Estrutura de Dados I

Capítulo 3: Bibliotecas e Interfaces

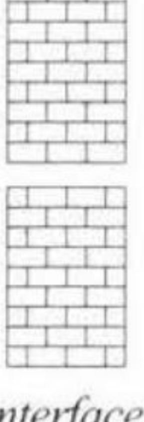
Por que aprender sobre bibliotecas e interfaces?

- 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**

Por que aprender sobre bibliotecas e interfaces?

- Termos iniciais:
  - **Biblioteca**: código escrito por outras pessoas que você utiliza nos seus próprios programas
  - **Cliente**: é o seu programa, o código que utiliza as bibliotecas
  - **Interface**: a fronteira que separa uma biblioteca de seus clientes

O conceito de interface



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

- No caso dos programas, **uma interface é uma separação abstrata entre duas coisas: IMPLEMENTAÇÃO e CLIENTES**:
  - 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**
  - cria um **canal de comunicação** e também uma **barreira** de separação
  - é uma **barreira abstrata** 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.)

Euclidiana entre ☰ e ☰ — **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

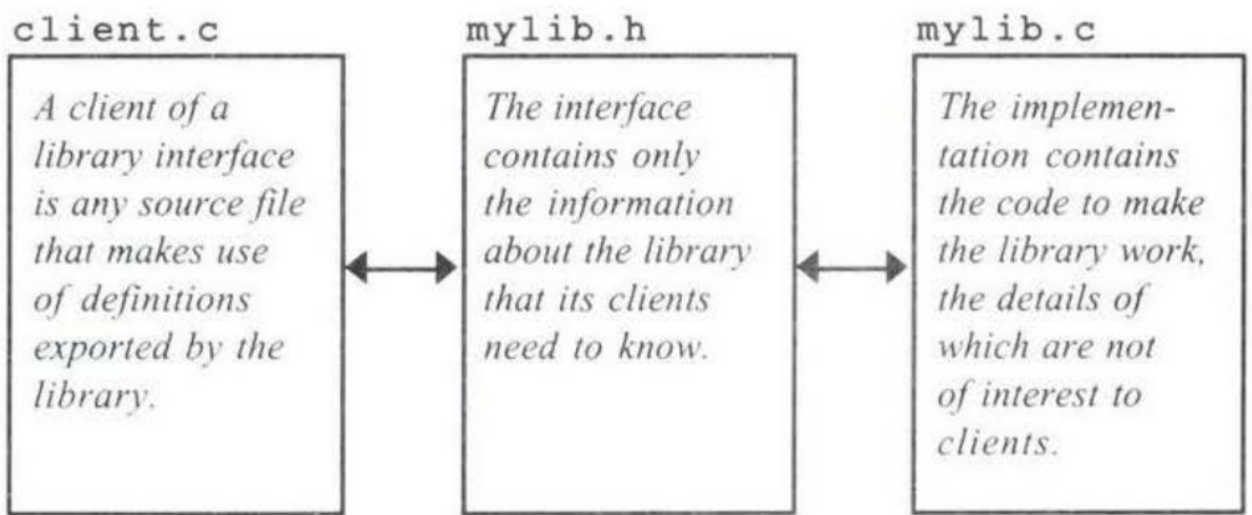
Interfaces e implementações em C

- A interface é um conceito abstrato, que separa o cliente da implementação de uma biblioteca. 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
  - protótipos de subprogramas
  - tipos de dados
  - constantes
  - etc.

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
  - cs50.h get_int, get_float, get_double, get_string, get_char
  - simpio.h GetInteger, GetReal, GetLine

- stdio.h fornece abstraçãos poderosas e flexíveis; cs50.h e simpio.h fornecem abstrações menos flexíveis mas mais fáceis para o iniciante
- Os pacotes implementam essas abstrações tornando-as "reais"

Princípios de bom projeto de interfaces

- Escrever interfaces (qualquer programa, na verdade) envolve:
    - Gerenciar complexidade
    - Criar algoritmos
    - Considerar casos especiais
    - Reduzir complexidade
    - Decompor problemas
    - Reconhecer padrões
    - Criar estruturas de dados adequadas
    - Criar abstrações adequadas
    - Criar subprogramas
    - …

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.: não colocar um get_int em uma interface de obtenção de dados)
  - **Gerais**: deve ser flexível o suficiente para atender múltiplos clientes (ex.: minhas_funcoes.h pode não ser 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)

Criando uma interface: **aleatorio.h**

- 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.

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

56
57 #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  * igual 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 retorna "true" com probabilidade indicada por "p", que deve
40  * ser um double entre 0.0 (nunca) e 1.0 (sempre). Por exemplo: chamar o
41  * predicado ao_acaso(0.30) returna "true" em 30% das vezes.
42  */
43
44 bool ao_acaso (double p);
45
```

Criando uma interface: **aleatorio.h**

```
46 /**
47  * Procedimento: randomizar
48  * Uso: randomizar();
49  * -------------------
50  * Este procedimento inicializa o gerador de números pseudo-aleatórios para que
51  * seus resultados sejam imprevisíveis. Se este procedimento não for chamado, os
52  * outros subprogramas retornarão sempre os mesmos valores.
53  */
54
55 void randomizar (void);
```

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)?

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**

```c
20 /* Protótipos */
21
22 static int jogar_dados (void);
23 static bool obter_pontuacao (int pontuacao);
24
```

Criando o cliente: **craps.c**

```
25 /* Programa principal */
26
27 int main (void)
28 {
29     int pontuacao = 0;
30
31     randomizar();
32
33     printf("Este programa joga uma partida de \"Craps\".\n");
34     pontuacao = jogar_dados();
35
36     switch (pontuacao)
37     {
38     case 7: case 11:
39         printf("Você obteve %d, um natural, e você ganhou!\n", pontuacao);
40         break;
41     case 2: case 3: case 12:
42         printf("Você obteve %d, um craps, e você perdeu.\n", pontuacao);
43     default:
44         printf("Sua pontuação é %d.\n", pontuacao);
45         if (obter_pontuacao(pontuacao))
46             printf("Você obteve a pontuação, você ganhou!\n");
47         else
48             printf("Você obteve um 7, você perdeu.\n");
49     }
50
51     return 0;
52 }
53
```

Criando o cliente: **craps.c**

```
54 /**
55  * Predicado: obter_pontuacao
56  * Uso: flag = obter_pontuacao(pontuacao);
57  * ---------------------------------------
58  * Este predicado é responsável pela parte do jogo onde você joga os dados
59  * repetidas vezes até que você obtenha sua pontuação ou um 7. O predicado
60  * retorna TRUE se você obteve a pontuação, e FALSE se obteve 7, o que ocorrer
61  * primeiro. Outros resultados não têm efeito nenhum.
62  */
63
64 static bool obter_pontuacao (int pontuacao)
65 {
66     int resultado = 0;
67     while (true)
68     {
69         resultado = jogar_dados();
70         if (resultado == pontuacao) return true;
71         if (resultado == 7) return false;
72     }
73 }
74
```

Criando o cliente: **craps.c**

```
75 /**
76  * Função: jogar_dados
77  * Uso: total = jogar_dados( );
78  * ----------------------------
79  * Esta função joga dois dados independentes e retorna a soma dos resultados.
80  * Como efeito colateral o resultado é exibido na tela.
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
```

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

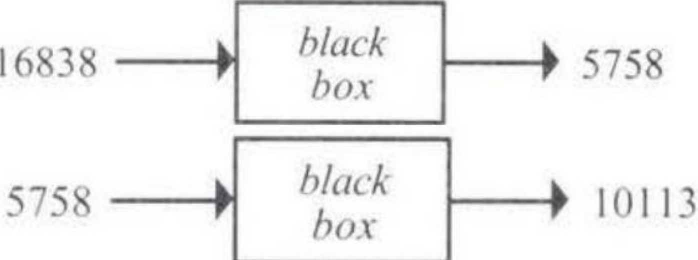
```
[abrantesasf@ideapad ~/ed1]$ gcc -c -std=c17 -Wall -Wpedantic -o craps.o craps.c

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

Implementando a interface: **aleatorio.c**

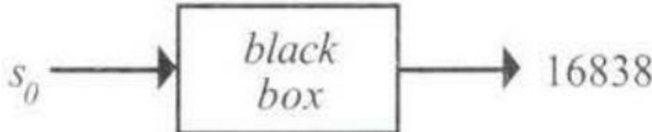
- Antes de implementarmos a interface, precisamos entender COMO algumas outras bibliotecas do padrão C funcionam para gerar números pseudo-aleatórios. As funções básicas para isso estão em **`stdlib.h`**.
  - **`int rand (void)`**
    - retorna um número **pseudo-aleatório não negativo**, **entre 0 e RAND_MAX** (uma constante definida em stdlib.h que depende de sua máquina/compilador)
    - 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
    - **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**

- 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**

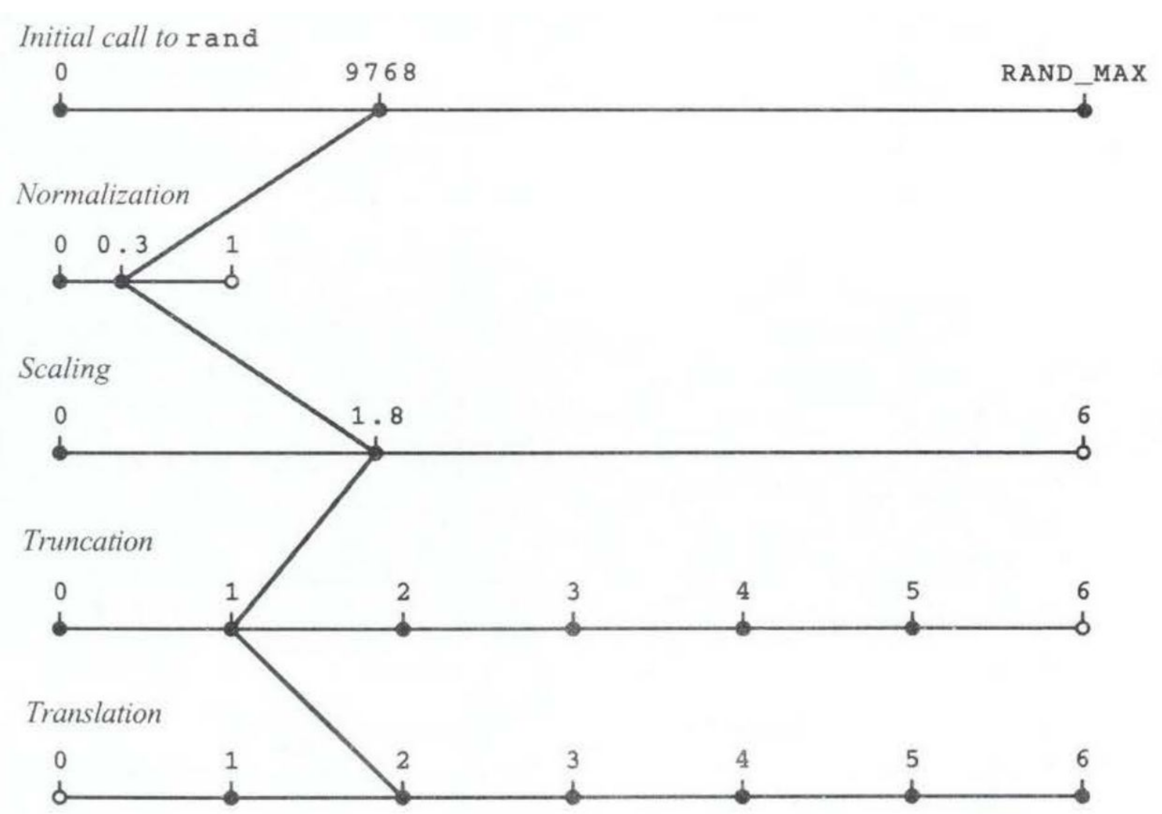
```
1  /**
2   * Arquivo: aleatorio.c
3   * ---------------------
4   * Este arquivo implementa a interface aleatorio.h.
5   */
6
7  #include "aleatorio.h"
8  #include <stdbool.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <time.h>
12
```

A implementação deve incluir a própria interface!

Implementando a interface: **aleatorio.c**

```
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  * Ela inicia usando rand para selecionar um inteiro no intervalo [0, RAND_MAX]
22  * e, então, converte esse inteiro para o intervalo desejado pelo usuário em
23  * [min, max] através dos seguintes passos:
24  *
25  * 1. Normalizar o valor para um double no intervalo [0, 1);
26  * 2. Escalar o resultado para um valor na faixa desejada;
27  * 3. Truncar o resultado para um inteiro;
28  * 4. Ajustar o resultado de acordo com o ponto de início apropriado.
29  */
30
31 int inteiro_aleatorio (int min, int max)
32 {
33     int k;
34     double d;
35
36     d = (double) rand() / ((double) RAND_MAX + 1);   // normaliza para [0, 1)
37     k = (int) (d * (max - min + 1));                 // escala e trunca
38     return (min + k);                                // ajusta para o início
39 }
40
```

Implementando a interface: **aleatorio.c**



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  * igual 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 + 1);                         // 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:

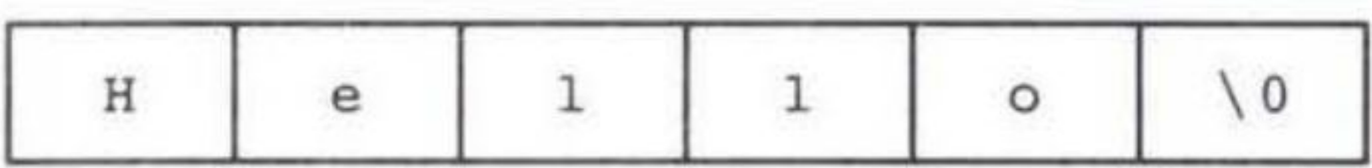
```
[abrantesasf@ideapad ~/ed1]$ gcc -std=c17 -Wall -Wpedantic -o craps craps.c aleatorio.o

[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 com um marcador de final, o caractere nulo (null, \0)**.



Outras bibliotecas importantes: strings

```c
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?
```

H	e	l	l	o	\0

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:

```
char ola[] = "Olá, mundo!";
printf("%s\n", ola);
```

Outras bibliotecas importantes: strings

- Idiomas comuns para processar os caracteres de uma string:

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

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

Outras bibliotecas importantes: strings

- Idiomas comuns para processar os caracteres de uma string:

```
static int contar_espacos2 (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

- Idiomas comuns para processar os caracteres de uma string:

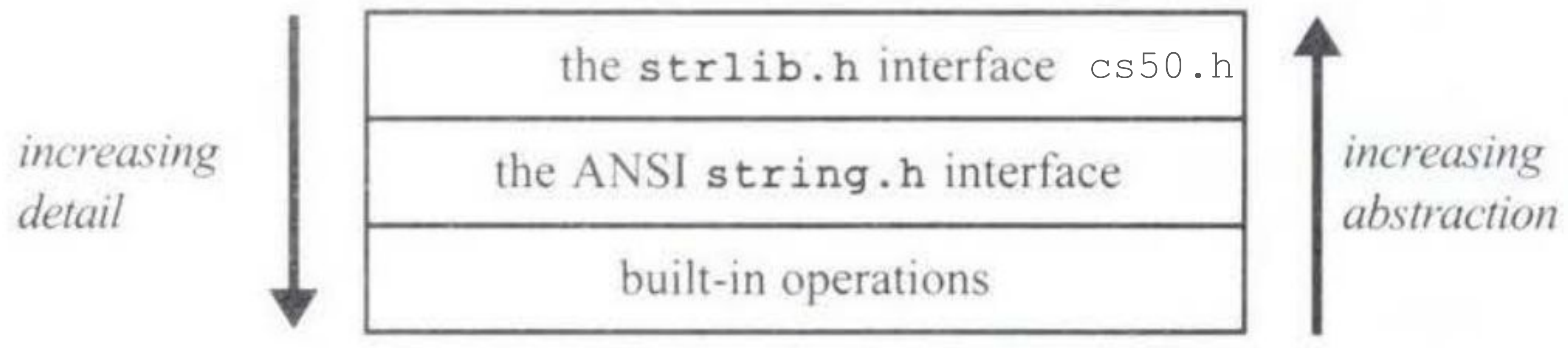
```c
typedef char *string;
```

- 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 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 e todas as operações precisam atuar diretamente sobre os caracteres. Por isso temos bibliotecas!

Outras bibliotecas importantes: strings

- Bibliotecas interessantes:



- A maior diferença está em COMO elas alocam memória para os caracteres em uma string. No caso da **`strlib.h`** e da **`cs50.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.

Biblioteca <string.h>

```
#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>

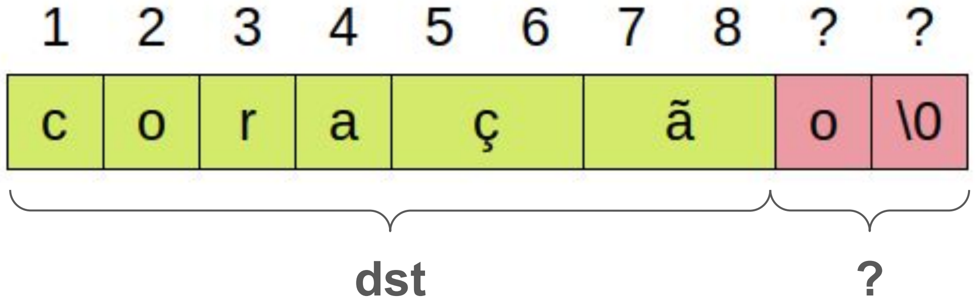
```
string src = "coração";
unsigned int tamanho = 8;    // ERRO! Por quê?
char dst[tamanho];
```

1	2	3	4	5	6	7	8	bytes (7 + 1)
c	o	r	a	c	a	o	\0	

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

Biblioteca **<string.h>**

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

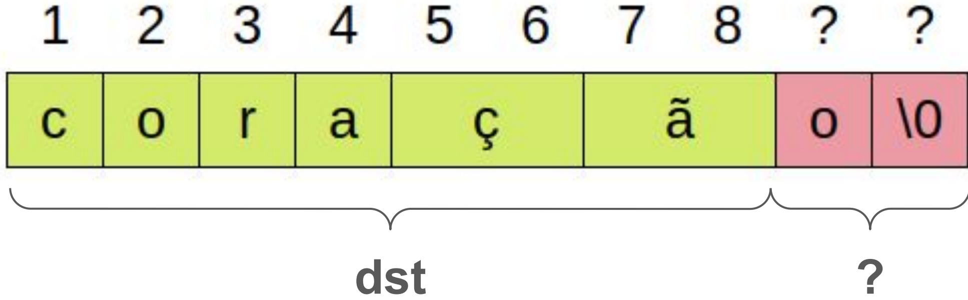


A operação de cópia está causando **buffer overflow**, e sobrescrevendo áreas de memória que NÃO PERTENCEM à variável dst!

Biblioteca <string.h>

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

```c
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.

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

Biblioteca <string.h>

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

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

dst (posições 1–8) | ? (posições ? ?)

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
�
ã
�o
c o r a � � � � o
```

```c
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?
```

0	1	2	3	4 5	6 7	?	?
c	o	r	a	ç	ã	o	\0

dst (posições 0 a 7) — ? (posições seguintes)

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
�
ã
�o
c o r a � � � � o
```

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

0	1	2	3	4	5	6	7	?	?
c	o	r	a	ç		ã		o	\0

dst ?

Como há um espaço entre os **%c multibytes 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);
}
```

Biblioteca `<strlib.h>`

- Biblioteca especialmente produzida para o 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

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);
```

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);
```

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);
```

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);
```

Biblioteca <strlib.h>

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

int StringLength(string s);
```

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);
```

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);
```

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);
```

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);
```

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);
```

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);
```

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);
```

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);
```

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);
```

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);
```

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);
```

CUIDADO!!!!!!!

- As bibliotecas padronizadas na linguagem C, ou as bibliotecas da CS50, ou as bibliotecas do livro de referência:
  C: <string.h> <ctype.h> <strings.h> …
  CS50: <cs50.h>
  Livro: <strlib.h>

  SÃO “IMPREVISÍVEIS” COM CARACTERES ACENTUADOS!

  O uso de caracteres acentuados, em C, é um tópico 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 o futuro:

- Ainda no Capítulo 3 do livro deveríamos estudar algumas outras bibliotecas, como:
  - stdio.h
  - stdlib.h
  - ctype.h
  - math.h

- Mas vamos deixar esse estudo para depois, quando formos utilizar essas bibliotecas de modo mais intenso (especialmente a stdio.h, para fazermos a leitura e escrita em arquivos de texto).