta função retorna um número inteiro aleatório no intervalo FECHADO
 * [min, max], significando que o resultado é sempre maior do que ou igual à
 * "min" e menor do que ou igual à "max".
 *
 * Ela inicia usando rand para selecionar um inteiro no intervalo [0, RAND_MAX]
 * e, então, converte esse inteiro para o intervalo desejado pelo usuário em
 * [min, max] através dos seguintes passos:
 *
 * 1. Normalizar o valor para um double no intervalo [0, 1);
 * 2. Escalar o resultado para um valor na faixa desejada;
 * 3. Truncar o resultado para um inteiro;
 * 4. Ajustar o resultado de acordo com o ponto de início apropriado.
 */

int inteiro_aleatorio (int min, int max)
{
    int k;
    double d;

    d = (double) rand() / ((double) RAND_MAX + 1);   // normaliza para [0, 1)
    k = (int) (d * (max - min + 1));                 // escala e trunca
    return (min + k);                                // ajusta para o início
}

```

Implementando a interface: **aleatorio.c**




```c
int inteiro_aleatorio (int min, int max)
{
    int k;
    double d;

    d = (double) rand() / ((double) RAND_MAX + 1);
    k = (int) (d * (max - min + 1));
    return (min + k);
}
```

Implementando a interface: **aleatorio.c**

```
41 /**
42  * Função: double_aleatorio
43  * Uso: d = double_aleatorio(min, max);
44  * ------------------------------------
45  * Esta função retorna um número double aleatório no intervalo SEMI-ABERTO À
46  * DIREITA [min, max), significando que o resultado seja sempre maior do que ou
47  * iqual à "min", mas estritamente menor do que "max".
48  *
49  * A implementação é semelhante à implementação da função "inteiro_aleatorio",
50  * sem o passo de truncar o resultado para um inteiro.
51  */
52
53 double double_aleatorio (double min, double max)
54 {
55     double d;
56
57     d = (double) rand() / ((double) RAND_MAX + 1);    // normaliza em [0, 1)
58     d = d * (max - min);                              // escala
59     return (min + d);                                 // ajusta para o início
60 }
61
```

Implementando a interface: **aleatorio.c**

```
62 /**
63  * Predicado: ao_acaso
64  * Uso: if (ao_acaso(p)) ...
65  * ------------------------
66  * Este predicado retorna "true" com probabilidade indicada por "p", que deve
67  * ser um double entre 0.0 (nunca) e 1.0 (sempre). Por exemplo: chamar o
68  * predicado ao_acaso(0.30) returna "true" em 30% das vezes.
69  *
70  * Usa a função "double_aleatorio" para gerar um número real entre [0, 1) e
71  * compara esse valor com a probabilidade p informada pelo usuário.
72  */
73
74 bool ao_acaso (double p)
75 {
76     return (double_aleatorio(0.0, 1.0) < p);
77 }
78
```

Implementando a interface: **aleatorio.c**

```
79 /**
80  * Procedimento: randomizar
81  * Uso: randomizar();
82  * ------------------
83  * Este procedimento inicializa o gerador de números pseudo-aleatórios para que
84  * seus resultados sejam imprevisíveis. Se este procedimento não for chamado, os
85  * outros subprogramas retornarão sempre os mesmos valores.
86  *
87  * Faz o ajuste do seed para a geração de númeos pseudo-aleatórios utilizando a
88  * hora atual do sistema e o procedimento srand. O procedimento srand é definido
89  * na biblioteca <stdlib.h> e requer como argumento um número inteiro. A função
90  * time é definida na biblioteca <time.h>.
91  */
92
93 void randomizar (void)
94 {
95     srand((unsigned int) time(NULL));
96 }
97
```

Implementando a interface: **aleatorio.c**

- A **documentação da implementação** NÃO É VOLTADA para o programador cliente mas, sim, para OUTROS IMPLEMENTADORES que poderão, no futuro, dar manutenção ou ampliar a biblioteca em si. Portanto, aqui, devemos incluir duas documentações:
  - a documentação de **o que** o subprograma faz (pode ser a mesma da interface)
  - a documentação de **como** o subprograma faz (voltada apenas para implementadores)

- **O CLIENTE NÃO DEVERIA NUNCA LER A DOCUMENTAÇÃO DE IMPLEMENTAÇÃO OU O CÓDIGO DA IMPLEMENTAÇÃO**. Se isso está sendo necessário, a interface está documentada de modo errado.

Implementando a interface: `aleatorio.c`

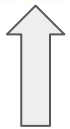
- Para gerar o arquivo objeto da biblioteca (`aleatorio.o`):

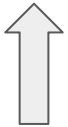
```
[abrantesasf@ideapad ~/ed1]$ gcc -c -std=c17 -Wall -Wpedantic -o aleatorio.o aleatorio.c

[abrantesasf@ideapad ~/ed1]$ ls aleatorio* craps*
aleatorio.c  aleatorio.h  aleatorio.o  craps.c  craps.o
```

Terminando a compilação do cliente: `craps.c`

- Em tese já temos tudo o que é necessário para terminar a compilação do cliente e gerar o programa executável:

```
[abrantesasf@ideapad ~/ed1]$ gcc -std=c17 -Wall -Wpedantic -o craps craps.c aleatorio.o

[abrantesasf@ideapad ~/ed1]$ ls aleatorio* craps*
aleatorio.c  aleatorio.h  aleatorio.o  craps  craps.c  craps.o
```





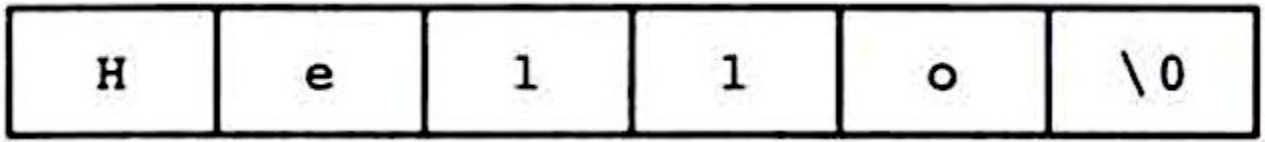
Testando o cliente: `craps`

```
[abrantesasf@ideapad ~/ed1]$ ./craps
Este programa joga uma partida de "Craps".
Jogando os dados . . .
Os dados foram 2 e 6, com total de 8.
Sua pontuação é 8.
Jogando os dados . . .
Os dados foram 4 e 2, com total de 6.
Jogando os dados . . .
Os dados foram 1 e 4, com total de 5.
Jogando os dados . . .
Os dados foram 4 e 6, com total de 10.
Jogando os dados . . .
Os dados foram 4 e 3, com total de 7.
Você obteve um 7, você perdeu.
```

Outras bibliotecas importantes: strings

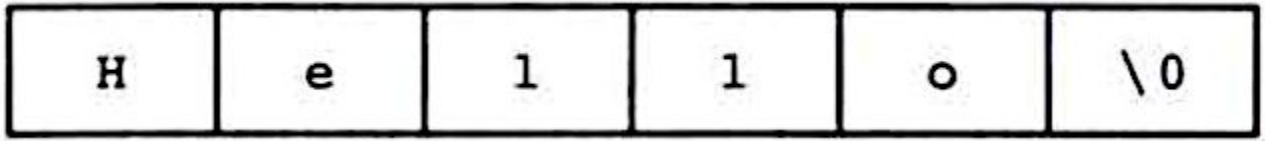
- Uma das aplicações mais importantes dos arrays em C é a representação de strings. Pode-se trabalhar apenas com arrays para fazer as operações com strings, mas C nos dá bibliotecas para facilitar.

- **Uma string é um array de caracteres em bytes consecutivos com um marcador de final, o caractere nulo (null, \0)**.



Outras bibliotecas importantes: strings

```
char ola1[]  = {'H', 'e', 'l', 'l', 'o', '\0'};
char ola2[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
char ola3[]  = "Hello";
char ola4[6] = "Hello";

//char ola5[5] = "Hello";       // ERRO! Por quê?
//char ola6[8] = "Coração";     // ERRO! Por quê?
//char ola7    = "Hello";       // ERRO! Por quê?
```



Outras bibliotecas importantes: strings

- **Idiomas comuns** para processar os caracteres de uma string:

```
char ola[] = "Olá, mundo!";

printf("%s\n", ola);
```

Outras bibliotecas importantes: strings

- **Idiomas comuns** para processar os caracteres de uma string:

```
char ola[] = "Olá, mundo!";

for (int i = 0; ola[i] != '\0', i++)
{
    printf("%c", ola[i]);
}
printf("\n");
```

```
for (int i = 0; ola[i]; i++)
```

Outras bibliotecas importantes: strings

- **Idiomas comuns** para processar os caracteres de uma string:
  - Considerando uma string como um array de caracteres

```
static int contar_espacos (char string[])
{
    int espacos = 0;

    for (int i = 0; string[i] != '\0', i++)
    {
        if (string[i] == ' ')
            espacos++;
    }

    return espacos;
}
```

Outras bibliotecas importantes: strings

- **Idiomas comuns** para processar os caracteres de uma string:
  - Considerando uma string como um ponteiro para char

```
static int contar_espacos (char *string)
{
    int espacos = 0;

    for (char *ps = string; *ps != '\0'; ps++)
    {
        if (*ps == ' ')
            espacos++;
    }

    return espacos;
}
```

Outras bibliotecas importantes: strings

- **Idiomas comuns** para processar os caracteres de uma string:
  - Considerando uma string como um ponteiro para char

```
static int contar_espacos (char *string)
{
    int espacos = 0;

    for (; *string; string++)
    {
        if (*string == ' ')
            espacos++;
    }

    return espacos;
}
```

Por que esse código NÃO ESTRAGA a referência para a string original?

Outras bibliotecas importantes: strings

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

int main (void)
{
    char ola1[] = "Ola, mundo!";
    char *ola2 = "Ola, mundo!";
    char *ola3 = malloc(sizeof(char) * (strlen("Ola, mundo!") + 1));
    strcpy(ola3, "Ola, mundo!");

    printf("%s\n%s\n%s\n", ola1, ola2, ola3);

    ola1[5] = 'M';
    ola2[5] = 'M';    // ERRO! Mas o compilador não reclama!
                      // O problema ocorre em execução!
    ola3[5] = 'M';

    printf("%s\n%s\n%s\n", ola1, ola2, ola3);

    free(ola3);
}
```

```
[abrantesasf@ideapad ~/ed1/cap03]$ ./strings_literais
Ola, mundo!
Ola, mundo!
Ola, mundo!
Segmentation fault (core dumped)
```

Strings declaradas como char * e que apontem para um literal ficam em memória read-only (são estáticas e ficam na área “rodata”).

Outras bibliotecas importantes: strings

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

int main (void)
{
    char ola1[] = "Ola, mundo!";
    char *ola2 = "Ola, mundo!";
    char *ola3 = malloc(sizeof(char) * (strlen("Ola, mundo!") + 1));
    strcpy(ola3, "Ola, mundo!");

    printf("%s\n%s\n%s\n", ola1, ola2, ola3);

    ola1[5] = 'M';
    //ola2[5] = 'M';    // ERRO! Mas o compilador não reclama!
                        // O problema ocorre em execução!
    ola3[5] = 'M';

    printf("%s\n%s\n%s\n", ola1, ola2, ola3);

    free(ola3);
}
```

```
[abrantesasf@ideapad ~/ed1/cap03]$ ./strings_literais
Ola, mundo!
Ola, mundo!
Ola, mundo!
Ola, Mundo!
Ola, mundo!
Ola, Mundo!
```

Strings declaradas como arrays dentro de funções são locais e ficam na stack, que é read-write.

Outras bibliotecas importantes: strings

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

int main (void)
{
    static char ola1[] = "Ola, mundo!";
    char *ola2 = "Ola, mundo!";
    char *ola3 = malloc(sizeof(char) * (strlen("Ola, mundo!") + 1));
    strcpy(ola3, "Ola, mundo!");

    printf("%s\n%s\n%s\n", ola1, ola2, ola3);

    ola1[5] = 'M';
    //ola2[5] = 'M';   // ERRO! Mas o compilador não reclama!
                       // O problema ocorre em execução!
    ola3[5] = 'M';

    printf("%s\n%s\n%s\n", ola1, ola2, ola3);

    free(ola3);
}
```

```
[abrantesasf@ideapad ~/ed1/cap03]$ ./strings_literais
Ola, mundo!
Ola, mundo!
Ola, mundo!
Ola, Mundo!
Ola, mundo!
Ola, Mundo!
```

Strings declaradas como arrays estáticos tem tempo de vida estático e ficam em memória read-write (área “rwdata”).

Outras bibliotecas importantes: strings

- **Idiomas comuns** para processar os caracteres de uma string:

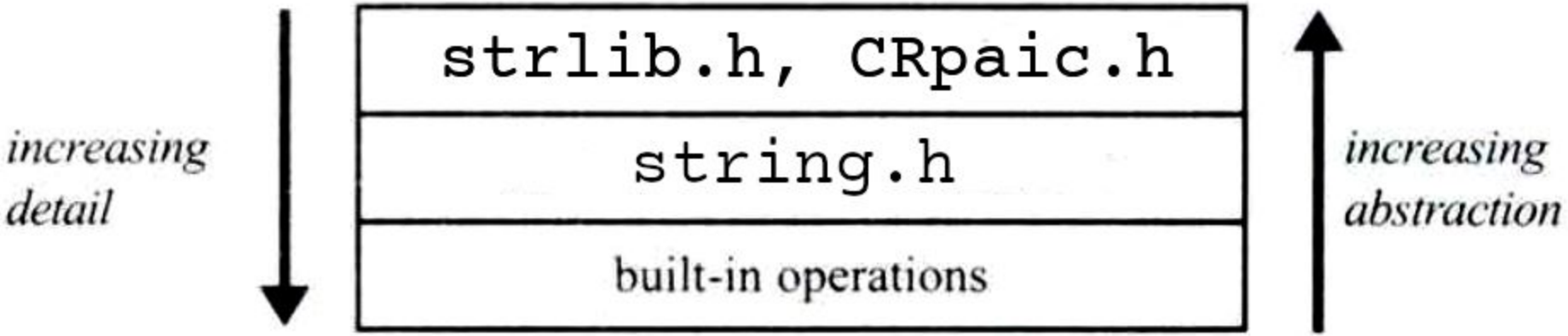
```
typedef char *string;
```

- Criamos um “tipo de dado” chamado string que é, na verdade, a mesma coisa de um ponteiro para char. São exatamente a mesma coisa do ponto de vista do compilador, mas passam “mensagens” diferentes para os humanos:
  - char *string = revela a representação interna subjacente como um ponteiro
  - string = o foco é na string como um todo

- O conjunto de valores (domínio) de string é o conjunto de todas as seqüências de caracteres (incluindo o vazio). E o conjunto de operações? C praticamente não tem nada definido sobre strings e todas as operações precisam atuar diretamente sobre os caracteres. Por isso temos bibliotecas!

Outras bibliotecas importantes: strings

- Bibliotecas interessantes e **níveis de abstração**:



- A maior diferença está em COMO elas alocam memória para os caracteres em uma string. No caso da **`strlib.h`** e da **`CRpaic.h`**, a alocação é dinâmica e feita automaticamente (a desalocação também). A **`string.h`** deixa a alocação para o cliente, mas é mais portável.

Biblioteca `<string.h>`

- Exporta um montão de subprogramas, muitos dos quais aplicáveis em situações muito específicas. As mais importantes são:
  - `strlen(s)`
  - `strcpy(dst, src)`
  - `strncpy(dst, src, n)`
  - `strcat(dst, src)`
  - `strncat(dst, src, n)`
  - `strcmp(s1, s2)`
  - `strncmp(s1, s2, n)`
  - `strchar(s, ch)`
  - `strrchr(s, ch)`
  - `strstr(s1, s2)`

Algumas funções examinam as strings sem alterar o conteúdo, são seguras.

O problema são funções que alteram de algum modo o conteúdo das strings. Deve-se ter muito cuidado!

Lembre-se de que **arrays são passados como ponteiros e qualquer descuido pode destruir a string original**!

Biblioteca `<string.h>`

Errado! Retorna a quantidade de bytes de s. Só é igual em ASCII.

`strlen(`*s*`)`	This function returns the length of the string *s*.
`strcpy(`*dst*`, `*src*`)`	This function copies characters from *src* to *dst* up to and including the first null character. As with most functions in the ANSI string library, it is the client's responsibility to ensure that there is sufficient memory space in the destination string.
`strncpy(`*dst*`, `*src*`, `*n*`)`	This function is similar to `strcpy` except that it never copies more than *n* characters, which makes it much easier to avoid overflowing the buffer used for *dst*. The ANSI definition requires `strncpy` to initialize all unused positions in the *dst* string to the null character, which leads to unnecessary inefficiency.

Biblioteca `<string.h>`

strcat(*dst*, *src*)	This function appends the characters from *src* to the end of *dst*. As with **strcpy**, this function provides no protection against overflowing the end of *dst* buffer.
strncat(*dst*, *src*, *n*)	This function appends at most *n* characters from *src* to the end of *dst*. Because the available buffer space depends on the number of characters already in the *dst* buffer as well as on the length of the *src* string, this function does not provide much help in avoiding buffer overflow.
strcmp(s_1, s_2)	This function compares the strings s_1 and s_2 and returns an integer that is less than 0 if s_1 comes before s_2 in lexicographic order, 0 if they are equal, and greater than 0 if s_1 comes after s_2.
strncmp(s_1, s_2, n)	This function is like **strcmp** but looks only at the first n characters of the two strings.

Biblioteca `<string.h>`

strchr(*s*, *ch*)	This function searches the string *s* for the character *ch* and returns a pointer to the first character position in which it appears. If *ch* does not appear in *s*, the function returns **NULL**.
strrchr(*s*, *ch*)	This function is similar to **strchr** except that it returns a pointer to the last position at which the character *ch* exists.
strstr(s_1, s_2)	This function searches for the string s_1 to see if it contains s_2 as a substring. If it does, **strstr** returns a pointer to the character position in s_1 at which the match begins. If not, it returns **NULL**.

Biblioteca `<string.h>`

- É **responsabilidade do CLIENTE** garantir que **exista espaço de armazenamento** para as operações com strings!
- **Fonte freqüente de bugs e problemas de segurança**!

- Analise o próximo trecho de código e descubra onde está o erro…

Biblioteca <string.h>

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

typedef char *string;

int main (void)
{
    string src = "Olá, mundo!";
    string dst;

    strcpy(dst, src);

    printf("Eu copiei a string: \"%s\"\n", dst);

    return 0;
}
```

Biblioteca `<string.h>`

- A solução é **pré-alocar um array** para armazenar a string de destino.

- **Arrays pré-alocados** que são depois usados são chamados de **buffers**. O tamanho do buffer deve ser o suficiente para armazenar todos os caracteres, INCLUINDO o nulo final.

- O buffer pode até ser maior do que o necessário, mas nunca pode ser menor do que o “**tamanho necessário + 1**”

Biblioteca <string.h>

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

typedef char *string;

int main (void)
{
    string src = "coracao";
    unsigned int tamanho = 8;    // OK, 7 + 1
    char dst[tamanho];           // OK, buffer suficiente.
                                 // Por que não fizemos: string dst[tamanho]?

    strcpy(dst, src);            // OK

    printf("Eu copiei a string: \"%s\"\n", dst);

    return 0;
}
```

```
[abrantesasf@ideapad ~/ed1]$ ./hello4
Eu copiei a string: "coracao"
```

Biblioteca <string.h>

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

typedef char *string;

int main (void)
{
    string src = "coração";
    unsigned int tamanho = 8;    // ERRO! Por quê?
    char dst[tamanho];

    strcpy(dst, src);            // ERRO GRAVE! Aparentemente funciona.
                                 // Por quê?

    printf("Eu copiei a string: \"%s\"\n", dst);  // ERRO, mas funciona.
                                                  // Por quê?

    char *ps = dst;
    for (; *ps != '\0'; ps++)
        printf("%c", *ps);       // ERRO, mas funciona. Por quê?
    printf("\n");

    printf("%c\n",   dst[6]);           // Imprime lixo. Por quê?
    printf("%c%c\n", dst[6], dst[7]);   // Imprime certo. Por quê?
    printf("%c%c\n", dst[7], dst[8]);   // Imprime lixo. Por quê?

    ps = dst;
    for (; *ps != '\0'; ps++)
        printf("%c ", *ps);      // Imprime lixos. Por quê?
    printf("\n");

    return 0;
}
```

```
[abrantesasf@ideapad ~/ed1]$ ./hello5
Eu copiei a string: "coração"
coração

ã
o
c o r a    o
```

Biblioteca <string.h>

```
string src = "coração";
unsigned int tamanho = 8;    // ERRO! Por quê?
char dst[tamanho];
```

1	2	3	4	5	6	7	8
c	o	r	a	c	a	o	\0

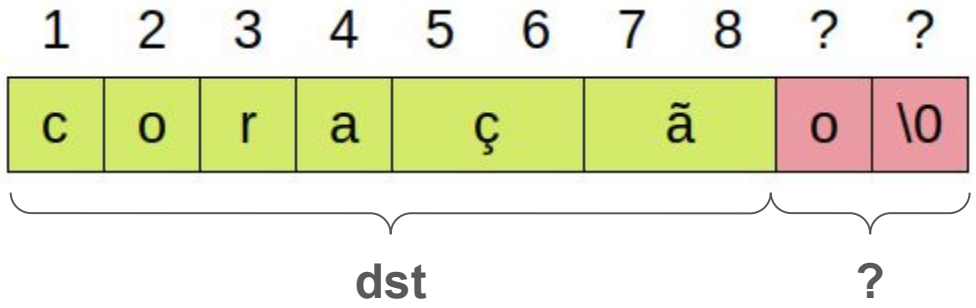
bytes (7 + 1)

1	2	3	4	5	6	7	8	9	10
c	o	r	a	ç		ã		o	\0

bytes (9 + 1)

Biblioteca <string.h>

```
strcpy(dst, src);          // ERRO GRAVE! Aparentemente funciona.
                           // Por quê?
```



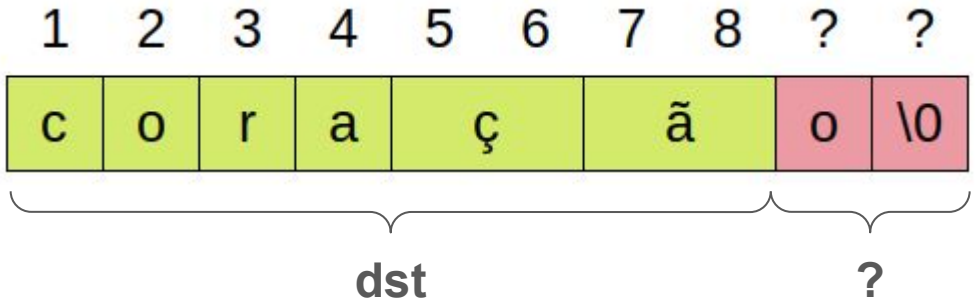
A operação de cópia está causando **buffer overflow**, e sobrescrevendo áreas de memória que NÃO PERTENCEM à variável dst!

BUFFER OVERFLOW: escrever além do tamanho de um buffer!

Biblioteca <string.h>

```
[abrantesasf@ideapad ~/ed1]$ ./hello5
Eu copiei a string: "coração"
coração
ã
o
c o r a ã o
```

```
printf("Eu copiei a string: \"%s\"\n", dst);  // ERRO, mas funciona.
                                              // Por quê?
```

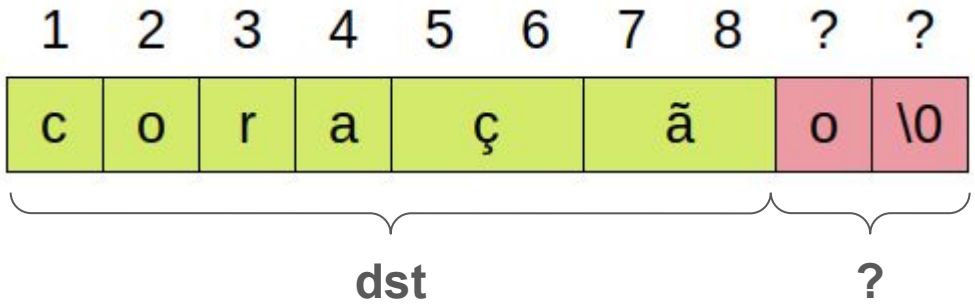


Funciona pois **%s** em printf significa:
Considere que dst é string e imprima DO INÍCIO ATÉ ENCONTRAR \0.

Biblioteca <string.h>

```
[abrantesasf@ideapad ~/ed1]$ ./hello5
Eu copiei a string: "coração"
coração
 
ã
o
c o r a
```

```
char *ps = dst;
for (; *ps != '\0'; ps++)
    printf("%c", *ps);      // ERRO, mas funciona. Por quê?
printf("\n");
```

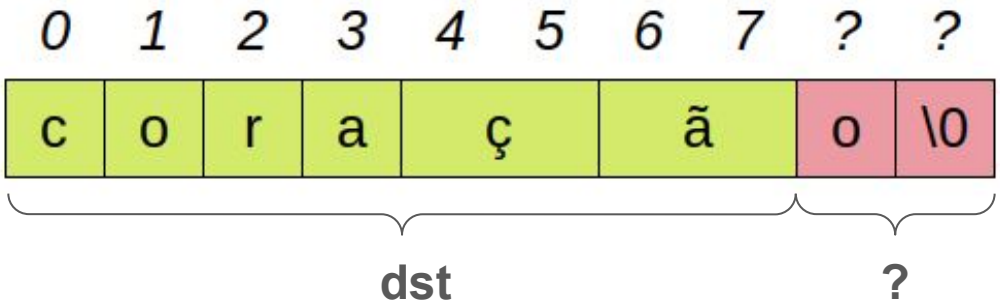


Funciona pois ao imprimir **%c multibytes contíguos**, printf consegue reproduzir o caractere multibyte corretamente até \0.

Biblioteca <string.h>

```
[abrantesasf@ideapad ~/ed1]$ ./hello5
Eu copiei a string: "coração"
coração
c o r a ç ã o
```

```
printf("%c\n",   dst[6]);           // Imprime lixo. Por quê?
printf("%c%c\n", dst[6], dst[7]);   // Imprime certo. Por quê?
printf("%c%c\n", dst[7], dst[8]);   // Imprime lixo. Por quê?
```

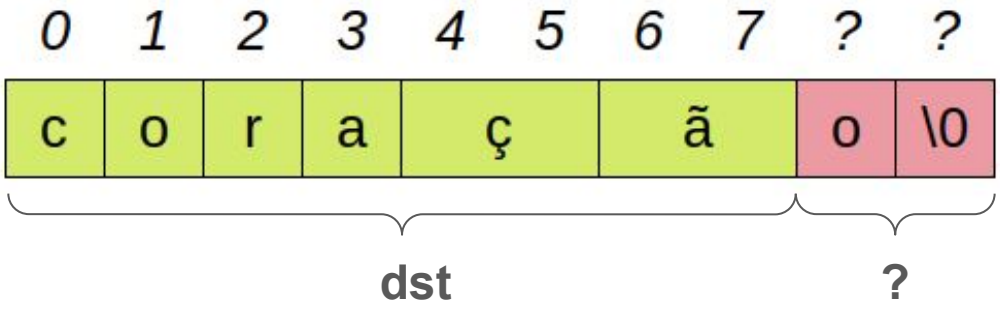


Se **%c for caractere de 1 byte ou caractere multibyte válido**, printf consegue reproduzir corretamente. Se for parte de um multibyte, não consegue.

Biblioteca <string.h>

```
[abrantesasf@ideapad ~/ed1]$ ./hello5
Eu copiei a string: "coração"
coração
c o r a
```

```
ps = dst;
for (; *ps != '\0'; ps++)
    printf("%c ", *ps);      // Imprime lixos. Por quê?
printf("\n");
```



Como há um espaço entre os **%c multibytes (e assim não são contíguos)**, printf não consegue reproduzir esse caractere multibyte.

Biblioteca <string.h>

- Em resumo:

  Ao usar funções para cópia de strings, você é **OBRIGADO A VERIFICAR O TAMANHO DA STRING DE ORIGEM E GARANTIR QUE O TAMANHO DA STRING DE DESTINO SEJA SUFICIENTE, PARA NÃO OCORRER O BUFFER OVERFLOW!**

```
string src = "coração";
unsigned int tamanho = strlen(src) + 1;
char dst[tamanho];

if (strlen(src) >= tamanho)
{
    printf("Buffer overflow!\n");
    return 1;
}
else
{
    strcpy(dst, src);
}
```

Exemplo: Pig Latin usando <string.h>

```c
/**
 * Arquivo: piglatim.c
 * Versão 2.0
 * Em 2025/03/09 15:15
 * -------------------
 * Implementa o jogo de palavras americano chamado de PigLatim, que converte
 * palavras para o vocabulário "Pig Latim" conforme as seguintes regras:
 *
 *   1. Se a palavra não contém vogais, nenhuma conversão é feita, ou seja,
 *      a palavra "traduzida" é idêntica à original;
 *   2. Se a palavra começa com uma vogal, adiciona-se o sufixo "way" ao final
 *      da palavra original: "any" torna-se "anyway"; e
 *   3. Se a palavra começa com uma consoante, extraímos a seqüencia de
 *      consoantes iniciais, até encontrar a primeira vogal, colocamos essa
 *      seqüência de consoantes como sufixo e, depois, adicionamos ainda mais
 *      um sufixo final, o "ay": "trash" torna-se "ashtray".
 */

/*** Includes ***/

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

Exemplo: Pig Latin usando <string.h>

```
26 /*** Constantes Simbólicas ***/
27
28 /**
29  * Constante: SUFMAX, SUFMIN
30  * -------------------------
31  * Definem o tamanho do maior e do menor sufixo a ser acrescentado à palavra
32  * original no momento da tradução para o Pig Latim. São três situações: ou a
33  * palavra não se altera (e, portanto não tem sufixo), ou é acrescida de "way"
34  * (que é o maior sufixo) ou é acrescida de "ay" (que é o menor sufixo).
35  */
36
37 #define SUFMAX 3
38 #define SUFMIN 2
39
40 /*** Tipos de Dados ***/
41
42 /**
43  * Tipo: en_status, statusT
44  * ------------------------
45  * Tipo de dado genérico para indicar retornos de funções. Se a função não foi
46  * executada com sucesso, retorna NOK; se foi executada com sucesso, retorna OK.
47  */
48
49 enum en_status
50 {
51     NOK,
52     OK
53 };
54
55 typedef enum en_status statusT;
```

Exemplo: Pig Latin usando <string.h>

```
57 /*** Definições de Subprogramas ***/
58
59 /**
60  * Predicado: e_vogal
61  * Uso: if (e_vogal(char)) . . .
62  * ----------------------------
63  * Este predicado recebe um único caractere e retorna TRUE se o caractere
64  * for uma vogal, e FALSE caso contrário.
65  *
66  * LIMITAÇÃO: só funciona com caracteres ASCII no momento!
67  */
68
69 bool e_vogal (const char c);
70
71 /**
72  * Função: achar_primeira_vogal
73  * Uso: p = achar_primeira_vogal(palavra);
74  * ---------------------------------------
75  * Esta função recebe uma string (como array ou ponteiro para char) e retorna um
76  * ponteiro para a primeira vogal na palavra. Se a palavra não tiver nenhuma
77  * vogal, retorna NULL.
78  */
79
80 static char *
81 achar_primeira_vogal (char *palavra);
82
```

Exemplo: Pig Latin usando <string.h>

```
83 /**
84  * Função: pig_latim
85  * Uso: if (pig_latim(palavra, buffer, tamanho_buffer)) . . .
86  * ----------------------------------------------------------
87  * Esta função traduz uma palavra qualquer em seu equivalente em PigLatim.
88  * A palavra traduzida é escrita no buffer, que tem um tamanho pré-alocado dado
89  * pelo último elemento. O código verifica se ocorrerá buffer overflow e gera
90  * uma mensagem de erro se isso ocorrer. O usuário tem a responsabilidade de
91  * passar uma palavra e um buffer válido (e o tamanho correto do buffer). Se
92  * a tradução foi feita corretamente, retorna um statusT de OK, caso contrário
93  * retorna um statusT de NOK.
94  */
95
96 static statusT
97 pig_latim (char *palavra, char buffer[], int tamanho_buffer);
98
```

Exemplo: Pig Latin usando <string.h>

```
 99 /*** Função MAIN ***/
100
101 int main (void)
102 {
103     // Obtém palava do usuário:
104     string palavra = get_string("Informe uma palavra: ");
105
106     // Cria buffer adequado para a palavra com o maior sufixo:
107     int tamanho_buffer = strlen(palavra) + 1 + SUFMAX;
108     char buffer[tamanho_buffer];
109     // Inicializa buffer inteiro como \0:
110     memset(buffer, 0, tamanho_buffer);
111
112     // Faz a tradução:
113     if (pig_latim(palavra, buffer, tamanho_buffer))
114         printf("Pig Latim: %s\n", buffer);
115     else
116     {
117         fprintf(stderr, "Erro na tradução!\n");
118         // Não precisa liberar "palavra", CRpaic faz isso aqui
119         // Não precisa liberar "buffer", pois está no stack
120         return EXIT_FAILURE;
121     }
122
123     // Finaliza:
124     // (não precisa liberar "palavra", CRpaic faz isso aqui)
125     return EXIT_SUCCESS;
126 }
```

Exemplo: Pig Latin usando <string.h>

```
128 /**
129  * Predicado: e_vogal
130  * Uso: if (e_vogal(char)) . . .
131  * ----------------------------
132  */
133
134 bool e_vogal (const char c)
135 {
136     switch (c)
137     {
138     case 'a': case 'e': case 'i': case 'o': case 'u':
139     case 'A': case 'E': case 'I': case 'O': case 'U':
140         return true;
141         break;
142     }
143     return false;
144 }
```

Exemplo: Pig Latin usando <string.h>

```
146 /**
147  * Função: achar_primeira_vogal
148  * Uso: p = achar_primeira_vogal(palavra);
149  * ------------------------------------
150  */
151
152 static char *
153 achar_primeira_vogal (char *palavra)
154 {
155     for (char *p = palavra; *p != '\0'; p++)
156     {
157         if (e_vogal(*p))
158             return p;
159     }
160
161     return NULL;
162 }
```

Exemplo: Pig Latin usando <string.h>

```
164 /**
165  * Função: pig_latim
166  * Uso: if (pig_latim(palavra, buffer, tamanho_buffer)) . . .
167  * ------------------------------------------------------------
168  */
169
170 static statusT
171 pig_latim (char *palavra, char buffer[], int tamanho_buffer)
172 {
173     if (!palavra || !buffer || tamanho_buffer <= 0)
174         return NOK;
175
176     int tam = strlen(palavra);
177
178     char *pv = achar_primeira_vogal(palavra);
179
180     if (pv == palavra)
181         tam += SUFMAX;
182     else if (pv != NULL)
183         tam += SUFMIN;
184
185     if (tam >= tamanho_buffer)
186         return NOK;
187
188     if (pv == NULL)
189         strcpy(buffer, palavra);
190     else if (pv == palavra)
191     {
192         strcpy(buffer, palavra);
193         strcat(buffer, "way");
194     }
195     else
196     {
197         strcpy(buffer, pv);
198         strncat(buffer, palavra, pv - palavra);
199         strcat(buffer, "ay");
200     }
201
202     return OK;
203 }
```

Faça o download deste programa e estude esta função para ter CERTEZA DE QUE VOCÊ ENTENDEU COMO ELA FUNCIONA!

ESTUDE! NÃO É TÃO SIMPLES QUANTO PARECE!

Biblioteca `<strlib.h>`

- Biblioteca especialmente produzida pelo autor de nosso livro de referência
- Idéia é simplificar a complexidade de `<string.h>` para não comprometer o estudo dos algoritmos e estruturas de dados.
- Trata as strings como valores abstratos, não como meros arrays de char
- Armazenamento na memória é feito de forma automática e dinâmica, o que livra o cliente de alocar a memória manualmente (as strings podem ser gigantes, até ocupar toda a heap)
- Está sendo incorporada à `CRpaic.h`

Biblioteca <strlib.h>

```
/*
 * Function: Concat
 * Usage: s = Concat(s1, s2);
 * --------------------------
 * This function concatenates two strings by joining them end to end.
 * For example, Concat("ABC", "DE") returns the string "ABCDE".
 */

string Concat(string s1, string s2);
```

Já está na CRpaic.h? SIM

`concat`

Biblioteca <strlib.h>

```
/*
 * Function: IthChar
 * Usage: ch = IthChar(s, i);
 * --------------------------
 * This function returns the character at position i in the string
 * s.  It is included in the library to make the type string a true
 * abstract type in the sense that all the necessary operations
 * can be invoked using functions. Calling IthChar(s, i) is like
 * selecting s[i], except that IthChar checks to see whether i is
 * within the range of legal index positions, which extend from 0
 * to StringLength(s). Calling IthChar(s, StringLength(s)) returns
 * the null character at the end of the string.
 */

char IthChar(string s, int i);
```

Já está na CRpaic.h? SIM

`ithcar`

Biblioteca <strlib.h>

```
/*
 * Function: SubString
 * Usage: t = SubString(s, p1, p2);
 * ------------------------------
 * SubString returns a copy of the substring of s consisting of
 * the characters between index positions p1 and p2, inclusive.
 * The following special cases apply:
 *
 * 1. If p1 is less than 0, it is assumed to be 0.
 * 2. If p2 is greater than StringLength(s) - 1, then p2 is assumed
 *    to be StringLength(s) - 1.
 * 3. If p2 < p1, SubString returns the empty string.
 */

string SubString(string s, int p1, int p2);
```

Já está na CRpaic.h? SIM

`substring`

Biblioteca <strlib.h>

```
/*
 * Function: CharToString
 * Usage: s = CharToString(ch);
 * ----------------------------
 * This function takes a single character and returns a one-character
 * string consisting of that character.  The CharToString function
 * is useful, for example, if you need to concatenate a string and
 * a character.  Since Concat requires two strings, you must first
 * convert the character into a string.
 */

string CharToString(char ch);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: StringLength
 * Usage: len = StringLength(s);
 * ----------------------------
 * This function returns the length of s.
 */

int StringLength(string s);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: CopyString
 * Usage: newstr = CopyString(s);
 * ------------------------------
 * CopyString copies the string s into dynamically allocated
 * memory and returns the new string.  This function is not required
 * if you use this library on its own, but is sometimes necessary
 * if you are working with the ANSI string library as well.
 */

string CopyString(string s);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: StringEqual
 * Usage: if (StringEqual(s1, s2)) . . .
 * ------------------------------------
 * This function returns TRUE if the strings s1 and s2 are equal.
 * For the strings to be considered equal, every character in one
 * string must precisely match the corresponding character in the
 * other.  Uppercase and lowercase characters are different.
 */

bool StringEqual(string s1, string s2);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: StringCompare
 * Usage: if (StringCompare(s1, s2) < 0) . . .
 * ------------------------------------------
 * This function returns a number less than 0 if string s1 comes
 * before s2 in alphabetical order, 0 if they are equal, and a
 * number greater than 0 if s1 comes after s2.  The order is
 * determined by the internal representation used for characters.
 */

int StringCompare(string s1, string s2);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: FindChar
 * Usage: p = FindChar(ch, text, start);
 * ------------------------------------
 * Beginning at position start in the string text, this
 * function searches for the character ch and returns the
 * first index at which it appears or -1 if no match is
 * found.
 */

int FindChar(char ch, string text, int start);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: FindString
 * Usage: p = FindString(str, text, start);
 * ----------------------------------------
 * Beginning at position start in the string text, this
 * function searches for the string str and returns the
 * first index at which it appears or -1 if no match is
 * found.
 */

int FindString(string str, string text, int start);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: ConvertToLowerCase
 * Usage: s = ConvertToLowerCase(s);
 * ---------------------------------
 * This function returns a new string with all
 * alphabetic characters converted to lower case.
 */

string ConvertToLowerCase(string s);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: ConvertToUpperCase
 * Usage: s = ConvertToUpperCase(s);
 * ---------------------------------
 * This function returns a new string with all
 * alphabetic characters converted to upper case.
 */

string ConvertToUpperCase(string s);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: IntegerToString
 * Usage: s = IntegerToString(n);
 * ------------------------------
 * This function converts an integer into the corresponding
 * string of digits.  For example, IntegerToString(123)
 * returns "123" as a string.
 */

string IntegerToString(int n);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: StringToInteger
 * Usage: n = StringToInteger(s);
 * ------------------------------
 * This function converts a string of digits into an integer.
 * If the string is not a legal integer or contains extraneous
 * characters, StringToInteger signals an error condition.
 */

int StringToInteger(string s);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: RealToString
 * Usage: s = RealToString(d);
 * ---------------------------
 * This function converts a floating-point number into the
 * corresponding string form.  For example, calling
 * RealToString(23.45) returns "23.45".  The conversion is
 * the same as that used for "%G" format in printf.
 */

string RealToString(double d);
```

Já está na CRpaic.h? NÃO

Biblioteca <strlib.h>

```
/*
 * Function: StringToReal
 * Usage: d = StringToReal(s);
 * --------------------------
 * This function converts a string representing a real number
 * into its corresponding value.  If the string is not a
 * legal floating-point number or if it contains extraneous
 * characters, StringToReal signals an error condition.
 */

double StringToReal(string s);
```

Já está na CRpaic.h? NÃO

CUIDADO!!!!!!!

- As bibliotecas padronizadas na linguagem C, ou as bibliotecas da CRpaic ou da CS50, ou as bibliotecas do livro de referência:

  C: <string.h> <ctype.h> <strings.h> …
  CRpaic: <CRpaic.h>
  CS50: <cs50.h>
  Livro: <strlib.h>

  SÃO “IMPREVISÍVEIS” COM CARACTERES ACENTUADOS!

  O uso de caracteres acentuados em C é extremamente complexo, que quase que mereceria uma disciplina só para isso. Ao utilizarmos essas bibliotecas para estudar estruturas de dados e algoritmos, sempre utilizaremos caracteres não acentuados para simplificar.

I/O padrão: `<stdio.h>`

- Além das funções mais básicas de todas (`printf` e `scanf`), `stdio.h` tem dezenas de outras funções para lidar com I/O. Nosso interesse agora é com a manipulação de **arquivos de dados** gravados como texto.

- Devemos entender os arquivos de texto como uma estrutura bi-dimensional:
  - Uma **seqüência de linhas compostas de caracteres individuais**

- Internamente um arquivo de texto é:
  - Uma **seqüência uni-dimensional de caracteres, contendo '\n'**

I/O padrão: <stdio.h>

```
[abrantesasf@ideapad ~/ed1/cap03]$ cat batatinha_sem_acentos.txt
Batatinha quando nasce,
Esparrama pelo chao.
Mamaezinha quando dorme,
Poe a mao no coracao.

[abrantesasf@ideapad ~/ed1/cap03]$ od -A d -c batatinha_sem_acentos.txt
0000000   B   a   t   a   t   i   n   h   a       q   u   a   n   d   o
0000016       n   a   s   c   e   ,  \n   E   s   p   a   r   r   a   m
0000032   a       p   e   l   o       c   h   a   o   .  \n   M   a   m
0000048   a   e   z   i   n   h   a       q   u   a   n   d   o       d
0000064   o   r   m   e   ,  \n   P   o   e       a       m   a   o
0000080   n   o       c   o   r   a   c   a   o   .  \n
0000092

[abrantesasf@ideapad ~/ed1/cap03]$ od -A d -c batatinha_sem_acentos_windows.txt
0000000   B   a   t   a   t   i   n   h   a       q   u   a   n   d   o
0000016       n   a   s   c   e   ,  \r  \n   E   s   p   a   r   r   a
0000032   m   a       p   e   l   o       c   h   a   o   .  \r  \n   M
0000048   a   m   a   e   z   i   n   h   a       q   u   a   n   d   o
0000064       d   o   r   m   e   ,  \r  \n   P   o   e       a       m
0000080   a   o       n   o       c   o   r   a   c   a   o   .  \r  \n
0000096
```

I/O padrão: `<stdio.h>`

- Arquivos de texto são similares às strings:
  - Coleção ordenada de caracteres
  - Marcador de fim (`\0` na string; `EOF` nos arquivos)

- Arquivos de texto são diferentes das strings:
  - Permanência dos dados
  - Strings são arrays de caracteres e podemos selecionar caracteres em qualquer ordem usando o índice desse caractere
  - Arquivos de texto são lidos ou escritos de modo seqüencial:
    - Quando um programa lê um arquivo, ele inicia no começo e lê os caracteres em seqüência, um por um, até que chegue no fim;
    - Quando um programa grava um arquivo, ele inicia escrevendo o primeiro caractere e continua em seqüência, escrevendo cada cada caractere subseqüente.

I/O padrão: `<stdio.h>`

- Uso de arquivos em C:
  a. **Declare uma variável do tipo ponteiro para arquivo (FILE *)**:

     FILE *entrada, *saida;

  b. **Abra o arquivo fazendo uma associação entre o ponteiro e o arquivo, com a função fopen**:

     entrada = fopen(“arquivo.txt”, <modo>)

     onde o <modo> pode ser: “r” (read) somente leitura
     “w” (write) escrita (e sobrescrita)
     “a” (append) adiciona ao final

     (teste se a abertura foi bem sucedida comparando o ponteiro com NULL)

  c. **Transfira os dados**:
     - **caractere a caractere** (`getc`, `ungetc`, `putc`)
     - **linha a linha** (`fgets`, `fputs`; `ReadLine` da `simpio.h`)
     - **dados formatados** (`fscanf`, `fprintf`)

  d. **Feche o arquivo com fclose**.

I/O padrão, caractere a caractere: <stdio.h>

```c
/**
 * Arquivo: batatinha1.c
 * Versão : 1.0
 * Data   : 2025-03-09 18:19
 * -------------------------
 * Este programa demonstra o uso de arquivos de texto em C, fazendo a
 * transferência de informações caractere a caractere com as funções:
 * getc, ungetc e putc.
 */

/*** Includes ***/

#include <CRpaic.h>
#include <stdio.h>
#include <stdlib.h>

/*** Declaração de Subprogramas ***/

/**
 * Procedimento: copiar_arquivo
 * Uso: copiar_arquivo(entrada, saida);
 * ------------------------------------
 * Este procedimento recebe dois ponteiros para arquivos, entrada e saída, e
 * faz a cópia do arquivo de entrada para o arquivo de saída. O usuário é
 * responsável por garantir que os ponteiros sejam válidos! Se o usuário não
 * passar arquivos válidos, o comportamento é indefinido.
 */

void copiar_arquivo (FILE *entrada, FILE *saida);
```

I/O padrão, caractere a caractere: <stdio.h>

```
31 /*** Função Main: ***/
32
33 int main (void)
34 {
35     // Cria os ponteiros para arquivos:
36     FILE *entrada, *saida;
37
38     // Abre o arquivo batatinha.txt para leitura:
39     entrada = fopen("batatinha.txt", "r");
40     if (!entrada)
41     {
42         fprintf(stderr, "Arquivo input não encontrado.\n");
43         return EXIT_FAILURE;
44     }
45
46     // Abre o arquivo batatinha_copia.txt para escrita:
47     saida = fopen("batatinha_copia.txt", "w");
48     if (!saida)
49     {
50         fprintf(stderr, "Arquivo output não encontrado.\n");
51         fclose(entrada);
52         return EXIT_FAILURE;
53     }
54
55     // Com os ponteiros válidos, faz a cópia:
56     copiar_arquivo(entrada, saida);
57
58     // Fecha arquivos:
59     fclose(entrada);
60     fclose(saida);
61
62     // Finaliza:
63     return EXIT_SUCCESS;
64 }
```

I/O padrão, caractere a caractere: `<stdio.h>`

```
66 /**
67  * Procedimento: copiar_arquivo
68  * Uso: copiar_arquivo(entrada, saida);
69  * ------------------------------------
70  */
71
72 void copiar_arquivo (FILE *entrada, FILE *saida)
73 {
74     int c;
75
76     while ((c = getc(entrada)) != EOF)
77         putc(c, saida);
78 }
```

Por que `int` e não `char`?

Idioma comum!

I/O padrão, caractere a caractere: rereading characters

- Ao usar a estratégia de ler caractere a caractere, uma situação relativamente comum que ocorre é você estar na posição de **não saber que você deveria ter parado de ler os caracteres até que você já tenha lido mais do que você deveria**. Ex.: um programa para remover os comentários entre “/*” e “*/”

- Um programa que copia um arquivo removendo os comentários deve:
  a. ler o arquivo e copiar os caracteres até encontrar um “/*”
  b. ler os caracteres mas não copiar
  c. após encontrar o “*/” deve voltar a ler e copiar os caracteres

- Ao ler caractere por caractere, o que fazer ao encontrar uma “/”? Pode ser o início do comentário, mas pode ser uma divisão. A única maneira é “olhar” se o próximo caractere é um “*”. Como fazer isso?

I/O padrão, caractere a caractere: rereading characters

- Usamos a função **`ungetc`**!

```
ungetc(char, entrada);
```

- Essa função coloca de volta o caractere lido no stream de entrada, para que ele seja lido novamente pela função **`getc`**.

- ATENÇÃO: `ungetc` só consegue colocar de volta apenas 1 único caractere! Não adianta tentar retornar mais de 1 caractere.

I/O padrão, caractere a caractere: rereading characters

```c
/**
 * Arquivo: remover_comentarios.c
 * Versão : 1.0
 * Data   : 2025-03-09 19:03
 * -------------------------
 * Este programa faz a cópia de um aquivo de entrada para um arquivo de saída,
 * removendo comentários entre /* e */

/*** Includes ***/

#include <CRpaic.h>
#include <stdio.h>
#include <stdlib.h>

/*** Declarações de Subprogramas ***/

/**
 * Procedimento: copiar_removendo_comentarios
 * Uso: copiar_removendo_comentario(entrada, saida);
 * ------------------------------------------------
 * Este procedimento faz a copia do arquivo de "entrada" para o arquivo de
 * "saida", removendo todos os comentários de bloco. Recebe dois ponteiros para
 * arquivos, que devem ser ponteiros válidos (o usuário é responsável de passar
 * ponteiros de arquivos válidos).
 */

void copiar_removendo_comentarios (FILE *entrada, FILE *saida);
```

I/O padrão, caractere a caractere: rereading characters

```
29 /*** Função Main: ***/
30
31 int main (void)
32 {
33     FILE *entrada, *saida;
34
35     entrada = fopen("comentarios.txt", "r");
36     if (!entrada)
37     {
38         fprintf(stderr, "Impossível abrir a entrada.\n");
39         return EXIT_FAILURE;
40     }
41
42     saida = fopen("comentarios_removidos.txt", "w");
43     if (!saida)
44     {
45         fprintf(stderr, "Impossível abrir a saída.\n");
46         fclose(entrada);
47         return EXIT_FAILURE;
48     }
49
50     copiar_removendo_comentarios(entrada, saida);
51
52     fclose(entrada);
53     fclose(saida);
54
55     return EXIT_SUCCESS;
56 }
```

I/O padrão, caractere a caractere: rereading characters

```
66 void copiar_removendo_comentarios (FILE *entrada, FILE *saida)
67 {
68     int c, prox_c;
69     bool comentario = false;
70
71     while ((c = getc(entrada)) != EOF)
72     {
73         if (comentario)
74         {
75             if (c == '*')
76             {
77                 prox_c = getc(entrada);
78                 if (prox_c == '/')
79                     comentario = false;
80                 else
81                     ungetc(prox_c, entrada);
82             }
83         }
84         else
85         {
86             if (c == '/')
87             {
88                 prox_c = getc(entrada);
89                 if (prox_c == '*')
90                     comentario = true;
91                 else
92                     ungetc(prox_c, entrada);
93             }
94
95             if (!comentario)
96                 putc(c, saida);
97         }
98     }
99 }
```

Faça o download deste programa e estude esta função para ter CERTEZA DE QUE VOCÊ ENTENDEU COMO ELA FUNCIONA!

ESTUDE! NÃO É TÃO SIMPLES QUANTO PARECE!

I/O padrão, caractere a caractere: atualizar arquivo

- Como você abriria um arquivo para poder atualizá-lo?
  - Na maioria dos sistemas é ilegal abrir um arquivo como output se ele já está aberto como também como input (isso pode até destruir o arquivo original)

- Solução comum: criar um arquivo temporário, atualizar o arquivo temporário e, depois atualizar o arquivo original. Passos necessários:
  - Abrir o arquivo original como input
  - Abrir um arquivo temporário como output (com outro nome)
  - Copiar o arquivo original para o temporário, fazendo as atualizações
  - Fechar os dois arquivos
  - Apagar o arquivo original
  - Renomear o arquivo temporário com o mesmo nome do original

- Funções necessárias (`stdio.h`): **`remove`**, **`rename`** (e **`mkstemp`** para criar nomes de arquivo temporários, mas é mais avançado e não veremos aqui)

I/O padrão, caractere a caractere: atualizar arquivo

```c
/**
 * Arquivo: atualizar_arquivo.c
 * Versão : 1.0
 * Data   : 2025-03-09 19:03
 * ---------------------------
 * Este programa atualiza um arquivo de entrada, removendo todos os comentários
 * de bloco. Isso é feito através da criação de um arquivo temporário.
 */

/*** Includes ***/

#include <CRpaic.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

/*** Declarações de Subprogramas ***/

/**
 * Procedimento: copiar_removendo_comentarios
 * Uso: copiar_removendo_comentario(entrada, saida);
 * -------------------------------------------------
 * Este procedimento faz a copia do arquivo de "entrada" para o arquivo de
 * "saida", removendo todos os comentários de bloco. Recebe dois ponteiros para
 * arquivos, que devem ser ponteiros válidos (o usuário é responsável de passar
 * ponteiros de arquivos válidos).
 */

void copiar_removendo_comentarios (FILE *entrada, FILE *saida);
```

I/O padrão, caractere a caractere: atualizar arquivo

```
37      // Cria nome de arquivo temporário (obs.: esta NÃO É A MELHOR PRÁTICA pois
38      // cria condições de corrida; o melhor seria usar mkstemp, mas isso fica
39      // para uma outra aula mais avançada)
40      string arq_temp = "arquivoTemporario";
41
62      copiar_removendo_comentarios(entrada, saida);
63
64      fclose(entrada);
65      fclose(saida);
66
67      // Remove arquivo original e altera o nome temporário para
68      // o original ("recriando" o arquivo):
69      if (remove(arquivo) != 0 || rename(arq_temp, arquivo) != 0)
70      {
71          fprintf(stderr, "Não foi possível renomear o arquivo temporário.\n");
72          return EXIT_FAILURE;
73      }
74
```

Faça o download deste programa e estude para ter certeza de que VOCÊ ENTENDEU COMO ATUALIZAR ARQUIVOS.

I/O padrão, linha a linha: `<stdio.h>`

- Pode ser necessário fazer a leitura de uma linha de cada vez, ao invés de caractere a caractere. Para isso a stdio.h nos dá:

```
char * fgets (char buffer[], int buf_size, FILE *infile);
```

- `fgets` copia uma linha do arquivo de entrada para o `buffer[]`, parando após encontrar um '`\n`', ou antes, se o tamanho do buffer dado por `buf_size` for ultrapassado. O último caractere no `buffer[]` será o '`\n`' a menos que a linha seja muito grande. Retorna o ponteiro para o `buffer[]` ou `NULL` se `fgets` chegar no final do arquivo.

I/O padrão, linha a linha: `<stdio.h>`

- Pode ser necessário fazer a gravação de uma linha de cada vez, ao invés de caractere a caractere. Para isso a stdio.h nos dá:

```
void fputs (char *str, FILE *outfile);
```

- fputs copia os caracteres da string para o arquivo de saída, até que o final da string seja alcançado.

I/O padrão, linha a linha: `<simpio.h>`

- Uma desvantagem grande de usar `fgets` é que o cliente precisa alocar o espaço do buffer, e não é fácil dimensionar o tamanho necessário. Todas as linhas do arquivo serão de mesmo tamanho?

- A biblioteca `simpio.h`, da CSLIB, fornece a função `ReadLine`, que oferece as seguintes vantagens sobre `fgets`:
  - Aloca seu próprio espaço na Heap, tornando impossível o buffer overflow;
  - Remove o caractere final '`\n`' e faz com que o retorno seja apenas os caracteres, padronizando a saída;
  - Cada string retornada é armazenada em sua própria área de memória;
  - Retorna NULL se alcançar o final do arquivo.

I/O padrão, formatado: `<stdio.h>`

- As funções para I/O formatado são emblemáticas da linguagem C.
- Existem diversas funções dessa categoria, tanto para output quanto para input.

I/O padrão, formatado: printf, `<stdio.h>`

- Para output temos:
  - **`printf(formato, . . .);`**
    - Sempre escreve no output padrão (geralmente o terminal)
  - **`fprintf(output, formato, . . .);`**
    - Idêntica à `printf`, mas recebe o ponteiro para um arquivo de saída no 1º argumento
  - **`sprintf(char array[], formato, . . .);`**
    - Recebe um array de caracteres no 1º argumento e escreve nesse array a mesma coisa que seria impressa pela `printf`. O usuário deve garantir que o array tenha tamanho suficiente para receber a escrita, ou causará buffer overflow!

I/O padrão, formatado: printf, `<stdio.h>`

- A **string de formato** informa o que será impresso e como os argumentos posteriores serão convertidos para o output.

- A **string de formato** contém **caracteres comuns** e **especificações de conversão** (começam com %)

```
%[flags][width: (* | int)][precision: .[*|int]][length mod.]<conversion specifier>
```

`%[flags][width: (* | int)][precision: .[*|int]][length mod.]<conversion specifier>`

I/O padrão, formatado: printf, `<stdio.h>`

- As **especificações de conversão** começam com %. Depois temos:
  - **Flags**: zero ou mais (modificam o significado do especificador de conversão)
  - **Width**: opcional, indica a largura mínima a ser impressa (preenche com espaços por padrão)
    - Pode ser um `*` ou um inteiro não negativo (se for 0 é uma flag)
  - **Precision**: opcional, no formato de um ponto (`.`) seguido por um `*` ou um inteiro não negativo. Tem vários significados:
    - Número mínimo de dígitos para as conversões: d, i, o, u, x, X
    - Número de dígitos após o ponto decimal para as conversões: a, A, e, E, f, F
    - Número máximo de dígitos significativos para as conversões: g, G
    - Número máximo de bytes para as conversões: s
  - **Length Modifier**: opcional, especifica o tamanho do argumento
  - **Conversion Specifier**: caractere obrigatório, especifica o tipo de conversão a ser aplicada

I/O padrão, formatado: printf, <stdio.h>

- Os **Conversion Specifiers** são:
  - `d, i`:
    argumento int é convertido para um inteiro signed
  - `o, u, x, X`:
    argumento unsigned int é convertido para unsigned octal (o), unsigned decimal (u) ou unsigned hexadecimal (x, X)
  - `f, F`:
    argumento double é convertido para notação decimal fracionária
  - `e, E`:
    argumento double é convertido em notação científica normalizada
  - `g, G`:
    argumento double é convertido em `f` ou `e`, ou então é convertido em `F` ou `E`

`%[flags][width: (* | int)][precision: .[*|int]][length mod.]<conversion specifier>`

I/O padrão, formatado: printf, `<stdio.h>`

- Os **Conversion Specifiers** são:
  - `a, A`:
    argumento double é convertido em notação hexadecimal fracionária
  - `c`:
    argumento int é convertido em unsigned char
  - `s`:
    argumento é ponteiro para um array de caracteres representando uma string
  - `p`:
    argumento é ponteiro para void
  - `%`:
    escreve o caractere %

`%[flags][width: (* | int)][precision: .[*|int]][length mod.]<conversion specifier>`

I/O padrão, formatado: printf, `<stdio.h>`

- As **Flags** são:
  - **-**:
    a conversão é justificada à esquerda (o padrão é à direita)
  - **+**:
    sempre será exibido um sinal (positivo ou negativo)
  - **0**:
    utiliza zeros ao invés de espaços para preencher a width
  - **#**:
    utiliza formato alternativo (ver documentação)

`%[flags][width: (* | int)][precision: .[*|int]][length mod.]<conversion specifier>`

I/O padrão, formatado: printf, `<stdio.h>`

- Os **Length Modifiers** são:
  - `hh`:
    o argumento é um signed char (d, i) ou unsigned char (o, u, x, X)
  - `h`:
    o argumento é um signed short int (d, i) ou unsigned shot int (o, u, x, X)
  - `l`:
    o argumento é um signed long int (d, i), unsigned long int (o, u, x, X), wint_t (c) ou wchar_t (s)
  - `ll`:
    o argumento é um signed long lont int (d, i) ou unsigned long long int (o, u, x, X)
  - `j`:
    o argumento é um intmax_t (d, i) ou uintmax_t (o, u, x, X)

`%[flags][width: (* | int)][precision: .[*|int]][length mod.]<conversion specifier>`

I/O padrão, formatado: printf, `<stdio.h>`

- Os **Length Modifiers** são:
  - `z`:
    o argumento é um size_t (d, i, o, u, x, X)
  - `t`:
    o argumento é um ptrdiff_t
  - `L`:
    o argumento é um long double (a, A, e, E, f, F, g, G)

I/O padrão, formatado: printf, <stdio.h>

- O **Width** e a **Precision** são particularmente úteis quando não especificamos diretamente um valor inteiro e usamos o *. Isso significa que a Widh e/ou a Precision não são fixos, mas são obtidos diretamente do próximo argumento (que deve ser um inteiro). Isso permite alinhamento de impressão em tempo de execução. Por exemplo:

```
printf("%*d\n", largura, valor);
```

- Cuidado ao imprimir números fracionários: o PONTO conta como caractere:

```
double d = 1234.5678;
printf("%9.4f\n", d);
printf("%10.4f\n", d);
```

```
[abrantesasf@ideapad ~/ed1/cap03]$ ./teste
1234.5678
 1234.5678
```

I/O padrão, formatado: scanf, `<stdio.h>`

- Para input temos:
  - **`scanf(formato, . . .);`**
    - Lê valores do input padrão (geralmente o terminal)
  - **`fscanf(input, formato, . . .);`**
    - Idêntica à `scanf`, mas recebe o ponteiro para um arquivo de enrada no 1º argumento
  - **`sscanf(char array[], formato, . . .);`**
    - Recebe um array de caracteres no 1º argumento e lê desse array.

- Como essas funções devem retornar o que foi lido, os argumentos são geralmente passados por “referência” (use o operador &).

I/O padrão, formatado: scanf `<stdio.h>`

`%d`	The next value from the input is scanned as a decimal integer. That integer value is then stored in the memory cell addressed by the next pointer argument. It is crucial that the size of the variable match the size indicated by the specification.

`%f`
`%e`
`%g`
	The next input value is scanned as a floating-point value and stored in the memory cell indicated by the next pointer in the argument list. The `%e` and `%g` codes are identical to `%f` and are included for symmetry with `printf`. In the simple form of these conversion specifications, the target of the pointer must be of type `float`. To read a value of type `double`, the key letter must be preceded by the letter `l`.

`%c`	The next character is read and stored at the address indicated by the next argument, which must be a character pointer. In contrast to the other specifications, the `%c` specification does not skip whitespace characters before conversion.
`%s`	Characters are read from the input and stored in successive elements of the character array indicated by the next argument. The caller must ensure that enough space has been allocated in the array to accommodate the value being read. Input is terminated by the first whitespace character.
`%[...]` `%[^...]`	The conversion specification may consist of a set of characters enclosed in square brackets. In this case, a string is read up to the first character that is not in the bracketed set. The string is stored at the address specified by the next argument to `scanf`, which must be a character array. If the set of characters begins with a circumflex (`^`), the characters that follow are instead interpreted as those that are *not* permitted in the input. For example, the specification `%[0123456789]` reads in the next sequence of digits as a string; the specification `%[^.!?]` reads in a string of characters up to the next end-of-sentence mark (period, exclamation point, or question mark).
`%%`	No conversion is done; a percent sign must follow in the input.

I/O padrão, formatado: scanf `<stdio.h>`

- P. J. Plauger:

  “You will find that the scan conversion specifications are not as complete as the print conversion specifications. Too often, you want to exercise control over an input scan. Or you may find it impossible to determine where a scan failed well enough to recover properly from the failure . . . **Be prepared . . . to give up on the scan functions** beyond a point. Their usefulness, over the years, has proved to be limited.”

I/O padrão: arquivos padrão (standard files) `<stdio.h>`

- Arquivos padronizados: **stdin**, **stdout** e **stderr**:
  - São identificadores especiais que atuam como constantes FILE * que estão disponíveis para todos os programas
    - `stdin`: é a entrada padrão (geralmente teclado)
    - `stdout`: é a saída padrão (geralmente terminal)
    - `stderr`: é a saída padrão de erro (geralmente terminal)
  - É possível alterar essas associações
  - Podem ser lidos/escritos:
    - caractere a caractere
    - linha a linha
    - dados formatados

Outras bibliotecas: `<stdlib.h>`

- Muito importante:
  - NULL
  - malloc
  - calloc
  - realloc
  - free
  - abs

Outras bibliotecas: `<ctype.h>`

- Trabalhar com caracteres:

`isupper(ch)`
`islower(ch)`
`isalpha(ch)`
	These functions return **TRUE** if *ch* is an uppercase letter, a lowercase letter, or any type of letter, respectively.

`isdigit(ch)`	This function returns **TRUE** if *ch* is one of the digit characters.
`isalnum(ch)`	This function returns **TRUE** if *ch* is **alphanumeric**, which means that it is either a letter or a digit.
`ispunct(ch)`	This function returns **TRUE** if *ch* is a punctuation symbol.
`isspace(ch)`	This function returns **TRUE** if *ch* is a **whitespace** character. These characters are `' '` (the space character), `'\t'`, `'\n'`, `'\f'`, or `'\r'`, all of which appear as blank space on the screen.
`isprint(ch)`	This function returns **TRUE** if *ch* is any printable character, including the whitespace characters.
`toupper(ch)` `tolower(ch)`	These functions first test *ch* to see if it is a letter. If so, *ch* is converted to the desired case, so that `tolower('A')` returns `'a'`. If *ch* is not a letter, its value is returned unchanged.

Outras bibliotecas: `<math.h>`

- Funções matemáticas diversas:

`fabs(x)`	This function returns the absolute value of a real number *x*. (Note: The function **abs**, which takes the absolute value of an integer, is exported by **stdlib.h**.)
`floor(x)`	This function returns the floating-point representation of the largest integer less than or equal to *x*.
`ceil(x)`	This function returns the floating-point representation of the smallest integer greater than or equal to *x*.
`fmod(x, y)`	This function returns the floating-point remainder of x / y.
`sqrt(x)`	This function returns the square root of *x*.
`pow(x, y)`	This function returns x^y.
`exp(x)`	This function returns e^x.
`log(x)`	This function returns the natural logarithm of *x*.
`sin(`*theta*`)` `cos(`*theta*`)`	These functions return the trigonometric sine and cosine of the angle *theta*, which is expressed in radians. You can convert from degrees to radians by multiplying by π / 180.
`atan(x)`	This function returns the trigonometric arctangent of the value *x*. The result is an angle expressed in radians between $-\pi/2$ and $+\pi/2$.
`atan2(y, x)`	This function returns the angle formed between the *x*-axis and the line extending from the origin through the point (*x*, *y*). As with the other trigonometric functions, the angle is expressed in radians.

Bibliotecas: criar ou usar?

- Sempre que possível, use bibliotecas prontas da C Standard Library:
  - São padronizadas, portáveis, compreensíveis por todos os programadores
  - São extremamente testadas e validadas, muito confiáveis e corretas
  - São extremamente otimizadas
- Se não existir nada que te sirva, aí sim crie sua biblioteca.