

Ulysses de Oliveira

# PROGRAMANDO EM C: VOLUME II

## A BIBLIOTECA PADRÃO DE C



**CM** EDITORA  
CIÊNCIA MODERNA

## ***DEDICATÓRIA***

*À minha mãe, Angelita.*

# *PREFÁCIO*

## **AO LEITOR**

Você tem em mãos o segundo e último volume de *Programando em C*. Este volume é dedicado principalmente à biblioteca padrão de C, examinando minuciosamente todos os componentes (tipos, macros, funções e variáveis globais) dos 24 cabeçalhos que a integram. Além disso, outros tópicos importantes em programação, como localização de programas, portabilidade de programas, processamento de caracteres extensos e multibytes, e tratamento de sinais e exceção, também são explorados. Acompanhando a discussão destes tópicos, são apresentados componentes da biblioteca padrão de C que podem ser usados na solução de problemas comuns associados a cada tópico. Por exemplo, o **Capítulo 5** faz uma introdução à localização de programa e, ao mesmo tempo, apresenta, com exemplos, os componentes do cabeçalho `<locale.h>`, que oferece suporte para resolução de problemas associados à localização de programas.

O presente volume possui as seguintes características:

- Explora completamente todos os componentes da biblioteca padrão de C, levando em consideração o padrão ISO/IEC 9899:1999 (C99), a Correção Técnica 1 (2001) e a Correção Técnica 2 (2004).
- Apresenta cerca de 400 exemplos de programação. Todos estes exemplos foram testados pelo autor e por monitores sob sua supervisão em compiladores que aderem ao padrão C99.
- Contém exercícios propostos de revisão do assunto ao final de cada capítulo.
- Contém um capítulo específico sobre portabilidade de programas.
- Discute e apresenta exemplos práticos de uso de padrões para códigos de caracteres universais (ISO 10464 e Unicode) e esquemas de representação UTF.
- Aborda tópicos intimamente associados a internacionalização de programas, tais como localização de programas, caracteres multibytes e colação de *strings*.
- Apresenta diversos exemplos de programas que usam localidades nacionais nos sistemas operacionais Linux e Windows.

- Apresenta uma referência completa de especificadores de formato para as famílias `scanf` e `printf` e para datas e horas.
- Apresenta uma lista de erros comuns de programação em C que ajuda o programador a encontrar ou prevenir eventuais erros em seus programas.

## PÚBLICO-ALVO

Este volume foi elaborado com dois objetivos gerais: (1) servir como referência a estudantes de programação e programadores que usam a linguagem C e (2) ensinar alguns tópicos avançados de programação usando esta linguagem. O livro destina-se primariamente a professores e estudantes de cursos da área de Computação e Informática, mas dificilmente poderá ser usado isoladamente como livro-texto. Além disso, a maior parte do texto não constitui material básico e requer algum conhecimento intermediário de programação em C.

O livro é recomendado idealmente como curso intermediário de programação quando o aluno já tem conhecimento básico de programação em C. Portanto, se este volume for utilizado como um curso de programação, ele deve ser complementado com um texto que preencha estes pré-requisitos. Em geral, o conteúdo do **Volume I** é suficiente como pré-requisito.

## ESCOPO

Alguns componentes da biblioteca padrão de C requerem conhecimento específico para ser utilizados. Em casos onde o conhecimento necessário está na área de programação, como tratamento de sinais e processamento de caracteres multibytes, é feita uma introdução ao assunto. Mas, o mesmo não ocorre quando o conhecimento requerido pertence a outra área de conhecimento, como números complexos ou estatística.

O padrão ISO C99 define dois tipos de sistemas: (1) sistemas com hospedeiro (sistema operacional) e (2) sistemas livres. Assim como o **Volume I**, o presente volume dá ênfase a sistemas com hospedeiro, mas o leitor não deverá ter dificuldades para utilizar o conhecimento apresentado aqui em sistemas livres, visto que, nestes sistemas, o padrão C99 requer apenas suporte limitado à biblioteca padrão de C (v. **Capítulo 1**).

## ORGANIZAÇÃO

Uma sinopse do conteúdo do **Volume I** será apresentada a seguir. Logo depois, o conteúdo de cada capítulo deste volume será descrito.

### VOLUME I

O **Capítulo 1** apresenta as construções básicas da linguagem C, como seus tipos de dados elementares e suas estruturas de controle.

O **Capítulo 2** ensina como construir programas monoarquivos usando um ambiente IDE ou um editor de programas em conjunto com um compilador. Este capítulo também ensina como usar as funções mais elementares de entrada e saída.

O **Capítulo 3** explora definição e uso de funções, assim como os conceitos fundamentais de endereços e ponteiros. Funções recursivas, *inline* e com listas de argumentos variáveis também são apresentadas neste capítulo.

O **Capítulo 4** ensina a construir programas multiarquivos e expõe conceitos tais como classes de armazenamento, tipos de dados derivados e qualificadores de tipos.

O **Capítulo 5** introduz o pré-processador de C e suas construções próprias.

O **Capítulo 6** discute legibilidade e depuração de programas.

O **Capítulo 7** enfoca arrays e ponteiros, e as relações entre eles.

O **Capítulo 8** apresenta *strings* e as funções para processamento de *strings* mais comuns. Este capítulo mostra ainda como definir a função **main()** com parâmetros.

Estruturas, uniões e enumerações são apresentadas no **Capítulo 9**, que inclui ainda iniciadores designados, arrays flexíveis e literais compostos.

O **Capítulo 10** discute as seguintes construções: arrays de ponteiros, ponteiros para ponteiros e ponteiros para funções.

O **Capítulo 11** introduz alocação dinâmica de memória, incluindo listas encadeadas e suas operações fundamentais.

O **Capítulo 12** expõe entrada e saída de dados (processamento de arquivos).

O **Capítulo 13** faz uma introdução à programação de baixo nível em C e apresenta aplicações práticas, como o uso de sinalizadores e criptografia.

O **Apêndice A** contém uma tabela completa com precedências e associatividades de todos os operadores da linguagem C, e o **Apêndice B** apresenta, de modo resumido, os especificadores de formato básicos utilizados por funções das famílias `printf` e `scanf`.

## VOLUME II

O **Capítulo 1** apresenta uma introdução e uma visão geral da biblioteca padrão de C. Aqui, além da apresentação do propósito de cada cabeçalho, os componentes são descritos de maneira suficientemente breve para permitir que o leitor tenha uma ideia do papel de cada componente. Assim, a síntese da biblioteca padrão apresentada neste capítulo tem mais utilidade como referência ou quando o programador já possui conhecimento sobre um determinado componente ou cabeçalho e deseja apenas refrescar a memória. Porém, o mais importante são as recomendações gerais e os alertas quanto ao uso dos componentes da biblioteca padrão que o programador deve levar em consideração.

O **Capítulo 2** começa com uma breve introdução aos tipos inteiros da linguagem C cujo objetivo é prover o conhecimento mínimo necessário para o entendimento dos cabeçalhos apresentados neste capítulo: `<limits.h>`, `<stdint.h>` e `<inttypes.h>`. Além de discutir os componentes destes cabeçalhos, este capítulo também apresenta funções declaradas no cabeçalho `<stdlib.h>` que executam operações aritméticas sobre inteiros.

O **Capítulo 3** introduz brevemente os tipos de ponto flutuante reais e explora em profundidade os componentes dos cabeçalhos `<float.h>`, `<math.h>` e `<fenv.h>`. É interessante notar que algumas funções do cabeçalho `<math.h>` são utilizadas em áreas específicas, tais como Física Matemática [e.g., `tgamma()`] e Estatística [e.g., `erf()`]. Portanto, entender exatamente aquilo que estas funções calculam requer um conhecimento mais profundo em Matemática do que o que é tipicamente ensinado em cursos de Computação. Prover este tipo de conhecimento está bem além do escopo deste livro.

O **Capítulo 4** faz uma breve introdução aos tipos de ponto flutuante complexos e apresenta os componentes dos cabeçalhos `<complex.h>` e `<tgmath.h>`. O entendimento da descrição da maioria das funções do cabeçalho `<complex.h>` requer conhecimento prévio de cálculo com variáveis complexas, que é tipicamente ensinado em cursos de Engenharia Elétrica/Eletrônica, mas o mesmo não ocorre em cursos de Computação. Se você não possui este conhecimento prévio, é recomendável consultar um bom texto sobre variáveis complexas antes de estudar este capítulo.

Nestes tempos de globalização, o **Capítulo 5** é extremamente importante, pois apresenta um tópico intimamente relacionado ao tema: localização de programas. Este capítulo discute os cabeçalhos `<locale.h>` e `<time.h>`, cujos componentes proveem suporte para localização e datação. Aspectos elementares de localização, medições de tempo e formatação de datas e horas também são discutidos neste capítulo.

O **Capítulo 6** lida com caracteres monobytes (i.e., do tipo **char**) e *strings* de caracteres monobytes. Estes tipos de caracteres e *strings* correspondem àqueles discutidos no **Volume I**. Neste capítulo, são estudados os cabeçalhos `<ctype.h>` e `<string.h>` e as funções declaradas no cabeçalho `<stdlib.h>` dedicadas à conversão de *strings* em números.

No **Capítulo 7**, faz-se uma introdução a dois temas complexos e importantes: caracteres extensos e multibytes. Devido à relativa complexidade dos temas expostos, este capítulo é essencialmente conceitual e deve ser visto como uma preparação básica para as aplicações práticas apresentadas no capítulo a seguir. Este capítulo também descreve o algoritmo de colação Unicode, que representa a abordagem de tratamento de colação mais aceita atualmente.

O **Capítulo 8** corresponde à aplicação prática dos conceitos apresentados no **Capítulo 7**. Este capítulo versa sobre os cabeçalhos `<wctype.h>` e `<wchar.h>`, mas as funções de entrada e saída envolvendo caracteres extensos declaradas neste último cabeçalho são apresentadas no **Capítulo 10**. Por outro lado, as funções usadas em conversão entre caracteres extensos e multibytes declaradas no cabeçalho `<stdlib.h>` são apresentadas aqui.

Funções com lista de argumentos variáveis são apresentadas no **Capítulo 9**. Apesar de o tema já ter sido abordado com relativa suficiência no **Volume I**, ele volta à tona neste capítulo. Aqui, além da macro **va\_copy()**, que não foi apresentada no **Volume I**, são expostos maiores detalhes sobre essa categoria de funções e vários exemplos novos são discutidos.

O **Capítulo 10** contém basicamente os mesmos tópicos do **Capítulo 12** do **Volume I**, mas a ênfase e o enfoque de exposição são diferentes nos dois casos. Aqui, os conceitos são apresentados com mais brevidade e as descrições dos componentes do cabeçalho `<stdio.h>` são mais detalhadas. Além disso, este capítulo aborda entrada e saída envolvendo caracteres e strings extensos e multibytes, o que não ocorre no **Volume I**.

O **Capítulo 11** apresenta os cabeçalhos: `<stdbool.h>`, `<iso646.h>`, `<errno.h>`, `<signal.h>` e `<setjmp.h>`. De algum modo, o uso dos dois primeiros cabeçalhos pode melhorar a legibilidade e a portabilidade para outras linguagens (e.g.,

C++) de programas escritos em C. Os três últimos cabeçalhos apresentados neste capítulo são tipicamente usados em tratamento de erros e exceções. Este capítulo também apresenta conceitos e exemplos relacionados a estes dois tópicos.

O **Capítulo 12** encerra o estudo da biblioteca padrão de C com a apresentação de dois cabeçalhos de propósito geral: `<stdlib.h>` e `<stddef.h>`. Parte dos componentes do cabeçalho `<stdlib.h>` é distribuída em outros capítulos devido à maior proximidade contextual: as funções declaradas neste cabeçalho dedicadas a operações aritméticas com inteiros são apresentadas no **Capítulo 2**, aquelas que realizam conversões de *strings* em número são apresentadas no **Capítulo 6**, e as funções envolvidas com conversões entre caracteres multibytes e extensos são exploradas no **Capítulo 8**.

O **Capítulo 13** discute portabilidade de programas escritos em C e este tópico não deve ser negligenciado, visto que esta é uma das principais deficiências da linguagem C.

O **Apêndice A** apresenta um resumo de todos os elementos de composição de programas escritos em C, que são, coletivamente, denominados *construtores*. Este apêndice menciona a seção desta obra onde cada construtor é explorado em detalhes, tornando-se, assim, bastante útil como referência. O **Apêndice B** apresenta exaustivamente todos os possíveis especificadores de formato que podem compor um *string* de formatação para qualquer membro das famílias `printf` e `scanf`. O **Apêndice C** descreve os possíveis componentes de *strings* de formatação de datas e horas que podem ser usados com as funções `strftime()` e `wcsftime()`. Finalmente, o **Apêndice D** apresenta uma lista de erros comuns de programação em C, que pode ser usada como lista de verificação, preventiva ou corretiva, de programas.

Ao final de cada capítulo, são incluídos **Exercícios de Revisão** que objetivam a verificação de aprendizagem do material exposto no respectivo capítulo. As respostas destes exercícios podem ser encontradas diretamente no texto ou usando-se um mínimo de dedução ou experimentação.



## EXEMPLOS E CÓDIGOS-FONTE

A maioria dos exemplos de programas foi testada usando o compilador gcc acompanhado de `-std=c99` como opção de compilação e `-lm` como opção de ligação<sup>1</sup>. A tabela a seguir descreve os ambientes nos quais a maioria dos programas apresentados como exemplos neste livro foi testada. Quando houver alguma divergência entre os dados apresentados nesta tabela e o modo como um dado programa foi compilado ou executado, o leitor será devidamente informado. Quando não houver menção a compilador ou a sistema operacional, espera-se que o resultado produzido seja o mesmo independentemente destes parâmetros.

QUANDO O TEXTO INFORMA QUE UM PROGRAMA FOI...	SIGNIFICA QUE...
<i>Compilado e executado no Linux</i>	<ul style="list-style-type: none"> <li>• O programa foi compilado com gcc 4.3.2 usando a biblioteca libc6 2.8.</li> <li>• O sistema no qual ele foi compilado e executado foi Linux Ubuntu 8.10.</li> <li>• O computador utilizado possuía CPU Pentium D945 com 2 GB de memória RAM.</li> </ul>
<i>Compilado e executado no Windows</i>	<ul style="list-style-type: none"> <li>• O programa foi compilado no ambiente Dev-C++ 4.9.9.2 usando glibc 2.2.3.</li> <li>• O sistema no qual ele foi compilado e executado foi Windows XP SP2.</li> <li>• A máquina utilizada foi um computador com CPU Pentium Quad Q6600 com 4 GB de memória RAM.</li> </ul>

Os resultados de execuções de alguns programas utilizados como exemplos são apresentados quando se julga que eles contribuem para melhorar o entendimento. Por outro lado, quando um resultado é trivial, ele não é apresentado. De qualquer modo, como todos os programas estarão disponíveis na internet (v. adiante), os resultados poderão ser verificados compilando-se e executando-se estes programas.

Exemplos de programas que envolvem localidade foram, em sua maioria, testados apenas no sistema operacional Linux (v. tabela anterior). A única alteração necessária para estes programas funcionarem em outro sistema operacional diz respeito à especificação de localidades no respectivo sistema. Em outros sistemas operacionais da família Unix, talvez nenhuma alteração desses programas seja necessária. Por outro

<sup>1</sup> Esta última opção só afeta programas que usam funções declaradas em `<math.h>`, `<complex.h>` e `<tgmath.h>`.

lado, a especificação de localidades em sistemas operacionais da família Windows é mais complicada e depende da versão do sistema considerada.

No site do livro da internet, <http://www.ulysseso.com/progc2.htm>, encontram-se os códigos-fonte de todos os exemplos apresentados no livro, além de outros programas não inseridos no texto. Este material é classificado de acordo com os capítulos correspondentes no livro e encontra-se comprimido em formato *zip*. Para baixá-lo, pode-se usar qualquer navegador de *web*.

## RECOMENDAÇÕES AO ESTUDANTE

Técnicas de programação não podem ser dominadas simplesmente estudando-se material escrito. Algumas sugestões para um melhor aprendizado das técnicas de programação apresentadas aqui são:

- *Edite, compile e execute os programas apresentados como exemplos no texto, tentando entender como eles funcionam.* Arquivos contendo estes exemplos podem ser obtidos via internet (v. mais adiante).
- *Utilize os exemplos apresentados como base para experimentos com os componentes da biblioteca padrão.* Muitos componentes, notadamente funções, podem ser usados de diversas maneiras que não foram totalmente exploradas nos exemplos apresentados. Nestes casos, recomenda-se que o estudante altere tais exemplos para testar outras opções oferecidas pelos respectivos componentes a que os exemplos se referem.
- *Use um depurador como ferramenta de aprendizagem de programação.* Utilizando um depurador para executar um programa mesmo quando ele não precisa ser depurado pode-se aprender muito mais sobre seu funcionamento. Esta sugestão é detalhada no **Volume I (Capítulo 6)**.
- *Desenvolva um estilo de programação.* Não existe um estilo único de escrita de programas que seja considerado o mais correto, mas, se você aderir aos conselhos básicos de estilo de programação sugeridos no **Volume I** e seguidos no presente volume, seu estilo estará bem fundamentado.
- *Tente responder todos os exercícios de revisão propostos ao final de cada capítulo.* Este volume não inclui seções contendo exclusivamente exercícios de programação, como ocorre no **Volume I**, mas muitos exercícios propostos no presente volume são de natureza prática e requerem a escrita de programas.

Além disso, muitas questões de natureza conceitual não podem ser relegadas a segundo plano.

O material complementar que o aluno deve ter disponível para acompanhamento do texto consiste em um computador e um ambiente de programação C. No site dedicado ao livro na internet encontram-se referências abundantes para páginas da web onde se podem encontrar ferramentas de programação, exemplos de códigos-fonte, artigos e outros materiais úteis na aprendizagem e no aprimoramento do conhecimento sobre programação em C.

## CONVENÇÕES ADOTADAS NO LIVRO

### *Convenções tipográficas*

*Itálico* é usado nas seguintes situações:

- Para enfatizar determinado ponto.
- Para representar componentes de construções da linguagem C ou de comandos de sistemas operacionais, compiladores, depuradores, etc. que devem ser substituídos por aquilo que realmente representam. Por exemplo, no comando do compilador gcc: `gcc -c nome-de-arquivo, nome-de-arquivo` é um guardador de lugar que deve ser substituído por um verdadeiro nome de arquivo quando o comando for utilizado.
- Em palavras que representam estrangeirismos. (Palavras de origem estrangeira, como array e buffer, reconhecidas pelos principais dicionários brasileiros não são representadas desta maneira.)

Utiliza-se **negrito** quando:

- Conceitos são definidos.
- Palavras-chave e identificadores reservados da linguagem C aparecem no corpo do texto, mas não é o caso quando eles aparecem em programas ou trechos de programas.

- Operadores da linguagem C são mencionados fora de programas ou trechos de programas.
- Em referências a capítulos, seções, tabelas, etc.

A fonte `courier` é utilizada nos seguintes casos:

- Na apresentação de programa ou trechos de programas.
- Na apresentação de comandos que aparecem numa interface de linha de comandos (*shell*).
- Em nomes de arquivos e diretórios.
- Na representação de constantes numéricas, caracteres e *strings*.
- Na representação gráfica de teclas ou combinações de teclas. Neste caso, as teclas são escritas em maiúsculas e colocadas entre colchetes, como, por exemplo, [CTRL-Z].

A fonte **`courier`** é utilizada para representar dados introduzidos por um usuário em exemplos de interação entre um programa e um usuário, enquanto que a fonte *`courier`* é utilizada para representar conteúdo impresso por um programa no meio de saída padrão.

## APRESENTAÇÃO DE COMPONENTES DA BIBLIOTECA PADRÃO DE C

As funções e macros com argumentos da biblioteca padrão são apresentadas seguindo o seguinte esquema (nesta ordem):

- *Nome-da-função* ou *nome-da-macro*
- **Incluir:** Cabeçalho que deve ser incluído para habilitar o uso da função ou macro.
- **Descrição:** Uma breve descrição da função ou macro.
- **Protótipo:** Protótipo da função ou da macro (v. a seguir).

- **Parâmetros:** Descrições dos parâmetros da função ou da macro.
- **Retorno:** Valor retornado pela função ou resultante da invocação da macro.
- **Observação(ões):** Informações adicionais que esclarecem o uso da função ou da macro, ou referência para outras seções do livro.
- **Exemplo:** Um exemplo de uso da função ou da macro, ou uma referência para o local onde tal exemplo pode ser encontrado.

Evidentemente, macros não possuem protótipo tal como este conceito é definido para funções, mas, para efeitos práticos de uso, pode-se imaginar que macros com argumentos possuem protótipos similares àqueles usados com funções.

Tipos, macros sem argumentos e variáveis globais da biblioteca padrão são apresentados em forma de tabela, a não ser que a relativa complexidade do componente justifique uma discussão mais longa. Neste caso, a discussão segue o seguinte esquema (nesta ordem):

- *Nome-do-componente*
- **Incluir:** Cabeçalho que deve ser incluído para permitir o uso do componente.
- **Descrição:** Descrição completa do componente.
- **Exemplo:** Um exemplo de uso do componente, ou uma referência para o local onde tal exemplo pode ser encontrado.

Para efeito de apresentação, rótulos de estruturas são tratados como se fossem tipos devido à proximidade com tipos derivados de C. De fato, se `rotulo` é um rótulo de estrutura, `struct rotulo` é um tipo.

## OUTRAS CONVENÇÕES

Alterações introduzidas pelo mais recente padrão ISO de C são identificadas no texto por (C99). É importante que o leitor dê atenção a esta indicação, pois uma dada característica introduzida por C99 pode não ter sido ainda implementada em seu compilador. Além disso, alguns compiladores requerem uma opção explícita (e.g., `-std=C99` no compilador gcc) para ativação do padrão C99.

O uso de acentuação infelizmente continua apresentando problemas para programadores cuja língua natural requer uso intensivo de letras acentuadas. Isso ocorre porque não há editores de programas que ofereçam suporte adequado ao uso de caracteres no formato UCN sugerido pelo padrão C99. Portanto, a maioria dos programas exibidos como exemplos apresenta palavras acentuadas no meio de saída apenas quando o objetivo é exatamente demonstrar como isso pode ser realizado.

Constantes são apresentadas do mesmo modo como elas são interpretadas em C. Na representação gráfica de números binários, o subscrito 2 (por exemplo, 11010010<sub>2</sub>) é utilizado apenas quando há iminência de ambiguidade. De modo análogo, quando existe impendente ambiguidade na representação gráfica de números octais e decimais, os subscritos 8 e 10 são utilizados, respectivamente.

Como, em C, o nome de uma função considerado isoladamente representa seu endereço, quando se faz referência a uma função (e não ao seu endereço), usa-se seu nome seguido de um par de parênteses [e.g., **printf()**]. Referências a arrays seguem raciocínio semelhante (e.g., `ar[]`), já que o nome de um array também representa seu endereço. Macros com argumentos não seguem o mesmo raciocínio, mas, devido à analogia entre macros com argumentos e funções, elas seguem a mesma notação usada com funções [e.g., **va\_copy()**].

Três pontos num fragmento de programa representam um trecho de programa (i.e., declarações e instruções) omitido por não ser relevante para a discussão em foco. Mas, três pontos em alusão ou cabeçalho de função representam uma lista de argumentos variáveis. Isto é, neste caso, os três pontos fazem realmente parte do cabeçalho ou da alusão.

As convenções utilizadas na escrita de identificadores são aquelas apresentadas no **Volume I (Capítulo 6)** e reproduzidas aqui para facilidade de referência:

- *Nomes de variáveis* começam com letra minúscula; quando o nome da variável é composto, utiliza-se letra maiúscula no início de cada palavra seguinte, incluindo palavras de ligação. Exemplo: `notaDoAluno`.
- *Nomes de tipos* seguem as mesmas regras para nomes de variáveis, mas começam sempre com a letra *t*. Exemplo: `tLista`.
- *Nomes de funções* começam com letra maiúscula e seguem as demais regras para nomes de variáveis. Exemplo: `OrdenaLista`.

- *Nomes de macros* utilizam apenas letras maiúsculas; se um nome de macro for composto, utiliza-se sublinha para separar os componentes. Exemplo: `VALOR_ABSOLUTO`.

Cabeçalho que fazem parte da biblioteca padrão de C são colocados entre “<” e “>” e escritos em *courier*. Por exemplo: `<stdio.h>`.

## SIMPLIFICAÇÕES

*Compilação versus construção de programa executável.* Compilar um arquivo-fonte não é o mesmo que transformá-lo em programa executável. Esta distinção é claramente apresentada no **Volume I**. Mesmo assim, como é comum, algumas vezes, *compilar um programa* é utilizado com o significado de *construir um programa executável*. Espera-se que o leitor possa deduzir do contexto o real significado de *compilação*.

*Declaração versus definição.* Definir uma variável ou função significa, em poucas palavras, causar a geração de código capaz de implementar a variável ou função. Por outro lado, declarar uma variável ou função significa simplesmente aludir à variável ou função. A linguagem C faz inequívoca distinção entre estes conceitos. Apesar disso, pode-se eventualmente utilizar *declaração* no texto quando o significado pretendido é *definição*. Novamente, espera-se que o leitor possa discernir os dois significados de *declaração*.

*Stream versus arquivo.* *Stream* é um conceito utilizado por C e outras linguagens de programação que permite que se processem arquivos sem dar atenção à origem ou destino de dados. Arquivo, por sua vez, representa qualquer dispositivo de onde se podem ler dados ou onde se podem depositá-los. Apesar de muitos leitores estarem familiarizados com o conceito mais usual de arquivo (e.g., uma coleção de bytes armazenada em disco rígido), eles não aparentam ter dificuldade em entender o conceito generalizado de arquivo utilizado por C e Unix. Assim, o termo *arquivo* é muitas vezes utilizado onde o termo mais adequado deveria ser *stream*.

*Unix versus Linux.* Do ponto de vista das referências feitas a estes sistemas neste livro, não há nenhuma diferença entre eles.

*Cabeçalho versus arquivo de cabeçalho.* O padrão ISO da linguagem C não especifica que devam existir arquivos de cabeçalhos contendo declarações dos diversos componentes da biblioteca padrão de C. O que esse padrão especifica é que devem existir *cabeçalhos* (e não exatamente *arquivos de cabeçalho*) cujos conteúdos especí-

ficos devem tornar-se disponíveis quando o programador utiliza tais cabeçalhos com diretivas **#include**. O modo como um dado compilador implementa isso não é especificado pelo padrão. Assim, uma diretiva como `#include <stdio.h>` não significa necessariamente incluir um arquivo denominado `stdio.h` no local onde esta diretiva se encontra. Em termos práticos, no entanto, esta sutileza só é importante para fabricantes de compiladores; i.e., para o programador, uma diretiva **#include** funciona do mesmo modo, quer um arquivo de cabeçalho seja realmente incluído ou não. Assim, tanto *arquivo de cabeçalho* quanto *cabeçalho* podem ser usados sem distinção.

*Indireção.* Esta palavra inexistente no vernáculo e é usada para representar *dereferencing* em inglês, que é o ato de acessar o conteúdo de uma porção de memória utilizando um ponteiro. Isto é, indireção significa acesso *indireto* a um conteúdo em memória por meio de um ponteiro, em vez do acesso direto promovido por variáveis comuns.

*Arrays como argumentos e retorno de funções.* Rigorosamente, em C, funções não recebem nem retornam arrays. Portanto, quando se fala no texto que uma função *recebe um array como argumento* ou *retorna um array*, o que se quer dizer é, respectivamente, que a função *recebe um ponteiro para o primeiro elemento de um array* ou *retorna um ponteiro para o primeiro elemento de um array*. A mesma simplificação elíptica aplica-se a *strings*, que também são arrays. Por outro lado, quando se menciona *ponteiro para um array* (ou *string*), o significado pretendido é *ponteiro para o endereço de um array* (ou *string*). Portanto, neste último caso, trata-se de um ponteiro para ponteiro.

*Caractere versus caractere monobyte.* No **Volume I** desta obra e na grande maioria dos livros de programação em C, existe apenas um tipo de caractere: aquele que pode ser armazenado numa variável do tipo **char** (i.e., num único byte). O presente volume lida com três categorias de caracteres: monobytes, multibytes e extensos, mas, por simplicidade, o termo *caractere monobyte* é usado apenas quando houver iminência de ambiguidade. Em outras palavras, o termo *caractere* (sem qualificação) denota *caractere monobyte*.

*String monobyte, string multibyte e string extenso.* Literalmente, *string monobyte* é um string constituído de um único byte. Neste livro, entretanto, para simplificar a terminologia, *string monobyte* tem o significado de *string constituído de caracteres monobytes*. De modo semelhante, *string multibyte* é um *string* constituído de caracteres multibytes, e *string extenso* é um *string* constituído de caracteres extensos.

*Unicode e ISO 10646 são códigos de caracteres?* Algumas vezes, Unicode e ISO 10646 são referidos no texto como códigos de caracteres quando, de fato, se deseja fazer referência aos códigos de caracteres especificados por estes padrões. Mas, con-



forme é explicitamente afirmado no texto, Unicode e ISO 10646 (principalmente, Unicode) vão bem mais além do que simplesmente estabelecer um mapeamento entre pontos de código e caracteres.

## PRÁTICAS INIMITÁVEIS

O livro adota práticas que o programador deve evitar em programação real. A adoção destas práticas possui justificativas no contexto do livro que são apresentadas a seguir:

- *Comentários didáticos.* Na prática, comentários devem ser escritos para programadores que conhecem a linguagem e não têm caráter didático como muitos comentários apresentados no texto. Naturalmente, os comentários apresentados aqui são didáticos porque estão inseridos num livro didático.
- *Números mágicos.* No **Volume I**, recomenda-se que números mágicos sejam substituídos por constantes simbólicas. Mesmo assim, há alguns números mágicos em exemplos apresentados no texto. A justificativa é que, nestes exemplos, não existe contexto necessário para encontrarem-se denominações significativas para associar a esses números.
- *Verificação de valores retornados por funções.* Muitas das funções da biblioteca padrão de C retornam valores que indicam se uma dada operação foi bem-sucedida ou não. Na prática, o programador deve sempre testar estes valores, em vez de assumir que uma dada operação sempre obtenha êxito. Em alguns exemplos apresentados no texto, estes valores não são testados para não desviar atenção daquilo que o exemplo pretende enfatizar.
- *Uso de tipos inteiros primitivos.* Os únicos tipos inteiros primitivos de C portáteis são **signed char**, **unsigned char** e **long long** (C99). Devido à natureza didática e a relativa simplicidade dos exemplos exibidos no livro, existe pouca possibilidade de ocorrerem problemas de portabilidade. Por isso, tipos inteiros não portáteis são fartamente utilizados. Na prática, para evitar problemas de portabilidade, o programador deve usar os tipos inteiros de larguras fixas definidos no cabeçalho `<stdint.h>` (v. **Capítulo 2**).

## CRÍTICAS, SUGESTÕES E COMENTÁRIOS

Como qualquer texto (ou programa) de considerável dimensão, este livro pode conter erros ou imperfeições que não puderam ser detectadas e corrigidas em tempo. Portanto, qualquer crítica ou indicação de erros encontrados no livro é bem-vinda.

Se, porventura, algum programa encontrado no livro ou no site dedicado a ele apresentar um comportamento inesperado que possa caracterizar um erro de programação (*bug*), não hesite em entrar em contato.

Qualquer questão de natureza técnica ou comentário útil pode ser enviado utilizando formulário próprio no site do livro na internet: <http://www.ulysseso.com/progc2.htm>. Neste site, também se encontra vasto material que complementa o livro, em particular, e sobre programação em C, em geral.

*Ulysses de Oliveira*

Julho de 2009

# SUMÁRIO

<b>Capítulo 1 - Introdução à biblioteca padrão de C.....</b>	<b>1</b>
1.1 Cabeçalhos .....	3
1.2 Componentes.....	4
1.3 Recomendações gerais de uso .....	5
1.4 Macros e funções.....	8
1.5 Cabeçalhos pré-compilados.....	10
1.6 Visão geral dos cabeçalhos padronizados .....	10
1.6.1 <assert.h>.....	11
1.6.2 <complex.h> (C99).....	11
1.6.3 <ctype.h> .....	13
1.6.4 <errno.h> .....	14
1.6.5 <env.h> (C99).....	14
1.6.6 <float.h>.....	15
1.6.7 <inttypes.h> (C99).....	17
1.6.8 <iso646.h> .....	18
1.6.9 <limits.h>.....	18
1.6.10 <locale.h> .....	19
1.6.11 <math.h> .....	20
1.6.12 <setjmp.h> .....	25
1.6.13 <signal.h> .....	25
1.6.14 <stdarg.h> .....	26
1.6.15 <stdbool.h> (C99).....	27
1.6.16 <stddef.h> .....	27
1.6.17 <stdint.h> (C99) .....	28
1.6.18 <stdio.h> .....	30
1.6.19 <stdlib.h> .....	33
1.6.20 <string.h>.....	36
1.6.21 <tgmath.h> (C99) .....	38
1.6.22 <time.h>.....	38
1.6.23 <wchar.h> .....	39
1.6.24 <wctype.h> .....	44
1.7 Componentes repetidos em cabeçalhos.....	45
1.7.1 Tipo size_t.....	45
1.7.2 Macro NULL .....	46
1.7.3 Outros componentes repetidos.....	47
1.8 Sistemas com hospedeiro e sistemas livres .....	47

1.9 Exercícios de Revisão .....	48
<b>Capítulo 2 - Números inteiros .....</b>	<b>51</b>
2.1 Introdução.....	53
2.2 Tipos inteiros primitivos.....	53
2.3 Propriedades dos tipos inteiros primitivos: <limits.h> .....	55
2.4 Portabilidade de inteiros I: <stdint.h> (C99).....	58
2.4.1 Tipos .....	58
2.4.2 Macros .....	59
2.5 Portabilidade de inteiros II: <inttypes.h> (C99) .....	62
2.5.1 Tipo imaxdiv_t.....	62
2.5.2 Macros .....	63
2.5.3 Funções.....	66
imaxdiv() (C99) .....	67
strtoimax() (C99) .....	68
strtoumax() (C99).....	70
westoimax() (C99) .....	72
westoumax() (C99) .....	73
2.6 Operações aritméticas inteiras.....	75
2.6.1 Tipos .....	75
div_t .....	75
ldiv_t .....	75
lldiv_t (C99).....	75
2.6.2 Funções.....	75
abs().....	75
div().....	76
labs() .....	77
llabs() (C99) .....	78
ldiv().....	79
lldiv() (C99) .....	81
2.7 Exercícios de Revisão .....	82
<b>Capítulo 3 - Números de ponto flutuante reais.....</b>	<b>85</b>
3.1 Introdução.....	87
3.2 Tipos primitivos de ponto flutuante reais .....	87
3.3 Conceitos fundamentais de aritmética de ponto flutuante.....	89
3.3.1 Underflow e overflow .....	89
3.3.2 Representações .....	89

3.3.3 Erros de domínio e de intervalo.....	90
Erros de domínio.....	91
Erros de intervalo.....	91
3.3.4 Exceções de ponto flutuante .....	91
3.3.5 Modos de arredondamento .....	92
3.3.6 Precisão.....	93
3.3.7 Ordenação.....	94
3.4 Pragmas para operações de ponto flutuante .....	94
3.4.1 Pragma FP_CONTRACT .....	94
3.4.2 Pragma FENV_ACCESS .....	95
3.5 Propriedades de números de ponto flutuante: <float.h>.....	96
3.6 Operações de ponto flutuante reais: <math.h> .....	102
3.6.1 Tipos .....	102
double_t (C99) .....	102
float_t (C99).....	103
3.6.2 Macros .....	103
FP_FAST_FMA (C99).....	103
FP_FAST_FMAF (C99) .....	103
FP_FAST_FMAL (C99) .....	104
FP_ILOGB0 (C99).....	104
FP_ILOGBNAN (C99).....	104
FP_INFINITE (C99).....	104
FP_NAN (C99).....	104
FP_NORMAL (C99).....	105
FP_SUBNORMAL (C99).....	105
FP_ZERO (C99) .....	105
HUGE_VAL.....	105
HUGE_VALF (C99) .....	105
HUGE_VALL (C99).....	106
INFINITY .....	106
MATH_ERRNO (C99) .....	106
MATH_ERREXCEPT (C99).....	106
math_errhandling (C99).....	107
NAN (C99).....	107
Exemplo de uso de macros definidas em <math.h>.....	107
3.6.3 Visão geral das funções declaradas em <math.h> .....	108
3.6.4 Funções trigonométricas.....	111
acos().....	111
asin() .....	111

atan()	112
atan2()	112
cos()	114
sin()	114
tan()	115
3.6.5 Funções hiperbólicas	115
acosh() (C99)	115
asinh() (C99)	116
atanh() (C99)	116
cosh()	117
sinh()	118
tanh()	118
3.6.6 Funções de arredondamento	119
ceil()	119
floor()	120
llrint() (C99)	121
llround() (C99)	121
lrint() (C99)	123
lround() (C99)	123
nearbyint() (C99)	124
nextafter() (C99)	125
nexttoward() (C99)	126
rint() (C99)	128
round() (C99)	129
trunc() (C99)	130
3.6.7 Funções de erro	131
erf() (C99)	131
erfc() (C99)	131
3.6.8 Funções exponenciais e logarítmicas	132
cbrt() (C99)	132
exp()	133
exp2() (C99)	134
expm1() (C99)	134
frexp()	135
ilogb() (C99)	137
ldexp()	137
log()	138
log10()	139
log1p() (C99)	140

log2() (C99) .....	140
logb() (C99) .....	142
pow() .....	143
scalbln() (C99) .....	144
scalbn() (C99) .....	145
sqrt() .....	146
3.6.9 Funções de comparação .....	147
fdim() (C99) .....	147
fmax() (C99) .....	147
fmin() (C99) .....	148
3.6.10 Funções gama .....	149
lgamma() (C99) .....	149
tgamma() (C99) .....	150
3.6.11 Funções de divisão .....	151
fmod() .....	151
remainder() (C99) .....	152
remquo() (C99) .....	152
3.6.12 Outras funções declaradas em <math.h> .....	154
copysign() (C99) .....	154
fabs() .....	155
fma() (C99) .....	155
hypot() (C99) .....	156
modf() .....	157
nan() (C99) .....	158
signbit() (C99) .....	159
3.6.13 Macros de classificação .....	160
fpclassify() (C99) .....	160
isfinite() (C99) .....	162
isgreater() (C99) .....	162
isgreaterequal() (C99) .....	163
isinf() (C99) .....	163
isless() (C99) .....	164
islessequal() (C99) .....	164
islessgreater() (C99) .....	165
isnan() (C99) .....	166
isnormal() (C99) .....	166
isunordered() (C99) .....	167
3.7 Tratamento de exceções e arredondamento: <fenv.h> (C99) .....	169
3.7.1 Tipos .....	169

fenv_t .....	169
fexcept_t .....	169
3.7.2 Macros .....	169
FE_ALL_EXCEPT .....	170
FE_DFL_ENV .....	170
FE_DIVBYZERO .....	170
FE_DOWNWARD .....	170
FE_INEXACT .....	171
FE_INVALID .....	171
FE_OVERFLOW .....	171
FE_TONEAREST .....	172
FE_TOWARDZERO .....	172
FE_UNDERFLOW .....	172
FE_UPWARD .....	172
3.7.3 Funções .....	173
feclearexcept() (C99) .....	173
fegetenv() (C99) .....	173
fegetexceptflag() (C99) .....	174
fegetround() (C99) .....	175
feholdexcept() (C99) .....	176
feraiseexcept() (C99) .....	176
fesetenv() (C99) .....	179
fesetexceptflag() (C99) .....	181
fesetround() (C99) .....	184
fetestexcept() (C99) .....	186
feupdateenv() (C99) .....	187
3.8 Exercícios de Revisão .....	191

## **Capítulo 4 - Números de ponto flutuante complexos e macros genéricas .....197**

4.1 Introdução .....	199
4.2 Tipos primitivos de ponto flutuante complexos .....	199
4.3 Pragma CX_LIMITED_RANGE .....	200
4.4 Suporte para números complexos: <complex.h> (C99) .....	202
4.4.1 Macros .....	202
4.4.2 Visão geral das funções declaradas em <complex.h> .....	202
4.4.3 Funções trigonométricas complexas .....	204
cacos() (C99) .....	204
casin() (C99) .....	204



catan() (C99) .....	205
ccos() (C99) .....	205
csin() (C99) .....	206
ctan() (C99) .....	206
4.4.4 Funções hiperbólicas complexas .....	208
cacosh() (C99) .....	208
casinh() (C99) .....	208
catanh() (C99) .....	209
ccosh() (C99) .....	209
csinh() (C99) .....	210
ctanh() (C99) .....	210
4.4.5 Funções exponenciais e logarítmicas complexas .....	212
cexp() (C99) .....	212
clog() (C99) .....	212
cpow() (C99) .....	213
csqrt() (C99) .....	213
4.4.6 Outras funções declaradas em <complex.h> .....	214
cabs() (C99) .....	215
carg() (C99) .....	215
cimag() (C99) .....	215
conj() (C99) .....	216
cproj() (C99) .....	216
creal() (C99) .....	217
4.5 Macros aritméticas genéricas: <tgmath.h> (C99) .....	218
4.6 Exercícios de Revisão .....	223

## **Capítulo 5 - Localização e datação.....225**

5.1 Introdução.....	227
5.1.1 Localidades .....	227
5.1.2 Implementação de localidades em sistemas operacionais da família Unix .....	228
5.1.3 Localização e internacionalização .....	230
5.1.4 A base de dados CLDR e a biblioteca ICU .....	230
5.2 Localização de programas: <locale.h> .....	231
5.2.1 Estruturas lconv .....	231
5.2.2 Macros .....	237
5.2.3 Funções .....	238
localeconv() .....	238
setlocale() .....	240
5.3 Datas e horas: <time.h> .....	243

5.3.1 Tipos .....	244
clock_t.....	244
size_t .....	244
time_t .....	244
tm .....	244
5.3.2 Macros .....	246
NULL.....	246
CLOCKS_PER_SEC .....	246
5.3.3 Funções.....	247
asctime().....	247
clock() .....	247
ctime().....	249
difftime() .....	250
gmtime().....	251
localtime() .....	252
mktime().....	253
strftime() .....	255
time().....	257
5.4 Exercícios de Revisão .....	258
<b>Capítulo 6 - Caracteres e strings monobytes.....</b>	<b>261</b>
6.1 Introdução.....	263
6.1.1 Conjunto básico de caracteres .....	263
6.1.2 Códigos de caracteres .....	265
6.1.3 Páginas de código .....	267
6.1.4 Problemas com caracteres monobytes.....	268
6.1.5 Caracteres constantes.....	269
6.1.6 Strings constantes .....	270
6.2 Classificação e transformação de caracteres: <ctype.h> .....	271
6.2.1 Funções de classificação de caracteres .....	271
isalnum() .....	271
isalpha().....	272
isblank() (C99) .....	273
iscntrl().....	274
isdigit().....	275
isgraph() .....	275
islower() .....	276
isprint().....	276
ispunct() .....	277

isspace()	278
isupper()	280
isxdigit()	280
6.2.2 Funções de transformação de caracteres	282
tolower()	282
toupper()	282
6.3 Processamento de strings e blocos: <string.h>	284
6.3.1 Tipo size_t	284
6.3.2 Macro NULL	284
6.3.3 Funções de processamento de blocos	284
memchr()	284
memcmp()	285
memcpy()	287
memmove()	289
memset()	290
6.3.4 Funções de processamento de strings	291
strcat()	291
strchr()	292
strcpy()	293
strcspn()	295
strerror()	297
strlen()	298
strncat()	299
strncpy()	301
strpbrk()	302
strrchr()	303
strspn()	304
strstr()	306
strtok()	307
6.4 Introdução à colação de strings	309
6.4.1 Colação versus ordenação	310
6.4.2 Funções de colação de strings	311
strcmp()	312
strncmp()	316
strcoll()	317
strxfrm()	319
6.5 Funções de conversão de strings em números	322
6.5.1 Conversões de strings em números inteiros	323

atoi(), atol() e atoll() (C99) .....	323
strtol(), strtoll() (C99), strtoul(), strtoull() (C99) .....	325
6.5.2 Conversões de strings em números de ponto flutuante reais .....	327
atof() .....	327
strtod(), strtodf() (C99), strtold() (C99) .....	329
6.5.3 Outras funções de conversão de strings em números .....	331
6.6 Exercícios de Revisão .....	332
<b>Capítulo 7 - Caracteres extensos e multibytes I: conceitos .....</b>	<b>335</b>
7.1 Introdução .....	337
7.2 Caracteres e strings extensos .....	337
7.3 Caracteres e strings multibytes .....	338
7.3.1 Codificações multibytes com estado .....	339
7.3.2 Codificações multibytes sem estado .....	340
7.4 Caracteres extensos versus caracteres multibytes .....	340
7.5 Códigos de caracteres extensos .....	341
7.5.1 Unicode .....	341
7.5.2 ISO 10646 .....	344
7.5.3 Diferenças entre Unicode e ISO 10646 .....	344
7.6 Esquemas de codificação de caracteres .....	344
7.6.1 UTF-8 .....	345
7.6.2 UTF-16, UTF-16BE e UTF-16LE .....	348
7.6.3 UTF-32, UTF-32BE e UTF-32LE .....	349
7.6.4 UCS-2 .....	350
7.6.5 UCS-4 .....	350
7.6.6 Escolha de um esquema de codificação .....	350
7.7 Colação avançada .....	351
7.7.1 Colação e localidade .....	351
7.7.2 Colação em múltiplos níveis .....	352
7.7.3 Casos especiais de colação .....	354
Acentuação francesa .....	354
Caracteres com contração .....	355
Caracteres com expansão .....	356
Colação aproximada .....	356
7.7.4 Algoritmo de Colação Unicode .....	356
Elementos de colação variáveis .....	357
Chaves de ordenação .....	359
Tabela DUCET .....	360
Normalização .....	362

Resumo do algoritmo UCA.....	363
7.7.5 Busca .....	364
7.7.6 Exemplo.....	364
7.8 Exercícios de Revisão .....	367
<b>Capítulo 8 - Caracteres extensos e multibytes II: suporte .....</b>	<b>369</b>
8.1 Introdução.....	371
8.2 Conceitos e terminologias .....	372
8.3 Implementações de caracteres extensos e multibytes em C.....	375
8.3.1 Caracteres e strings extensos em C.....	375
8.3.2 Caracteres e strings multibytes em C .....	377
8.4 Conversões entre caracteres e strings extensos e multibytes: <stdlib.h>....	379
8.4.1 Preliminares.....	379
8.4.2 Funções de conversão entre caracteres e strings extensos e multibytes I.....	380
mblen().....	380
mbstowcs().....	381
mbtowc().....	385
wcstombs().....	388
wctomb() .....	393
8.5 Suporte para caracteres multibytes e extensos: <wchar.h> .....	395
8.5.1 Tipos .....	396
mbstate_t.....	396
size_t .....	396
tm .....	396
wchar_t.....	397
wctype_t.....	397
wint_t .....	397
8.5.2 Macros .....	397
NULL .....	397
WCHAR_MAX .....	398
WCHAR_MIN .....	398
WEOF .....	398
8.5.3 Funções de conversão entre caracteres e strings extensos e multibytes II.....	399
btowc() .....	399
mbrlen().....	401
mbrtowc().....	405
mbsinit().....	408
mbsrtowcs().....	409
wctomb().....	412

wcsrtombs()	414
wctob()	417
8.5.4 Funções de processamento de arrays de caracteres extensos	419
wmemchr()	419
wmemcmp()	420
wmemcpy()	421
wmemmove()	422
wmemset()	424
8.5.5 Funções de processamento de strings extensos	425
wscat()	425
wcschr()	426
wscmp()	428
wscoll()	429
wscpy()	433
wscspn()	434
wcsftime()	435
wcslen()	437
wcsncat()	438
wcsncmp()	440
wcsncpy()	442
wcsprk()	443
wcsrchr()	445
wcssp()	446
wcsstr()	447
wcstok()	448
wcxfrm()	450
8.5.6 Funções de conversão de strings extensos em números	453
wcstol(), wcstoll() (C99), wcstoul(), wcstoull() (C99)	454
wcstod(), wcstof() (C99), wcstold()	456
8.6 Classificação e transformação de caracteres extensos: <wctype.h>	459
8.6.1 Tipos	459
wctrans_t	459
wctype_t	459
wint_t	460
8.6.2 Macro WEOF	460
8.6.3 Funções de classificação de caracteres extensos	460
iswctype()	467
wctype()	469
8.6.4 Funções de transformação de caracteres extensos	471

towctrans() .....	471
tolower() .....	471
toupper() .....	472
wctrans() .....	473
8.7 Exercícios de Revisão .....	475
<b>Capítulo 9 - Funções com listas de argumentos variáveis .....</b>	<b>479</b>
9.1 Introdução .....	481
9.2 Suporte para listas de argumentos variáveis: <stdarg.h> .....	482
9.2.1 Tipo va_list .....	482
9.2.2 Macros .....	483
va_start() .....	483
va_arg() .....	483
va_end() .....	484
va_copy() (C99) .....	485
9.3 Como criar funções com listas de argumentos variáveis .....	486
9.3.1 Uso direto de listas de argumentos variáveis .....	486
9.3.2 Uso indireto de listas de argumentos variáveis .....	488
9.4 Exemplos de funções com listas de argumentos variáveis .....	490
9.5 Exercícios de Revisão .....	498
<b>Capítulo 10 - Entrada e saída .....</b>	<b>501</b>
10.1 Introdução .....	503
10.2 Conceitos fundamentais de entrada e saída .....	503
10.2.1 Processamento de entrada e saída .....	503
10.2.2 Streams .....	503
10.2.3 Processamento de arquivos .....	503
10.2.4 Formatos de arquivos .....	504
10.2.5 Acesso a arquivos .....	504
10.2.6 Streams padronizados: stdin, stdout e stderr .....	505
10.2.7 Entrada e saída formatadas .....	505
10.2.8 Buffering .....	506
10.2.9 Orientação de <i>streams</i> .....	506
10.3 Processamento de arquivos em C na prática .....	508
10.4 Tipos .....	510
fpos_t .....	510
FILE .....	510
10.5 Macros .....	511

10.6 Variáveis globais.....	512
10.7 Funções.....	513
10.7.1 Abertura e fechamento de arquivos .....	513
fopen().....	513
fclose() .....	515
freopen().....	517
10.7.2 Gerenciamento de buffers.....	519
setbuf() .....	519
setvbuf() .....	520
fflush().....	523
10.7.3 Processamento de caracteres monobytes .....	524
getc() e fgetc().....	524
putc() e fputc() .....	526
getchar() .....	527
putchar() .....	528
10.7.4 Processamento de linhas .....	529
fgets() .....	529
fputs() .....	531
gets() .....	532
puts() .....	534
10.7.5 Processamento de blocos .....	535
fread().....	535
fwrite() .....	537
10.7.6 Entrada formatada: a família de funções scanf.....	539
fscanf() .....	541
scanf() .....	542
vfscanf() .....	544
vscanf() .....	547
10.7.7 Saída formatada: a família de funções printf.....	549
fprintf().....	552
printf() .....	553
vfprintf().....	554
vprintf() .....	555
10.7.8 Formatação em memória .....	557
sscanf().....	558
vsscanf() (C99).....	560
snprintf() (C99) .....	562
sprintf() .....	566
vsprintf() .....	568



vsnprintf() (C99) .....	570
10.7.9 Funções de posicionamento (acesso direto) .....	572
fseek() .....	572
ftell() .....	576
fgetpos() .....	579
fsetpos().....	580
rewind().....	582
10.7.10 Gerenciamento de arquivos .....	584
remove().....	584
rename().....	585
10.7.11 Arquivos temporários.....	586
tmpfile().....	586
tmpnam().....	587
10.7.12 Detecção de erros em <i>streams</i> .....	<b>590</b>
feof() .....	590
clearerr().....	591
ferror().....	592
10.7.13 Funções perror() e ungetc() .....	593
perror() .....	593
ungetc() .....	594
10.8 Funções de entrada e saída de caracteres e strings extensos.....	597
fgetwc() e getwc() .....	597
fgetws() .....	598
fputwc() e putwc().....	600
fputws() .....	601
fwide().....	602
fwprintf().....	605
fwscanf() .....	606
getwchar() .....	607
putwchar() .....	608
swprintf().....	608
swscanf() .....	610
ungetwc() .....	611
vfwprintf().....	612
vfwscanf() (C99) .....	615
vswprintf().....	616
vswscanf() (C99).....	618
vwprintf() .....	621
vwscanf() (C99) .....	622

wprintf()	624
wscanf()	624
10.9 Exercícios de Revisão	626

## **Capítulo 11 - Suporte para legibilidade e robustez .....633**

11.1 Introdução	635
11.2 Macros para o tipo booleano: <stdbool.h> (C99)	636
11.3 Nomes legíveis para operadores: <iso646.h>	637
11.4 Macro assert(): <assert.h>	639
11.5 Classificação de erros: <errno.h>	640
11.5.1 Macros	641
11.5.2 Variável Global errno	641
11.6 Tratamento de sinais: <signal.h>	644
11.6.1 Tipo sig_atomic_t	646
11.6.2 Macros	647
SIGFPE	648
SIGILL	648
SIGSEGV	649
SIGABRT	649
SIGTERM	649
SIGINT	650
SIG_DFL	650
SIG_IGN	650
SIG_ERR	651
11.6.3 Funções	651
raise()	651
signal()	652
11.7 Desvios generalizados: <setjmp.h>	663
11.7.1 Tipo jmp_buf	663
11.7.2 Funções longjmp() e setjmp()	663
setjmp()	664
longjmp()	665
11.7.3 Como entender longjmp() e setjmp()	668
11.7.4 Tratamento de exceções usando longjmp() e setjmp()	674
11.8 Relação entre sinais, interrupções e exceções	681
11.9 Exercícios de Revisão	682

<b>Capítulo 12- Miscelânea de tipos, funções e macros .....</b>	<b>687</b>
12.1 Introdução.....	689
12.2 Cabeçalho <stdlib.h>.....	689
12.2.1 Tipos .....	690
size_t .....	690
div_t, ldiv_t, lldiv_t.....	690
wchar_t.....	690
12.2.2 Macros .....	690
EXIT_FAILURE.....	690
EXIT_SUCCESS.....	690
MB_CUR_MAX.....	691
RAND_MAX.....	691
NULL.....	691
12.2.3 Funções de controle de processos.....	691
abort().....	691
atexit().....	693
_Exit() (C99).....	694
exit().....	695
getenv() .....	696
system().....	697
12.2.4 Funções de geração de números aleatórios.....	700
rand().....	700
srand() .....	701
12.2.5 Funções de alocação dinâmica de memória .....	703
calloc() .....	703
malloc().....	704
realloc().....	706
free().....	708
12.2.6 Funções de busca e ordenação de dados.....	710
bsearch().....	710
qsort().....	713
12.2.7 Funções de aritmética inteira.....	715
12.2.8 Funções de conversão de <i>strings</i> em números inteiros .....	715
12.2.9 Funções de conversão de <i>strings</i> em números de ponto flutuante reais..	716
12.2.10 Funções de conversão entre caracteres extensos e multibytes .....	716
12.3 Miscelânea de tipos e macros: <stddef.h> .....	717
12.3.1 Tipos .....	717
ptrdiff_t.....	717
size_t .....	718

wchar_t.....	718
12.3.2 Macros .....	718
NULL .....	718
offsetof().....	718
12.4 Exercícios de Revisão .....	720
<b>Capítulo 13 - Portabilidade de programas em C.....</b>	<b>723</b>
13.1 Introdução .....	725
13.2 Portabilidade e padronização .....	725
13.3 Ordenação de bytes (endianess).....	728
13.4 Alinhamento de variáveis e preenchimento de estruturas .....	739
13.5 Aritmética inteira.....	749
13.5.1 Tipos inteiros não portáveis.....	749
13.5.2 Overflow .....	749
13.5.3 signed char e unsigned char.....	751
13.6 Caracteres e strings.....	752
13.7 Compiladores .....	756
13.8 Sistemas operacionais .....	757
13.9 Representações de quebra de linha.....	758
13.10 Aspectos pragmáticos de portabilidade .....	760
13.11 Exercícios de Revisão .....	762
<b>Apêndice A - Construtores da linguagem C.....</b>	<b>767</b>
A.1 Introdução.....	769
A.1.1 Palavras-chave .....	769
A.1.2 Identificadores reservados .....	770
A.2 Referências de construtores usados em C .....	771
A.3 Identificadores reservados para uso futuro .....	805
A.4 Estatísticas da linguagem C.....	806
A.4.1 Linguagem C .....	806
A.4.2 Biblioteca padrão de C.....	807
<b>Apêndice B - Especificadores de formato de entrada e saída.....</b>	<b>809</b>
B.1 Introdução.....	811
B.1.1 Recomendações de uso .....	811
B.1.2 Notação .....	812
B.2 Especificadores de formato da família printf.....	813
B.2.1 Formato geral .....	813

B.2.2 Sinalizadores .....	813
B.2.3 Largura .....	814
B.2.4 Precisão .....	814
B.2.5 Tipo de especificador .....	814
B.2.6 Prefixo .....	815
B.2.7 Composições de especificadores de formato .....	817
a e A (C99) .....	817
c .....	818
d e i .....	819
e, E .....	820
f .....	821
F (C99) .....	822
g e G .....	822
n .....	823
o .....	824
p .....	825
s .....	826
u .....	827
x e X .....	828
% .....	829
B.2.8 Exemplos adicionais .....	829
B.2.9 Resumo de especificadores de formato .....	831
B.3 Especificadores de formato da família scanf .....	834
B.3.1 Formato geral .....	834
B.3.2 Asterisco .....	834
B.3.3 Largura .....	836
B.3.4 Tipo de especificador .....	836
B.3.5 Prefixo .....	837
B.3.6 Composições de especificadores de formato .....	837
a e A (C99) .....	838
c .....	838
d .....	839
e, E .....	840
f .....	840
F (C99) .....	841
g e G .....	841
i .....	841
n .....	843
o .....	844

p.....	845
s.....	846
u.....	847
x e X.....	848
[caracteres].....	849
%.....	850
B.3.7 Resumo de especificadores de formato.....	850
B.4 Diferenças entre especificadores de formato das famílias printf e scanf....	854
<b><i>Apêndice C - Especificadores de formato de datas e horas .....</i></b>	<b><i>855</i></b>
<b><i>Apêndice D - Erros comuns de programação em C .....</i></b>	<b><i>871</i></b>
D.1 Introdução.....	873
D.2 Operadores.....	873
D.2.1 Uso de atribuição em vez de igualdade ou vice-versa.....	873
D.2.2 Uso Incorreto de regras de precedência e associatividade.....	874
D.2.3 Uso de && em vez de    ou vice-versa.....	875
D.2.4 Uso de & em vez de &&.....	876
D.2.5 Uso de   em vez de    .....	876
D.2.6 Operadores lógicos de C não são comutativos .....	876
D.2.7 Separação indevida de símbolos .....	877
D.2.8 Suposições sobre ordem de avaliação de operandos .....	877
D.3 Estruturas de controle.....	878
D.3.1 Uso indevido de ponto e vírgula .....	878
D.3.2 Instrução switch-case sem break .....	879
D.3.3 Instrução do-while confundida com REPEAT-UNTIL.....	880
D.3.4 else que não corresponde ao if desejado.....	881
D.4 Definições incorretas de funções.....	882
D.4.1 Recursão sem fim.....	882
D.4.2 Retorno de zumbis .....	883
D.4.3 Chamadas sem o devido retorno.....	886
D.4.4 Desconhecimento de regras de escopo .....	887
D.4.5 Vazamento de memória.....	888
D.4.6 Uso incorreto de variáveis locais de duração fixa .....	888
D.5 Entrada e saída.....	891
D.5.1 Uso incorreto de scanf().....	891
D.5.2 Uso incorreto de printf() .....	893
D.5.3 String de formatação incorreto .....	894

D.5.4 Lixo no buffer associado à entrada padrão .....	896
D.5.5 Não existe uso correto para gets() .....	897
D.5.6 Uso incorreto de EOF .....	897
D.5.7 Uso incorreto de feof().....	898
D.5.8 Prompts que o usuário não lê.....	899
D.6 Chamadas incorretas de funções .....	899
D.6.1 Suposições sobre ordem de avaliação de parâmetros .....	899
D.6.2 Argumentos incorretos.....	900
D.6.3 Omissão de teste de condição de exceção .....	901
D.6.4 Omissão de teste em alocação dinâmica de memória.....	904
D.6.5 Implementações incorretas da biblioteca padrão.....	905
D.6.6 Chamadas de funções sem parâmetros .....	905
D.6.7 Alusões e ponteiros para funções sem protótipos .....	906
D.7 Pré-processador .....	907
D.7.1 Arquivos de programa não devem ser incluídos.....	907
D.7.2 Inclusão múltipla de arquivos.....	907
D.7.3 Inclusão recursiva de arquivos.....	907
D.7.4 Definições de tipos usando #define.....	908
D.7.5 Definições incorretas de macros .....	908
D.7.6 Chamadas incorretas de macros.....	908
D.8 Ponteiros .....	909
D.8.1 Ponteiros não iniciados .....	909
D.8.2 Ponteiros órfãos .....	909
D.8.3 Ponteiro incrementado passa a apontar para outro endereço.....	911
D.8.4 Indireção de ponteiro nulo .....	915
D.9 Arrays e strings.....	916
D.9.1 Desrespeito aos limites de arrays.....	916
D.9.2 Strings constantes devem ser considerados constantes .....	919
D.9.3 Comparação incorreta de strings .....	919
D.9.4 Strings constantes sem acessibilidade .....	920
D.9.5 Uso de sizeof em vez de strlen().....	921
D.9.6 Funções que não limitam o número de caracteres escritos.....	921
D.9.7 Funções que nem sempre produzem strings .....	922
D.9.8 Strings constantes versus caracteres constantes .....	923
D.9.9 Alocação de espaço insuficiente para conter um string .....	924
D.10 Alocação dinâmica de memória .....	924
D.10.1 Zumbis também assombram o heap .....	924
D.10.2 Uso incorreto de free().....	927
D.10.3 Lista encadeada não é array.....	928

D.11 Operações inteiras .....	928
D.11.1 Overflow .....	928
D.11.2 Erro de sinal .....	930
D.12 Operações de ponto flutuante .....	933
D.12.1 Arredondamentos .....	933
D.12.2 Comparações .....	935
D.12.3 Overflow e underflow .....	937
D.13 Comentários .....	938
D.13.1 Comentários não terminados .....	938
D.13.2 Comentários mal posicionados .....	938
D.13.3 Comentários aninhados .....	939
D.14 Erros de portabilidade .....	939
D.14.1 void main() .....	940
D.14.2 fflush(stdin) .....	940
D.14.3 conio.h .....	940
D.15 Outros erros comuns .....	941
D.15.1 Uso de variáveis não iniciadas .....	941
D.15.2 Conversões inadequadas de tipos .....	942
D.15.3 Ambiguidades em definições e alusões de variáveis globais .....	943
D.15.4 Nomes de identificadores trocados .....	943
D.15.5 Colisões de identificadores .....	944

<b><i>Bibliografia</i></b> .....	<b>945</b>
----------------------------------	------------



# *Capítulo 1*

---

*Introdução à biblioteca padrão de C*

## 1.1 CABEÇALHOS

A biblioteca padrão de C complementa esta linguagem provendo recursos adicionais que facilitam a escrita de programas. Os componentes da biblioteca padrão de C são declarados ou definidos em **cabeçalhos padronizados**. Atualmente, existem 24 cabeçalhos padronizados, apresentados na **Tabela 1-1** juntamente com a versão do padrão ISO que introduziu cada um deles.

CABEÇALHO	PADRÃO ISO
<assert.h>	C89
<complex.h>	C99
<ctype.h>	C89
<errno.h>	C89
<fenv.h>	C99
<float.h>	C89
<inttypes.h>	C99
<iso646.h>	Emenda 1 (1995)
<limits.h>	C89
<locale.h>	C89
<math.h>	C89
<setjmp.h>	C89
<signal.h>	C89
<stdarg.h>	C89
<stdbool.h>	C99
<stddef.h>	C89
<stdint.h>	C99
<stdio.h>	C89
<stdlib.h>	C89
<string.h>	C89
<tgmath.h>	C99
<time.h>	C89
<wchar.h>	Emenda 1 (1995)
<wctype.h>	Emenda 1 (1995)

Tabela 1-1: Cabeçalhos da biblioteca padrão de C.

Na maioria das implementações, o nome de um cabeçalho coincide com o nome de um arquivo que implementa o correspondente cabeçalho<sup>1</sup>. O padrão C99, no entanto, não requer que esta abordagem seja utilizada, de modo que um fabricante de compilador tem liberdade de, por exemplo, implementar cabeçalhos padronizados no próprio compilador.

Para usar algum componente de um cabeçalho padronizado, deve-se incluir o respectivo cabeçalho por meio de uma diretiva **#include**. Uma inclusão de cabeçalho padronizado pode ser feita em qualquer ordem com relação a outras inclusões e pode ocorrer mais de uma vez, embora isto nunca seja necessário. Além disso, dois ou mais cabeçalhos padronizados que definem um mesmo componente (e.g., `<stddef.h>` e `<stdlib.h>`) podem ser incluídos sem problemas.

Os símbolos `< e >` que envolvem o nome de um arquivo de cabeçalho informam o pré-processador que ele deve dar preferência à inclusão de um cabeçalho padronizado em detrimento de um eventual arquivo com o mesmo nome. Por outro lado, o uso de aspas em torno de um nome de arquivo de cabeçalho indica a intenção de incluir um arquivo específico de um dado projeto.

## 1.2 COMPONENTES

Os componentes dos cabeçalhos padronizados são divididos nas seguintes categorias:

- Definições de tipos e rótulos de estruturas
- Definições de macros
- Declarações (ou alusões) de variáveis globais
- Declarações (ou alusões) de funções

As apresentações dos cabeçalhos padronizados neste livro (inclusive nas seções a seguir) serão feitas segundo a ordem das categorias mencionadas. Além disto, os componentes de cada categoria serão apresentados em ordem alfabética, a não ser que haja uma dependência entre eles que justifique uma ordem de apresentação mais lógica [e.g., o uso da função **setjmp()** deve sempre preceder o uso de **longjmp()**, de modo que esta é a ordem de apresentação destes componentes].

---

<sup>1</sup> É por esta razão que, frequentemente, se utiliza a expressão *arquivo de cabeçalho* em vez de simplesmente *cabeçalho*.

O número de componentes num cabeçalho da biblioteca padrão varia desde um único componente (e.g., `<assert.h>`) até um grande número de componentes (e.g., `<inttypes.h>`). Além disso, qualquer componente pode aparecer em mais de um cabeçalho (v. **Seção 1.7**).

Como regra geral, para não criar conflitos de identificadores, é recomendável evitar a redefinição de identificadores correntemente usados pela biblioteca padrão, bem como aqueles reservados para uso futuro (v. **Apêndice A**).

## 1.3 RECOMENDAÇÕES GERAIS DE USO

Qualquer função da biblioteca padrão de C pode ser usada sem que seja necessário incluir o cabeçalho onde ela é declarada. Neste caso, basta fazer alusão à função antes de chamá-la. No entanto, se uma dada função utiliza um tipo definido num cabeçalho, deve-se incluir o cabeçalho que contém a definição do tipo ou copiar esta definição de tipo antes da alusão da função. Em qualquer situação, é sempre melhor incluir os devidos cabeçalhos.

Para assegurar que uma macro não seja redefinida acidentalmente, sugere-se que todas as inclusões de cabeçalhos da biblioteca padrão sejam colocadas no início de cada arquivo-fonte e antes da inclusão de arquivos de cabeçalho criados para o programa.

Quando um argumento de uma função é um ponteiro, deve-se passar um endereço válido do respectivo tipo<sup>2</sup>. As únicas exceções ocorrem quando a descrição da função informa explicitamente que um ponteiro nulo pode ser usado. Exemplos:

```
strcpy(str, NULL); /* Inválido */
memcpy(str, 0, 0); /* Inseguro */
realloc(NULL, sizeof(int)); /* OK: o mesmo que malloc(sizeof(int)) */
```

Quando uma função não é reentrante, duas chamadas dela não podem ser executadas concorrentemente num mesmo programa. Uma função é **reentrante** quando<sup>3</sup>:

- Não usa nenhuma variável de duração fixa.
- Funciona apenas com os dados passados como argumentos.
- Não chama outra função que não seja reentrante.

<sup>2</sup> Por válido, entenda-se: um ponteiro para um espaço em memória suficientemente grande para conter o resultado.

<sup>3</sup> Esta definição poderia ser mais precisa, mas é suficiente para os objetivos do presente texto.

O padrão ISO não garante que nenhuma função da biblioteca padrão seja reentrante<sup>4</sup>. Ou seja, é responsabilidade de cada implementação da biblioteca padrão especificar quais funções são reentrantes.

Apesar de todas as funções da biblioteca padrão serem prototipadas, o que permite ao compilador checar erros de passagens de parâmetros, o programador é o verdadeiro responsável pela passagem de valores corretos para macros e funções. A seguir são apresentados exemplos de erros comuns em chamadas de funções.

***Erro: Passagem de ponteiro não iniciado para uma função.***

**Exemplo:**

```
char *str;
...
strcpy(str, "Desastre a vista");
```

Neste exemplo, o ponteiro `str` não está apontando para nenhuma posição válida em memória. Portanto, a função **`strcpy()`** irá escrever o resultado da cópia de *string* numa posição aleatória e provavelmente o programa será abortado devido à violação de memória.

***Erro: Passagem de ponteiro válido, mas que aponta para espaço insuficiente em memória para conter o resultado.***

**Exemplo:**

```
char str[5];
...
strcpy(str, "Outro desastre a vista");
```

Neste exemplo, o ponteiro `str` aponta para uma posição válida em memória, que é o array de cinco elementos do tipo **`char`** alocado. No entanto, a chamada de **`strcpy()`** requer que haja espaço para conter pelo menos 23 caracteres. Novamente, o programa será provavelmente abortado.

***Erro: Passagem de ponteiro válido, apontando para espaço suficiente em memória para conter o resultado, mas o espaço é reservado apenas para leitura.***

---

<sup>4</sup> A justificativa para esta falta de garantia é que muitas funções podem usar variáveis de duração fixa.

**Exemplo:**

```
char str = "Um string contendo muito espaco";
...
strcpy(str, "Mais um provavel desastre");
```

O erro apresentado no último exemplo é decorrente da violação de uma regra simples, mas desconhecida por muitos programadores: *strings constantes devem realmente ser considerados constantes!* Isto é, algumas implementações armazenam *strings* constantes em posições de memória reservadas apenas para leitura. Nestas implementações, o programa contendo o trecho de código anterior será definitivamente abortado.

***Erro: Passagem de um valor que está fora do domínio da função.*****Exemplo:**

```
double x = -2.5;
...
sqrt(x);
```

A função **sqrt()** calcula a raiz quadrada de seu argumento e seu domínio consiste nos valores do tipo **double** maiores do que ou iguais a zero. Este tipo de erro ocorre principalmente com funções declaradas em `<math.h>` e não causa aborto do programa, mas, se o problema não for detectado e corrigido, certamente resultará em erros lógicos no programa.

***Erro: Passagem de um argumento para alguma função da família printf ou scanf que não é compatível com o respectivo especificador de formato.***

```
int x;
...
printf("Conteudo do string: %s", x);
```

No último exemplo, o especificador `%s` deveria corresponder a um *string*, mas, em vez disso, encontra-se uma variável do tipo **int**. Apesar da aparência ingênua, este tipo de erro pode causar o aborto do programa ou, na melhor hipótese, impressão de *lixo* no meio de saída padrão.

## 1.4 MACROS E FUNÇÕES

Muitas funções da biblioteca padrão de C podem ser implementadas como macros em seus respectivos cabeçalhos. Macros com argumentos requerem mais cuidado quando chamadas do que funções porque, diferentemente do que ocorre com funções, o compilador não é capaz de checar se os tipos dos parâmetros passados para macros correspondem aos tipos esperados.

O padrão ISO C99 requer que cada argumento de uma macro da biblioteca padrão seja avaliado apenas uma vez no corpo da macro. Isto visa garantir que macros possam ser chamadas passando-lhes expressões como parâmetros sem a ocorrência de efeitos indesejados (v. **Capítulo 5 do Volume I**)<sup>5</sup>. Para ficar do lado mais seguro, é recomendável não invocar macros da biblioteca padrão com argumentos que sofrem a ação de operadores com efeito colateral ou, então, usar funções equivalentes. Por exemplo, em vez de usar o seguinte trecho de programa:

```
char c;
FILE *stream;
...
putc(++c, stream); /* Inseguro: putc() pode ser macro */
```

é recomendável usar:

```
char c;
FILE *stream;
...
++c;
putc(c, stream); /* OK */
```

ou

```
char c;
FILE *stream;
...
fputc(++c, stream); /* OK: fputc() é sempre função */
```

---

<sup>5</sup> Mas, o próprio padrão é inconsistente ao afirmar que as macros **getc()** e **putc()** podem avaliar seus argumentos mais de uma vez (v. **Seção 10.7**).

Há outras razões pelas quais pode não se utilizar uma macro em substituição a uma função equivalente. Por exemplo, a expansão de uma macro pode causar confusão enquanto se está depurando um programa e aumentar o tamanho do programa executável (v. **Capítulo 5 do Volume I**).

Qualquer definição em forma de macro de uma função pode ser suprimida envolvendo-se o respectivo identificador entre parênteses porque, neste caso, ele não é seguido imediatamente por um abre-parênteses e, assim, não é reconhecido como macro com argumentos. Pode-se ainda usar **#undef** para remover uma definição de macro e assegurar que a função correspondente seja chamada. Os exemplos a seguir ilustram o que foi exposto.

***Exemplo 1 – Talvez, uma macro seja invocada:***

```
#include <stdlib.h>
const char *str;
...
int i = atoi(str);
```

***Exemplo 2 – Com certeza, uma função será chamada:***

```
#include <stdlib.h>
#undef atoi
char *str;
...
int i = atoi(str);
```

***Exemplo 3 – Com certeza, uma função será chamada:***

```
#include <stdlib.h>
char *str;
...
int i = (atoi)(str);
```

***Exemplo 4 – Com certeza, uma função será chamada:***

```
extern int atoi(const char *);
char *str;
...
int i = atoi(str);
```



## 1.5 CABEÇALHOS PRÉ-COMPILADOS

Alguns compiladores oferecem um recurso denominado **cabeçalho pré-compilado** que visa acelerar o processo de compilação. A abordagem utilizada consiste em compilar (ou, mais precisamente, pré-processar) arquivos de cabeçalho que raramente são alterados transformando-os em código intermediário denominado cabeçalho pré-compilado. Então, compilações subsequentes de arquivos que utilizam estes cabeçalhos pré-compilados tornam-se mais rápidas. Tipicamente, um arquivo de cabeçalho pré-compilado usa a extensão `.pch`.

Suponha que se tenham dois arquivos denominados `arq1.h` e `arq1.c` e que este último arquivo inclua o primeiro. Então, quando um compilador usa pré-compilação de cabeçalhos, na primeira vez que o arquivo `arq1.c` é compilado, o arquivo de cabeçalho `arq1.h` é pré-compilado, obtendo-se assim um arquivo denominado, por exemplo, `arq1.pch`. Assim, da próxima vez que o arquivo `arq1.c` ou qualquer outro arquivo que inclua `arq1.h` for compilado, o compilador utilizará o arquivo `arq1.pch`, em vez de `arq1.h`.

O uso de cabeçalhos pré-compilados é indicado para grandes projetos de programação nos quais muitos cabeçalhos são incluídos. Neste caso, o tempo que o compilador leva para processar estes arquivos repetidas vezes torna-se muito grande comparado ao tempo de compilação do projeto inteiro. Mas, apesar de atraente, esta ideia nem sempre é adequada, pois cabeçalhos pré-compilados podem se tornar relativamente grandes e retardar a compilação, em vez de acelerá-la.

O uso de cabeçalhos pré-compilados com um dado compilador segue algumas regras e possui restrições. Se seu projeto de programação justifica o uso desta facilidade, consulte o manual do seu compilador.

## 1.6 VISÃO GERAL DOS CABEÇALHOS PADRONIZADOS

Esta seção apresenta uma visão geral dos cabeçalhos padronizados de C e dos componentes que fazem parte de cada cabeçalho. Estes cabeçalhos e componentes serão apresentados aqui em ordem alfabética e acompanhados de breves descrições. Assim, as descrições apresentadas nesta seção devem ser úteis provavelmente apenas nas seguintes situações:

- Quando se deseja ter uma visão geral de cada cabeçalho.

- Quando as informações apresentadas aqui são usadas como referência para as respectivas seções que aprofundam o estudo de determinados componentes ou cabeçalhos.

### 1.6.1 <assert.h>

O cabeçalho <assert.h> contém apenas um componente, que é a macro **assert()** usada em teste e depuração de programas. Este cabeçalho é apresentado no **Capítulo 11**.

### 1.6.2 <complex.h> (C99)

O propósito do cabeçalho <complex.h> é prover suporte para operações envolvendo números complexos (ou, mais precisamente, com valores de tipos de ponto flutuante complexos).

As macros definidas em <complex.h> são apresentadas na **Tabela 1-2**.

MACRO	EXPANSÃO
<b>complex</b>	A palavra reservada <b>_Complex</b> .
<b>_Complex_I</b>	Uma expressão do tipo <b>const float _Complex</b> , cuja parte real é zero e cuja parte imaginária é 1.
<b>I</b>	<b>_Imaginary_I</b> , se a macro <b>imaginary</b> for definida; <b>_Complex_I</b> , caso contrário.
<b>imaginary</b>	A palavra reservada <b>_Imaginary</b> , se esta palavra reservada for definida pela implementação.
<b>_Imaginary_I</b>	<b>(const float _Imaginary) 1</b> , se a macro <b>_Imaginary_I</b> for definida.

Tabela 1-2: Macros definidas em <complex.h>.

Todas as funções declaradas em `<complex.h>` têm nomes começando com *c* e o resto do nome da maioria delas é derivado de nomes de funções semelhantes declaradas em `<math.h>`. Por exemplo, a função **ccos()** declarada em `<complex.h>` é uma função que calcula o cosseno de um número complexo e tem seu nome derivado de **cos()**, que é a função semelhante declarada em `<math.h>`. Também, assim como as funções declaradas em `<math.h>`, cada operação provida pelas funções declaradas em `<complex.h>` possui três versões; uma para cada tipo complexo primitivo. Em termos de nomenclatura, as funções para o tipo **float \_Complex** têm acrescido *f* com relação aos nomes respectivos para o tipo **double \_Complex** e aquelas para o tipo **long double \_Complex** têm *l* acrescido ao final. As funções declaradas em `<complex.h>` são apresentadas na **Tabela 1-3**.

FUNÇÃO PARA O TIPO...			
double _Complex	float _Complex	long double _Complex	CALCULA
<b>cabs()</b>	<b>cabsf()</b>	<b>cabsl()</b>	Módulo
<b>cacos()</b>	<b>cacosf()</b>	<b>cacosl()</b>	Arco cosseno
<b>cacosh()</b>	<b>cacoshf()</b>	<b>cacoshl()</b>	Arco cosseno hiperbólico
<b>carg()</b>	<b>cargf()</b>	<b>cargl()</b>	Ângulo de fase
<b>casin()</b>	<b>casinf()</b>	<b>casinl()</b>	Arco seno
<b>casinh()</b>	<b>casinhf()</b>	<b>casinhl()</b>	Arco seno hiperbólico
<b>catan()</b>	<b>catanf()</b>	<b>catanl()</b>	Arco tangente
<b>catanh()</b>	<b>catanhf()</b>	<b>catanhl()</b>	Arco tangente hiperbólica
<b>ccos()</b>	<b>ccosf()</b>	<b>ccosl()</b>	Cosseno
<b>ccosh()</b>	<b>ccoshf()</b>	<b>ccoshl()</b>	Cosseno hiperbólico
<b>cexp()</b>	<b>cexpf()</b>	<b>cexpl()</b>	Exponencial
<b>cimag()</b>	<b>cimagf()</b>	<b>cimagl()</b>	Parte imaginária
<b>clog()</b>	<b>clogf()</b>	<b>clogl()</b>	Logaritmo
<b>conj()</b>	<b>conjf()</b>	<b>conjl()</b>	Conjugado
<b>cpow()</b>	<b>cpowf()</b>	<b>cpowl()</b>	Potenciação
<b>cproj()</b>	<b>cprojf()</b>	<b>cprojl()</b>	Projeção na esfera de Riemann
<b>creal()</b>	<b>crealf()</b>	<b>creall()</b>	Parte real
<b>csin()</b>	<b>csinf()</b>	<b>csinl()</b>	Seno
<b>csinh()</b>	<b>csinhf()</b>	<b>csinhl()</b>	Seno hiperbólico
<b>csqrt()</b>	<b>csqrtf()</b>	<b>csqrtl()</b>	Raiz quadrada

FUNÇÃO PARA O TIPO...			
double _Complex	float _Complex	long double _Complex	CALCULA
ctan()	ctanf()	ctanl()	Tangente
ctanh()	ctanhf()	ctanhl()	Tangente hiperbólica

Tabela 1-3: Funções declaradas em <complex.h>.

O cabeçalho <complex.h> e seus componentes são apresentados detalhadamente no **Capítulo 4**.

### 1.6.3 <ctype.h>

No cabeçalho <ctype.h> são declaradas funções de classificação de caracteres e transformação de letras maiúsculas em minúsculas e vice-versa. Estas funções, que são usualmente implementadas como macros, são brevemente descritas na **Tabela 1-4**.

FUNÇÃO	DESCRIÇÃO BREVE
isalnum()	Verifica se um caractere é alfanumérico.
isalpha()	Verifica se um caractere é uma letra.
isblank() (C99)	Verifica se um caractere é um espaço em branco.
isctrl()	Verifica se um caractere é um caractere de controle.
isdigit()	Verifica se um caractere é um dígito.
isgraph()	Verifica se um caractere pode ser impresso.
islower()	Verifica se um caractere é uma letra minúscula.
isprint()	Verifica se um caractere pode ser impresso (incluindo espaços).
ispunct()	Verifica se um caractere é um símbolo de pontuação.
isspace()	Verifica se um caractere é um espaço em branco.
isupper()	Verifica se um caractere é uma letra maiúscula.
isxdigit()	Verifica se um caractere é dígito hexadecimal.
tolower()	Retorna a letra minúscula correspondente a um dado caractere.
toupper()	Retorna a letra maiúscula correspondente a um dado caractere.

Tabela 1-4: Funções declaradas em <ctype.h>.

As funções declaradas no cabeçalho `<ctype.h>` serão exploradas no **Capítulo 6**.

### 1.6.4 `<errno.h>`

O cabeçalho `<errno.h>` contém uma alusão à variável global **errno** e definições de constantes (macros) associadas a condições de erros. Essa variável e essas macros são utilizadas por várias funções da biblioteca padrão para sinalizar o recebimento de parâmetros inadequados ou a produção de resultados incorretos. A **Tabela 1-5** apresenta as macros definidas em `<errno.h>` acompanhadas de breves descrições.

MACRO	INTERPRETAÇÃO
<b>EDOM</b>	Argumento inválido passado para uma função matemática.
<b>EILSEQ</b>	Sequência inválida de bytes representando um caractere multibyte.
<b>ERANGE</b>	Valor resultante de uma operação aritmética excessivamente grande.

Tabela 1-5: Macros definidas em `<errno.h>`.

As funções da biblioteca padrão que mais fazem uso das macros **EDOM** e **ERANGE** são aquelas declaradas no cabeçalho `<math.h>`, enquanto que macro **EILSEQ** é utilizada por algumas funções que processam caracteres extensos e multibytes.

As macros **EDOM** e **ERANGE** são descritas em detalhes no **Capítulo 3** e os demais componentes do cabeçalho `<errno.h>` são discutidos no **Capítulo 11**.

### 1.6.5 `<fenv.h>` (C99)

O cabeçalho `<fenv.h>` contém tipos, macros e funções que permitem testar e controlar exceções e modos de arredondamento de números de ponto flutuante. Este cabeçalho define os dois tipos apresentados na **Tabela 1-6**.

TIPO	DESCRIÇÃO BREVE
<b>fenv_t</b>	Tipo de uma variável capaz de conter toda a configuração de ambiente de ponto flutuante.
<b>fexcept_t</b>	Tipo de uma variável capaz de representar todos os sinalizadores de exceções de ponto flutuante.

Tabela 1-6: Tipos definidos em `<fenv.h>`.

A única macro definida em `<fenv.h>` requerida pelo padrão C99 é **FE\_DFL\_ENV**, que descreve a configuração de processamento de números de ponto flutuante quando o programa inicia. Além desta macro, o padrão C99 sugere a implementação de outras macros que serão discutidas no **Capítulo 3**.

As funções declaradas no cabeçalho `<fenv.h>` são apresentadas na **Tabela 1-7**.

FUNÇÃO	DESCRIÇÃO BREVE
<b>feclearexcept()</b>	Desliga exceções de ponto flutuante.
<b>fegetenv()</b>	Armazena configurações de ponto flutuante.
<b>fegetexceptflag()</b>	Armazena uma representação de exceções de ponto flutuante.
<b>fegetround()</b>	Retorna o modo corrente de arredondamento.
<b>feholdexcept()</b>	Armazena a configuração do ambiente de ponto flutuante.
<b>feraiseexcept()</b>	Liga sinalizadores de exceção de ponto flutuante.
<b>fesetenv()</b>	Restaura configurações de controle de ponto flutuante.
<b>fesetexceptflag()</b>	Liga exceções de ponto flutuante.
<b>fesetround()</b>	Estabelece um novo modo de arredondamento.
<b>fetestexcept()</b>	Liga um sinalizador de exceção de ponto flutuante.
<b>feupdateenv()</b>	Liga sinalizadores de exceção de ponto flutuante.

Tabela 1-7: Funções declaradas em `<fenv.h>`.

O cabeçalho `<fenv.h>` é completamente explorado no **Capítulo 3**.

### 1.6.6 `<float.h>`

O cabeçalho `<float.h>` define macros que representam propriedades de números de ponto flutuante. A **Tabela 1-8** apresenta resumidamente estas macros.

MACRO	SIGNIFICADO
<b>FLT_DIG</b> <b>DBL_DIG</b> <b>LDBL_DIG</b>	Precisão em casas decimais para os tipos <b>float</b> , <b>double</b> e <b>long double</b> .

MACRO	SIGNIFICADO
<b>FLT_EPSILON</b> <b>DBL_EPSILON</b> <b>LDBL_EPSILON</b>	O menor X do tipo <b>float</b> , <b>double</b> ou <b>long double</b> tal que: $1.0 + X \neq 1.0$
<b>FLT_MANT_DIG</b> <b>DBL_MANT_DIG</b> <b>LDBL_MANT_DIG</b>	O número de dígitos na mantissa de um valor do tipo <b>float</b> , <b>double</b> ou <b>long double</b> , usando-se a base <b>FLT_RADIX</b> .
<b>FLT_MAX</b> <b>DBL_MAX</b> <b>LDBL_MAX</b>	O maior valor finito e representável do tipo <b>float</b> , <b>double</b> ou <b>long double</b> .
<b>FLT_MAX_10_EXP</b> <b>DBL_MAX_10_EXP</b> <b>LDBL_MAX_10_EXP</b>	O maior inteiro X, tal que $10^X$ é um valor finito e representável do tipo <b>float</b> , <b>double</b> ou <b>long double</b> .
<b>FLT_MAX_EXP</b> <b>DBL_MAX_EXP</b> <b>LDBL_MAX_EXP</b>	O maior inteiro X, tal que $\text{FLT\_RADIX}^{(X - 1)}$ é um valor finito e representável do tipo <b>float</b> , <b>double</b> ou <b>long double</b> .
<b>FLT_MIN</b> <b>DBL_MIN</b> <b>LDBL_MIN</b>	O menor valor normalizado, finito e representável do tipo <b>float</b> , <b>double</b> ou <b>long double</b> .
<b>FLT_MIN_10_EXP</b> <b>DBL_MIN_10_EXP</b> <b>LDBL_MIN_10_EXP</b>	O menor inteiro X, tal que $10^X$ é um valor normalizado, finito e representável do tipo <b>float</b> , <b>double</b> ou <b>long double</b> .
<b>FLT_MIN_EXP</b> <b>DBL_MIN_EXP</b> <b>LDBL_MIN_EXP</b>	O menor inteiro X, tal que $\text{FLT\_RADIX}^{(X - 1)}$ é um valor normalizado, finito e representável do tipo <b>float</b> , <b>double</b> ou <b>long double</b> .
<b>DECIMAL_DIG</b> (C99)	O número mínimo de casas decimais necessárias para representar todos os dígitos significativos de um valor do tipo <b>long double</b> .
<b>FLT_EVAL_METHOD</b> (C99)	Um valor que descreve o modo de avaliação para operações de ponto flutuante.
<b>FLT_RADIX</b>	A base de todas as representações de números de ponto flutuante.

MACRO	SIGNIFICADO
<b>FLT_ROUNDS</b>	Um valor que descreve o modo de arredondamento para operações de ponto flutuante.

Tabela 1-8: Macros definidas em &lt;float.h&gt;.

O **Capítulo 3** discute em detalhes as macros definidas no cabeçalho <float.h> e apresenta exemplos de uso delas.

### 1.6.7 <inttypes.h> (C99)

O cabeçalho <inttypes.h> define o tipo **imaxdiv\_t**, que é o tipo do valor retornado pela função **imaxdiv()**, e pode definir muitas macros utilizadas como especificadores de formato de números inteiros em *strings* de formatação das famílias printf e scanf (v. **Capítulo 10**). As definições dessas macros variam entre implementações e são discutidas em detalhes no **Capítulo 2**. Neste cabeçalho, também são declaradas as funções apresentadas na **Tabela 1-9**.

FUNÇÃO	DESCRIÇÃO BREVE
<b>imaxabs()</b>	Retorna o valor absoluto de um inteiro do tipo <b>intmax_t</b> .
<b>imaxdiv()</b>	Retorna, numa estrutura do tipo <b>imaxdiv_t</b> , o quociente e o resto da divisão do seu primeiro parâmetro pelo segundo.
<b>strtoimax()</b>	Converte um <i>string</i> num número inteiro do tipo <b>intmax_t</b> .
<b>strtoumax()</b>	Converte um <i>string</i> num número inteiro do tipo <b>uintmax_t</b> .
<b>wcstoimax()</b>	Converte um <i>string</i> extenso num número inteiro do tipo <b>intmax_t</b> .
<b>wcstoumax()</b>	Converte um <i>string</i> extenso num número inteiro do tipo <b>uintmax_t</b> .

Tabela 1-9: Funções declaradas em &lt;inttypes.h&gt;.

O cabeçalho <inttypes.h> é estudado em detalhes no **Capítulo 2**.



## 1.6.8 <iso646.h>

O cabeçalho <iso646.h> define macros, apresentadas na **Tabela 1-10**, usadas como identificadores alternativos para alguns operadores de C. O objetivo original destas macros é compensar a ausência de caracteres necessários para composição de operadores em alguns conjuntos de caracteres especificados pelo padrão ISO 646 (v. **Seção 11.3**).

MACRO	EXPANSÃO
<b>and</b>	<b>&amp;&amp;</b>
<b>and_eq</b>	<b>&amp;=</b>
<b>bitand</b>	<b>&amp;</b>
<b>bitor</b>	<b> </b>
<b>compl</b>	<b>~</b>
<b>not</b>	<b>!</b>
<b>not_eq</b>	<b>!=</b>
<b>or</b>	<b>  </b>
<b>or_eq</b>	<b> =</b>
<b>xor</b>	<b>^</b>
<b>xor_eq</b>	<b>^=</b>

Tabela 1-10: Macros definidas em <iso646.h>.

O **Capítulo 11** apresenta maiores detalhes sobre o cabeçalho <iso646.h>.

## 1.6.9 <limits.h>

O cabeçalho <limits.h> define macros, apresentadas na **Tabela 1-11**, relacionadas com diversas propriedades de tipos inteiros. A maioria destas macros é expandida em limites inferiores e superiores dos tipos inteiros primitivos de C.

MACRO	SIGNIFICADO
<b>CHAR_BIT</b>	Número de bits utilizados para representar um valor do tipo <b>char</b> .
<b>CHAR_MAX</b>	Maior valor do tipo <b>char</b> .
<b>CHAR_MIN</b>	Menor valor do tipo <b>char</b> .
<b>INT_MAX</b>	Maior valor do tipo <b>int</b> .

MACRO	SIGNIFICADO
<b>INT_MIN</b>	Menor valor do tipo <b>int</b> .
<b>LLONG_MAX</b> (C99)	Maior valor do tipo <b>long long</b> .
<b>LLONG_MIN</b> (C99)	Menor valor do tipo <b>long long</b> .
<b>LONG_MAX</b>	Maior valor do tipo <b>long</b> .
<b>LONG_MIN</b>	Menor valor do tipo <b>long</b> .
<b>MB_LEN_MAX</b>	Maior número de bytes que podem constituir um caractere multibyte.
<b>SCHAR_MAX</b>	Maior valor do tipo <b>signed char</b> .
<b>SCHAR_MIN</b>	Menor valor do tipo <b>signed char</b> .
<b>SHRT_MAX</b>	Maior valor do tipo <b>short</b> .
<b>SHRT_MIN</b>	Menor valor do tipo <b>short</b> .
<b>UCHAR_MAX</b>	Maior valor do tipo <b>unsigned char</b> .
<b>UINT_MAX</b>	Maior valor do tipo <b>unsigned int</b> .
<b>ULLONG_MAX</b> (C99)	Maior valor do tipo <b>unsigned long long</b> .
<b>ULONG_MAX</b>	Maior valor do tipo <b>unsigned long</b> .
<b>USHRT_MAX</b>	Maior valor do tipo <b>unsigned short</b> .

Tabela 1-11: Macros definidas em &lt;limits.h&gt;.

O **Capítulo 2** apresenta exemplo de uso das macros definidas em <limits.h>.

## 1.6.10 <locale.h>

Localização de programas é o propósito do cabeçalho <locale.h>. Este cabeçalho define um rótulo de estrutura, denominado **lconv**, contendo campos que especificam como valores numéricos são formatados numa dada localidade.

Informações sobre localidade são divididas em categorias representadas por macros definidas no cabeçalho <locale.h> e descritas brevemente na **Tabela 1-12**.

CATEGORIA	REPRESENTA...
<b>LC_ALL</b>	Todas as demais categorias de localidade.
<b>LC_COLLATE</b>	Colação de <i>strings</i> (i.e., como dois <i>strings</i> são comparados) numa dada localidade.
<b>LC_CTYPE</b>	Como caracteres são classificados numa localidade.

CATEGORIA	REPRESENTA...
<b>LC_MONETARY</b>	Formatos monetários de uma localidade.
<b>LC_NUMERIC</b>	Formatos numéricos não monetários de uma localidade.
<b>LC_TIME</b>	Como datas e horas são representadas numa localidade.

Tabela 1-12: Macros definidas em &lt;locale.h&gt;.

No cabeçalho <locale.h>, são declaradas as funções resumidas na **Tabela 1-13**.

FUNÇÃO	DESCRIÇÃO BREVE
<b>localeconv()</b>	Provê informações sobre a localidade corrente.
<b>setlocale()</b>	Seleciona ou consulta informações sobre localidades.

Tabela 1-13: Funções declaradas em &lt;locale.h&gt;.

O cabeçalho <locale.h> é minuciosamente discutido no **Capítulo 5**.

## 1.6.11 <math.h>

O cabeçalho <math.h> provê suporte para operações de ponto flutuante real. Neste cabeçalho são definidos os tipos **double\_t** e **float\_t**, que podem corresponder a **float**, **double** ou **long double**, dependendo do valor da macro **FLT\_EVAL\_METHOD**, definida em <float.h>.

As macros definidas no cabeçalho <math.h> são apresentadas resumidamente na **Tabela 1-14**.

MACRO	DESCRIÇÃO BREVE
<b>FP_FAST_FMA</b> (C99) <b>FP_FAST_FMAF</b> (C99) <b>FP_FAST_FMAL</b> (C99)	Definida se uma chamada da função <b>fma()</b> é executada tão rapidamente quanto é avaliada uma expressão equivalente do tipo <b>double</b> ( <b>FP_FAST_FMA</b> ), <b>float</b> ( <b>FP_FAST_FMAF</b> ) ou <b>long double</b> ( <b>FP_FAST_FMAL</b> ).
<b>FP_ILOGB0</b> (C99)	Resulta no valor retornado por uma chamada da função <b>ilogb(x)</b> , quando $x \neq 0.0$ ou $-0.0$ .

MACRO	DESCRIÇÃO BREVE
<b>FP_ILOGBNAN</b> (C99)	Resulta no valor retornado por uma chamada da função <code>ilogb(x)</code> , quando $x$ é NaN.
<b>FP_INFINITE</b> (C99)	Expandida no valor resultante de uma invocação da macro <code>fpclassify(x)</code> , quando $x$ é $+\infty$ ou $-\infty$ .
<b>FP_NAN</b> (C99)	Expandida no valor resultante de uma invocação da macro <code>fpclassify(x)</code> , quando $x$ é NaN.
<b>FP_NORMAL</b> (C99)	Expandida no valor resultante de uma invocação da macro <code>fpclassify(x)</code> , quando $x$ é finito e normalizado.
<b>FP_SUBNORMAL</b> (C99)	Expandida no valor resultante de uma invocação da macro <code>fpclassify(x)</code> , quando $x$ é finito e não normalizado.
<b>FP_ZERO</b> (C99)	Expandida no valor resultante de uma invocação da macro <code>fpclassify(x)</code> , quando $x$ é $+0.0$ ou $-0.0$ .
<b>HUGE_VAL</b> <b>HUGE_VALF</b> (C99) <b>HUGE_VALL</b> (C99)	Representa um valor grande demais para ser representado como <b>double</b> ( <b>HUGE_VAL</b> ), <b>float</b> ( <b>HUGE_VALF</b> ) ou <b>long double</b> ( <b>HUGE_VALL</b> ).
<b>INFINITY</b>	Representação de $+\infty$ do tipo <b>float</b> .
<b>MATH_ERRNO</b> (C99)	Usada em conjunto com <b>math_errhandling</b> para verificar se as funções declaradas em <code>&lt;math.h&gt;</code> indicam erros usando a variável global <b>errno</b> .
<b>MATH_ERREXCEPT</b> (C99)	Usada em conjunto com <b>math_errhandling</b> para verificar se as funções declaradas em <code>&lt;math.h&gt;</code> indicam erros usando sinalizadores de exceção.
<b>math_errhandling</b> (C99)	Especifica como funções declaradas em <code>&lt;math.h&gt;</code> comunicam um erro de domínio ou intervalo.
<b>NAN</b> (C99)	Resulta na representação do tipo <b>float</b> de NaN.

Tabela 1-14: Macros sem argumentos definidas em `<math.h>`.

O cabeçalho `<math.h>` também define macros com argumentos usadas para classificar números de ponto flutuante reais que foram introduzidas pelo padrão C99 e são apresentadas na **Tabela 1-15**.

MACRO	DESCRIÇÃO BREVE
<b>fpclassify()</b>	Classifica números de ponto flutuante.
<b>isfinite()</b>	Verifica se um valor de ponto flutuante é finito.
<b>isgreater()</b>	Verifica se um valor é maior do que outro.
<b>isgreaterequal()</b>	Verifica se um valor é maior do que ou igual a outro.
<b>isinf()</b>	Verifica se um valor de ponto flutuante é $+\infty$ ou $-\infty$ .
<b>isless()</b>	Verifica se um valor é menor do que outro.
<b>islessequal()</b>	Verifica se um valor é menor do que ou igual ao outro.
<b>islessgreater()</b>	Verifica se um valor é menor ou maior do que outro.
<b>isnan()</b>	Verifica se um valor é NaN.
<b>isnormal()</b>	Verifica se um valor é finito e normalizado.
<b>isunordered()</b>	Verifica se há relação de ordem entre dois valores de ponto flutuante.
<b>signbit()</b>	Verifica se o bit de sinal de um valor de ponto flutuante está ligado.

Tabela 1-15: Macros com argumentos definidas em `<math.h>`.

A maioria das funções declaradas em `<math.h>` recebe um ou dois parâmetros de um tipo primitivo de ponto flutuante e os utiliza para calcular e retornar um valor inteiro ou de ponto flutuante. As funções declaradas em `<math.h>` são resumidas na **Tabela 1-16**.

FUNÇÃO	CALCULA
<b>acos(), acosf(), acosl()</b>	Arco cosseno.
<b>acosh(), acoshf(), acoshl() (C99)</b>	Arco cosseno hiperbólico.
<b>asin(), asinf(), asinl()</b>	Arco seno.
<b>asinh(), asinhf(), asinhl() (C99)</b>	Arco seno hiperbólico.
<b>atan(), atanf(), atanl()</b>	Arco tangente.
<b>atan2(), atan2f(), atan2l()</b>	Arco tangente de um quociente.
<b>atanh(), atanhf(), atanhhl() (C99)</b>	Arco tangente hiperbólica.
<b>cbrt(), cbrtf(), cbrtl() (C99)</b>	Raiz cúbica.

FUNÇÃO	CALCULA
<b>ceil()</b> , <b>ceilf()</b> , <b>ceil()</b>	Arredondamento para o menor inteiro.
<b>copysign()</b> , <b>copysignf()</b> , <b>copysignl()</b> (C99)	Troca o sinal de um valor pelo sinal de outro.
<b>cos()</b> , <b>cosf()</b> , <b>cosl()</b>	Cosseno.
<b>cosh()</b> , <b>coshf()</b> , <b>coshl()</b>	Cosseno hiperbólico.
<b>erf()</b> , <b>erff()</b> , <b>erfl()</b> (C99)	Função de erro.
<b>erfc()</b> , <b>erfcf()</b> , <b>erfcl()</b> (C99)	Função de erro complementar.
<b>exp()</b> , <b>expf()</b> , <b>expl()</b>	Exponencial neperiana.
<b>exp2()</b> , <b>exp2f()</b> , <b>exp2l()</b> (C99)	Exponencial na base dois.
<b>expm1()</b> , <b>expm1f()</b> , <b>expm1l()</b> (C99)	$e^x - 1$ , onde $x$ é o argumento.
<b>fabs()</b> , <b>fabsf()</b> , <b>fabsl()</b>	Valor absoluto.
<b>fdim()</b> , <b>fdimf()</b> , <b>fdiml()</b> (C99)	Compara a diferença entre dois argumentos com zero e retorna o maior valor.
<b>floor()</b> , <b>floorf()</b> , <b>floorl()</b>	Arredondamento para o próximo maior inteiro.
<b>fma()</b> , <b>fmaf()</b> , <b>fmal()</b> (C99)	FMA.
<b>fmax()</b> , <b>fmaxf()</b> , <b>fmaxl()</b> (C99)	Maior de dois valores.
<b>fmin()</b> , <b>fminf()</b> , <b>fminl()</b> (C99)	Menor de dois valores.
<b>fmod()</b> , <b>fmodf()</b> , <b>fmodl()</b>	Resto da divisão.
<b>frexp()</b> , <b>frexpf()</b> , <b>frexpl()</b>	Fração normalizada.
<b>hypot()</b> , <b>hypotf()</b> , <b>hypotl()</b> (C99)	Hipotenusa.
<b>ilogb()</b> , <b>ilogbf()</b> , <b>ilogbl()</b> (C99)	Expoente na base 2 em formato inteiro.
<b>ldexp()</b> , <b>ldexpf()</b> , <b>ldexpl()</b>	$x * 2^y$ , onde $x$ e $y$ são os parâmetros.
<b>lgamma()</b> , <b>lgammaf()</b> , <b>lgammal()</b> (C99)	Logaritmo natural do valor absoluto da função gama.
<b>llrint()</b> , <b>llrintf()</b> , <b>llrintl()</b> (C99)	Arredondamento para o inteiro do tipo <b>long long</b> mais próximo.
<b>llround()</b> , <b>llroundf()</b> , <b>llroundl()</b> (C99)	Arredondamento para o inteiro do tipo <b>long long</b> mais próximo.
<b>log()</b> , <b>logf()</b> , <b>logl()</b>	Logaritmo natural.
<b>log10()</b> , <b>log10f()</b> , <b>log10l()</b>	Logaritmo na base 10.

FUNÇÃO	CALCULA
<b>log1p()</b> , <b>log1pf()</b> , <b>log1pl()</b> (C99)	Logaritmo natural mais um.
<b>log2()</b> , <b>log2f()</b> , <b>log2l()</b> (C99)	Logaritmo na base 2.
<b>logb()</b> , <b>logbf()</b> , <b>logbl()</b> (C99)	Expoente na base 2.
<b>lrint()</b> , <b>lrintf()</b> , <b>lrintl()</b> (C99)	Arredondamento para o inteiro do tipo <b>long</b> mais próximo.
<b>lround()</b> , <b>lroundf()</b> , <b>lroundl()</b> (C99)	Arredondamento para o inteiro do tipo <b>long</b> mais próximo.
<b>modf()</b> , <b>modff()</b> , <b>modfl()</b>	Partes inteira e fracionária.
<b>nan()</b> , <b>nanf()</b> , <b>nanl()</b> (C99)	Converte um <i>string</i> em NaN.
<b>nearbyint()</b> , <b>nearbyintf()</b> , <b>nearbyintl()</b> (C99)	Arredondamento para o inteiro mais próximo usando o arredondamento corrente.
<b>nextafter()</b> , <b>nextafterf()</b> , <b>nextafterl()</b> (C99)	Valor representável mais próximo de outro.
<b>nexttoward()</b> , <b>nexttowardf()</b> , <b>nexttowardl()</b> (C99)	Valor representável mais próximo de outro.
<b>pow()</b> , <b>powf()</b> , <b>powl()</b>	Potenciação.
<b>remainder()</b> , <b>remainderf()</b> , <b>remainderl()</b> (C99)	Resto da divisão.
<b>remquo()</b> , <b>remquof()</b> , <b>remquol()</b> (C99)	Resto e quociente de divisão.
<b>rint()</b> , <b>rintf()</b> , <b>rintl()</b> (C99)	Arredondamento para o inteiro mais próximo.
<b>round()</b> , <b>roundf()</b> , <b>roundl()</b> (C99)	Arredondamento para o inteiro mais próximo.
<b>scalbln()</b> , <b>scalblnf()</b> , <b>scalblnl()</b> (C99)	$x * FLT\_RADIX^y$ , onde $x$ é o primeiro argumento e $y$ é o segundo.
<b>scalbn()</b> , <b>scalbnf()</b> , <b>scalbnl()</b> (C99)	$x * FLT\_RADIX^y$ , onde $x$ é o primeiro argumento e $y$ é o segundo.
<b>sin()</b> , <b>sinf()</b> , <b>sinl()</b>	Seno.
<b>sinh()</b> , <b>sinhf()</b> , <b>sinhl()</b>	Seno hiperbólico.
<b>sqrt()</b> , <b>sqrtf()</b> , <b>sqrtl()</b>	Raiz quadrada.
<b>tan()</b> , <b>tanf()</b> , <b>tanl()</b>	Tangente.
<b>tanh()</b> , <b>tanhf()</b> , <b>tanhl()</b>	Tangente hiperbólica.
<b>tgamma()</b> , <b>tgammaf()</b> , <b>tgammal()</b> (C99)	Função gama.

FUNÇÃO	CALCULA
<b>trunc()</b> , <b>truncf()</b> , <b>trunci()</b> (C99)	Arredondamento para o inteiro mais próximo.

Tabela 1-16: Funções declaradas em &lt;math.h&gt;.

Na **Tabela 1-16**, funções que terminam com *f* são dirigidas para o tipo **float**, aquelas que terminam com *l* são dirigidas para o tipo **long double** e as demais funções aplicam-se ao tipo **double**.

Os componentes do cabeçalho <math.h> são pormenorizados no **Capítulo 3**.

### 1.6.12 <setjmp.h>

O cabeçalho <setjmp.h> provê suporte para desvios entre funções. Neste cabeçalho, são declaradas as funções apresentadas na **Tabela 1-17**.

FUNÇÃO	DESCRIÇÃO BREVE
<b>setjmp()</b>	Prepara um desvio generalizado.
<b>longjmp()</b>	Realiza um desvio generalizado.

Tabela 1-17: Funções declaradas em &lt;setjmp.h&gt;.

O cabeçalho <setjmp.h> também define o tipo **jmp\_buf** usado pela função **setjmp()** para armazenar informações necessárias para efetivação de um desvio generalizado e pela função **longjmp()** para restaurar as mesmas informações logo antes de realizar o respectivo desvio.

O **Capítulo 11** explica como efetuar desvios generalizados usando os componentes do cabeçalho <setjmp.h>.

### 1.6.13 <signal.h>

O cabeçalho <signal.h> dá suporte para tratamento de sinais em C. Um sinal pode indicar uma ocorrência excepcional dentro de um programa (e.g., divisão por zero) ou algum evento ocorrido fora dele (e.g., digitação de [CTRL-C]).

No cabeçalho <signal.h>, são definidas as macros apresentas na **Tabela 1-18**.



SINAL	DESCRIÇÃO BREVE
<b>SIGFPE</b>	Sinal que representa uma tentativa de execução de uma operação aritmética ilegal.
<b>SIGILL</b>	Sinal que representa uma tentativa de execução de uma instrução ilegal.
<b>SIGSEGV</b>	Sinal que representa uma tentativa de acesso ilegal à memória.
<b>SIGABRT</b>	Sinal que informa que um programa será abortado.
<b>SIGTERM</b>	Sinal solicitando o encerramento de um programa.
<b>SIGINT</b>	Sinal emitido quando um usuário digita [CTRL-C].
<b>SIG_DFL</b>	Ponteiro para função representando a ação padrão de tratamento de um sinal.
<b>SIG_IGN</b>	Ponteiro para função que representa ignorar um sinal.

Tabela 1-18: Macros definidas em &lt;signal.h&gt;.

As duas funções apresentadas na **Tabela 1-19** são declaradas no cabeçalho <signal.h>.

FUNÇÃO	DESCRIÇÃO BREVE
<b>raise()</b>	Envia um sinal para o próprio programa.
<b>signal()</b>	Instala uma função de tratamento de sinal.

Tabela 1-19: Funções declaradas em &lt;signal.h&gt;.

O cabeçalho <signal.h> também define o tipo **sig\_atomic\_t**. O padrão de C garante que uma variável deste tipo nunca seja lida ou escrita parcialmente.

O cabeçalho <signal.h> é explorado em profundidade no **Capítulo 11**.

## 1.6.14 <stdarg.h>

O cabeçalho <stdarg.h> define um tipo, **va\_list**, e quatro macros utilizadas no processamento de listas de argumentos variáveis. O tipo **va\_list** é usado para definir variáveis associadas a listas de argumentos variáveis e as macros, apresentadas na **Tabela 1-20**, são utilizadas no processamento destes argumentos.

MACRO	DESCRIÇÃO BREVE
<b>va_start()</b>	Inicia o processamento de uma lista de argumentos variáveis.

<b>va_arg()</b>	Retorna no próximo argumento de uma lista de argumentos variáveis.
<b>va_end()</b>	Encerra o processamento de uma lista de argumentos variáveis.
<b>va_copy()</b> (C99)	Copia o conteúdo de uma variável do tipo <b>va_list</b> para outra do mesmo tipo.

Tabela 1-20: Macros definidas em <stdarg.h>.

O estudo das macros apresentadas na **Tabela 1-20** é aprofundado com auxílio de diversos exemplos no **Capítulo 9**.

### 1.6.15 <stdbool.h> (C99)

O cabeçalho <stdbool.h> provê macros que representam identificadores alternativos para o tipo **\_Bool**. O objetivo destas macros, enumeradas na **Tabela 1-21**, é melhorar a legibilidade de programas escritos em C e facilitar o transporte de programas escritos nesta linguagem para algumas outras linguagens, como C++ ou Java.

MACRO	EXPANSÃO
<b>bool</b>	<b>_Bool</b>
<b>false</b>	0
<b>true</b>	1
<b>__bool_true_false_are_defined</b>	1

Tabela 1-21: Macros definidas em <stdbool.h>.

O **Capítulo 11** apresenta mais detalhes sobre o cabeçalho <stdbool.h>.

### 1.6.16 <stdint.h>

O cabeçalho <stdint.h> define três tipos, apresentados na **Tabela 1-22**, e duas macros, apresentadas na **Tabela 1-23**.

TIPO	DESCRIÇÃO BREVE
<b>ptrdiff_t</b>	Tipo inteiro com sinal que representa o tipo do resultado da subtração de dois ponteiros.

TIPO	DESCRIÇÃO BREVE
<b>size_t</b>	Tipo inteiro sem sinal definido em vários cabeçalhos (v. <b>Seção 1.7.1</b> ).
<b>wchar_t</b> <sup>6</sup>	Tipo inteiro usado para representar caracteres extensos.

Tabela 1-22: Tipos definidos em <stddef.h>.

MACRO	EXPANSÃO
<b>NULL</b>	Ponteiro nulo (macro definida em vários cabeçalhos – v. <b>Seção 1.7.2</b> ).
<b>offsetof()</b>	O deslocamento, em bytes, de um membro de uma estrutura a partir do início dela.

Tabela 1-23: Macros definidas em <stddef.h>.

O **Capítulo 12** apresenta em detalhes o cabeçalho <stddef.h>.

### 1.6.17 <stdint.h> (C99)

O padrão C99 introduziu o cabeçalho <stdint.h> com o objetivo de prover os tipos inteiros portáveis apresentados na **Tabela 1-24**. Além destes tipos, o padrão C99 também sugere outros tipos que são discutidos no **Capítulo 2**.

TIPO	DESCRIÇÃO BREVE
<b>int8_t</b>	Inteiro com sinal representado em exatamente 8 bits.
<b>int16_t</b>	Inteiro com sinal representado em exatamente 16 bits.
<b>int32_t</b>	Inteiro com sinal representado em exatamente 32 bits.
<b>int64_t</b>	Inteiro com sinal representado em exatamente 64 bits.
<b>uint8_t</b>	Inteiro sem sinal representado em exatamente 8 bits.
<b>uint16_t</b>	Inteiro sem sinal representado em exatamente 16 bits.
<b>uint32_t</b>	Inteiro sem sinal representado em exatamente 32 bits.
<b>uint64_t</b>	Inteiro sem sinal representado em exatamente 64 bits.
<b>intmax_t</b>	Inteiro com sinal de maior largura.
<b>uintmax_t</b>	Inteiro sem sinal de maior largura.

Tabela 1-24: Tipos inteiros definidos em <stdint.h>.

---

6 Este tipo também é definido em <stdlib.h> e <wchar.h>.

De acordo com o padrão C99, o cabeçalho `<stdint.h>` deve definir pelo menos as macros apresentadas na **Tabela 1-25**. Além destas macros, o padrão C99 também sugere outras que são discutidas no **Capítulo 2**.

MACRO	EXPANSÃO
<b>INT8_MAX</b>	Maior valor do tipo <b>int8_t</b> .
<b>INT16_MAX</b>	Maior valor do tipo <b>int16_t</b> .
<b>INT32_MAX</b>	Maior valor do tipo <b>int32_t</b> .
<b>INT64_MAX</b>	Maior valor do tipo <b>int64_t</b> .
<b>INT8_MIN</b>	Menor valor do tipo <b>int8_t</b> .
<b>INT16_MIN</b>	Menor valor do tipo <b>int16_t</b> .
<b>INT32_MIN</b>	Menor valor do tipo <b>int32_t</b> .
<b>INT64_MIN</b>	Menor valor do tipo <b>int64_t</b> .
<b>UINT8_MAX</b>	Maior valor do tipo <b>uint8_t</b> .
<b>UINT16_MAX</b>	Maior valor do tipo <b>uint16_t</b> .
<b>UINT32_MAX</b>	Maior valor do tipo <b>uint32_t</b> .
<b>UINT64_MAX</b>	Maior valor do tipo <b>uint64_t</b> .
<b>INTMAX_MAX</b>	Maior valor do tipo <b>intmax_t</b> .
<b>INTMAX_MIN</b>	Menor valor do tipo <b>intmax_t</b> .
<b>UINTMAX_MAX</b>	Maior valor do tipo <b>uintmax_t</b> .
<b>PTRDIFF_MAX</b>	Limite máximo do tipo <b>ptrdiff_t</b> .
<b>PTRDIFF_MIN</b>	Limite mínimo do tipo <b>ptrdiff_t</b> .
<b>SIG_ATOMIC_MAX</b>	Limite máximo do tipo <b>sig_atomic_t</b> .
<b>SIG_ATOMIC_MIN</b>	Limite mínimo do tipo <b>sig_atomic_t</b> .
<b>SIZE_MAX</b>	Limite máximo do tipo <b>size_t</b> .
<b>WCHAR_MAX</b> <sup>7</sup>	Limite máximo do tipo <b>wchar_t</b> .
<b>WCHAR_MIN</b> <sup>8</sup>	Limite mínimo do tipo <b>wchar_t</b> .
<b>WINT_MAX</b>	Limite máximo do tipo <b>wint_t</b> .
<b>WINT_MIN</b>	Limite mínimo do tipo <b>wint_t</b> .

Tabela 1-25: Macros definidas em `<stdint.h>`.

<sup>7</sup> Esta macro também é definida em `<wchar.h>`.

<sup>8</sup> Esta macro também é definida em `<wchar.h>`.

## 1.6.18 <stdio.h>

O cabeçalho <stdio.h> contém declarações de todas as funções básicas de entrada e saída da biblioteca padrão, assim como tipos e macros que dão suporte para estas operações. Os tipos definidos no cabeçalho <stdio.h> aparecem na **Tabela 1-27**.

TIPO	DESCRIÇÃO BREVE
<b>fpos_t</b>	Tipo de um valor que representa uma posição num arquivo.
<b>FILE</b>	Tipo utilizado para implementar o conceito de <i>stream</i> .

Tabela 1-26: Tipos definidos em <stdio.h>.

As macros definidas no cabeçalho <stdio.h> são resumidas na **Tabela 1-27**.

MACRO	EXPANSÃO
<b>_IOFBF</b>	Valor inteiro indicando <i>buffering</i> de bloco.
<b>_IOLBF</b>	Valor inteiro indicando <i>buffering</i> de linha.
<b>_IONBF</b>	Valor inteiro indicando ausência de <i>buffering</i> .
<b>BUFSIZ</b>	Tamanho do buffer utilizado pela função <b>setbuf()</b> .
<b>EOF</b>	Valor que indica final de um arquivo ou uma condição de erro gerada numa operação de entrada ou saída.
<b>SEEK_CUR</b>	Valor inteiro que indica a posição corrente do apontador de posição do arquivo.
<b>SEEK_END</b>	Valor inteiro que indica a posição inicial de um arquivo.
<b>SEEK_SET</b>	Valor inteiro que indica a posição final de um arquivo.
<b>FILENAME_MAX</b>	Tamanho máximo de um <i>string</i> utilizado para representar um nome de arquivo.
<b>FOPEN_MAX</b>	Número máximo de arquivos que podem estar simultaneamente abertos.
<b>L_tmpnam</b>	Número mínimo de caracteres requeridos para representar o nome de um arquivo criado pela função <b>tmpnam()</b> .
<b>TMP_MAX</b>	Número mínimo de nomes de arquivos distintos criados pela função <b>tmpnam()</b> .

Tabela 1-27: Macros definidas em <stdio.h>.

Três variáveis globais, representando os *streams* padronizados de entrada e saída, são aludidas em `<stdio.h>` e apresentadas na **Tabela 1-28**.

VARIÁVEL GLOBAL	DESCRIÇÃO BREVE
<b>stdin</b>	<i>Stream</i> associado à entrada padrão.
<b>stdout</b>	<i>Stream</i> associado à saída padrão.
<b>stderr</b>	<i>Stream</i> associado à saída padrão de mensagens de erro.

Tabela 1-28: Variáveis globais declaradas em `<stdio.h>`.

As funções apresentadas na **Tabela 1-29** são declaradas no cabeçalho `<stdio.h>`.

FUNÇÃO	DESCRIÇÃO BREVE
<b>clearerr()</b>	Remove sinalização de erro e de final de arquivo.
<b>fclose()</b>	Fecha um arquivo.
<b>feof()</b>	Testa se o final de um <i>stream</i> foi atingido.
<b>ferror()</b>	Verifica se existe alguma sinalização de erro num <i>stream</i> .
<b>fflush()</b>	Descarrega a área de buffer associada a um <i>stream</i> de saída.
<b>fgetc()</b>	Lê um caractere num <i>stream</i> .
<b>fgetpos()</b>	Informa onde se encontra o apontador de posição de um arquivo.
<b>fgets()</b>	Lê caracteres num <i>stream</i> .
<b>fopen()</b>	Abre um arquivo.
<b>fprintf()</b>	Escreve dados formatados num <i>stream</i> de texto.
<b>fputc()</b>	Escreve um caractere num <i>stream</i> .
<b>fputs()</b>	Escreve caracteres num <i>stream</i> .
<b>fread()</b>	Lê um bloco num <i>stream</i> .
<b>freopen()</b>	Associa um arquivo a um <i>stream</i> associado a outro arquivo já aberto.
<b>fscanf()</b>	Lê dados formatados num <i>stream</i> de texto.
<b>fseek()</b>	Move o apontador de posição de um arquivo para um local especificado.
<b>fsetpos()</b>	Move o apontador de posição de um arquivo.
<b>ftell()</b>	Informa onde se encontra o apontador de posição de um arquivo.

FUNÇÃO	DESCRIÇÃO BREVE
<b>fwrite()</b>	Escreve um bloco num <i>stream</i> .
<b>getc()</b>	Lê um caractere num <i>stream</i> .
<b>getchar()</b>	Lê um caractere no meio de entrada padrão.
<b>gets()</b>	Lê uma linha em <b>stdin</b> .
<b>perror()</b>	Escreve em <b>stderr</b> uma mensagem de erro correspondente ao valor armazenado na variável global <b>errno</b> .
<b>printf()</b>	Escreve dados formatados em <b>stdout</b> .
<b>putc()</b>	Escreve um caractere num <i>stream</i> .
<b>putchar()</b>	Escreve um caractere em <b>stdout</b> .
<b>puts()</b>	Escreve um <i>string</i> em <b>stdout</b> .
<b>remove()</b>	Apaga um arquivo.
<b>rename()</b>	Altera o nome de um arquivo.
<b>rewind()</b>	Faz o apontador de posição de um arquivo retornar ao seu início.
<b>scanf()</b>	Lê dados formatados em <b>stdin</b> .
<b>setbuf()</b>	Associa um buffer a um <i>stream</i> .
<b>setvbuf()</b>	Associa um buffer a um <i>stream</i> .
<b>snprintf()</b> (C99)	Escreve dados formatados num array de caracteres, limitando o número de caracteres escritos.
<b>sprintf()</b>	Escreve dados formatados num array de caracteres.
<b>sscanf()</b>	Lê dados formatados num <i>string</i> .
<b>tmpfile()</b>	Cria um arquivo temporário e abre-o para escrita.
<b>tmpnam()</b>	Cria um nome de arquivo temporário.
<b>ungetc()</b>	Insera um caractere num <i>stream</i> de entrada.
<b>vfprintf()</b>	Escreve dados formatados num <i>stream</i> de texto usando um argumento do tipo <b>va_list</b> .
<b>vfscanf()</b> (C99)	Lê dados formatados num <i>stream</i> de texto usando um argumento do tipo <b>va_list</b> .
<b>vprintf()</b>	Escreve dados formatados em <b>stdout</b> usando um argumento do tipo <b>va_list</b> .
<b>vscanf()</b>	Lê dados formatados em <b>stdin</b> usando um argumento do tipo <b>va_list</b> .
<b>vsnprintf()</b> (C99)	Escreve dados formatados num <i>string</i> usando um argumento do tipo <b>va_list</b> .

FUNÇÃO	DESCRIÇÃO BREVE
<b>vsprintf()</b>	Escreve dados formatados num <i>string</i> usando um argumento do tipo <b>va_list</b> .
<b>vsscanf()</b> (C99)	Lê dados formatados num <i>string</i> usando um argumento do tipo <b>va_list</b> .

Tabela 1-29: Funções declaradas em &lt;stdio.h&gt;.

O cabeçalho <stdio.h> é apresentado em profundidade no **Capítulo 10**.

### 1.6.19 <stdlib.h>

O cabeçalho <stdlib.h> é um dos mais antigos cabeçalhos da biblioteca padrão de C e contém uma miscelânea de tipos, macros e funções. Os tipos definidos em <stdlib.h> são apresentados na **Tabela 1-30**.

TIPO	DESCRIÇÃO BREVE
<b>size_t</b>	Tipo inteiro sem sinal definido em vários cabeçalhos (v. <b>Seção 1.7.1</b> ).
<b>div_t</b>	Tipo da estrutura retornada pela função <b>div()</b> .
<b>ldiv_t</b>	Tipo da estrutura retornada pela função <b>ldiv()</b> .
<b>lldiv_t</b> (C99)	Tipo da estrutura retornada pela função <b>lldiv()</b> .
<b>wchar_t</b> <sup>9</sup>	Tipo usado para representar caracteres extensos.

Tabela 1-30: Tipos definidos em &lt;stdlib.h&gt;.

A **Tabela 1-31** mostra um resumo das macros definidas no cabeçalho <stdlib.h>.

MACRO	EXPANSÃO
<b>EXIT_FAILURE</b>	Valor retornado pela função <b>exit()</b> ou <b>main()</b> quando o programa termina de modo malsucedido.
<b>EXIT_SUCCESS</b>	Valor retornado pela função <b>exit()</b> ou <b>main()</b> quando o programa termina de modo bem-sucedido.
<b>MB_CUR_MAX</b>	Número máximo de caracteres que constituem um caractere multibyte na localidade corrente.
<b>RAND_MAX</b>	Maior valor que pode ser retornado pela função <b>rand()</b> .

<sup>9</sup> Este tipo também é definido em <wchar.h> e <stddef.h>.



MACRO	EXPANSÃO
<b>NULL</b>	Ponteiro nulo; esta macro é definida em vários cabeçalhos (v. <b>Seção 1.7.2</b> ).

Tabela 1-31: Macros definidas em &lt;stdlib.h&gt;.

A **Tabela 1-32** apresenta as funções declaradas no cabeçalho <stdlib.h>.

FUNÇÃO	DESCRIÇÃO BREVE
<b>_Exit()</b> (C99)	Encerra a execução do programa.
<b>abort()</b>	Causa o encerramento abrupto de um programa.
<b>abs()</b>	Calcula valor absoluto.
<b>atexit()</b>	Registra uma função a ser executada no encerramento de um programa quando ele for finalizado normalmente.
<b>atof()</b>	Converte um <i>string</i> num valor do tipo <b>float</b> .
<b>atoi()</b>	Converte um <i>string</i> num valor do tipo <b>int</b> .
<b>atol()</b>	Converte um <i>string</i> num valor do tipo <b>long</b> .
<b>atoll()</b> (C99)	Converte um <i>string</i> num valor do tipo <b>long long</b> .
<b>bsearch()</b>	Executa busca binária num array ordenado.
<b>calloc()</b>	Aloca dinamicamente um array de blocos e zera seu conteúdo.
<b>div()</b>	Calcula quociente e resto de divisão.
<b>exit()</b>	Encerra a execução de um programa.
<b>free()</b>	Libera o espaço ocupado por um bloco alocado dinamicamente.
<b>getenv()</b>	Retorna um ponteiro para um <i>string</i> contendo o valor de uma variável de ambiente.
<b>labs()</b>	Calcula valor absoluto.
<b>ldiv()</b>	Calcula quociente e resto de divisão.
<b>llabs()</b> (C99)	Calcula valor absoluto.
<b>lldiv()</b> (C99)	Calcula quociente e resto de divisão.
<b>malloc()</b>	Aloca dinamicamente um bloco de tamanho especificado.
<b>mblen()</b>	Calcula o número de bytes de um caractere multibyte.
<b>mbtowc()</b>	Retorna o caractere extenso correspondente a um caractere multibyte.
<b>mbstowcs()</b>	Converte um <i>string</i> multibyte num <i>string</i> extenso.
<b>qsort()</b>	Ordena um array usando o método <i>quicksort</i> .
<b>rand()</b>	Retorna um número aleatório.

FUNÇÃO	DESCRIÇÃO BREVE
<b>realloc()</b>	Redimensiona um bloco alocado dinamicamente.
<b>srand()</b>	Inicia o gerador de números aleatórios.
<b>strtod()</b>	Converte um <i>string</i> num valor do tipo <b>double</b> .
<b>strtof()</b> (C99)	Converte um <i>string</i> num valor do tipo <b>float</b> .
<b>strtol()</b>	Converte um <i>string</i> num valor do tipo <b>long</b> .
<b>strtold()</b> (C99)	Converte um <i>string</i> num valor do tipo <b>long double</b> .
<b>strtoll()</b> (C99)	Converte um <i>string</i> num valor do tipo <b>long long</b> .
<b>strtoul()</b>	Converte um <i>string</i> num valor do tipo <b>unsigned long</b> .
<b>strtoull()</b> (C99)	Converte um <i>string</i> num valor do tipo <b>unsigned long long</b> .
<b>system()</b>	Passa um <i>string</i> representando um comando para o processador de comandos do sistema operacional.
<b>wctomb()</b>	Retorna o caractere multibyte correspondente a um caractere extenso.
<b>wcstombs()</b>	Converte um <i>string</i> extenso num <i>string</i> multibyte.

Tabela 1-32: Funções declaradas em &lt;stdlib.h&gt;.

As seguintes funções declaradas em <stdlib.h> e que executam operações aritméticas inteiras são apresentadas em detalhes no **Capítulo 2 (Seção 2.6)**:

- **abs()**
- **labs()**
- **llabs()**
- **div()**
- **ldiv()**
- **lldiv()**

As seguintes funções de conversão de *strings* em números declaradas em <stdlib.h> são apresentadas em detalhes no **Capítulo 6 (Seção 6.5)**:

- **atof()**
- **atoi()**
- **atol()**

- **atoll()**
- **strtod()**
- **strtof()**
- **strtoul()**
- **strtold()**
- **strtoll()**
- **strtoul()**
- **strtoull()**

As seguintes funções declaradas em `<stdlib.h>` e que executam conversões entre caracteres extensos e multibytes são apresentadas em detalhes no **Capítulo 8** (Seção 8.4.2):

- **mblen()**
- **mbtowc()**
- **mbstowcs()**
- **wctomb()**
- **wcstombs()**

As demais funções declaradas em `<stdlib.h>` são pormenorizadas no **Capítulo 12**.

## 1.6.20 `<string.h>`

O cabeçalho `<string.h>` é dedicado ao processamento de *strings* e arrays de bytes (blocos). Neste cabeçalho, são definidos o tipo **size\_t** e a macro **NULL** (v. **Seção 1.7**). As funções para tratamento de *strings* e blocos declaradas neste cabeçalho são brevemente descritas na **Tabela 1-33**.

FUNÇÃO	DESCRIÇÃO BREVE
<b>memchr()</b>	Procura a primeira ocorrência de um caractere num bloco, limitando o número de bytes examinados.

FUNÇÃO	DESCRIÇÃO BREVE
<b>memcmp()</b>	Compara um número limitado de bytes de dois blocos de memória.
<b>memcpy()</b>	Copia um número limitado de bytes de um bloco de memória para outro.
<b>memmove()</b>	Copia um número de bytes de um bloco de memória para outro, permitindo sobreposição dos blocos.
<b>memset()</b>	Preenche um número limitado de bytes de um bloco com um determinado valor.
<b>strcat()</b>	Concatena dois <i>strings</i> .
<b>strchr()</b>	Procura a primeira ocorrência de um caractere num <i>string</i> .
<b>strcpy()</b>	Copia o conteúdo de um <i>string</i> num array.
<b>strcspn()</b>	Examina um <i>string</i> até encontrar algum caractere presente em outro <i>string</i> .
<b>strerror()</b>	Retorna um <i>string</i> descrevendo o erro associado com um inteiro.
<b>strlen()</b>	Calcula o comprimento de um <i>string</i> .
<b>strncat()</b>	Concatena um número limitado de caracteres ao final de um <i>string</i> .
<b>strncpy()</b>	Copia um número limitado de caracteres de um <i>string</i> para um array de caracteres.
<b>strpbrk()</b>	Encontra num <i>string</i> a primeira ocorrência de qualquer caractere presente noutro <i>string</i> .
<b>strrchr()</b>	Procura a última ocorrência de um caractere em um <i>string</i> .
<b>strspn()</b>	Procura o segmento inicial de um <i>string</i> que contém todos os caracteres de outro <i>string</i> .
<b>strstr()</b>	Procura a primeira ocorrência de um <i>string</i> em outro <i>string</i> .
<b>strtok()</b>	Divide um <i>string</i> em partes.
<b>strcmp()</b>	Compara dois <i>strings</i> .
<b>strncmp()</b>	Compara um número limitado de caracteres de dois <i>strings</i> .
<b>strcoll()</b>	Compara dois <i>strings</i> usando a ordem de colação especificada pela categoria de localidade <b>LC_COLLATE</b> .
<b>strxfrm()</b>	Cria uma chave de ordenação de acordo com a ordem de colação definida na categoria <b>LC_COLLATE</b> .

Tabela 1-33: Funções declaradas em &lt;string.h&gt;.

O cabeçalho <string.h> é explorado no **Capítulo 6**.

## 1.6.21 <tgmath.h> (C99)

O cabeçalho `<tgmath.h>` declara apenas macros genéricas utilizadas em operações matemáticas com números de ponto flutuante reais e complexos. O objetivo dessas macros é decidir, com base nos tipos dos parâmetros com os quais são invocadas, qual função declarada em `<math.h>` ou `<complex.h>` deve ser chamada.

Cada função declarada em `<math.h>` possui uma macro genérica com o mesmo nome da função. Quando uma destas macros é invocada tendo um valor de um tipo complexo como um dos seus argumentos, ela chama a função correspondente declarada em `<complex.h>`, se esta função existir; se tal função não existir, o resultado será indefinido. Se uma destas macros for invocada sem nenhum argumento complexo, ela chamará a função correspondente declarada em `<math.h>`.

Cada função declarada em `<complex.h>` que não possui contrapartida em `<math.h>` possui uma macro genérica com o mesmo nome da função. Quando uma destas macros for invocada com um parâmetro com tipo inteiro ou de ponto flutuante real em lugar de um parâmetro complexo, ocorrerá a devida conversão para o tipo complexo, e a função correspondente declarada em `<complex.h>` será chamada normalmente.

Maiores detalhes sobre o cabeçalho `<tgmath.h>` são apresentados no **Capítulo 4**.

## 1.6.22 <time.h>

O cabeçalho `<time.h>` provê componentes que facilitam a manipulação de datas e horas. Neste cabeçalho, são definidos os tipos apresentados na **Tabela 1-34** e as macros apresentadas na **Tabela 1-35**.

TIPO	DESCRIÇÃO BREVE
<b>clock_t</b>	Representa intervalos de tempo de processamento da CPU.
<b>size_t</b>	Tipo inteiro sem sinal definido em vários cabeçalhos (v. <b>Seção 1.7.1</b> ).
<b>time_t</b>	Tipo usado para representar datas e horas.
<b>tm</b> <sup>10</sup>	Rótulo de estruturas cujos membros podem armazenar informações sobre um dado instante de tempo.

Tabela 1-34: Tipos definidos em `<time.h>`.

<sup>10</sup> Este rótulo de estrutura também é definido em `<wchar.h>`.

MACRO	EXPANSÃO
<b>NULL</b>	Ponteiro nulo (definido em vários cabeçalhos – v. <b>Seção 1.7.2</b> ).
<b>CLOCKS_PER_SEC</b>	Número de tiques retornados pela função <b>clock()</b> em um segundo.

Tabela 1-35: Macros definidas em &lt;time.h&gt;.

Na **Tabela 1-36**, são apresentadas as funções declaradas em <time.h>.

FUNÇÃO	DESCRIÇÃO BREVE
<b>asctime()</b>	Converte em <i>string</i> a data e a hora armazenadas numa estrutura do tipo <b>tm</b> .
<b>clock()</b>	Fornece o número de tiques decorridos desde que o programa começou a ser executado.
<b>ctime()</b>	Converte em <i>string</i> a data e a hora armazenadas num valor do tipo <b>time_t</b> .
<b>difftime()</b>	Calcula o intervalo de tempo, em segundos, decorrido entre dois instantes representados por valores do tipo <b>time_t</b> .
<b>gmtime()</b>	Converte data e hora, representadas num valor do tipo <b>time_t</b> , no padrão UTC.
<b>localtime()</b>	Converte um valor do tipo <b>time_t</b> , representando data e hora, numa estrutura do tipo <b>tm</b> .
<b>mktime()</b>	Calcula o intervalo de tempo decorrido desde o instante de referência até um instante especificado.
<b>strftime()</b>	Cria um <i>string</i> contendo data e hora de acordo com uma especificação de formato.
<b>time()</b>	Determina os segundos decorridos desde o instante de referência.

Tabela 1-36: Funções declaradas em &lt;time.h&gt;.

O **Capítulo 5** apresenta em detalhes os componentes do cabeçalho <time.h>.

### 1.6.23 <wchar.h>

Os componentes do cabeçalho <wchar.h> são usados em processamento de caracteres extensos e multibytes. A **Tabela 1-37** enumera os tipos definidos neste cabeçalho com suas descrições resumidas.

TIPO	DESCRIÇÃO BREVE
<b>mbstate_t</b>	Tipo de uma variável que pode representar um estado de mudança para caracteres multibytes com estado.
<b>size_t</b>	Tipo inteiro sem sinal definido em vários cabeçalhos (v. <b>Seção 1.7.1</b> ).
<b>tm</b> <sup>11</sup>	Rótulo de estruturas contendo membros que descrevem várias propriedades de data e hora.
<b>wchar_t</b> <sup>12</sup>	Tipo de um caractere extenso.
<b>wint_t</b> <sup>13</sup>	Tipo inteiro usado para representar caracteres extensos quando eles são passados como parâmetros ou retornados por funções.

Tabela 1-37: Tipos definidos em &lt;wchar.h&gt;.

A **Tabela 1-38** a seguir apresenta as macros definidas em <wchar.h> e suas respectivas expansões.

MACRO	EXPANSÃO
<b>NULL</b>	Ponteiro nulo (definido em vários cabeçalhos – v. <b>Seção 1.7.2</b> ).
<b>WCHAR_MAX</b> <b>WCHAR_MIN</b>	Valor máximo/mínimo do tipo <b>wchar_t</b> <sup>14</sup> .
<b>WEOF</b> <sup>15</sup>	Valor usado para marcar o final de um <i>stream</i> de caracteres extensos ou indicar uma condição de erro.

Tabela 1-38: Macros definidas em &lt;wchar.h&gt;.

O cabeçalho <wchar.h> declara funções para processamento de *strings* extensos semelhantes àquelas que executam tarefas similares com *strings* monobytes. Este cabeçalho também contém declarações de funções para conversão de caracteres extensos em multibytes e vice-versa. Uma sinopse das funções declaradas em <wchar.h> é apresentada na **Tabela 1-41**.

---

11 Este rótulo de estrutura também é definido no cabeçalho <time.h>.

12 Este tipo também é definido em <stdlib.h> e <stddef.h>.

13 Este tipo também é definido em <wctype.h>.

14 Estas macros também são definidas em <stdint.h>.

15 Esta macro também é definida em <wctype.h>.

FUNÇÃO	DESCRIÇÃO BREVE
<b>btowc()</b>	Converte um caractere monobyte em caractere extenso.
<b>fgetwc()</b>	Lê um caractere extenso num <i>stream</i> .
<b>fgetws()</b>	Lê um <i>string</i> extenso num <i>stream</i> até encontrar uma quebra de linha, limitando o número de caracteres lidos.
<b>fputwc()</b>	Escreve um caractere extenso num <i>stream</i> .
<b>fputws()</b>	Escreve um <i>string</i> extenso num <i>stream</i> .
<b>fwide()</b>	Testa ou altera a orientação de um <i>stream</i> .
<b>fwprintf()</b>	Escreve num <i>stream</i> dados formatados usando um <i>string</i> extenso.
<b>fwscanf()</b>	Lê num <i>stream</i> dados formatados usando um <i>string</i> extenso.
<b>getwc()</b>	Lê um caractere extenso num <i>stream</i> .
<b>getwchar()</b>	Lê um caractere extenso em <b>stdin</b> .
<b>mbrlen()</b>	Determina o número de bytes num <i>string</i> multibyte, permitindo reinício.
<b>mbrtowc()</b>	Converte um caractere multibyte num caractere extenso, permitindo reinício.
<b>mbsinit()</b>	Testa se um valor representa um estado inicial de mudança.
<b>mbsrtowcs()</b>	Converte um <i>string</i> de caracteres multibytes num <i>string</i> de caracteres extensos, limitando o número de caracteres extensos armazenados.
<b>putwc()</b>	Escreve um caractere extenso num <i>stream</i> especificado.
<b>putwchar()</b>	Escreve um caractere extenso em <b>stdout</b> .
<b>swprintf()</b>	Escreve dados formatados usando um <i>string</i> extenso de formatação num array de caracteres extensos.
<b>swscanf()</b>	Lê num <i>string</i> extenso dados formatados usando um <i>string</i> extenso.
<b>ungetwc()</b>	Insere um caractere extenso num <i>stream</i> de leitura.
<b>vfwprintf()</b>	Escreve dados formatados num <i>stream</i> usando um <i>string</i> extenso de formatação e um argumento do tipo <b>va_list</b> .
<b>vfwscanf()</b> (C99)	Lê num <i>stream</i> dados formatados usando um <i>string</i> extenso de formatação e um argumento do tipo <b>va_list</b> .



FUNÇÃO	DESCRIÇÃO BREVE
<b>vswprintf()</b>	Escreve dados formatados num array de caracteres extensos usando um argumento do tipo <b>va_list</b> .
<b>vswscanf()</b> (C99)	Lê dados formatados num <i>string</i> extenso usando um <i>string</i> extenso de formatação e um argumento do tipo <b>va_list</b> .
<b>wprintf()</b>	Escreve dados formatados em <b>stdout</b> usando um <i>string</i> extenso de formatação e um argumento do tipo <b>va_list</b> .
<b>wscanf()</b> (C99)	Lê dados formatados em <b>stdin</b> usando um <i>string</i> extenso e um argumento do tipo <b>va_list</b> .
<b>wrtomb()</b>	Converte um caractere extenso num caractere multibyte.
<b>wscat()</b>	Concatena dois <i>strings</i> extensos.
<b>wcschr()</b>	Procura a primeira ocorrência de um caractere extenso num <i>string</i> extenso.
<b>wscmp()</b>	Compara dois <i>strings</i> extensos.
<b>wscoll()</b>	Compara dois <i>strings</i> extensos usando informações sobre colação na localidade corrente.
<b>wscpy()</b>	Faz uma cópia de um <i>string</i> extenso.
<b>wscspn()</b>	Examina um <i>string</i> extenso até encontrar algum caractere extenso que faça parte de outro <i>string</i> extenso.
<b>wcsftime()</b>	Constrói um <i>string</i> extenso contendo data e hora especificadas utilizando um <i>string</i> de formatação extenso.
<b>wcslen()</b>	Calcula o comprimento de um <i>string</i> extenso.
<b>wcsncat()</b>	Concatena um número limitado de caracteres extensos como final de um <i>string</i> extenso.
<b>wcsncmp()</b>	Compara um número limitado de caracteres de dois <i>strings</i> extensos.
<b>wcsncpy()</b>	Copia um número limitado de caracteres de um <i>string</i> extenso.
<b>wcsprbrk()</b>	Procura num <i>string</i> extenso a primeira ocorrência de qualquer caractere extenso contido noutra <i>string</i> extenso.
<b>wcsrchr()</b>	Procura a última ocorrência de um caractere extenso num <i>string</i> extenso.
<b>wcsrtombs()</b>	Converte um <i>string</i> extenso num <i>string</i> multibyte.
<b>wcsspn()</b>	Procura o segmento inicial de um <i>string</i> extenso contendo todos os caracteres extensos de outro <i>string</i> extenso.

FUNÇÃO	DESCRIÇÃO BREVE
<b>wcsstr()</b>	Procura a primeira ocorrência de um <i>string</i> extenso em outro.
<b>wctod()</b>	Converte um <i>string</i> extenso num valor do tipo <b>double</b> .
<b>wcstof()</b> (C99)	Converte um <i>string</i> extenso num valor do tipo <b>float</b> .
<b>wctok()</b>	Divide um <i>string</i> extenso em partes.
<b>wcstol()</b>	Converte um <i>string</i> extenso num valor do tipo <b>long</b> .
<b>wcstold()</b>	Converte um <i>string</i> extenso num valor do tipo <b>long double</b> .
<b>wcstoll()</b> (C99)	Converte um <i>string</i> extenso num valor do tipo <b>long long</b> .
<b>wctoul()</b>	Converte um <i>string</i> extenso num valor do tipo <b>unsigned long</b> .
<b>wcstoull()</b> (C99)	Converte um <i>string</i> extenso num valor do tipo <b>unsigned long long</b> .
<b>wcsxfrm()</b>	Cria uma chave de ordenação para um <i>string</i> extenso usando a ordem de colação estabelecida na categoria de localidade <b>LC_COLLATE</b> .
<b>wctob()</b>	Converte um caractere extenso em monobyte.
<b>wmemchr()</b>	Procura a primeira ocorrência de um caractere extenso num <i>string</i> extenso limitando o número de caracteres examinados.
<b>wmemcmp()</b>	Compara um número limitado de caracteres extensos de dois arrays de caracteres extensos.
<b>wmemcpy()</b>	Copia um número limitado de caracteres extensos de um array de caracteres extensos para outro.
<b>wmemmove()</b>	Copia um número limitado de caracteres extensos de um array de caracteres extensos para outro, permitindo a sobreposição dos arrays.
<b>wmemset()</b>	Preenche os primeiros elementos de um array de caracteres extensos com um dado valor.
<b>wprintf()</b>	Escreve em <b>stdout</b> dados formatados usando um <i>string</i> extenso.
<b>wscanf()</b>	Lê em <b>stdin</b> dados formatados usando um <i>string</i> extenso.

Tabela 1-39: Funções declaradas em &lt;wchar.h&gt;.

As funções declaradas em `<wchar.h>` que executam operações de entrada e saída usando caracteres extensos são apresentadas em detalhes no **Capítulo 10**. As demais funções declaradas neste cabeçalho são descritas no **Capítulo 8**.

### 1.6.24 `<wctype.h>`

O cabeçalho `<wctype.h>` dedica-se à classificação e transformação de caracteres extensos. Neste cabeçalho, são definidos três tipos, apresentados na **Tabela 1-40**, e a macro **WEOF**, que é equivalente, para caracteres extensos, à macro **EOF** utilizada para caracteres monobytes (v. **Seção 1.6.18**).

TIPO	DESCRIÇÃO BREVE
<b>wctrans_t</b>	Representa valores associados a transformações de caracteres extensos específicos de localidade.
<b>wctype_t</b>	Representa valores associados a classificações de caracteres extensos específicos de localidade.
<b>wint_t</b> <sup>16</sup>	Tipo usado para representar caracteres extensos quando eles são passados como parâmetros ou retornados por funções.

Tabela 1-40: Tipos definidos em `<wctype.h>`.<sup>16</sup>

As funções apresentadas na **Tabela 1-41** são declaradas no cabeçalho `<wctype.h>`.

FUNÇÃO	DESCRIÇÃO BREVE
<b>iswalnum()</b>	Verifica se um caractere extenso é alfanumérico.
<b>iswalpha()</b>	Verifica se um caractere extenso é letra.
<b>iswblank() (C99)</b>	Verifica se um caractere extenso é espaço em branco que separa palavras numa linha.
<b>iswcntrl()</b>	Verifica se um caractere extenso é caractere de controle.
<b>iswdigit()</b>	Verifica se um caractere extenso é dígito (0–9).
<b>iswgraph()</b>	Verifica se um caractere extenso é imprimível (exceto espaço em branco).
<b>iswlower()</b>	Verifica se um caractere extenso é letra minúscula.
<b>iswprint()</b>	Verifica se um caractere extenso é imprimível (incluindo espaço em branco).
<b>iswpunct()</b>	Verifica se um caractere extenso é símbolo de pontuação.

<sup>16</sup> Este tipo também é definido em `<wchar.h>`.

FUNÇÃO	DESCRIÇÃO BREVE
<b>isspace()</b>	Verifica se um caractere extenso é espaço em branco (sem restrição).
<b>iswupper()</b>	Verifica se um caractere extenso é letra maiúscula.
<b>iswxdigit()</b>	Verifica se um caractere extenso é dígito hexadecimal.
<b>iswctype()</b>	Verifica se um caractere extenso pertence a uma dada categoria de caracteres extensos.
<b>wctype()</b>	Retorna uma regra de classificação de caracteres extensos que pode ser utilizada com a função <b>iswctype()</b> .
<b>towctrans()</b>	Retorna o caractere extenso recebido como parâmetro convertido de acordo com uma dada transformação.
<b>tolower()</b>	Retorna a letra minúscula correspondente a um dado caractere extenso.
<b>toupper()</b>	Retorna a letra maiúscula correspondente a um dado caractere extenso.
<b>wctrans()</b>	Retorna um valor que representa um mapeamento de um conjunto de caracteres extensos para outro.

Tabela 1-41: Funções declaradas em &lt;wctype.h&gt;.

Os componentes do cabeçalho <wctype.h> são discutidos minuciosamente no **Capítulo 8**.

## 1.7 COMPONENTES REPETIDOS EM CABEÇALHOS

Alguns tipos e macros aparecem repetidos em vários cabeçalhos. Os componentes definidos repetidamente na biblioteca padrão de C serão apresentados a seguir.

### 1.7.1 TIPO `size_t`

**size\_t** é um tipo inteiro sem sinal que representa o tipo do valor resultante da aplicação do operador **sizeof** (v. **Volume I**). Este tipo é definido nos seguintes cabeçalhos:

- <time.h> – v. **Seção 5.3.1**

- `<string.h>` – v. **Seção 6.3.1**
- `<wchar.h>` – v. **Seção 8.5.1**
- `<stdlib.h>` – v. **Seção 12.2.1**
- `<stddef.h>` – v. **Seção 12.3.1**

#### **Observações:**

- Este tipo é comumente utilizado para definir variáveis que representam tamanhos ou índices de arrays.
- O uso adequado deste tipo num programa promove sua portabilidade e legibilidade.

## **1.7.2 MACRO NULL**

A macro **NULL** representa um ponteiro nulo e é definida nos seguintes cabeçalhos:

- `<time.h>` – v. **Seção 5.3.2**
- `<string.h>` – v. **Seção 6.3.2**
- `<wchar.h>` – v. **Seção 8.5.2**
- `<stdlib.h>` – v. **Seção 12.2.2**
- `<stddef.h>` – v. **Seção 12.3.2**

#### **Observações:**

- O padrão ISO de C garante que este ponteiro representa um endereço reconhecidamente inválido em qualquer implementação.
- O **Volume I** descreve a importância dessa macro e apresenta vários exemplos práticos de seu uso.

### 1.7.3 OUTROS COMPONENTES REPETIDOS

Além de **size\_t** e **NULL**, outros componentes que aparecem repetidamente em cabeçalhos padronizados são enumerados na **Tabela 1-42**. Esta tabela indica os cabeçalhos nos quais cada componente é definido e a seção no presente texto onde ele é descrito.

COMPONENTE	CATEGORIA	CABEÇALHO	REFERÊNCIA
<b>tm</b>	Rótulo de estrutura	<b>&lt;time.h&gt;</b>	<b>Seção 5.3.1</b>
		<b>&lt;wchar.h&gt;</b>	<b>Seção 8.5.1</b>
<b>WCHAR_MAX</b>	Macro	<b>&lt;stdint.h&gt;</b>	<b>Seção 2.4.2</b>
		<b>&lt;wchar.h&gt;</b>	<b>Seção 8.5.2</b>
<b>WCHAR_MIN</b>	Macro	<b>&lt;stdint.h&gt;</b>	<b>Seção 2.4.2</b>
		<b>&lt;wchar.h&gt;</b>	<b>Seção 8.5.2</b>
<b>wchar_t</b>	Tipo	<b>&lt;wchar.h&gt;</b>	<b>Seção 8.5.1</b>
		<b>&lt;stdlib.h&gt;</b>	<b>Seção 12.2.1</b>
		<b>&lt;stddef.h&gt;</b>	<b>Seção 12.3.1</b>
<b>wctype_t</b>	Tipo	<b>&lt;wchar.h&gt;</b>	<b>Seção 8.5.1</b>
		<b>&lt;wctype.h&gt;</b>	<b>Seção 8.6.1</b>
<b>WEOF</b>	Macro	<b>&lt;wchar.h&gt;</b>	<b>Seção 8.5.2</b>
		<b>&lt;wctype.h&gt;</b>	<b>Seção 8.6.2</b>
<b>wint_t</b>	Tipo	<b>&lt;wchar.h&gt;</b>	<b>Seção 8.5.1</b>
		<b>&lt;wctype.h&gt;</b>	<b>Seção 8.6.1</b>

Tabela 1-42: Componentes repetidos da biblioteca padrão de C.

## 1.8 SISTEMAS COM HOSPEDEIRO E SISTEMAS LIVRES

O padrão ISO C99 define dois tipos de sistemas de execução de programas escritos em C: (1) **sistemas com hospedeiro** e (2) **sistemas livres**. Programas executados em sistemas com hospedeiro são, tipicamente, programas executados sob a supervisão e com suporte de um sistema operacional. Estes programas devem definir uma função **main()**, que é a primeira função chamada quando o programa é executado. Para programas desta natureza, o padrão de C requer que haja suporte completo para a biblioteca padrão.

Programas executados em sistemas livres não possuem hospedeiro ou sistema de arquivos e uma implementação de C não precisa atender a todas as recomendações impostas para sistemas com hospedeiro. Num programa deste tipo, o nome e o tipo da primeira função chamada quando o programa é executado são determinados pela implementação. O padrão ISO não requer que implementações de C para sistemas livres deem suporte completo para a biblioteca padrão de C. Neste caso, os programas não podem usar tipos complexos e o suporte de biblioteca pode resumir-se aos componentes dos cabeçalhos: `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>` e `<stdint.h>`.

## 1.9 EXERCÍCIOS DE REVISÃO

1. (a) Qual é a diferença entre *cabeçalho* e *arquivo de cabeçalho*? (b) Por que o padrão ISO C99 dá preferência ao uso do termo *cabeçalho* em vez de *arquivo de cabeçalho*?
2. Como são categorizados os componentes típicos de um cabeçalho da biblioteca padrão?
3. (a) Quando se usa uma função da biblioteca padrão num programa, é necessário incluir o cabeçalho no qual a função é declarada? (b) E quando se usam outros componentes?
4. O que é uma função reentrante?
5. Cite alguns erros comuns que podem ocorrer quando se usam funções e macros da biblioteca padrão de C.
6. Suponha que uma operação identificada por `Op` e declarada no cabeçalho `<cabecalho.h>` da biblioteca padrão possa ser implementada tanto em forma de função quanto em forma de macro. Quais dos seguintes trechos de programas garantem que a função será chamada? Explique.

(a)

```
Op ( ) ;
```

(b)

```
#undef Op
...
Op( );
```

(c)

```
(Op)( );
```

(d)

```
extern void Op(void);
...
Op( );
```

7. (a) O que é um cabeçalho pré-compilado? (b) Que vantagem é obtida com o uso de cabeçalhos pré-compilados?

8. Descreva sucintamente o propósito geral dos cabeçalhos da biblioteca padrão de C:

- (a) <assert.h>
- (b) <complex.h>
- (c) <ctype.h>
- (d) <errno.h>
- (e) <fenv.h>
- (f) <float.h>
- (g) <inttypes.h>
- (h) <iso646.h>
- (i) <limits.h>
- (j) <locale.h>
- (k) <math.h>
- (l) <setjmp.h>



- (m) <signal.h>
- (n) <stdarg.h>
- (o) <stdbool.h>
- (p) <stddef.h>
- (q) <stdint.h>
- (r) <stdio.h>
- (s) <stdlib.h>
- (t) <string.h>
- (u) <tgmath.h>
- (v) <time.h>
- (w) <wchar.h>
- (x) <wctype.h>

9. Descreva (a) sistema com hospedeiro e (b) sistema livre.

10. Qual é o suporte, em termos de biblioteca padrão, recomendado pelo padrão C99 para sistemas livres?

# *Capítulo 2*

---

*Números inteiros*

## 2.1 INTRODUÇÃO

Este capítulo inicia fazendo uma breve exposição sobre os tipos inteiros primitivos da linguagem C. Espera-se com esta curta introdução meramente situar o leitor de tal modo que ele acompanhe as descrições dos componentes dos cabeçalhos que serão apresentados mais adiante. Qualquer dúvida referente aos tipos inteiros primitivos de C pode ser dirimida estudando-se o **Capítulo 1 do Volume I**.

Neste capítulo serão descritos os seguintes cabeçalhos:

- `<limits.h>` – que define macros que descrevem propriedades dos tipos inteiros primitivos de C.
- `<stdint.h>` – que define tipos e macros associados a inteiros com larguras definidas.
- `<inttypes.h>` – que contém componentes que permitem controlar conversões de valores entre os tipos inteiros definidos em `<stdint.h>`.

Este capítulo apresenta ainda tipos e funções usados em operações aritméticas inteiras que fazem parte do cabeçalho `<stdlib.h>`.

## 2.2 TIPOS INTEIROS PRIMITIVOS

Os tipos inteiros primitivos da linguagem C aparecem em negrito na Figura 2-1. Como mostra esta figura, existem quatro tipos básicos de inteiros em C: **int**, **char**, **\_Bool** e **long long int** (que pode ser abreviado para **long long**). O padrão de C especifica apenas as larguras dos tipos **char** – que sempre ocupa um byte – e **long long** (C99) – que sempre ocupa 8 bytes. As larguras dos demais tipos são dependentes de implementação.

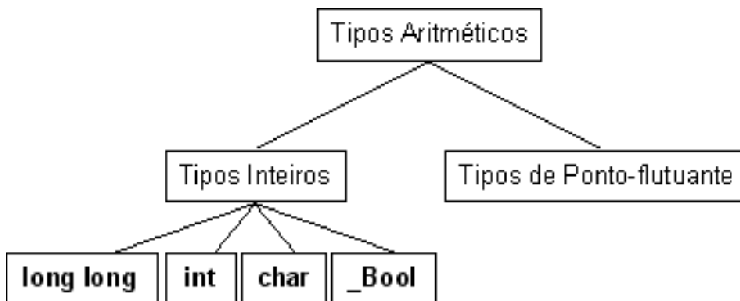


Figura 2-1: Tipos inteiros primitivos.

O tipo **int** pode ser qualificado com as seguintes palavras-chave:

- **short** ou **long**, que se referem à largura do tipo.
- **signed** ou **unsigned**, que determinam se o tipo é considerado com sinal ou sem sinal, respectivamente.

Os qualificadores **signed** e **unsigned** podem também ser utilizados com os tipos **char** e **long long**. A qualificação de um tipo é feita antepondo-se o qualificador ao nome do tipo (por exemplo, **signed char**, **unsigned long int**).

Por padrão, os tipos **int** e **long long** apresentam-se com sinal, enquanto que o tipo **char** pode ter sinal ou não, dependendo do compilador utilizado. As interpretações de **short** e **long** para o tipo **int** variam de acordo com a implementação, mas **short int** deve sempre ocupar menos espaço do que **long int**.

Constantes inteiras podem ser de três tipos, de acordo com a base numérica utilizada:

- **Decimais.** São escritas utilizando-se os dígitos de 0 a 9, sendo que o primeiro dígito não pode ser zero, a não ser que o próprio valor seja zero. Exemplos de constantes inteiras decimais: 0, 12345, -67.
- **Octais.** Constantes inteiras octais devem começar com zero e utilizar os dígitos de 0 a 7. Exemplos válidos: 00, 07744.
- **Hexadecimais.** Constantes inteiras hexadecimais devem começar com 0x ou 0X, e utilizar os dígitos de 0 a 9 e as letras de A a F (ou a a f). Exemplos válidos: 0x7FFA, 0X230, 0xf2de.

Pode-se explicitamente sugerir que uma constante inteira seja interpretada como **long** ou **long long** usando-se os sufixos **L** ou **LL**, respectivamente. Constantes inteiras na base decimal são sempre consideradas **signed**, mas pode-se sugerir que uma constante seja considerada **unsigned** usando-se o sufixo **U** (ou **u**). Este sufixo pode ser combinado com **L** ou **LL**, como, por exemplo: 10UL. Constantes inteiras na base octal ou hexadecimal podem ser consideradas **signed** ou **unsigned** dependendo do primeiro tipo que a constante couber na sequência: **int** → **unsigned int** → **long** → **unsigned long** → **long long** → **unsigned long long**. As regras completas de interpretação de constantes inteiras introduzidas pelo padrão C99 são apresentadas no **Capítulo 1 do Volume I**.

Conforme pode-se perceber, os tipos inteiros de C, com exceção de **long long**, que foi introduzido em C99, não são portáteis. Reconhecidamente, esta é uma falha da especificação original da linguagem que se perpetuou devido a razões de compatibilidade histórica. Portanto, o programador deve tomar cuidado quando utilizar estes tipos em situações práticas<sup>17</sup>.

Para resolver esses problemas de portabilidade, o padrão C99 introduziu o cabeçalho `<stdint.h>`, no qual são definidos vários tipos inteiros de tamanhos definidos. O uso desses tipos ao invés dos tipos inteiros primitivos de C é altamente recomendável para evitar problemas de portabilidade envolvendo tipos inteiros.

## 2.3 PROPRIEDADES DOS TIPOS INTEIROS PRIMITIVOS: <limits.h>

O cabeçalho `<limits.h>` define apenas macros que representam várias propriedades dos tipos inteiros primitivos de C. A **Tabela 2-1** apresenta as macros definidas no cabeçalho `<limits.h>` e seus respectivos significados.

MACRO	SIGNIFICADO
<b>CHAR_BIT</b>	O número de bits utilizados para representar um valor do tipo <b>char</b> .
<b>CHAR_MAX</b>	O maior valor permitido para o tipo <b>char</b> . Este valor será igual a <b>SCHAR_MAX</b> se o tipo <b>char</b> for <b>signed</b> ou <b>UCHAR_MAX</b> em caso contrário.
<b>CHAR_MIN</b>	O menor valor permitido para o tipo <b>char</b> . Este valor será igual a <b>SCHAR_MIN</b> se <b>char</b> for <b>signed</b> ou zero em caso contrário.
<b>INT_MAX</b>	O maior valor do tipo <b>int</b> .
<b>INT_MIN</b>	O menor valor do tipo <b>int</b> .
<b>LLONG_MAX</b> (C99)	O maior valor do tipo <b>long long</b> .
<b>LLONG_MIN</b> (C99)	O menor valor do tipo <b>long long</b> .
<b>LONG_MAX</b>	O maior valor do tipo <b>long</b> .

---

<sup>17</sup> Ao longo deste livro, tipos inteiros não portáteis são utilizados em diversos exemplos, pois, dada a natureza didática e a relativa simplicidade destes exemplos, existe pouca possibilidade de ocorrerem problemas de portabilidade.

MACRO	SIGNIFICADO
<b>LONG_MIN</b>	O menor valor do tipo <b>long</b> .
<b>MB_LEN_MAX</b>	O maior número de bytes que podem constituir um caractere multibyte, independentemente da localidade utilizada. Este valor é maior do que ou igual à macro <b>MB_CUR_MAX</b> definida em <code>&lt;stdlib.h&gt;</code> (v. <b>Capítulo 12</b> ).
<b>SCHAR_MAX</b>	O maior valor do tipo <b>signed char</b> .
<b>SCHAR_MIN</b>	O menor valor do tipo <b>signed char</b> .
<b>SHRT_MAX</b>	O maior valor do tipo <b>short</b> .
<b>SHRT_MIN</b>	O menor valor do tipo <b>short</b> .
<b>UCHAR_MAX</b>	O maior valor do tipo <b>unsigned char</b> .
<b>UINT_MAX</b>	O maior valor do tipo <b>unsigned int</b> .
<b>ULLONG_MAX</b> (C99)	O maior valor do tipo <b>unsigned long long</b> .
<b>ULONG_MAX</b>	O maior valor do tipo <b>unsigned long</b> .
<b>USHRT_MAX</b>	O maior valor do tipo <b>unsigned short</b> .

Tabela 2-1: Macros definidas em `<limits.h>` e respectivos significados.

Para conhecer os valores das macros definidas em `<limits.h>`, compile e execute o programa a seguir.

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

int main()
{
    printf("Numero de bits utilizados para representar "
           "um valor do tipo char: %d\n", CHAR_BIT);
    printf("Maior valor permitido para o tipo char: %d\n",
           CHAR_MAX);
    printf("Menor valor permitido para o tipo char: %d\n",
           CHAR_MIN);
    printf("Maior valor para o tipo int: %d\n",
           INT_MAX);
    printf("Menor valor para o tipo int: %d\n",
           INT_MIN);
    printf("Maior valor do tipo long long: %lli "
           "(C99)\n", LLONG_MAX); /* (C99) */
    printf("Menor valor do tipo long long: %lli "
```

```

        "(C99)\n", LLONG_MIN); /* (C99) */
printf("Maior valor do tipo long: %ld\n",
        LONG_MAX);
printf("Menor valor do tipo long: %ld\n",
        LONG_MIN);
printf("Maior numero de bytes que podem constituir "
        "um caractere multibyte: %d\n", MB_LEN_MAX);
printf("Maior valor do tipo signed char: %d\n",
        SCHAR_MAX);
printf("Menor valor do tipo signed char: %d\n",
        SCHAR_MIN);
printf("Maior valor do tipo short: %d\n",
        SHRT_MAX);
printf("Menor valor do tipo short: %d\n",
        SHRT_MIN);
printf("Maior valor do tipo unsigned char: %d\n",
        UCHAR_MAX);
printf("Maior valor do tipo unsigned int: %u\n",
        UINT_MAX);
printf("Maior valor do tipo unsigned long long: "
        "%llu (C99)\n", ULLONG_MAX); /* (C99) */
printf("Maior valor do tipo unsigned long: "
        "%lu\n", ULONG_MAX);
printf("Maior valor do tipo unsigned short: "
        "%hu\n", USHRT_MAX);

return 0;
}

```

Quando o programa anterior é compilado e executado no Linux, ele apresenta o seguinte resultado:

```

Numero de bits utilizados para representar um valor do tipo
char: 8
Maior valor permitido para o tipo char: 127
Menor valor permitido para o tipo char: -128
Maior valor do tipo int: 2147483647
Menor valor do tipo int: -2147483648
Maior valor do tipo long long: 9223372036854775807 (C99)
Menor valor do tipo long long: -9223372036854775808 (C99)
Maior valor do tipo long: 2147483647
Menor valor do tipo long: -2147483648
Maior numero de bytes que podem constituir um caractere multi-
byte: 16

```

```

Maior valor do tipo signed char: 127
Menor valor do tipo signed char: -128
Maior valor do tipo short: 32767
Menor valor do tipo short: -32768
Maior valor do tipo unsigned char: 255
Maior valor do tipo unsigned int: 4294967295
Maior valor do tipo unsigned long long: 18446744073709551615
(C99)
Maior valor do tipo unsigned long: 4294967295
Maior valor do tipo unsigned short: 65535

```

## 2.4 PORTABILIDADE DE INTEIROS I: <stdint.h> (C99)

O cabeçalho <stdint.h> define vários tipos inteiros com larguras (número de bytes) definidas, bem como várias macros relacionadas a estes tipos. Estes componentes, que são bastante úteis para melhorar a portabilidade de programas, variam de acordo com a implementação de C. Aqui serão descritos apenas os tipos e macros exigidos pelo padrão C99.

### 2.4.1 TIPOS

A **Tabela 2-2** apresenta os tipos inteiros com larguras exatas requeridos pelo padrão C99.

TIPO	INTERPRETAÇÃO
<b>int8_t</b>	Inteiro com sinal representado em exatamente 8 bits.
<b>int16_t</b>	Inteiro com sinal representado em exatamente 16 bits.
<b>int32_t</b>	Inteiro com sinal representado em exatamente 32 bits.
<b>int64_t</b>	Inteiro com sinal representado em exatamente 64 bits.
<b>uint8_t</b>	Inteiro sem sinal representado em exatamente 8 bits.
<b>uint16_t</b>	Inteiro sem sinal representado em exatamente 16 bits.
<b>uint32_t</b>	Inteiro sem sinal representado em exatamente 32 bits.
<b>uint64_t</b>	Inteiro sem sinal representado em exatamente 64 bits.

Tabela 2-2: Tipos inteiros com larguras exatas definidos em <stdint.h>.

A **Tabela 2-3** apresenta os tipos inteiros com larguras máximas requeridos pelo padrão C99.



TIPO	INTERPRETAÇÃO
<b>intmax_t</b>	Inteiro com sinal que possui a maior largura numa dada implementação.
<b>uintmax_t</b>	Inteiro sem sinal que possui a maior largura numa dada implementação.

Tabela 2-3: Tipos inteiros com larguras máximas definidos em &lt;stdint.h&gt;.

Além dos tipos apresentados anteriormente, o padrão C99 sugere (mas não requer) a presença de tipos que representem inteiros classificados nas seguintes categorias:

- **Inteiros que tenham pelo menos certa largura.** Estes inteiros seriam denominados `int_least8_t`, `int_least16_t`, `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, `uint_least64_t`, etc.
- **Inteiros rápidos que tenham pelo menos certa largura**<sup>18</sup>. Estes inteiros seriam denominados `int_fast8_t`, `int_fast16_t`, `uint_fast8_t`, `uint_fast16_t`, etc.
- **Inteiros capazes de representar ponteiros.** Estes inteiros seriam denominados `intptr_t` e `uintptr_t`.

Os tipos `intptr_t` e `uintptr_t` proveem um modo portátil de conversão entre ponteiros e inteiros. Por exemplo:

```
tEstrutura *pe = ...;
intptr_t ptr = (void *) pe; /* Conversão portátil */
...
pe = (tEstrutura *) ptr; /* Conversão portátil */
```

Exemplos de usos destes tipos são apresentados na **Seção 2.5.2**.

## 2.4.2 MACROS

As macros apresentadas na **Tabela 2-4** a seguir são requeridas pelo padrão C99 e representam valores máximos e mínimos dos tipos inteiros com larguras exatas apresentados na **Tabela 2-2**.

<sup>18</sup> **Inteiros rápidos** são inteiros que permitem a execução de operações sobre si próprios de modo mais rápido do que ocorre com outros inteiros que não fazem parte desta categoria.

MACRO	INTERPRETAÇÃO
<b>INT8_MAX</b>	Maior valor do tipo <b>int8_t</b>
<b>INT16_MAX</b>	Maior valor do tipo <b>int16_t</b>
<b>INT32_MAX</b>	Maior valor do tipo <b>int32_t</b>
<b>INT64_MAX</b>	Maior valor do tipo <b>int64_t</b>
<b>INT8_MIN</b>	Menor valor do tipo <b>int8_t</b>
<b>INT16_MIN</b>	Menor valor do tipo <b>int16_t</b>
<b>INT32_MIN</b>	Menor valor do tipo <b>int32_t</b>
<b>INT64_MIN</b>	Menor valor do tipo <b>int64_t</b>
<b>UINT8_MAX</b>	Maior valor do tipo <b>uint8_t</b>
<b>UINT16_MAX</b>	Maior valor do tipo <b>uint16_t</b>
<b>UINT32_MAX</b>	Maior valor do tipo <b>uint32_t</b>
<b>UINT64_MAX</b>	Maior valor do tipo <b>uint64_t</b>

Tabela 2-4: Limites máximos e mínimos dos tipos inteiros com larguras exatas.

Para conhecer as expansões das macros apresentadas na **Tabela 2-4** utilize o seguinte raciocínio:

- Macros identificadas como **INTN\_MAX** são expandidas como:  $2^{N-1} - 1$ .
- Macros identificadas como **INTN\_MIN** são expandidas como:  $-2^{N-1}$ .
- Macros identificadas como **UINTN\_MAX** são expandidas como:  $2^N - 1$ .

A **Tabela 2-5** apresenta macros que representam limites dos tipos inteiros com larguras máximas requeridos pelo padrão C99.

TIPO	EXPANSÃO
<b>INTMAX_MAX</b>	Valor máximo do tipo inteiro com sinal que possui maior largura numa dada implementação.
<b>INTMAX_MIN</b>	Valor mínimo do tipo inteiro com sinal que possui maior largura numa dada implementação.
<b>UINTMAX_MAX</b>	Valor máximo do tipo inteiro sem sinal que possui maior largura numa dada implementação.

Tabela 2-5: Valores máximos e mínimos dos tipos inteiros com maiores larguras.

A **Tabela 2-6** apresenta macros que representam limites de tipos inteiros definidos em outros cabeçalhos da biblioteca padrão de C.

MACRO	EXPANSÃO
<b>PTRDIFF_MAX</b>	Valor máximo do tipo <b>ptrdiff_t</b> definido em <code>&lt;stddef.h&gt;</code> (v. <b>Seção 12.3.1</b> ).
<b>PTRDIFF_MIN</b>	Valor mínimo do tipo <b>ptrdiff_t</b> definido em <code>&lt;stddef.h&gt;</code> .
<b>SIG_ATOMIC_MAX</b>	Valor máximo do tipo <b>sig_atomic_t</b> definido em <code>&lt;signal.h&gt;</code> (v. <b>Seção 11.6.1</b> ).
<b>SIG_ATOMIC_MIN</b>	Valor mínimo do tipo <b>sig_atomic_t</b> definido em <code>&lt;signal.h&gt;</code> .
<b>SIZE_MAX</b>	Valor máximo do tipo <b>size_t</b> definido em <code>&lt;stddef.h&gt;</code> e outros cabeçalhos (v. <b>Seção 1.7</b> ).
<b>WCHAR_MAX</b> <sup>19</sup>	Valor máximo do tipo <b>wchar_t</b> definido em <code>&lt;wchar.h&gt;</code> (v. <b>Seção 8.5.1</b> ).
<b>WCHAR_MIN</b> <sup>20</sup>	Valor mínimo do tipo <b>wchar_t</b> definido em <code>&lt;wchar.h&gt;</code> .
<b>WINT_MAX</b>	Valor máximo do tipo <b>wint_t</b> definido em <code>&lt;wchar.h&gt;</code> (v. <b>Seção 8.5.1</b> ).
<b>WINT_MIN</b>	Valor mínimo do tipo <b>wint_t</b> definido em <code>&lt;wchar.h&gt;</code> .

Tabela 2-6: Limites de tipos inteiros definidos em outros cabeçalhos.

Se uma dada implementação incluir os tipos opcionais descritos na **Seção 2.4.1**, ela também deve incluir as seguintes macros:

- **Macros que representam limites de tipos inteiros que tenham pelo menos certo tamanho.** Estas macros seriam denominadas `INT_LEAST8_MIN`, `INT_LEAST8_MAX`, `UINT_LEAST8_MAX`, `INT_LEAST16_MIN`, `INT_LEAST16_MAX`, `UINT_LEAST16_MAX`, etc.
- **Macros que representam limites de tipos inteiros rápidos que tenham pelo menos certo tamanho.** Estas macros seriam denominadas `INT_FAST8_MIN`, `INT_FAST8_MAX`, `UINT_FAST8_MAX`, `INT_FAST16_MIN`, `INT_FAST16_MAX`, `UINT_FAST16_MAX`, etc.

---

<sup>19</sup> Esta macro também é definida em `<wchar.h>`.

<sup>20</sup> Esta macro também é definida em `<wchar.h>`.

- **Macros que representam limites de tipos inteiros capazes de representar ponteiros.** Estas macros seriam denominadas `INTPTR_MAX`, `INTPTR_MIN` e `UINTPTR_MAX`.

Exemplos de uso de algumas dessas macros serão apresentados na **Seção 2.5.2**.

## 2.5 PORTABILIDADE DE INTEIROS II: `<inttypes.h>` (C99)

O cabeçalho `<inttypes.h>` (C99) define tipos, funções e macros que constituem uma extensão do cabeçalho `<stdint.h>` (v. **Seção 2.4**). O cabeçalho `<inttypes.h>` inclui o cabeçalho `<stdint.h>` (C99), de modo que, se o primeiro cabeçalho for incluído num programa, este último não precisará ser.

Há duas razões pelas quais o cabeçalho `<inttypes.h>` foi criado, em vez de seus componentes terem sido incluídos no cabeçalho `<stdint.h>`:

1. O padrão C99 requer que o cabeçalho `<stdint.h>` esteja presente em implementações para sistemas livres, mas este não é o caso do cabeçalho `<inttypes.h>` (v. **Seção 1.8**).
2. O cabeçalho `<inttypes.h>` inclui inúmeras macros que, por alguma razão (e.g., aumento no tempo de compilação), podem ser indesejadas.

### 2.5.1 TIPO `imaxdiv_t`

**Incluir:** `<inttypes.h>`

**Descrição:** O tipo `imaxdiv_t` é o tipo do valor retornado pela função `imaxdiv()`. Ele consiste em uma estrutura com dois campos: `quot`, que representa o quociente, e `rem`, que representa o resto de uma divisão calculada usando a função `imaxdiv()` com operandos do tipo `intmax_t`.

**Exemplo:** Veja o exemplo da função `imaxdiv()` (**Seção 2.5.3**).

## 2.5.2 MACROS

O cabeçalho `<inttypes.h>` define uma grande variedade de macros que podem ser utilizadas como especificadores de formato de leitura e impressão para os tipos de dados definidos no cabeçalho `<stdint.h>`. Em vez de enumerar todas estas macros, é mais instrutivo mostrar como os nomes destas macros são formados, o que será feito a seguir.

Inicialmente, essas macros são divididas em dois grandes grupos:

1. **Macros especificadoras de formato de impressão.** Todos os nomes das macros deste grupo começam com *PRI*. Este prefixo é derivado das três letras iniciais de `printf`, que é o nome que identifica a família de funções de saída formatada.
2. **Macros especificadoras de formato de leitura.** Todos os nomes das macros deste grupo começam com *SCN*. Este prefixo é derivado das três consoantes iniciais de `scanf`, que é o nome que identifica a família de funções de leitura formatada.

Em seguida, essas macros são divididas em dois outros grandes grupos:

1. **Macros especificadoras de formato de inteiros com sinal.** Todos os nomes das macros deste grupo têm *d* ou *i* como quarta letra constituinte. A escolha destas letras deriva do fato de elas constituírem os especificadores mais comuns de leitura ou impressão de inteiros (i.e., `%d` e `%i`). Não existe diferença entre escolher uma macro que use *d* ou uma macro que use *i* se esta macro for usada para impressão, mas há diferença se ela for usada para leitura (v. **Apêndice B**).
2. **Macros especificadoras de formato de inteiros sem sinal.** Todos os nomes das macros deste grupo têm como quarto constituinte uma letra derivada de um especificador de formato para inteiros sem sinal. Agora, a escolha é mais ampla e a letra pode ser uma daquelas apresentadas na tabela a seguir, de acordo com a base de representação utilizada.

LETRA	BASE	ESPECIFICADOR DE FORMATO CORRESPONDENTE
<b>o</b>	Octal	<b>%o</b>
<b>u</b>	Decimal	<b>%u</b>
<b>X ou x</b>	Hexadecimal	<b>%X ou %x</b>

Os grandes grupos de macros apresentados são mutuamente exclusivos. Combinados, portanto, existem quatro grandes grupos de macros especificadoras de formato:

1. Macros especificadoras de formato de impressão de inteiros com sinal
2. Macros especificadoras de formato de impressão de inteiros sem sinal
3. Macros especificadoras de formato de leitura de inteiros com sinal
4. Macros especificadoras de formato de leitura de inteiros sem sinal

As letras que compõem o final (sufixo) das macros especificadoras de formato são derivadas dos nomes dos tipos (v. **Seção 2.4.1**) aos quais elas estão associadas, conforme apresentado na tabela a seguir:

CATEGORIA DE TIPOS	SUFIXO
Inteiros com larguras exatas	$n$ , onde $n$ é a largura do tipo
Inteiros que têm pelo menos certa largura	LEAST $n$ , onde $n$ é a largura do tipo
Inteiros rápidos que têm pelo menos certa largura	FAST $n$ , onde $n$ é a largura do tipo
Inteiros capazes de representar ponteiros	PTR
Inteiros de maior largura (com ou sem sinal)	MAX

Finalmente, usando-se as regras de formação descritas, podem-se apresentar esquematicamente todas as macros especificadoras de formato, o que é feito nas quatro tabelas a seguir.

PRId $n$	PRIdLEAST $n$	PRIdFAST $n$	PRIdMAX	PRIdPTR
PRId $n$	PRIdLEAST $n$	PRIdFAST $n$	PRIdMAX	PRIdPTR

Tabela 2-7: Macros especificadoras de formato de impressão de inteiros com sinal.

PRIo $n$	PRIoLEAST $n$	PRIoFAST $n$	PRIoMAX	PRIoPTR
PRIo $n$	PRIoLEAST $n$	PRIoFAST $n$	PRIoMAX	PRIoPTR
PRIo $n$	PRIoLEAST $n$	PRIoFAST $n$	PRIoMAX	PRIoPTR
PRIo $n$	PRIoLEAST $n$	PRIoFAST $n$	PRIoMAX	PRIoPTR

Tabela 2-8: Macros especificadoras de formato de impressão de inteiros sem sinal.

SCNd $n$	SCNdLEAST $n$	SCNdFAST $n$	SCNdMAX	SCNdPTR
SCNi $n$	SCNiLEAST $n$	SCNiFAST $n$	SCNiMAX	SCNiPTR

Tabela 2-9: Macros especificadoras de formato de leitura de inteiros com sinal.

SCNo <i>n</i>	SCNoLEAST <i>n</i>	SCNoFAST <i>n</i>	SCNoMAX	SCNoPTR
SCNu <i>n</i>	SCNuLEAST <i>n</i>	SCNuFAST <i>n</i>	SCNuMAX	SCNuPTR
SCNx <i>n</i>	SCNxLEAST <i>n</i>	SCNxFAST <i>n</i>	SCNxMAX	SCNxPTR

Tabela 2-10: Macros especificadoras de formato de leitura de inteiros sem sinal.

Ao usar estas macros, não esqueça que *n* representa um espaço a ser preenchido por um número que especifica a largura do tipo que o especificador de formato representa. Por exemplo, se você deseja imprimir um valor do tipo `int16_t`, utilize a macro especificadora de formato **PRId16** ou **PRi16**; se você deseja imprimir um valor do tipo `uint_least32_t` no formato hexadecimal, utilize a macro especificadora de formato **PRIxLEAST32** ou **PRIXLEAST32**, e assim por diante.

A utilização dessas macros em funções de entrada e saída formatadas pode gerar algumas dúvidas que poderão ser dissipadas se você lembrar os seguintes fatos:

- Todas essas macros expandem-se em especificadores de formatos em forma de *strings*. Por exemplo, a macro **PRId32** pode ser expandida no especificador de formato "d" ou "ld", dependendo da implementação. Mas, independentemente de implementação, a expansão de qualquer dessas macros sempre resultará num *string*.
- O *string* resultante da expansão de uma macro especificadora de formato nunca inclui o caractere %, que é necessário na formação de qualquer especificador de formato.
- Macros são sempre expandidas pelo pré-processador e este não processa o que se encontra no interior de *strings*. Isto significa que essas macros não devem ser inseridas no interior de *strings* de formatação de funções de entrada ou saída formatada.

A seguir são apresentados dois exemplos de uso incorreto de macros especificadoras de formato e suas respectivas possíveis correções.

### Exemplo 1

```
uint32_t umInt;
...
printf("PRIu32\n", umInt);
```

Como resultado da execução deste trecho de programa, seria impresso o seguinte no meio de saída padrão:

```
PRIu32
```

O uso correto da macro **PRIu32** neste exemplo seria:

```
printf("%" PRIu32 "\n", umInt);
```

Note que o uso de espaços em branco em torno da macro **PRIu32** não é necessário, mas ajuda a melhorar a legibilidade.

### Exemplo 2

```
intmax_t outroInt;
...
printf("Valor de outroInt = %PRIdMAX\n", outroInt);
```

Como resultado da execução deste trecho de programa, seria impresso o seguinte no meio de saída padrão:

```
Valor de outroInt = PRIdMAX
```

O uso correto da macro **PRIdMAX** neste exemplo seria:

```
printf("Valor de outroInt = %" PRIdMAX "\n", outroInt);
```

Novamente, os espaços em branco em torno da macro **PRIu32** foram usados para melhorar a legibilidade.

## 2.5.3 FUNÇÕES

*imaxabs()* (C99)

**Incluir:** <inttypes.h>



**Descrição:** A função **imaxabs()** (C99) retorna o valor absoluto de um inteiro do tipo **intmax\_t**.

**Protótipo:**

```
intmax_t imaxabs(intmax_t umInt)
```

**Parâmetro:** **umInt** – valor do tipo **intmax\_t** que terá seu valor absoluto calculado.

**Exemplo:**

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

int main(void)
{
    intmax_t x = INTMAX_MIN/5;

    printf( "Valor absoluto de %" PRIdMAX ": %" PRIdMAX,
           x, imaxabs(x) );

    putchar('\n');

    return 0;
}
```

*imaxdiv()* (C99)

**Incluir:** <inttypes.h>

**Descrição:** A função **imaxdiv()** (C99) retorna, numa estrutura do tipo **imaxdiv\_t**, o quociente e o resto da divisão do seu primeiro parâmetro pelo segundo.

**Protótipo:**

```
imaxdiv_t imaxdiv(intmax_t numerador, intmax_t denominador)
```

**Parâmetros:**

- **numerador** – valor inteiro a ser dividido.

- `denominador` – o divisor.

### Observações:

- O membro `quot` da estrutura retornada é o quociente dos argumentos truncado.
- A seguinte relação é satisfeita:

```
numerador == E.quot*denominador + E.rem
```

onde `E` é a estrutura retornada.

- Compare esta função com `div()`, `ldiv()` e `lldiv()` declaradas em `<stdlib.h>` (v. **Seção 2.6.2**).

### Exemplo:

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

int main(void)
{
    intmax_t    numerador = INTMAX_MAX,
               denominador = 5;
    imaxdiv_t    resultado;

    resultado = imaxdiv(numerador, denominador);
    printf("numerador = %" PRIuMAX ", denominador = %"
           PRIuMAX "\n", numerador, denominador );
    printf("quociente = %" PRIuMAX ", resto = %"
           PRIuMAX "\n", resultado.quot, resultado.rem);

    return 0;
}
```

### *strtoimax()* (C99)

**Incluir:** `<inttypes.h>`

**Descrição:** A função `strtoimax()` (C99) converte um *string* num número inteiro do tipo `intmax_t`.

**Protótipo:**

```
intmax_t strtoumax( const char *restrict string,
                   char **restrict final, int base )
```

**Parâmetros:**

- *string* – *string* a ser convertido.
- *final* – ponteiro para a porção do *string* original que não será convertida, se este parâmetro não for **NULL**.
- *base* – a base a ser usada na conversão.

**Retorno:** Valor, do tipo **intmax\_t**, resultante da conversão, se não ocorrer erro; caso contrário, zero.

**Observações:**

- A conversão efetuada por esta função é semelhante àquela efetuada por **strtoul()** (v. **Seção 6.5.1**).
- A conversão encerra quando um caractere inválido é encontrado.
- A base (terceiro parâmetro) deve ser zero ou estar entre 2 e 36. Se este valor for igual a 1, menor que zero ou maior que 36, este valor será considerado inválido. Qualquer valor inválido para *base* faz com que o retorno seja zero e *final* aponte para o início do parâmetro *string*.
- Se o valor do parâmetro *base* for igual a zero, então a base do número será determinada pelo formato dele. Isto é, se o número começar com 0x ou 0X, ele será considerado hexadecimal; se ele começar com zero, a base octal será assumida; em outros casos, a base será considerada decimal.
- Apenas os dígitos e as letras correspondentes à base fornecida são reconhecidos durante a conversão.
- Se o valor resultante for grande demais para ser representado como um valor do tipo **intmax\_t**, a função armazenará o valor **ERANGE** na variável **errno** e retornará **INTMAX\_MAX**, se o resultado for positivo, ou **INTMAX\_MIN**, se o resultado for negativo.
- Compare esta função com as funções **atoi()**, **atol()**, **strtoul()** e **strtoull()**.

**Exemplo:**

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

int main(void)
{
    intmax_t umIntMax = 0;
    char      *string = "-123487866480098445567abc",
              *resto;

    umIntMax = strtoumax(string, &resto, 0);

    printf("String original: \"%s\"\n", string);
    printf("\nValor convertido para intmax_t: %"
           PRIuMAX "\n", umIntMax);
    printf("\nResto do string original que nao foi "
           "convertido: \"%s\"\n", resto);

    if (errno == ERANGE)
        printf("\nO valor convertido e' demasiadamente "
              "grande ou pequeno\n");

    return 0;
}
```

***strtoumax()* (C99)****Incluir:** <inttypes.h>

**Descrição:** A função **strtoumax()** (C99) converte um *string* num número inteiro do tipo **uintmax\_t**.

**Protótipo:**

```
uintmax_t strtoumax( const char *restrict string,
                    char **restrict final, int base )
```

**Parâmetros:**

- *string* – *string* a ser convertido.

- *final* – ponteiro para a porção do *string* original que não será convertida, se este parâmetro não for **NULL**.
- *base* – especifica a base a ser usada na conversão. Este valor deve ser zero ou estar entre 2 e 36.

**Retorno:** Valor do tipo **uintmax\_t** resultante da conversão, se não ocorrer erro; caso contrário, zero.

#### Observações:

- Esta função funciona de modo similar a **strtoimax()**; a principal diferença é o tipo do valor convertido.
- Se o valor resultante for grande demais para ser representado como um valor do tipo **uintmax\_t**, a função armazenará o valor **ERANGE** na variável **errno** e retornará **UINTMAX\_MAX**.
- Consulte **strtoimax()** para obter maiores informações.

#### Exemplo:

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

int main(void)
{
    intmax_t  umIntMax;
    char      *string = "123487866480098445567abc",
              *resto;

    umIntMax = strtoumax(string, &resto, 0);

    printf("String original: \"%s\"\n", string);
    printf( "Valor convertido para intmax_t: %"
            PRIuMAX "\n", umIntMax );
    printf( "Resto do string original que nao "
            "foi convertido: \"%s\"\n", resto);

    if (errno == ERANGE)
        printf("\nO valor convertido e' grande demais\n");
}
```

```
    return 0;
}
```

### *wcstoimax()* (C99)

**Incluir:** `<inttypes.h>`

**Descrição:** A função **wcstoimax()** (C99) converte um *string* extenso num número inteiro do tipo **intmax\_t**.

**Protótipo:**

```
intmax_t wcstoimax( const wchar_t *restrict strExt,
                    wchar_t **restrict final, int base )
```

**Parâmetros:**

- `strExt` – *string* extenso a ser convertido.
- `final` – ponteiro para a porção do *string* original que não será convertida, se este parâmetro não for **NULL**.
- `base` – especifica a base a ser usada na conversão.

**Retorno:** Valor do tipo **intmax\_t** resultante da conversão, se não ocorrer erro; caso contrário, zero.

**Observações:**

- Esta função funciona de modo similar a **strtoimax()**; a principal diferença é que **strtoimax()** converte *strings* de caracteres monobytes, enquanto que **wcstoimax()** converte *strings* extensos. Consulte **strtoimax()** para obter maiores informações.
- Se o valor resultante da conversão for grande demais para ser representado no tipo **intmax\_t**, a função atribuirá **ERANGE** à variável **errno** e retornará **INTMAX\_MAX**, se o valor for positivo, ou **INTMAX\_MIN**, se o valor for negativo.
- Compare esta função com outras funções de conversão de *strings* extensos em números apresentadas na **Seção 8.5.6**.

**Exemplo:**

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

int main()
{
    intmax_t  im;
    wchar_t   *string = L"-1234567abc", *resto;

    im = wcstoimax(string, &resto, 0);

    printf("String original: \"%ls\"\n", string);
    printf("Valor convertido para intmax_t: %"
           PRIdMAX "\n", im );
    printf("Resto do string original que nao "
           "foi convertido: \"%ls\"\n", resto );

    return 0;
}
```

***wcstoumax()* (C99)****Incluir:** <inttypes.h>

**Descrição:** A função **wcstoumax()** (C99) converte um *string* extenso num número inteiro do tipo **uintmax\_t**.

**Protótipo:**

```
uintmax_t wcstoumax(const wchar_t *restrict strExt,
                    wchar_t **restrict final, int base)
```

**Parâmetros:**

- *strExt* – o *string* extenso a ser convertido.
- *final* – ponteiro para a porção do *string* original que não será convertida, se este parâmetro não for **NULL**.
- *base* – especifica a base a ser usada na conversão.

**Retorno:** Valor do tipo **uintmax\_t** resultante da conversão, se não ocorrer erro; caso contrário, zero.

### Observações:

- Esta função funciona de modo similar a **strtoumax()**; a principal diferença é que **strtoumax()** converte *strings* de caracteres monobytes, enquanto que **wcstoumax()** converte *strings* extensos. Consulte **strtoimax()** para obter maiores informações.
- Se o valor resultante da conversão for grande demais para ser representado no tipo **uintmax\_t**, a função atribuirá **ERANGE** à variável **errno** e retornará **UINTMAX\_MAX**.
- Compare esta função com outras funções de conversão de *strings* extensos em números apresentadas na **Seção 8.5.6**.

### Exemplo:

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

int main()
{
    uintmax_t  uim;
    wchar_t    *string = L"1234567abc", *resto;

    uim = wcstoumax(string, &resto, 0);

    printf("String original: \"%ls\"\n", string);
    printf("Valor convertido para uintmax_t: %"
           PRIuMAX "\n", uim );
    printf("Resto do string original que nao "
           "foi convertido: \"%ls\"\n", resto );

    return 0;
}
```



## 2.6 OPERAÇÕES ARITMÉTICAS INTEIRAS

As funções e os tipos apresentados nesta seção são declarados no cabeçalho `<stdlib.h>`, que é um cabeçalho de propósito geral apresentado no **Capítulo 12**.

### 2.6.1 TIPOS

*div\_t*

**Incluir:** `<stdlib.h>`

**Descrição:** **div\_t** é o tipo da estrutura retornada pela função **div()**. Uma estrutura deste tipo contém dois campos:

- `quot` – representa o quociente de uma divisão de dois valores do tipo **int**.
- `rem` – representa o resto da divisão de dois valores do tipo **int**.

*ldiv\_t*

**Incluir:** `<stdlib.h>`

**Descrição:** **ldiv\_t** é o tipo da estrutura retornada pela função **ldiv()**. Este tipo é semelhante à **div\_t**, mas aplica-se ao tipo **long int**.

*lldiv\_t (C99)*

**Incluir:** `<stdlib.h>`

**Descrição:** **lldiv\_t** é o tipo da estrutura retornada pela função **lldiv()**. Este tipo é semelhante à **div\_t**, mas aplica-se ao tipo **long long int**.

### 2.6.2 FUNÇÕES

*abs()*

**Incluir:** `<stdlib.h>`

**Descrição:** A função **abs()** retorna o valor absoluto do parâmetro recebido.

**Protótipo:**

```
int abs(int valorInt)
```

**Parâmetro:** `valorInt` – valor do tipo **int** cujo valor absoluto será obtido.

**Observações:**

- **abs()** pode ser definida como macro.
- Note que um valor do tipo **int** não pode conter o valor absoluto de **INT\_MIN** porque **-INT\_MIN** resulta novamente em **INT\_MIN**; i.e., `abs(INT_MIN)` resulta em **INT\_MIN** [v. exemplo de **labs()**].

**Exemplo:**

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

int main(void)
{
    printf("abs(-10) = %d\n", abs(-10));

    return 0;
}
```

*div()*

**Incluir:** `<stdlib.h>`

**Descrição:** A função **div()** calcula o quociente e o resto da divisão de dois valores do tipo **int**.

**Protótipo:**

```
div_t div(int numerador, int denominador)
```

**Parâmetros:**

- `numerador` – valor a ser dividido.

- denominador – divisor.

**Retorno:** Um valor do tipo `div_t` (v. **Seção 2.6.1**) contendo o resultado da divisão.

**Observação:** Esta função é semelhante às funções `ldiv()` e `lldiv()`. A principal diferença entre estas funções é o tipo de dado sobre o qual cada uma delas atua.

### Exemplo:

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

int main(void)
{
    int    num = 10, den = 3;
    div_t  res = div(10, 3);

    printf( "numerador = %d, denominador = %d\n",
            num, den );
    printf( "quociente = %d, resto = %d\n",
            res.quot, res.rem );

    return 0;
}
```

### *labs()*

**Incluir:** `<stdlib.h>`

**Descrição:** A função `labs()` retorna o valor absoluto de seu argumento do tipo **long**.

**Protótipo:**

<code>long labs(long d)</code>
--------------------------------

**Parâmetro:** `d` – valor do qual o valor absoluto é desejado.

**Retorno:** O valor absoluto do argumento.

**Observação:** Um valor do tipo **long int** não pode representar o valor absoluto de `LONG_MIN` porque `-LONG_MIN` resulta novamente em `LONG_MIN`; i.e., `labs(LONG_MIN)` resulta em `LONG_MIN` [v. exemplo de `llabs()` adiante].

**Exemplo:**

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

int main()
{
    /* Se o valor LLONG_MIN fosse usado, ocorreria */
    /* overflow pois não há valor do tipo long long */
    /* que possa conter o resultado. Neste caso, o */
    /* valor retornado por labs() seria LLONG_MIN, */
    /* que é um valor negativo. */
    printf( "Valor absoluto de %lld = %lld\n",
           LLONG_MIN, labs(LLONG_MIN) );

    return 0;
}
```

Quando compilado e executado no Linux, o último programa imprime o seguinte no meio de saída padrão:

```
Valor absoluto de -2147483647 = 2147483647
```

Compare o resultado apresentado por este último programa com aquele resultante do programa apresentado como exemplo da função **labs()**.

*llabs() (C99)*

**Incluir:** <stdlib.h>

**Descrição:** A função **llabs()** (C99) retorna o valor absoluto de seu argumento do tipo **long long**.

**Protótipo:**

```
long long llabs(long long d)
```

**Parâmetro:** d – valor do qual o valor absoluto é desejado.

**Retorno:** O valor absoluto do argumento.

**Observações:**

- Esta função é semelhante à **abs()** e **labs()**; a diferença é que aqui o argumento e o valor retornado são do tipo **long long**.
- Compare esta função com **abs()** e **labs()**.

**Exemplo:**

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

int main()
{
    /* Se o valor LLONG_MIN for usado, ocorrerá */
    /* overflow pois não há valor do tipo long */
    /* long que possa conter o resultado. Neste */
    /* caso, o valor retornado por labs() será */
    /* LLONG_MIN, que é um valor negativo.      */
    printf( "Valor absoluto de %lld = %lld\n",
            LLONG_MIN, llabs(LLONG_MIN) );

    return 0;
}
```

Quando compilado e executado no Linux, o último programa imprime o seguinte no meio de saída padrão:

```
Valor absoluto de -9223372036854775808 = -9223372036854775808
```

Observe que apesar das semelhanças do último programa com aquele apresentado como exemplo da função **labs()**, os dois programas produzem resultados substancialmente diferentes. O programa que demonstra o uso de **labs()** produz um resultado esperado, enquanto o último programa apresentado produz um resultado incorreto devido à ocorrência de *overflow*.

*ldiv()*

**Incluir:** <stdlib.h>

**Descrição:** A função **ldiv()** calcula o quociente e o resto da divisão de dois números do tipo **long**.

**Protótipo:**

`ldiv_t ldiv(long numerador, long denominador)`

**Parâmetros:**

- `numerador` – valor a ser dividido.
- `denominador` – o divisor.

**Retorno:** Estrutura do tipo **ldiv\_t** contendo o resultado da divisão.

**Observações:**

- Esta função é semelhante à **div()**; a diferença é que aqui os argumentos são do tipo **long** e o valor retornado é do tipo **ldiv\_t**.
- Compare esta função com **div()** e **lldiv()**.

**Exemplo:**

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

int main ()
{
    int    quantia = -1234;
    char   moeda[] = "R$";
    char   sinal;
    ldiv_t reaisComCentavos;

    if (quantia > 0)
        sinal = '+';
    else if (quantia < 0)
        sinal = '-';
    else
        sinal = ' ';

    reaisComCentavos = ldiv(abs(quantia), 100);
```

```

printf( "\nSeu saldo e' %c%s%ld.%2ld\n",
        sinal, moeda, reaisComCentavos.quot,
        reaisComCentavos.rem );

return 0;
}

```

### *lldiv() (C99)*

**Incluir:** <stdlib.h>

**Descrição:** A função **lldiv()** (C99) calcula o quociente e o resto da divisão de dois números do tipo **long long**.

**Protótipo:**

```

lldiv_t lldiv( long long numerador,
               long long denominador )

```

**Parâmetros:**

- `numerador` – valor do tipo **long long** a ser dividido.
- `denominador` – divisor.

**Retorno:** Estrutura do tipo **lldiv\_t** contendo o resultado da divisão.

**Observações:**

- Esta função é semelhante à **div()**; a diferença é que aqui os argumentos são do tipo **long long** e o valor retornado é do tipo **lldiv\_t**.
- Compare esta função com **div()** e **ldiv()**.

**Exemplo:** Veja exemplos das funções **div()** e **ldiv()**.

## 2.7 EXERCÍCIOS DE REVISÃO

1. (a) Por que a maioria dos tipos inteiros primitivos da linguagem C não é portátil? (b) Quais são os únicos tipos inteiros primitivos portáteis de C?
2. Em linhas gerais, qual é o conteúdo do cabeçalho `<limits.h>`?
3. Explique a utilidade da constante **CHAR\_BIT**.
4. Qual é o objetivo do cabeçalho `<stdint.h>`?
5. Quais são os tipos providos pelo cabeçalho `<stdint.h>`?
6. O que é um inteiro rápido?
7. Quais são as interpretações dadas aos seguintes tipos providos pelo cabeçalho `<stdint.h>`?

(a) **int32\_t**

(b) **uint16\_t**

(c) **uintmax\_t**

(d) **int\_fast8\_t**

8. Qual é o significado de cada uma das seguintes macros?

(a) **INT16\_MAX**

(b) **INT16\_MIN**

(c) **UINT16\_MAX**

9. Como cada uma das macros a seguir poderia ser expandida numa dada implementação?

(a) **PRId8**

(b) **PRIdLEAST32**

(c) **PRIdFAST16**

(d) **PRIdMAX**

(e) **PRIdPTR**



(f) **PRiMAX**

(g) **SCNiFAST32**

(h) **SCNuLEAST8**

10. O tipo **char** pode ser usado em vez de **int8\_t** de modo portátil? Explique.
11. (a) Qual é a expansão da macro **SIZE\_MAX**? (b) Por que não existe uma macro denominada **SIZE\_MIN**?
12. O que há de errado no seguinte fragmento de programa?
 

```
uint32_t umInt;
...
printf("Valor de umInt = %PRi32\n", umInt);
```
13. Qual é a vantagem do uso da função **div()** com relação ao uso dos operadores **/** e **%**?

# *Capítulo 3*

---

*Números de ponto flutuante reais*

## 3.1 INTRODUÇÃO

Este capítulo começa com uma breve exposição sobre os tipos de ponto flutuante reais primitivos da linguagem C. Em seguida, serão apresentados diversos conceitos básicos relacionados a operações de ponto flutuante. O objetivo dessas duas seções iniciais é prover o conhecimento mínimo necessário para entender bem o suporte para operações de ponto flutuante provido pela biblioteca padrão de C. Entretanto, o presente texto não provê conhecimento matemático específico para entendimento da maioria das operações matemáticas representados por funções declaradas nos cabeçalhos aqui apresentados. Por exemplo, está fora do escopo deste livro dotar o leitor de conhecimento básico necessário para fazê-lo entender funções gama ou hiperbólicas.

Este capítulo apresenta cabeçalhos cujos componentes proveem a maioria das operações matemáticas envolvendo os tipos de ponto flutuante reais de C. Estes cabeçalhos são:

- `<float.h>` – que descreve, por meio de macros, propriedades dos números de ponto flutuante.
- `<math.h>` – que dá suporte para operações matemáticas de ponto flutuante real.
- `<fenv.h>` – que provê meios para controlar como as operações de ponto flutuante são efetuadas.

## 3.2 TIPOS PRIMITIVOS DE PONTO FLUTUANTE REAIS

Os tipos de ponto flutuante primitivos de C são divididos em duas categorias: **tipos reais** e **tipos complexos**, conforme mostra a **Figura 3-1**. Nesta figura, os três tipos primitivos de ponto flutuante reais de C, **float**, **double** e **long double**, aparecem em negrito. O padrão de C não especifica os tamanhos destes tipos, mas especifica que o conjunto de valores do tipo **float** é um subconjunto do conjunto de valores do tipo **double**, que, por sua vez, é um subconjunto do conjunto de valores do tipo **long double**.

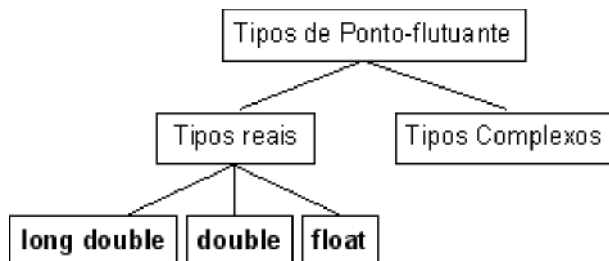


Figura 3-1: Tipos primitivos de ponto Flutuante reais.

Constantes de ponto flutuante reais podem ser escritas de três maneiras.

- **Notação convencional.** Esta notação consiste em simplesmente uma parte inteira e outra decimal separadas por um ponto decimal. Exemplos: 3.2, 7., .5.
- **Notação científica.** Nesta notação, um número de ponto flutuante consiste em duas partes: (1) **mantissa** – que é um número de ponto flutuante em notação convencional – e (2) **expoente** – que é um inteiro representando uma potência de 10. Estas duas partes são separadas pela letra **e** ou **E**. Por exemplo, 2.5E-3 representa o número  $2.5 \times 10^{-3}$ , ou seja, 0.0025.
- **Formato hexadecimal.** Neste formato, introduzido pelo padrão C99, uma constante de ponto flutuante em formato hexadecimal é constituída por<sup>21</sup>:
  - Um prefixo hexadecimal (0x ou 0X).
  - Algarismos hexadecimais significativos representando a parte inteira e a parte fracionária.
  - Um expoente binário representando uma potência de 2 que deve ser multiplicada pela parte significativa.
  - Uma letra P ou p separando os algarismos significativos do expoente.

Uma constante de ponto flutuante em qualquer dos formatos mencionados pode incluir um sufixo, que pode ser **F** ou **L**, que informa que a constante deve ser interpretada como **float** ou **long double**, respectivamente. Na ausência de sufixo, a constante é interpretada como **double**.

<sup>21</sup> Constantes de ponto flutuante em formato hexadecimal são descritas em maiores detalhes no **Volume I**.

### 3.3 CONCEITOS FUNDAMENTAIS DE ARITMÉTICA DE PONTO FLUTUANTE

Esta seção apresenta vários conceitos a que se fazem referências nas seções seguintes. Estes conceitos são apresentados com o propósito de prover o mínimo conhecimento necessário para melhor entendimento dos componentes dos cabeçalhos descritos neste capítulo<sup>22</sup>.

#### 3.3.1 UNDERFLOW E OVERFLOW

**Underflow** é uma condição de erro que ocorre quando um computador tenta representar um número cuja magnitude é pequena demais (i.e., muito próximo de zero) para ser representada. Por exemplo, se uma dada representação de ponto flutuante permite uma precisão de quatro dígitos e o menor expoente é  $-50$ , então o menor número positivo suportado é  $1.000 \times 10^{-50}$ . Se uma dada operação de ponto flutuante produz um valor cuja magnitude é menor do que esse valor (e.g.,  $0.5 \times 10^{-50}$ ), então ocorre uma condição de underflow. De modo análogo, **overflow** é uma condição de erro que ocorre quando o computador tenta representar um número cuja magnitude é grande demais para ser representada<sup>23</sup>.

Um programa pode responder a uma condição de *overflow* ou *underflow* de modos diferentes. Alguns programas assinalam a ocorrência de uma exceção (v. **Seção 3.3.4**), enquanto outros fazem aproximações e continuam o processamento normalmente.

#### 3.3.2 REPRESENTAÇÕES

Um número real na base decimal está em **forma normalizada** quando é escrito como:

$$\pm d_0.d_1d_2d_3\dots \times 10^n$$

onde  $1 \leq d_0 \leq 9$ ,  $d_1, d_2, d_3, \dots$  são dígitos decimais e  $n$  é um valor inteiro. Qualquer número exceto zero pode ser colocado nesta forma.

<sup>22</sup> Ao leitor que desejar obter conhecimento mais profundo sobre estes conceitos, recomenda-se o excelente artigo *What Every Computer Scientist Should Know About Floating-Point Arithmetic* (v. **Bibliografia**).

<sup>23</sup> *Overflow* também pode ocorrer em operações inteiras. Este capítulo lida apenas com ocorrências de overflow em operações de ponto flutuante.

De modo geral, um número real (diferente de zero) na base  $b$  pode ser normalizado como:

$$\pm d_0 . d_1 d_2 d_3 \dots \times b^n$$

onde  $1 \leq d_0 \leq b - 1$ ,  $d_1, d_2, d_3, \dots$  são dígitos da base  $b$  e  $n$  é um valor inteiro.

**Números subnormais** são aqueles cujos valores absolutos são menores do que o menor valor representável (i.e., valores que causam *underflow*).

Um valor que não representa um número de ponto flutuante válido é denominado NaN<sup>24</sup>. Quando a avaliação de uma expressão de ponto flutuante resulta em NaN, pode ser que um sinalizador de exceção de ponto flutuante correspondente seja ligado para indicar a ocorrência de um valor inválido. Quando isto não ocorre, diz-se que ocorrência de NaN é **silenciosa**.

### 3.3.3 ERROS DE DOMÍNIO E DE INTERVALO

Muitas das macros definidas em `<math.h>` são utilizadas em tratamentos de erros que possam ocorrer durante a execução de alguma função declarada neste cabeçalho. Estes erros são classificados em duas categorias: erro de domínio e erro de intervalo.

Independentemente da categoria de um erro, uma função pode relatar sua ocorrência de duas maneiras : (1) atribuindo um valor diferente de zero à variável **errno** (v. **Seção 11.5.2**) ou (2) usando uma **variável sinalizadora de exceções**. O papel das macros **math\_errhandling**, **MATH\_ERRNO** e **MATH\_ERREXCEPT**, definidas em `<math.h>`, é exatamente especificar qual destas alternativas é utilizada pelas funções declaradas nesse mesmo cabeçalho.

---

<sup>24</sup> NaN significa *not a number* em inglês.

### *Erros de domínio*

Um **erro de domínio** ocorre quando o valor de pelo menos um dos argumentos de uma função não corresponde a um valor esperado por ela<sup>25</sup>. Por exemplo, a função **atanh()**, descrita na **Seção 3.6.4**, espera receber um valor compreendido entre  $-1$  e  $1$  (sem incluir estes valores). Assim, se o valor do parâmetro passado para esta função estiver fora deste intervalo, ocorrerá um erro de domínio.

Uma função pode comunicar a ocorrência de um erro de domínio de duas maneiras (não mutuamente exclusivas):

1. Armazenando o valor da macro **EDOM** na variável global **errno** e retornando um valor dependente de implementação que corresponde ao tipo específico de erro ocorrido.
2. Utilizando um sinalizador de exceção de ponto flutuante (v. **Seção 3.3.4**).

### *Erros de intervalo*

Um **erro de intervalo** ocorre quando a magnitude do valor a ser retornado por uma função é grande ou pequeno demais para ser representado no tipo declarado como tipo de retorno da função. Novamente, uma função pode comunicar a ocorrência de um erro de intervalo de duas maneiras (não mutuamente exclusivas):

1. Armazenando o valor da macro **ERANGE** na variável global **errno** e retornando zero ou o valor de uma das macros: **HUGE\_VAL**, **HUGE\_VALF**, **HUGE\_VALL**, **-HUGE\_VAL**, **-HUGE\_VALF** e **HUGE\_VALL** (v. **Seção 3.6.2**).
2. Utilizando um sinalizador de exceção de ponto flutuante (v. **Seção 3.3.4**).

## **3.3.4 EXCEÇÕES DE PONTO FLUTUANTE**

Uma **exceção de ponto flutuante** resulta de uma situação anormal que pode ocorrer durante a execução de uma operação de ponto flutuante<sup>26</sup>. Como exemplos de tais exceções, podem ser citadas: uma tentativa de divisão por zero e uma operação que resulta em *underflow* ou *overflow* e, por isso, não pode ser representada.

---

<sup>25</sup> Note que isto não tem relação com a ocorrência ou não de conversão de tipos durante a passagem de parâmetros para uma função. Por exemplo, se um argumento de uma função for do tipo **double** e o argumento correspondente passado para a função for do tipo **int**, ocorrerá conversão automática conforme descrito no **Volume I** e este fato em si não acarreta em nenhum erro de domínio.

<sup>26</sup> O conceito geral de exceção é descrito em detalhes no **Capítulo 11**.

Um **senalizador de exceção** é uma variável associada a um tipo específico de exceção e que representa a ocorrência ou não de uma exceção deste tipo. Portanto, um sinalizador precisa apenas de um bit para definir o status (ocorrência/não-ocorrência) da exceção que o sinalizador representa. Assim, é comum utilizarem-se os termos **ligado** – significando a ocorrência de exceção – e **desligado** – significando a não-ocorrência de exceção.

Um sinalizador pode ser ligado automaticamente ou explicitamente pelo programa, mas, tipicamente, ele deve ser desligado explicitamente pelo programa após ter sido levado em consideração. Se um sinalizador de exceção não for desligado após ter sido levado em conta, ele permanecerá ligado e dificultará a identificação da operação que ligou o sinalizador. Por exemplo, se, após a execução de uma operação, for verificado que o sinalizador de *overflow* está ligado, só será possível concluir que foi esta última operação que ligou o sinalizador se se tiver certeza de que este sinalizador estava desligado antes desta mesma operação.

### 3.3.5 MODOS DE ARREDONDAMENTO

Existem várias maneiras pelas quais se pode arredondar um número de ponto flutuante. Tipicamente, uma implementação de C dispõe de alguns modos de arredondamento comuns e permite que o modo de arredondamento corrente seja alterado durante a execução de um programa. Algumas formas de arredondamento comuns são descritas a seguir.

- **Arredondamento com afastamento de zero.** Neste modo de arredondamento, o valor é arredondado para o próximo valor com o número de algarismos significativos desejado mais afastado de zero. Por exemplo, se for desejado um algarismo significativo, 4.1 e 4.8 serão arredondados para 5.0, enquanto que -4.1 e -4.8 serão arredondados para -5.0. Isto é, os valores com um dígito significativo mais próximos de 4.1 e 4.8 são, respectivamente, 4.0 e 5.0, sendo este último o valor mais afastado de zero. No segundo exemplo, os valores com um dígito significativo mais próximos de -4.1 e -4.8 são, respectivamente, -4.0 e -5.0, e -5.0 é o valor mais afastado de zero.
- **Arredondamento com aproximação de zero.** Aqui, o valor é arredondado para o próximo valor com o número de algarismos significativos desejado mais próximo de zero. Por exemplo, se for desejado um algarismo significativo, 4.1 e 4.8 serão arredondados para 4.0, enquanto que -4.1 e -4.8 serão arredondados para -4.0. Isto é, os valores com um dígito significativo mais



próximos de 4.1 e 4.8 são, respectivamente, 4.0 e 5.0, sendo que o primeiro valor é mais próximo de zero. No segundo exemplo, os inteiros mais próximos de -4.1 e -4.8 são, respectivamente, -4.0 e -5.0, e -4.0 é o valor mais próximo de zero. Este modo de arredondamento é também conhecido como **truncamento**.

- **Arredondamento para o mais próximo.** Este é o arredondamento mais comum no cotidiano e considerado padrão pela maioria das implementações. Neste modo de arredondamento, o valor é arredondado para o valor mais próximo com o número de algarismos significativos desejado. Por exemplo, 4.1 é arredondado para 4.0 e 4.8 é arredondado para 5.0, enquanto que -4.1 é arredondado para -4.0 e -4.8 é arredondado para -5.0. Quando o valor a ser arredondado é equidistante de dois possíveis resultados do arredondamento, o arredondamento é feito para aquele valor que for par. Por exemplo, 4.5 é equidistante de 4.0 e 5.0 e, seguindo esta última regra, é arredondado para 4.0; por outro lado, 1.5, que é equidistante de 1.0 e 2.0, é arredondado para 2.0.
- **Arredondamento para baixo.** Aqui, o arredondamento é feito para o valor mais próximo que não seja maior do que o valor a ser arredondado. Por exemplo, 2.2, 2.5 e 2.9 são arredondados para 2.0, enquanto que -2.2, -2.5 e -2.9 são arredondados para -3.0.
- **Arredondamento para cima.** Neste modo, o arredondamento é feito para o valor mais próximo que não seja menor do que o valor a ser arredondado. Por exemplo, 2.2, 2.5 e 2.9 são arredondados para 3.0, enquanto que -2.2, -2.5 e -2.9 são arredondados para -2.0.

O modo de arredondamento corrente afeta o resultado da maioria das operações de ponto flutuante.

### 3.3.6 PRECISÃO

Para uma dada base, a **precisão** com a qual os números são representados é determinada pelo número de dígitos no significando. Por outro lado, o **intervalo de valores** de um tipo de ponto flutuante é determinado pelo maior e menor valores de expoentes suportados pelo tipo. Para cada tipo de ponto flutuante real, existem macros definidas em `<float.h>` que representam esses valores (v. **Seção 3.5**).

### 3.3.7 ORDENAÇÃO

Dois números de ponto flutuante serão considerados **ordenados** se, e somente se, um deles for menor do que, maior do que, ou igual ao outro. Se um dos dois valores não for considerado um número (i.e., se for representado como NaN), então os dois números serão considerados **desordenados**.

## 3.4 PRAGMAS PARA OPERAÇÕES DE PONTO FLUTUANTE

### 3.4.1 PRAGMA FP\_CONTRACT

A diretiva **#pragma FP\_CONTRACT**, que foi introduzida pelo padrão C99, controla o comportamento de contração de expressões de ponto flutuante. Os possíveis valores do parâmetro desta diretiva com seus respectivos efeitos são apresentados na **Tabela 3-1**.

PARÂMETRO	EFEITO
ON	O compilador pode avaliar uma expressão mais rapidamente, mas, talvez, omita erros de arredondamento e deixe de sinalizar a ocorrência de exceções de ponto flutuante (v. <b>Seção 3.7</b> ).
OFF	A contração de expressões é desabilitada. A avaliação de expressões de ponto flutuante pode ser mais lenta, mas a precisão do resultado pode ser mais confiável.
DEFAULT	Restaura o estado original de contração de expressões, que depende de implementação.

Tabela 3-1: Possíveis valores do parâmetro da diretiva `pragma FP_CONTRACT`.

Levando em consideração os valores possíveis para o único parâmetro desta diretiva **#pragma**, ela pode ser usada de três formas:

- `#pragma STD FP_CONTRACT ON`
- `#pragma STD FP_CONTRACT OFF`
- `#pragma STD FP_CONTRACT DEFAULT`

Se essa diretiva **#pragma** for utilizada dentro de um bloco, ela deverá preceder qualquer declaração ou instrução e seu efeito terá vigência até que seja invalidada por

outra diretiva do mesmo tipo, ou até o final do referido bloco. Por outro lado, se essa diretiva for utilizada fora de qualquer bloco, o efeito dela permanecerá até ser anulado por outra diretiva do mesmo tipo.

### 3.4.2 PRAGMA FENV\_ACCESS

A diretiva **#pragma FENV\_ACCESS**, introduzida pelo padrão C99, tem o objetivo de informar ao compilador se o programa controla e testa o status de arredondamento e sinalização de exceções do ambiente de ponto flutuante. Os possíveis valores do parâmetro desta diretiva com seus respectivos efeitos são apresentados na **Tabela 3-2**.

PARÂMETRO	EFEITO
ON	O programa pode utilizar as funções declaradas em <code>&lt;fenv.h&gt;</code> para controlar e testar o status do ambiente de ponto flutuante.
OFF	O programa <i>não</i> pode utilizar as funções declaradas em <code>&lt;fenv.h&gt;</code> para controle e teste do status do ambiente de ponto flutuante <sup>27</sup> .
DEFAULT	Restaura a opção original, que pode ser o mesmo que ON ou OFF, dependendo da implementação.

Tabela 3-2: Possíveis valores do parâmetro da diretiva pragma FENV\_ACCESS

Levando-se em consideração os valores possíveis do parâmetro desta diretiva pragma, ela pode ser utilizada de três formas:

- `#pragma STD FENV_ACCESS ON`
- `#pragma STD FENV_ACCESS OFF`
- `#pragma STD FENV_ACCESS DEFAULT`

Se essa diretiva pragma for utilizada dentro de um bloco, ela deverá preceder qualquer declaração ou instrução e seu efeito terá vigência até que seja invalidada por outra diretiva do mesmo tipo ou até o final do referido bloco. Por outro lado, se essa diretiva pragma for utilizada fora de qualquer bloco, o efeito dela permanecerá

---

<sup>27</sup> A existência da opção OFF pode intrigar o programador. Afinal, não é sempre bom poder testar e controlar o status do ambiente de ponto flutuante? Acontece que, utilizando-se a opção OFF, se permite ao compilador efetuar otimizações de código que lida com operações de ponto flutuante que não são possíveis quando se utiliza a opção ON.

até ser anulado por outra diretiva semelhante. Observe ainda que, quando existe uma transição da opção `OFF` para a opção `ON`, os sinalizadores de status tornam-se indeterminados e o controle do ambiente de ponto flutuante assume seu estado padrão.

**Exemplo:** Veja exemplo de uso da função `fesetenv()` (Seção 3.7.3).

### 3.5 PROPRIEDADES DE NÚMEROS DE PONTO FLUTUANTE: `<float.h>`

O cabeçalho `<float.h>` define macros que representam várias propriedades dos tipos de ponto flutuante reais primitivos de C. Nomes de macros que começam com `FLT_` representam propriedades do tipo **float**, macros que começam com `DBL_` representam propriedades do tipo **double** e aquelas que começam com `LDBL_` representam propriedades do tipo **long double**. A única exceção a esta regra de denominação é a macro `FLT_RADIX` que descreve uma propriedade relativa aos três tipos de ponto flutuante reais. A **Tabela 3-3** apresenta as macros definidas no cabeçalho `<float.h>` e suas respectivas interpretações.

MACRO	EXPANSÃO
<b>DBL_DIG</b>	Precisão em casas decimais para o tipo <b>double</b> .
<b>DBL_EPSILON</b>	O menor $X$ do tipo <b>double</b> tal que: $1.0 + X \neq 1.0$
<b>DBL_MANT_DIG</b>	O número de dígitos na mantissa (significando), na base <code>FLT_RADIX</code> , de um valor do tipo <b>double</b> .
<b>DBL_MAX</b>	O maior valor finito e representável do tipo <b>double</b> .
<b>DBL_MAX_10_EXP</b>	O maior inteiro $X$ , tal que $10^X$ é um valor finito e representável do tipo <b>double</b> .
<b>DBL_MAX_EXP</b>	O maior inteiro $X$ , tal que $\text{FLT\_RADIX}^{(X-1)}$ é um valor finito e representável do tipo <b>double</b> .
<b>DBL_MIN</b>	O menor valor normalizado, finito e representável do tipo <b>double</b> .
<b>DBL_MIN_10_EXP</b>	O menor inteiro $X$ , tal que $10^X$ é um valor normalizado, finito e representável do tipo <b>double</b> .
<b>DBL_MIN_EXP</b>	O menor inteiro $X$ , tal que $\text{FLT\_RADIX}^{(X-1)}$ é um valor normalizado, finito e representável do tipo <b>double</b> .

MACRO	EXPANSÃO
<b>DECIMAL_DIG (C99)</b>	O número mínimo de casas decimais necessárias para representar todos os dígitos significativos de um valor do tipo <b>long double</b> .
<b>FLT_DIG</b>	Precisão em casas decimais para o tipo <b>float</b> .
<b>FLT_EPSILON</b>	O menor $X$ do tipo <b>float</b> tal que: $1.0 + X \neq 1.0$
<b>FLT_EVAL_METHOD (C99)</b>	Um valor que descreve o modo de avaliação para operações de ponto flutuante. Os valores possíveis são: <ul style="list-style-type: none"> <li>• <math>-1</math>, se o modo é indeterminado.</li> <li>• Zero, se não há promoção de valores.</li> <li>• <math>1</math>, se valores do tipo <b>float</b> são promovidos a <b>double</b>.</li> <li>• <math>2</math>, se valores dos tipos <b>float</b> e <b>double</b> são promovidos a <b>long double</b>.</li> </ul>
<b>FLT_MANT_DIG</b>	O número de dígitos na mantissa (significando), na base <b>FLT_RADIX</b> , de um valor do tipo <b>float</b> .
<b>FLT_MAX</b>	O maior valor finito e representável do tipo <b>float</b> .
<b>FLT_MAX_10_EXP</b>	O maior inteiro $X$ , tal que $10^X$ é um valor finito e representável do tipo <b>float</b> .
<b>FLT_MAX_EXP</b>	O maior inteiro $X$ , tal que $\text{FLT\_RADIX}^{(X - 1)}$ é um valor finito e representável do tipo <b>float</b> .
<b>FLT_MIN</b>	O menor valor normalizado, finito e representável do tipo <b>float</b> .
<b>FLT_MIN_10_EXP</b>	O menor inteiro $X$ , tal que $10^X$ é um valor normalizado, finito e representável do tipo <b>float</b> .
<b>FLT_MIN_EXP</b>	O menor inteiro $X$ , tal que $\text{FLT\_RADIX}^{(X - 1)}$ é um valor normalizado, finito e representável do tipo <b>float</b> .
<b>FLT_RADIX</b>	A base de todas as representações de números de ponto flutuante. Tipicamente, este valor é $2$ .

MACRO	EXPANSÃO
<b>FLT_ROUNDS</b>	Um valor que descreve o modo de arredondamento para operações de ponto flutuante. Os valores possíveis são (v. <b>Seção 3.3.5</b> ): <ul style="list-style-type: none"> <li>• -1, se o modo é indeterminado.</li> <li>• Zero, se o arredondamento é com aproximação de zero.</li> <li>• 1, se o arredondamento é para o mais próximo valor representável.</li> <li>• 2, se o arredondamento é para cima.</li> <li>• 3, se o arredondamento é para baixo.</li> </ul>
<b>LDBL_DIG</b>	Precisão em casas decimais para o tipo <b>long double</b> .
<b>LDBL_EPSILON</b>	O menor $X$ do tipo <b>long double</b> tal que: $1.0 + X \neq 1.0$
<b>LDBL_MANT_DIG</b>	O número de dígitos na mantissa, na base <b>FLT_RADIX</b> , de um valor do tipo <b>long double</b> .
<b>LDBL_MAX</b>	O maior valor finito e representável do tipo <b>long double</b> .
<b>LDBL_MAX_10_EXP</b>	O maior inteiro $X$ , tal que $10^X$ é um valor finito e representável do tipo <b>long double</b> .
<b>LDBL_MAX_EXP</b>	O maior inteiro $X$ , tal que $\text{FLT\_RADIX}^{(X-1)}$ é um valor finito e representável do tipo <b>long double</b> .
<b>LDBL_MIN</b>	O menor valor normalizado, finito e representável do tipo <b>double</b> .
<b>LDBL_MIN_10_EXP</b>	O menor inteiro $X$ , tal que $10^X$ é um valor normalizado, finito e representável do tipo <b>long double</b> .
<b>LDBL_MIN_EXP</b>	O menor inteiro $X$ , tal que $\text{FLT\_RADIX}^{(X-1)}$ é um valor normalizado, finito e representável do tipo <b>long double</b> .

Tabela 3-3: Macros definidas em &lt;float.h&gt;.

**Exemplo:** Para conhecer os valores das macros definidas no arquivo <float.h>, execute o programa a seguir.

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

int main(void)
```

```

{
    printf("\nPrecisao em casas decimais para o tipo "
           "double: %d", DBL_DIG);
    printf("\nMenor X do tipo double tal que "
           "1.0 + X != 1.0: %f", DBL_EPSILON);
    printf("\nNumero de digitos na mantissa, na base "
           "FLT_RADIX, para o tipo double: %d",
           DBL_MANT_DIG);
    printf("\nMaior valor finito e representavel do "
           "tipo double: %E", DBL_MAX);
    printf("\nMaior inteiro X, tal que 10^X e' um "
           "valor finito e representavel do tipo "
           "double: %d", DBL_MAX_10_EXP);
    printf("\nMaior inteiro X, tal que FLT_RADIX^(X - 1)"
           "e' um valor finito e representavel do tipo "
           "double: %d", DBL_MAX_EXP);
    printf("\nMenor valor normalizado, finito e "
           "representavel do tipo double: %f", DBL_MIN);
    printf("\nMenor inteiro X, tal que 10^X e' um valor"
           "normalizado, finito e representavel do "
           "tipo double: %d", DBL_MIN_10_EXP);
    printf("\nMenor inteiro X, tal que FLT_RADIX^(X - 1)"
           "e' um valor normalizado, finito e "
           "representavel do tipo double: %d",
           DBL_MIN_EXP);
    printf("\nNumero minimo de casas decimais necessarias"
           "para representar todos os digitos "
           "significativos para o tipo long double: %d",
           DECIMAL_DIG); // (C99)
    printf("\nPrecisao em casas decimais para o tipo "
           "float: %d", FLT_DIG);
    printf("\nMenor X do tipo float tal que "
           "1.0 + X != 1.0: %f", FLT_EPSILON);

    printf("\nModo de avaliacao para operacoes de "
           "ponto flutuante: ");

    switch (FLT_EVAL_METHOD) { // (C99)
        case -1:
            printf("indeterminado");
            break;
        case 0:
            printf("nao ha promocao de valores");
    }
}

```

```

        break;
    case 1:
        printf("valores do tipo float sao "
               "promovidos a double");
        break;
    case 2:
        printf("valores do tipo float e double sao "
               "promovidos a long double");
        break;
}

printf("\nNumero de digitos na mantissa, na base "
       "FLT_RADIX, para o tipo float: %d",
       FLT_MANT_DIG);
printf("\nMaior valor finito e representavel do "
       "tipo float: %E", FLT_MAX);
printf("\nMaior inteiro X, tal que 10^X e' um valor"
       " finito e representavel do tipo float: %d",
       FLT_MAX_10_EXP);
printf("\nMaior inteiro X, tal que FLT_RADIX^(X - 1)"
       " e' um valor finito e representavel do tipo"
       " float: %d", FLT_MAX_EXP);
printf("\nMenor valor normalizado, finito e "
       "representavel do tipo float: %f", FLT_MIN);
printf("\nMenor inteiro X, tal que 10^X e' um valor"
       " normalizado, finito e representavel do "
       "tipo float: %d", FLT_MIN_10_EXP);
printf("\nMenor inteiro X, tal que FLT_RADIX^(X - 1)"
       " e' um valor normalizado, finito e "
       "representavel do tipo float: %d",
       FLT_MIN_EXP);
printf("\nBase de todas as representacoes de "
       "numeros de ponto flutuante: %d", FLT_RADIX);

printf("\nO modo de arredondamento para operacoes"
       " de ponto flutuante e' ");

switch (FLT_ROUNDS) {
    case -1:
        printf("indeterminado");
        break;
    case 0:
        printf("direcionado para zero");

```



```

        break;
    case 1:
        printf("para o mais proximo valor "
               "representavel");
        break;
    case 2:
        printf("direcionado para mais infinito");
        break;
    case 3:
        printf("direcionado para menos infinito");
        break;
}

printf("\nPrecisao em casas decimais para o tipo "
       "long double: %d", LDBL_DIG);
printf("\nMenor X do tipo long double tal que "
       "1.0 + X != 1.0: %Lf", LDBL_EPSILON);
printf("\nNumero de digitos na mantissa, na base "
       "FLT_RADIX, para o tipo long double: %d",
       LDBL_MANT_DIG);
printf("\nMaior valor finito e representavel do "
       "tipo long double: %LE", LDBL_MAX);
printf("\nMaior inteiro X, tal que 10^X e' um valor"
       " finito e representavel do tipo long double:"
       " %d", LDBL_MAX_10_EXP);
printf("\nMaior inteiro X, tal que FLT_RADIX^(X - 1)"
       " e' um valor finito e representavel do tipo"
       " long double: %d", LDBL_MAX_EXP);
printf("\nMenor valor normalizado, finito e "
       "representavel do tipo long double: %Lf",
       LDBL_MIN);
printf("\nMenor inteiro X, tal que 10^X e' um valor"
       " normalizado, finito e representavel do "
       "tipo long double: %d", LDBL_MIN_10_EXP);
printf("\nMenor inteiro X, tal que FLT_RADIX^(X - 1)"
       " e' um valor normalizado, finito e "
       "representavel do tipo long double: %d\n",
       LDBL_MIN_EXP);

return 0;
}

```

## 3.6 OPERAÇÕES DE PONTO FLUTUANTE REAIS: `<math.h>`

O cabeçalho `<math.h>` apresenta uma variedade de funções que representam operações matemáticas comuns sobre números de ponto flutuante reais.

Uma informação importante para aqueles que utilizam o compilador gcc é que o *linker* que acompanha este compilador não faz ligação automática com as funções declaradas em `<math.h>`. Portanto, para possibilitar a ligação entre chamadas de funções declaradas neste cabeçalho com o código compilado dessas funções, deve-se utilizar o *linker* com a opção `-lm`. Caso contrário, poderá ocorrer um erro de ligação apresentado pelo gcc como<sup>28</sup>:

```
undefined reference to <nome-da-função>
```

### 3.6.1 TIPOS

Os dois tipos definidos em `<math.h>` representam a precisão usada internamente na avaliação de expressões de ponto flutuante e têm como objetivo evitar que haja conversão implícita de alargamento de valores dos tipos **float** (conversão em **double** ou **long double**) e **double** (conversão em **long double**)<sup>29</sup>.

*double\_t* (C99)

**Incluir:** `<math.h>`

**Descrição:** O tipo **double\_t** (C99) corresponde a um dos tipos primitivos de ponto flutuante de C. O tipo correspondente depende do valor da macro **FLT\_EVAL\_METHOD** definida em `<float.h>` (v. **Seção 3.5**), de acordo com os seguintes critérios:

- Se **FLT\_EVAL\_METHOD** resultar em 0 ou 1, o tipo será **double**.
- Se **FLT\_EVAL\_METHOD** resultar em 2, o tipo será **long double**.
- Caso **FLT\_EVAL\_METHOD** não resulte em nenhum desses valores anteriores, o tipo terá, no mínimo, a capacidade do tipo **float\_t** (v. a seguir).

---

<sup>28</sup> Nem todas as chamadas de funções declaradas no cabeçalho `<math.h>` provocam esse tipo de erro quando o *linker* é invocado sem a opção `-lm`. Isto é, algumas funções são embutidas no próprio gcc e não precisam de ligação convencional.

<sup>29</sup> Consulte o **Capítulo 1** do **Volume I** para obter mais informações sobre conversões implícitas.

Usando-se este tipo em substituição a **float**, garante-se que não haverá conversão.

*float\_t* (C99)

**Incluir:** `<math.h>`

**Descrição:** Do mesmo modo que o tipo **double\_t**, o tipo **float\_t** (C99) também corresponde a um tipo primitivo de ponto flutuante de C. O tipo correspondente depende do valor da macro **FLT\_EVAL\_METHOD** definida em `<float.h>` (v. **Seção 3.5**) de acordo com os seguintes critérios:

- Se **FLT\_EVAL\_METHOD** resultar em zero, o tipo será **float**.
- Se **FLT\_EVAL\_METHOD** resultar em 1, o tipo será **double**.
- Se **FLT\_EVAL\_METHOD** resultar em 2, o tipo será **long double**.
- Se **FLT\_EVAL\_METHOD** não resultar em nenhum desses valores anteriores, o tipo terá, no máximo, a capacidade do tipo **double\_t**.

### 3.6.2 MACROS

A maioria das macros do cabeçalho `<math.h>` foi introduzida pelo padrão C99. Estas macros juntamente com suas respectivas interpretações são apresentadas a seguir.

*FP\_FAST\_FMA* (C99)

**Incluir:** `<math.h>`

**Descrição:** A macro **FP\_FAST\_FMA** (C99) é definida apenas se uma chamada da função `fma(x, y, z)` é executada tão rapidamente quanto a avaliação da expressão do tipo **double**:

`x * y + z`

*FP\_FAST\_FMAF* (C99)

**Incluir:** `<math.h>`

**Descrição:** A macro **FP\_FAST\_FMAF** (C99) é similar à macro **FP\_FAST\_FMA**, mas aplica-se à função **fmaf()** e ao tipo **float**.

*FP\_FAST\_FMAL (C99)*

**Incluir:** `<math.h>`

**Descrição:** A macro **FP\_FAST\_FMAL** (C99) é similar à macro **FP\_FAST\_FMA**, mas aplica-se à função **fmal()** e ao tipo **long double**.

*FP\_ILOGB0 (C99)*

**Incluir:** `<math.h>`

**Descrição:** A macro **FP\_ILOGB0** (C99) resulta no valor retornado por uma chamada da função `ilogb(x)`, quando  $x$  é  $+0.0$  ou  $-0.0$ . Portanto, o valor desta macro é **INT\_MIN** ou **-INT\_MAX** (v. **Seção 2.3**).

*FP\_ILOGBNAN (C99)*

**Incluir:** `<math.h>`

**Descrição:** A macro **FP\_ILOGBNAN** (C99) resulta no valor retornado por uma chamada da função `ilogb(x)`, quando  $x$  é NaN (v. **Seção 3.3.2**). Portanto, o valor desta macro é **INT\_MIN** ou **INT\_MAX** (v. **Seção 2.3**).

*FP\_INFINITE (C99)*

**Incluir:** `<math.h>`

**Descrição:** **FP\_INFINITE** (C99) é o valor resultante de uma invocação de macro `fpclassify(x)`, quando  $x$  é  $+\infty$  ou  $-\infty$ .

*FP\_NAN (C99)*

**Incluir:** `<math.h>`

**Descrição:** **FP\_NAN** (C99) é o valor resultante de uma invocação de macro `fpclassify(x)`, quando `x` é NaN.

*FP\_NORMAL (C99)*

**Incluir:** `<math.h>`

**Descrição:** **FP\_NORMAL** (C99) é o valor resultante de uma invocação de macro `fpclassify(x)`, quando `x` é finito e normalizado.

*FP\_SUBNORMAL (C99)*

**Incluir:** `<math.h>`

**Descrição:** **FP\_SUBNORMAL** (C99) é o valor resultante de uma invocação de macro `fpclassify(x)`, quando `x` é finito e não normalizado.

*FP\_ZERO (C99)*

**Incluir:** `<math.h>`

**Descrição:** **FP\_ZERO** (C99) é o valor resultante de uma invocação de macro `fpclassify(x)`, quando `x` é `+0.0` ou `-0.0`.

*HUGE\_VAL*

**Incluir:** `<math.h>`

**Descrição:** A macro **HUGE\_VAL** representa um valor grande demais para ser representado como **double**. Este valor pode ser uma representação de  $+\infty$ .

*HUGE\_VALF (C99)*

**Incluir:** `<math.h>`

**Descrição:** A macro **HUGE\_VALF** (C99) representa um valor grande demais para ser representado como **float**. Este valor pode ser uma representação de  $+\infty$ .

*HUGE\_VALL (C99)*

**Incluir:** `<math.h>`

**Descrição:** A macro **HUGE\_VALL** (C99) representa um valor grande demais para ser representado como **long double**. Este valor pode ser uma representação de  $+\infty$ .

*INFINITY*

**Incluir:** `<math.h>`

**Descrição:** A macro **INFINITY** resulta na representação do tipo **float** de  $+\infty$ .

*MATH\_ERRNO (C99)*

**Incluir:** `<math.h>`

**Descrição:** A macro **MATH\_ERRNO** (C99) resulta em 1 e é usada em conjunto com a macro **math\_errhandling** para verificar se alguma função declarada no cabeçalho `<math.h>` indica uma condição de erro armazenando um valor diferente de zero na variável global **errno**.

*MATH\_ERREXCEPT (C99)*

**Incluir:** `<math.h>`

**Descrição:** A macro **MATH\_ERREXCEPT** (C99) resulta em 2 e é usada em conjunto com a macro **math\_errhandling** para verificar se alguma função declarada no cabeçalho `<math.h>` indica uma condição de erro utilizando um sinalizador de exceção de ponto flutuante (v. **Seção 3.7**).

*math\_errhandling (C99)***Incluir:** `<math.h>`

**Descrição:** A macro **math\_errhandling** (C99) especifica como funções declaradas no cabeçalho `<math.h>` comunicam um erro de domínio ou um erro de intervalo (v. **Seção 3.3.3**). Precisamente:

- Se o resultado da operação `math_errhandling & MATH_ERRNO` for diferente de zero, qualquer função declarada em `<math.h>` comunicará um erro de domínio armazenando um valor diferente de zero na variável global **errno** e retornará um valor que caracteriza o tipo específico de erro.
- Se o resultado da operação `math_errhandling & MATH_ERREXCEPT` for diferente de zero, qualquer função declarada em `<math.h>` comunicará um erro de domínio utilizando um sinalizador de exceção de ponto flutuante (v. **Seção 3.7**). Neste caso, as macros **FE\_DIVBYZERO**, **FE\_INVALID** e **FE\_OVERFLOW** serão todas definidas (v. **Seção 3.7.2**).

O valor da macro **math\_errhandling** não é alterado durante a execução de um programa.

*NAN (C99)***Incluir:** `<math.h>`

**Descrição:** A macro **NAN** (C99) resulta na representação do tipo **float** de NaN.

*Exemplo de uso de macros definidas em <math.h>*

A seguir, é apresentado um exemplo de uso de várias macros definidas no cabeçalho `<math.h>`. A **Seção 3.6.14** apresenta exemplo de uso das macros **FP\_INFINITE**, **FP\_NAN**, **FP\_NORMAL**, **FP\_SUBNORMAL** e **FP\_ZERO**.

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

```
int main()
{
```

```

printf( "fma(x, y, z) %se' tao rapida quanto a "
        "avaliacao da expressao double: x * y + z\n",
        FP_FAST_FMA ? " " : "nao " );

printf( "Valor retornado por uma chamada da funcao "
        "ilogb(x), quando x e' +0.0 ou -0.0: %d\n",
        FP_ILOGB0 );

printf( "Valor retornado por uma chamada da funcao "
        "ilogb(x), quando x e' NaN: %d\n",
        FP_ILOGBNAN );

printf( "Representação do tipo float de +infinito:"
        " %f\n", INFINITY );

printf( "Representação do tipo float de  NAN: %f\n",
        NAN );

if (math_errhandling & MATH_ERRNO)
    printf( "As funcoes declaradas em <math.h> "
            "comunicam erros de dominio usando "
            "a variavel errno\n" );
else if (math_errhandling & MATH_ERREXCEPT)
    printf( "As funcoes declaradas em <math.h> "
            "comunicam erros de dominio usando "
            "sinalizadores de exceção\n" );
else
    printf("Esta instrucao nao deve ser executada\n");

if (cosh(DBL_MAX) == HUGE_VAL)
    printf("O resultado da operacao e' grande demais"
           " para caber num valor do tipo double\n");

return 0;
}

```

### 3.6.3 VISÃO GERAL DAS FUNÇÕES DECLARADAS EM <math.h>

A partir do padrão C99, algumas operações que antes existiam apenas para o tipo **double** passaram a ter versões correspondentes para os tipos **float** e **long double**. As funções apresentadas em cada linha da **Tabela 3-4** são consideradas equivalentes, pois



executam essencialmente as mesmas operações. A única diferença entre estas funções são os tipos de dados sobre os quais elas atuam. Assim, no restante desta seção, apenas as versões destas funções para o tipo **double** serão apresentadas, pois este conhecimento é suficiente para entender o funcionamento das versões correspondentes para **float** e **long double**.

VERSÃO PARA O TIPO double	VERSÃO PARA O TIPO float	VERSÃO PARA O TIPO long double
<b>acos()</b>	<b>acosf()</b>	<b>acosl()</b>
<b>acosh()</b>	<b>acoshf()</b>	<b>acoshl()</b>
<b>asin()</b>	<b>asinf()</b>	<b>asinl()</b>
<b>asinh()</b>	<b>asinhf()</b>	<b>asinhf()</b>
<b>atan()</b>	<b>atanf()</b>	<b>atanl()</b>
<b>atanh()</b>	<b>atanhf()</b>	<b>atanhl()</b>
<b>atan2()</b>	<b>atan2f()</b>	<b>atan2l()</b>
<b>cbrt()</b>	<b>cbrtf()</b>	<b>cbrtl()</b>
<b>ceil()</b>	<b>ceilf()</b>	<b>ceilf()</b>
<b>copysign()</b>	<b>copysignf()</b>	<b>copysignl()</b>
<b>cos()</b>	<b>cosf()</b>	<b>cosl()</b>
<b>cosh()</b>	<b>coshf()</b>	<b>coshl()</b>
<b>erf()</b>	<b>erff()</b>	<b>erfl()</b>
<b>erfc()</b>	<b>erfcf()</b>	<b>erfcl()</b>
<b>exp()</b>	<b>expf()</b>	<b>expl()</b>
<b>exp2()</b>	<b>exp2f()</b>	<b>exp2l()</b>
<b>expm1()</b>	<b>expm1f()</b>	<b>expm1l()</b>
<b>fabs()</b>	<b>fabsf()</b>	<b>fabsl()</b>
<b>fdim()</b>	<b>fdimf()</b>	<b>fdiml()</b>
<b>floor()</b>	<b>floorf()</b>	<b>floorl()</b>
<b>fma()</b>	<b>fmaf()</b>	<b>fmal()</b>
<b>fmax()</b>	<b>fmaxf()</b>	<b>fmaxl()</b>
<b>fmin()</b>	<b>fminf()</b>	<b>fminl()</b>
<b>fmod()</b>	<b>fmodf()</b>	<b>fmodl()</b>
<b>frexp()</b>	<b>frexpf()</b>	<b>frexpl()</b>
<b>hypot()</b>	<b>hypotf()</b>	<b>hypotl()</b>
<b>ilogb()</b>	<b>ilogbf()</b>	<b>ilogbl()</b>

<b>VERSÃO PARA O TIPO double</b>	<b>VERSÃO PARA O TIPO float</b>	<b>VERSÃO PARA O TIPO long double</b>
<b>ldexp()</b>	<b>ldexpf()</b>	<b>ldexpl()</b>
<b>lgamma()</b>	<b>lgammaf()</b>	<b>lgammal()</b>
<b>llrint()</b>	<b>llrintf()</b>	<b>llrintl()</b>
<b>llround()</b>	<b>llroundf()</b>	<b>llroundl()</b>
<b>log()</b>	<b>logf()</b>	<b>logl()</b>
<b>log10()</b>	<b>log10f()</b>	<b>log10l()</b>
<b>log1p()</b>	<b>log1pf()</b>	<b>log1pl()</b>
<b>log2()</b>	<b>log2f()</b>	<b>log2l()</b>
<b>logb()</b>	<b>logbf()</b>	<b>logbl()</b>
<b>lrint()</b>	<b>lrintf()</b>	<b>lrintl()</b>
<b>lround()</b>	<b>lroundf()</b>	<b>lroundl()</b>
<b>modf()</b>	<b>modff()</b>	<b>modfl()</b>
<b>nan()</b>	<b>nanf()</b>	<b>nanl()</b>
<b>nearbyint()</b>	<b>nearbyintf()</b>	<b>nearbyintl()</b>
<b>nextafter()</b>	<b>nextafterf()</b>	<b>nextafterl()</b>
<b>nexttoward()</b>	<b>nexttowardf()</b>	<b>nexttowardl()</b>
<b>pow()</b>	<b>powf()</b>	<b>powl()</b>
<b>remainder()</b>	<b>remainderf()</b>	<b>remainderl()</b>
<b>remquo()</b>	<b>remquof()</b>	<b>remquol()</b>
<b>rint()</b>	<b>rintf()</b>	<b>rintl()</b>
<b>round()</b>	<b>roundf()</b>	<b>roundl()</b>
<b>scalbln()</b>	<b>scalblnf()</b>	<b>scalblnl()</b>
<b>scalbn()</b>	<b>scalbnf()</b>	<b>scalbnl()</b>
<b>sin()</b>	<b>sinf()</b>	<b>sinl()</b>
<b>sinh()</b>	<b>sinhf()</b>	<b>sinhl()</b>
<b>sqrt()</b>	<b>sqrtf()</b>	<b>sqrtl()</b>
<b>tan()</b>	<b>tanf()</b>	<b>tanl()</b>
<b>tanh()</b>	<b>tanhf()</b>	<b>tanhl()</b>
<b>tgamma()</b>	<b>tgammaf()</b>	<b>tgammal()</b>
<b>trunc()</b>	<b>truncf()</b>	<b>truncl()</b>

Tabela 3-4: Versões de funções declaradas em &lt;math.h&gt; para double, float e long double.

### 3.6.4 FUNÇÕES TRIGONOMÉTRICAS

*acos()*

**Incluir:** `<math.h>`

**Descrição:** A função **acos()** calcula o arco cosseno do seu argumento em radianos.

**Protótipo:**

```
double acos(double umCosseno)
```

**Parâmetro:** `umCosseno` – valor do tipo **double** entre -1 e 1 que representa um cosseno.

**Retorno:** Um valor entre zero e  $\pi$  que representa o arco cosseno do argumento recebido em radianos.

**Observação:** A função atribuirá o valor **EDOM** à variável global **errno** se o valor do argumento estiver fora do intervalo  $[-1, 1]$ .

**Exemplo:** Veja o exemplo da função **atan()**.

*asin()*

**Incluir:** `<math.h>`

**Descrição:** A função **asin()** calcula o arco seno do seu argumento em radianos.

**Protótipo:**

```
double asin(double umSeno)
```

**Parâmetro:** `umSeno` – um valor do tipo **double** entre -1 até 1 que representa um seno.

**Retorno:** Um valor entre  $-\pi/2$  e  $\pi/2$  representando o arco seno do argumento recebido em radianos.

**Observação:** Se o valor do argumento estiver fora do intervalo  $[-1, 1]$ , esta função atribuirá o valor da macro **EDOM** à variável global **errno**.

**Exemplo:** Veja o exemplo da função **atan()**.

*atan()*

**Incluir:** `<math.h>`

**Descrição:** A função **atan()** calcula o arco tangente de seu argumento em radianos.

**Protótipo:**

double atan(double tan)
-------------------------

**Parâmetro:** *tan* – um valor do tipo **double** do qual se deseja calcular o arco tangente.

**Retorno:** Um valor entre  $-\pi/2$  e  $\pi/2$  representando o arco tangente de *tan* em radianos.

**Exemplo:** O programa a seguir demonstra o uso das funções **acos()**, **asin()** e **atan()**.

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

int main()
{
    printf( "Arco cosseno de %3.2f = %3.2f\n", 1.0,
            acos(1.0) );
    printf( "Arco seno de %3.2f = %3.2f\n", 1.0,
            asin(1.0) );
    printf( "Arco tangente de %3.2f (atan) = %3.2f\n",
            1.0, atan(1.0) );

    return 0;
}
```

*atan2()*

**Incluir:** `<math.h>`

**Descrição:** A função **atan2()** calcula o valor principal do arco tangente de  $\tan1/\tan2$ , onde  $\tan1$  e  $\tan2$  são os argumentos, usando os sinais desses argumentos para determinar o quadrante do valor retornado.

**Protótipo:**

```
double atan2(double tan1, double tan2)
```

Parâmetros:

- $\tan1$  – primeira tangente usada no cálculo.
- $\tan2$  – segunda tangente usada no cálculo.

**Retorno:** Valor principal do arco tangente de  $\tan1/\tan2$  no intervalo  $[-\pi, \pi]$  e em radianos.

**Observações:**

- A função **atan2()** atribuirá o valor **EDOM** à variável global **errno** se os valores de  $\tan1$  e  $\tan2$  forem ambos iguais a zero.
- Esta função é usada frequentemente para converter coordenadas retangulares em coordenadas polares, como mostra o exemplo a seguir.

**Exemplo:**

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

int main()
{
    double r, teta, x = 2.0, y = -2.0;

    r = sqrt(x*x + y*y);

    teta = atan2(y, x);

    printf( "Ponto em coordenadas retangulares (x, y): "
           "(%5.3f, %5.3f)\n", x, y );
    printf( "Ponto em coordenadas polares (r, teta): "
           "(%5.3f, %5.3f)\n", r, teta );
```

```
    return 0;  
}
```

*cos()*

**Incluir:** `<math.h>`

**Descrição:** A função **cos()** calcula o cosseno de seu argumento.

**Protótipo:**

`double cos(double arco)`

**Parâmetro:** `arco` – o arco em radianos cujo cosseno será calculado.

**Retorno:** O cosseno do argumento.

**Exemplo:** Veja o exemplo da função **tan()**.

*sin()*

**Incluir:** `<math.h>`

**Descrição:** A função **sin()** calcula o seno do argumento recebido.

**Protótipo:**

`double sin(double arco)`

**Parâmetro:** `arco` – o arco em radianos cujo seno será computado.

**Retorno:** O seno do argumento recebido.

**Exemplo:** Veja o exemplo da função **tan()**.

*tan()*

**Incluir:** `<math.h>`

**Descrição:** A função **tan()** calcula a tangente do seu argumento.

**Protótipo:**

```
double tan(double arco)
```

**Parâmetro:** `arco` – arco em radianos cuja tangente será computada.

**Retorno:** Tangente do argumento recebido.

**Exemplo:** O programa a seguir demonstra o uso das funções **cos()**, **sin()**, e **tan()**.

```
#define PI 3.141592

#include <stdio.h>
#include <math.h>

int main()
{
    double oArco = PI/2;

    printf( "Cosseno de %3.2f = %3.2f\n",
           oArco, cos(oArco) );
    printf("Seno de %3.2f = %3.2f\n", oArco, sin(oArco));
    printf("Tangente de %3.2f = %3.2f\n", PI, tan(PI));

    return 0;
}
```

### 3.6.5 FUNÇÕES HIPERBÓLICAS

*acosh()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **acosh()** (C99) calcula o arco cosseno hiperbólico do parâmetro fornecido.

**Protótipo:**

```
double acosh(double x)
```

**Parâmetro:**  $x$  – um valor do tipo **double** cujo arco cosseno hiperbólico será calculado.

**Retorno:** Arco cosseno hiperbólico de  $x$ .

**Observação:** Se  $x < 1$ , ocorrerá um erro de domínio.

**Exemplo:** Veja o exemplo de **atanh()**.

*asinh()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **asinh()** (C99) calcula o arco seno hiperbólico do parâmetro fornecido.

**Protótipo:**

```
double asinh(double x)
```

**Parâmetro:**  $x$  – um valor do tipo **double** do qual se deseja calcular o arco seno hiperbólico.

**Retorno:** Arco seno hiperbólico de  $x$ .

**Exemplo:** Veja o exemplo de **atanh()**.

*atanh()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **atanh()** (C99) calcula o arco tangente hiperbólica do parâmetro recebido.



**Protótipo:**

`double atanh(double x)`

**Parâmetro:**  $x$  – um valor do tipo **double** do qual se deseja calcular o arco tangente hiperbólica.

**Retorno:** O arco tangente hiperbólica de  $x$ .

**Observação:** Ocorrerá um erro de domínio se  $x \leq -1$  ou  $x \geq 1$ .

**Exemplo:** O programa a seguir demonstra o uso de **acosh()**, **asinh()** e **atanh()**.

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

int main()
{
    double x = 0.5, y = 1.5;

    printf( "Arco cosseno hiperbolico de %3.2f = %3.2f\n",
           y, acosh(y) );
    printf( "Arco seno hiperbolico de %3.2f = %3.2f\n",
           x, asinh(x) );
    printf( "Arco tangente hiperbolica de %3.2f = %3.2f\n",
           x, atanh(x) );

    return 0;
}
```

*cosh()*

**Incluir:** `<math.h>`

**Descrição:** A função **cosh()** calcula o cosseno hiperbólico de seu argumento.

**Protótipo:**

`double cosh(double x)`

**Parâmetro:**  $x$  – valor cujo cosseno hiperbólico será computado.

**Retorno:** O cosseno hiperbólico do argumento recebido.

**Observação:** Quando ocorre *overflow*, esta função retorna o valor **HUGE\_VAL** (com o sinal apropriado) e a variável global **errno** assume o valor **ERANGE**.

**Exemplo:** Veja o exemplo da função **tanh()**.

*sinh()*

**Incluir:** `<math.h>`

**Descrição:** A função **sinh()** calcula o seno hiperbólico de seu argumento.

**Protótipo:**

```
double sinh(double x)
```

**Parâmetro:**  $x$  – valor cujo seno hiperbólico será computado.

**Retorno:** O seno hiperbólico de  $x$ .

**Observação:** Quando ocorre *overflow*, esta função retorna o valor **HUGE\_VAL** (com o sinal apropriado) e a variável global **errno** assume o valor **ERANGE**.

**Exemplo:** Veja o exemplo da função **tanh()**.

*tanh()*

**Incluir:** `<math.h>`

**Descrição:** A função **tanh()** calcula a tangente hiperbólica do seu argumento.

**Protótipo:**

```
double tanh(double x)
```

**Parâmetro:**  $x$  – valor cuja tangente hiperbólica é desejada.

**Retorno:** A tangente hiperbólica do seu argumento.

**Exemplo:** O programa a seguir demonstra o uso das funções **cosh()**, **sinh()** e **tanh()**.

```
#define PI 3.141592

#include <stdio.h>
#include <math.h>

int main()
{
    double oArco = PI/2;

    printf( "Cosseno hiperbolico de %3.2f = %3.2f\n",
           oArco, cosh(oArco) );
    printf( "Seno hiperbolico de %3.2f = %3.2f\n",
           oArco, sinh(oArco) );
    printf( "Tangente hiperbolica de %3.2f = %3.2f\n",
           oArco, tanh(oArco) );

    return 0;
}
```

### 3.6.6 FUNÇÕES DE ARREDONDAMENTO

*ceil()*

**Incluir:** <math.h>

**Descrição:** A função **ceil()** arredonda para o menor inteiro, convertido em **double**, maior do que ou igual ao argumento recebido.

**Protótipo:**

```
double ceil(double valor)
```

**Parâmetro:** `valor` – valor que será arredondado.

**Retorno:** O menor inteiro, convertido em **double**, que não é menor do que o argumento recebido.

**Exemplo:** Veja o exemplo de **floor()**.

*floor()*

**Incluir:** `<math.h>`

**Descrição:** A função **floor()** calcula o maior inteiro, convertido em **double**, menor do que ou igual ao argumento recebido.

**Protótipo:**

```
double floor(double d)
```

**Parâmetro:** `d` – valor a ser arredondado.

**Retorno:** O maior inteiro, convertido em **double**, menor do que ou igual ao argumento recebido.

**Observação:** Compare esta função com **ceil()**.

**Exemplo:** O programa a seguir demonstra o uso das funções **ceil()** e **floor()**.

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

int main(void)
{
    double x = -2.00001;
    double y = 3.1415;

    printf( "O valor de ceil(%f) e' %f\n",
           x, ceil(x) );
    printf( "O valor de floor(%f) e' %f\n",
           y, floor(y) );

    return 0;
}
```

Quando executado, o último programa produz o seguinte resultado:

```
O valor de ceil(-2.000010) e' -2.000000
O valor de floor(3.141500) e' 3.000000
```

### *llrint()* (C99)

**Incluir:** <math.h>

**Descrição:** A função **llrint()** (C99) calcula o inteiro do tipo **long long** mais próximo do valor recebido como argumento utilizando o modo de arredondamento corrente.

**Protótipo:**

```
long long llrint(double x)
```

**Parâmetro:** *x* – valor que será arredondado.

**Retorno:** O inteiro mais próximo do valor recebido como argumento utilizando o modo de arredondamento corrente.

**Observação:** A função **llrint()** lança uma exceção de ponto flutuante se o valor arredondado é grande demais para ser representado num valor **long long**.

**Exemplo:** Veja o exemplo de **llround()**.

### *llround()* (C99)

**Incluir:** <math.h>

**Descrição:** A função **llround()** (C99) calcula o inteiro do tipo **long long** mais próximo do valor recebido como argumento arredondado com afastamento de zero, independentemente do modo de arredondamento corrente.

**Protótipo:**

```
long long llround(double x)
```

**Parâmetro:**  $x$  – valor a ser arredondado.

**Retorno:** O inteiro mais próximo do valor recebido como argumento arredondado com afastamento de zero.

**Observações:**

- Consulte a **Seção 3.3.5** para entender este modo de arredondamento.
- Compare esta função com **llrint()** e **trunc()**.

**Exemplo:** O programa a seguir demonstra o uso de **llrint()** e **llround()**.

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

int main()
{
    double x = 2.4, y = -2.55;

    printf( "O valor de %3.2f arredondado usando "
           "llrint e' %lld\n", 2.0/3.0,
           llrint(2.0/3.0) );
    printf( "O valor de %3.2f arredondado usando "
           "llround e' %lld\n", 2.0/3.0,
           llround(2.0/3.0) );

    printf( "O valor de %3.2f arredondado usando "
           "llrint e' %lld\n", y, llrint(y) );
    printf( "O valor de %3.2f arredondado usando "
           "llround e' %lld\n", y, llround(y) );

    return 0;
}
```

Quando executado, o último programa produz o seguinte resultado:

```
O valor de 0.67 arredondado usando llrint e' 1
O valor de 0.67 arredondado usando llround e' 1
O valor de -2.55 arredondado usando llrint e' -3
O valor de -2.55 arredondado usando llround e' -3
```

*lrint()* (C99)**Incluir:** <math.h>

**Descrição:** A função **lrint()** (C99) determina o inteiro do tipo **long int** mais próximo do valor recebido como argumento utilizando o modo de arredondamento corrente.

**Protótipo:**

```
long int lrint(double x)
```

**Parâmetro:** x – valor que será arredondado.

**Retorno:** O valor de x arredondado conforme a descrição da função.

**Observações:**

- Esta função lançará uma exceção de ponto flutuante se o valor arredondado for grande demais para ser representado como um valor **long int** (v. **Seção 3.7**).
- Esta função é semelhante à **llrint()**; a diferença é que **llrint()** faz aproximação para **long long**, e **lrint()** aproxima para **long int**.

**Exemplo:** Veja o exemplo de **lround()**.

*lround()* (C99)**Incluir:** <math.h>

**Descrição:** A função **lround()** (C99) determina o inteiro do tipo **long int** mais próximo do valor recebido como argumento, arredondando com afastamento de zero, independentemente do modo de arredondamento corrente.

**Protótipo:**

```
long int lround(double x)
```

**Parâmetro:** x – valor que será arredondado.

**Retorno:** O valor de *x* arredondado conforme a descrição da função.

**Observação:** Esta função é semelhante à função **llround()**; a diferença é que **llround()** faz aproximação para **long long**, e **lround()** aproxima para **long int**.

**Exemplo:** O programa a seguir demonstra o uso das funções **lrint()** e **lround()**.

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

int main()
{
    double x = 2.4, y = -2.55;

    printf( "O valor de %3.2f arredondado usando "
           "lrint e' %ld\n", x, lrint(x) );
    printf( "O valor de %3.2f arredondado usando "
           "lround e' %ld\n", x, lround(x) );

    printf( "O valor de %3.2f arredondado usando "
           "lrint e' %ld\n", y, lrint(y) );
    printf( "O valor de %3.2f arredondado usando "
           "lround e' %ld\n", y, lround(y) );

    return 0;
}
```

Quando executado, o último programa produz o seguinte resultado:

```
O valor de 2.40 arredondado usando lrint e' 2
O valor de 2.40 arredondado usando lround e' 2
O valor de -2.55 arredondado usando lrint e' -3
O valor de -2.55 arredondado usando lround e' -3
```

*nearbyint()* (C99)

**Incluir:** <math.h>

**Descrição:** A função **nearbyint()** (C99) arredonda seu argumento para o inteiro mais próximo convertido em **double** utilizando o modo de arredondamento corrente.



**Protótipo:**

```
double nearbyint(double x)
```

**Parâmetro:**  $x$  – valor do tipo **double** que será arredondado.

**Retorno:** O valor de  $x$  arredondado para o inteiro mais próximo convertido em **double** utilizando o modo de arredondamento corrente.

**Observações:**

- Compare esta função com **lrint()**, **lround()** e **rint()**.
- Diferentemente de **rint()**, **nearbyint()** não lança exceções de ponto flutuante.

**Exemplo:** Veja o exemplo de **rint()**.

*nextafter()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **nextafter()** (C99) calcula o valor representável do tipo **double** mais próximo do primeiro argumento na direção do segundo argumento.

**Protótipo:**

```
double nextafter(double x, double y)
```

**Parâmetros:**

- $x$  – valor que será arredondado.
- $y$  – valor que direciona o arredondamento.

**Retorno:**

- O valor representável do tipo **double** mais próximo de  $x$  e que sucede este parâmetro, se  $x < y$ .
- $y$ , se  $x == y$ .

- O valor representável do tipo **double** mais próximo de  $x$  e que antecede este parâmetro, se  $x > y$ .

**Observação:** Compare esta função com **nexttoward()**.

**Exemplo:** Veja o exemplo de **nexttoward()**.

*nexttoward()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **nexttoward()** (C99) calcula o valor representável do tipo **double** mais próximo do primeiro argumento na direção do segundo argumento.

**Protótipo:**

```
double nexttoward(double x, long double y)
```

**Parâmetros:**

- $x$  – valor que será arredondado.
- $y$  – valor que direciona o arredondamento.

Retorno:

- O valor representável do tipo **double** mais próximo de  $x$  e que sucede este parâmetro, se  $x < y$ .
- $y$ , se  $x == y$ .
- O valor representável do tipo **double** mais próximo de  $x$  e que antecede este parâmetro, se  $x > y$ .

**Observação:** As funções **nextafter()** e **nexttoward()** são muito parecidas. A diferença entre estas funções está no segundo argumento. Na primeira função, o segundo argumento é do tipo **double**, enquanto que, na segunda função, o segundo argumento é do tipo **long double**.

**Exemplo:** O programa a seguir demonstra o uso das funções **nextafter()** e **nexttoward()**.

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

int main()
{
    double    x = 2.9, y = 2.5, z = -2.5;

    printf( "Valor mais proximo de %3.2f usando "
            "nextafter(%3.2f, %3.2f): %3.2f\n",
            x, x, y, nextafter(x, y) );

    printf( "Valor mais proximo de %3.2f usando "
            "nextafter(%3.2f, %3.2f): %3.2f\n",
            x, x, z, nextafter(x, z) );

    y = DBL_MAX;
    z = DBL_MIN;

    printf( "Valor mais proximo de %3.2f usando "
            "nexttoward(%3.2f, %3.2E): %3.2f\n",
            x, x, y, nexttoward(x, y) );

    printf( "Valor mais proximo de %3.2f usando "
            "nexttoward(%3.2f, %3.2E): %3.2f\n",
            x, x, z, nexttoward(x, z) );

    return 0;
}
```

Quando executado, o último programa produz o seguinte resultado:

```
Valor mais proximo de 2.90 usando nextafter(2.90, 2.50): 2.90
Valor mais proximo de 2.90 usando nextafter(2.90, -2.50): 2.90
Valor mais proximo de 2.90 usando nexttoward(2.90, 1.80E+308): 2.90
Valor mais proximo de 2.90 usando nexttoward(2.90, 2.23E-
308): 2.90
```

*rint()* (C99)**Incluir:** <math.h>

**Descrição:** A função **rint()** (C99) arredonda seu argumento para o inteiro mais próximo convertido em **double** utilizando o modo de arredondamento corrente.

**Protótipo:**

double rint(double x)
-----------------------

**Parâmetro:** x – valor que será arredondado.

**Retorno:** Valor de x arredondado para o inteiro mais próximo convertido em **double** utilizando o modo de arredondamento corrente.

**Observação:** Compare esta função com **nearbyint()**.

**Exemplo:** O programa a seguir demonstra o uso de **nearbyint()** e **rint()**.

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

int main()
{
    double x = 2.4, y = -2.55;

    printf( "O valor de %3.2f arredondado usando "
           "nearbyint e' %3.2f\n", x, nearbyint(x) );
    printf( "O valor de %3.2f arredondado usando "
           "nearbyint e' %3.2f\n", y, nearbyint(y) );

    printf( "O valor de %3.2f arredondado usando "
           "rint e' %3.2f\n", x, rint(x) );
    printf( "O valor de %3.2f arredondado usando "
           "rint e' %3.2f\n", y, rint(y) );

    return 0;
}
```

Quando executado, o último programa produz o seguinte resultado:

```
O valor de 2.40 arredondado usando nearbyint e' 2.00
O valor de -2.55 arredondado usando nearbyint e' -3.00
O valor de 2.40 arredondado usando rint e' 2.00
O valor de -2.55 arredondado usando rint e' -3.00
```

### *round()* (C99)

**Incluir:** <math.h>

**Descrição:** A função **round()** (C99) arredonda seu argumento para o inteiro mais próximo convertido em **double**. Se o argumento for equidistante de dois inteiros, o arredondamento será feito para o maior deles.

**Protótipo:**

```
double round(double x)
```

**Parâmetro:** *x* – valor do tipo **double** que será arredondado.

**Retorno:** O valor do parâmetro arredondado para o inteiro mais próximo convertido em **double**.

**Observações:** Compare esta função com outras funções de arredondamento [**lrint()**, **rint()**, **nearbyint()**, etc.].

**Exemplo:**

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

int main()
{
    double x = 2.4, y = -2.55;

    printf( "O valor de %3.2f arredondado usando "
           "round e' %3.2f\n", x, round(x) );
    printf( "O valor de %3.2f arredondado usando "
           "round e' %3.2f\n", y, round(y) );

    return 0;
}
```

Quando executado, o último programa produz o seguinte resultado:

*O valor de 2.40 arredondado usando round e' 2.00*  
*O valor de -2.55 arredondado usando round e' -3.00*

*trunc() (C99)*

**Incluir:** <math.h>

**Descrição:** A função **trunc()** (C99) arredonda seu argumento para o inteiro mais próximo convertido em **double** que não seja maior do que o valor absoluto do mesmo argumento.

**Protótipo:**

```
double trunc(double x)
```

**Retorno:** O valor do parâmetro recebido arredondado de acordo com a descrição.

**Observações:**

- Este é um modo de arredondamento que é denominado **arredondamento com aproximação de zero** (v. **Seção 3.3.5**).
- Compare esta função com outras funções de arredondamento [**lrint()**, **rint()**, **nearbyint()**, etc.].

**Exemplo:**

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

int main()
{
    double x = 2.99, y = -2.9;

    printf( "O valor de %3.2f arredondado usando "
           "trunc e' %3.2f\n", x, trunc(x) );
    printf( "O valor de %3.2f arredondado usando "
           "trunc e' %3.2f\n", y, trunc(y) );
}
```

```
    return 0;
}
```

Quando executado, o último programa produz o seguinte resultado:

```
O valor de 2.99 arredondado usando trunc e' 2.00
O valor de -2.90 arredondado usando trunc e' -2.00
```

### 3.6.7 FUNÇÕES DE ERRO

Funções de erro são comumente usadas em Estatística e em Ciências. Está fora do escopo deste livro discutir o significado das duas funções de erro da biblioteca padrão de C que serão apresentadas a seguir.

*erf()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **erf()** (C99) calcula o resultado da aplicação da função erro sobre o parâmetro recebido.

**Protótipo:**

```
double erf(double x)
```

**Parâmetro:**  $x$  – valor do tipo **double** usado no cálculo da função erro.

**Retorno:** O resultado da aplicação da função erro sobre o argumento recebido.

**Observação:** Para grandes valores de  $x$ , utiliza-se a função **erfc()**.

**Exemplo:** Veja o exemplo de **erfc()**.

*erfc()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **erfc()** (C99) calcula o resultado da aplicação da função erro complementar sobre o parâmetro recebido.

**Protótipo:**

`double erfc(double x)`

**Parâmetro:** *x* – valor do tipo **double** usado no cálculo da função erro complementar.

**Retorno:** O valor resultante da aplicação da função erro complementar sobre o parâmetro recebido.

**Observação:** Veja também **erf()**.

**Exemplo:** O programa a seguir demonstra o uso das funções **erf()** e **erfc()**.

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

int main(void)
{
    double x = 1.0;

    printf("Funcao erro de %f:\t\t %f\n", x, erf(x));
    printf("Funcao erro complementar de %f:\t %f\n",
           x, erfc(x) );

    return 0;
}
```

### 3.6.8 FUNÇÕES EXPONENCIAIS E LOGARÍTMICAS

*cbrt()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **cbrt()** (C99) calcula a raiz cúbica do parâmetro recebido.



**Protótipo:**

`double cbrt(double x)`

**Parâmetro:**  $x$  – valor do tipo **double** cuja raiz cúbica é calculada.

**Retorno:** Raiz cúbica do parâmetro recebido.

**Exemplo:**

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

int main()
{
    double x = 8.0;

    printf( "A raiz cubica de %3.2f e' %3.2f\n",
           x, cbrt(x) );

    return 0;
}
```

*exp()*

**Incluir:** <math.h>

**Descrição:** A função **exp()** calcula a exponencial neperiana de seu argumento (i.e.,  $e^x$ , onde  $x$  é o argumento recebido).

**Protótipo:**

`double exp(double x)`

**Parâmetro:**  $x$  – valor cuja exponencial é calculada.

**Retorno:**  $e^x$ .

**Observação:** Se houver *overflow* no resultado, será retornado o valor da macro **HUGE\_VAL** (definida em <math.h>) e a variável global **errno** assumirá o valor da macro **ERANGE** (definida em <errno.h>).

**Exemplo:** Veja o exemplo de **expm1()**.

*exp2()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **exp2()** (C99) calcula a exponencial na base dois de seu argumento (i.e.,  $2^x$ , onde  $x$  é o valor fornecido como argumento).

**Protótipo:**

```
double exp2(double x)
```

**Parâmetro:**  $x$  – valor cuja exponencial será calculada.

**Retorno:** O valor de  $2^x$ .

**Observação:** Se houver *overflow* no resultado, será retornado o valor da macro **HUGE\_VAL** (definida em `<math.h>`) e a variável global **errno** assumirá o valor da macro **ERANGE** (definida em `<errno.h>`).

**Exemplo:** Veja o exemplo de **expm1()**.

*expm1()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **expm1()** (C99) calcula  $e^x - 1$ , onde  $x$  é o valor fornecido como argumento.

**Protótipo:**

```
double expm1(double x)
```

**Parâmetro:**  $x$  – valor do tipo **double** usado no cálculo.

**Retorno:**  $e^x - 1$ .

**Observações:**

- Uma chamada `expm1(x)` deve resultar num valor mais preciso do que o obtido com a avaliação da expressão `exp(x) - 1`, principalmente quando o valor de `x` é próximo de zero.
- Se houver *overflow* no resultado, será retornado o valor da macro **HUGE\_VAL** (definida em `<math.h>`) e a variável global **errno** assumirá o valor da macro **ERANGE** (definida em `<errno.h>`).

**Exemplo:** O programa a seguir demonstra o uso das funções **exp()**, **expm1()** e **exp2()**.

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

int main(void)
{
    double x = 5.0;

    printf( "Exponencial de %f na base 'e': %f\n",
           x, exp(x) );
    printf( "Exponencial de %f na base 'e' "
           "menos um: %f\n", x, expm1(x) );
    printf( "Exponencial de %f na base 2: %f\n",
           x, exp2(x) );

    return 0;
}
```

*frexp()*

**Incluir:** `<math.h>`

**Descrição:** A função **frexp()** separa um valor do tipo **double** em mantissa e expoente.

**Protótipo:**

`double frexp(double d, int *e)`

**Parâmetros:**

- `d` – valor do tipo **double** a ser separado em mantissa e expoente.
- `e` – ponteiro para um inteiro que receberá o expoente.

**Retorno:** A mantissa de `d`.

**Observações:**

- Esta função calcula a mantissa `m` (um valor maior do que ou igual a 0.5 e menor que 1) e o valor inteiro `n` tal que `d` (o valor original do primeiro parâmetro) é igual a  $m \cdot 2^n$ . O valor de `n` é armazenado na variável apontada por `e` (segundo parâmetro).
- Esta função executa a operação inversa daquela executada pela função **ldexp()**.
- Consulte a descrição da função **ldexp()**.

**Exemplo:**

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

int main(void)
{
    double mantissa, numero;
    int     expoente;

    printf("Digite um numero de ponto flutuante: ");
    scanf("%lf", &numero);

    mantissa = frexp(numero, &expoente);

    printf("\nO numero %f e' igual a ", numero);
    printf(" %f vezes 2 elevado a %d\n",
           mantissa, expoente );

    return 0;
}
```

*ilogb()* (C99)**Incluir:** <math.h>**Descrição:** A função **ilogb()** (C99) calcula o expoente em formato inteiro do valor recebido como argumento.**Protótipo:**

<pre>int ilogb(double x)</pre>
--------------------------------

**Parâmetro:**  $x$  – valor cujo expoente inteiro será calculado.**Retorno:**

- Se  $x = \text{NaN}$  (i.e., se  $x$  não for um número), a função retorna o valor da macro **FP\_ILOGBNAN** (v. **Seção 3.6.2**).
- Se  $x = 0$ , a função retorna o valor da macro **FP\_ILOGB0** (v. **Seção 3.6.2**).
- Se  $x = +\infty$  ou  $x = -\infty$ , a função retorna o valor da macro **INT\_MAX** (v. **Seção 2.3**).
- Para qualquer outro valor de  $x$ , a função retorna  $(\text{int}) \log_b(x)$ .

**Observações:**

- Se o argumento não for normalizado, a função retornará seu expoente normalizado.
- Consulte descrição da função **logb()**.

**Exemplo:** Veja o exemplo da função **logb()**.*ldexp()***Incluir:** <math.h>**Descrição:** A função **ldexp()** calcula o valor do tipo **double** que possui a mantissa e o expoente recebidos como argumentos. Isto é, esta função calcula  $\text{mantissa} \times 2^{\text{expoente}}$ , onde mantissa e expoente são seus parâmetros.

**Protótipo:**

```
double ldexp(double mantissa, int expoente)
```

**Parâmetros:**

- `mantissa` – a mantissa do valor a ser calculado.
- `expoente` – o expoente do valor a ser calculado.

**Retorno:** O valor do tipo **double** que possui a mantissa e o expoente recebidos como argumentos.

**Observações:**

- Esta função executa a operação inversa daquela executada pela função **frexp()**.
- Consulte a descrição da função **frexp()**.

**Exemplo:**

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

int main(void)
{
    double x = 0.785398;
    int    y = 2;

    printf("ldexp(%f, %d) = %f\n", x, y, ldexp(x,y));

    return 0;
}
```

*log()*

**Incluir:** `<math.h>`

**Descrição:** A função **log10()** calcula o logaritmo natural (base e) do valor recebido como argumento.

**Protótipo:**

```
double log(double d)
```

**Parâmetro:**  $d$  – valor do qual o logaritmo natural será calculado.

**Retorno:** O logaritmo natural de  $d$ , se  $d > 0$ ; caso contrário, um valor indefinido.

**Observações:**

- Se o argumento passado para esta função for menor do que zero, a variável global **errno** receberá o valor **EDOM** (erro de domínio).
- Se o argumento for igual a zero, a variável global **errno** receberá o valor **ERANGE**.

**Exemplo:** Veja o exemplo de **log2()**.

*log10()*

**Incluir:** `<math.h>`

**Descrição:** A função **log10()** calcula o logaritmo na base 10 do valor recebido como argumento.

**Protótipo:**

```
double log10(double d)
```

**Parâmetro:**  $d$  – valor do qual o logaritmo na base 10 será calculado.

**Retorno:** Se  $d > 0$ , o logaritmo na base 10 de  $d$ ; caso contrário, um valor indefinido.

**Observações:**

- Se o argumento passado para esta função for menor do que zero, a variável global **errno** receberá o valor **EDOM** (erro de domínio).

- Se o argumento for igual a zero, a variável global **errno** receberá o valor **ERANGE**.

**Exemplo:** Veja o exemplo de **log2()**.

*log1p()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **log1p()** (C99) calcula o logaritmo natural da soma do valor recebido como parâmetro mais um.

**Protótipo:**

```
double log1p(double x)
```

**Parâmetro:**  $x$  – valor do tipo **double** usado no cálculo.

**Retorno:** O logaritmo natural de  $x + 1$ , se  $x > -1$ ; um valor indefinido caso contrário.

**Observações:**

- O resultado obtido com o uso desta função deve ser mais preciso do que aquele obtido com o uso da função **log()**. Isto é, o resultado obtido com a chamada `log1p(x)` deve ser mais preciso do que o obtido com a chamada `log(x + 1)`, principalmente quando o valor de  $x$  é próximo de zero.
- Ocorrerá um erro de domínio se  $x \leq -1$ .
- Se  $x = -1$ , poderá ocorrer um erro de intervalo, dependendo da implementação.

**Exemplo:** Veja o exemplo de **log2()**.

*log2()* (C99)

**Incluir:** `<math.h>`



**Descrição:** A função **log2()** (C99) calcula o logaritmo na base 2 do parâmetro recebido.

**Protótipo:**

```
double log2(double x)
```

**Parâmetro:**  $x$  – valor do tipo **double** cujo logaritmo na base 2 será calculado.

**Retorno:** O logaritmo na base 2 de  $x$ .

**Observações:**

- Ocorrerá um erro de domínio se  $x < 0$ .
- Poderá ocorrer um erro de intervalo se  $x = 0$ , dependendo da implementação.

**Exemplo:** O programa a seguir demonstra o uso das funções **log()**, **log10()**, **log1p()** e **log2()**.

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

int main()
{
    double x = 2.5;

    printf( "Logaritmo na base 'e' de %3.2f: %f\n",
           x, log(x) );
    printf( "Logaritmo na base 10 de %3.2f: %f\n",
           x, log10(x) );
    printf( "Logaritmo na base 'e' de %3.2f + 1: %f\n",
           x, log1p(x) );
    printf( "Logaritmo na base 2 de %3.2f: %f\n",
           x, log2(x) );

    return 0;
}
```

*logb()* (C99)**Incluir:** `<math.h>`**Descrição:** A função **logb()** (C99) calcula o expoente do seu argumento.**Protótipo:**

<pre>double logb(double x)</pre>
----------------------------------

**Parâmetro:** `x` – valor cujo expoente será calculado.**Retorno:** O valor do expoente de seu argumento. Se o argumento não for normalizado, seu expoente normalizado será retornado.**Observação:** Ocorrerá um erro de domínio se `x = 0`.**Exemplo:** O programa a seguir demonstra o uso das funções **logb()** e **ilogb()**.

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

int main(void)
{
    double x = 0.0, y = INFINITY, z = NAN, w = 3.5;

    printf("\n**** logb ****\n");
    printf("\tlogb(%f) = %f\n", x, logb(x));
    printf("\tlogb(%f) = %f\n", y, logb(y));
    printf("\tlogb(%f) = %f\n", z, logb(z));
    printf("\tlogb(%f) = %f\n", w, logb(w));

    printf("\n\n**** ilogb ****\n");
    printf("\tilogb(%f) = %d\n", x, ilogb(x));
    printf("\tilogb(%f) = %d\n", y, ilogb(y));
    printf("\tilogb(%f) = %d\n", z, ilogb(z));
    printf("\tilogb(%f) = %d\n", w, ilogb(w));

    return 0;
}
```

Resultado do programa quando compilado e executado no Windows:

```
**** logb ****
    logb(0.000000) = -1.#INF00
    logb(1.#INF00) = 1.#INF00
    logb(-1.#IND00) = -1.#IND00
    logb(3.500000) = 1.000000

**** ilogb ****
    ilogb(0.000000) = -2147483648
    ilogb(1.#INF00) = 2147483647
    ilogb(-1.#IND00) = -2147483648
    ilogb(3.500000) = 1
```

*pow()*

**Incluir:** <math.h>

**Descrição:** A função **pow()** calcula o valor do primeiro parâmetro elevado ao segundo.

**Protótipo:**

```
double pow(double x, double y)
```

**Parâmetros:**

- *x* – número a ser elevado à potência *y*.
- *y* – potência a qual *x* será elevado.

**Retorno:** O valor de *x* elevado à potência *y*.

**Observações:**

- Quando o resultado da operação produz *overflow*, a função retorna o valor **HUGE\_VAL** e a variável global **errno** recebe o valor **ERANGE**.
- Se o argumento *x* for menor do que zero e *y* for um número com parte fracionária não nula, ou for feita a chamada `pow(0, 0)`, a variável global **errno** receberá o valor **EDOM**.
- Consulte também **exp()**.

**Exemplo:**

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

int main(void)
{
    printf("3 elevado a 5 = %5.3f\n", pow(3.0, 5.0));

    return 0;
}
```

*scalbln()* (C99)**Incluir:** <math.h>

**Descrição:** A função **scalbln()** (C99) calcula o resultado da multiplicação de um valor do tipo **double** por uma potência inteira da constante **FLT\_RADIX** (v. **Seção 3.5**). Isto é, esta função calcula o valor da expressão  $x * \text{FLT\_RADIX}^{\text{expoente}}$ , onde  $x$  e  $\text{expoente}$  são os argumentos recebidos pela função.

**Protótipo:**

```
double scalbln(double x, long int expoente)
```

**Parâmetros:**

- $x$  – valor do tipo **double** usado no cálculo.
- $\text{expoente}$  – inteiro do tipo **long int** usado no cálculo.

**Retorno:**  $x * \text{FLT\_RADIX}^{\text{expoente}}$ .

**Observações:**

- O cálculo efetuado por esta função é mais eficiente do seria um cálculo equivalente usando operadores.
- Usualmente, **FLT\_RADIX** vale 2, de modo que, neste caso, esta função retorna:  $x * 2^{\text{expoente}}$ , que é o mesmo valor retornado por **ldexp()**.
- Compare esta função com **scalbn()**.
- Consulte também **ldexp()**.

**Exemplo:**

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

int main()
{
    double x = -3.4;
    long   expoente = 2L;

    printf( "O valor de scalbln(%3.2f, %d) e' %f\n",
           x, expoente, scalbln(x, expoente) );

    return 0;
}
```

*scalbn() (C99)***Incluir:** <math.h>

**Descrição:** A função **scalbn()** (C99) calcula o resultado da multiplicação de um valor do tipo **double** por uma potência inteira da constante **FLT\_RADIX** (v. **Seção 3.5**). Isto é, esta função calcula o valor da expressão  $x * FLT\_RADIX^{\text{expoente}}$ , onde  $x$  e  $\text{expoente}$  são os argumentos recebidos pela função.

**Protótipo:**

```
double scalbn(double x, int expoente)
```

**Parâmetros:**

- $x$  – valor do tipo **double** usado no cálculo.
- $\text{expoente}$  – inteiro do tipo **long int** usado no cálculo.

**Retorno:**  $x * FLT\_RADIX^{\text{expoente}}$ .**Observações:**

- Compare esta função com **scalbln()** e note que as duas funções diferem apenas no tipo do segundo argumento.
- Não deixe de consultar a descrição da função **scalbln()**.

**Exemplo:**

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

int main()
{
    double x = -3.4;
    int     expoente = 2;

    printf( "O valor de scalbn(%3.2f, %d) e' %f\n",
           x, expoente, scalbn(x, expoente) );

    return 0;
}
```

*sqrt()***Incluir:** <math.h>**Descrição:** A função **sqrt()** calcula a raiz quadrada positiva de seu argumento.**Protótipo:**

double sqrt(double d)
-----------------------

**Parâmetro:** d – valor cuja raiz quadrada será calculada.**Retorno:** A raiz quadrada positiva do argumento recebido.**Observação:** Se o argumento for negativo, a variável global **errno** recebe o valor **EDOM** (erro de domínio).**Exemplo:**

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

int main(void)
{
    return !printf("sqrt(2) = %.4f\n", sqrt(2));
}
```

### 3.6.9 FUNÇÕES DE COMPARAÇÃO

*fdim()* (C99)

**Incluir:** <math.h>

**Descrição:** A função **fdim()** (C99) compara  $x - y$  com zero e retorna o maior valor.

**Protótipo:**

```
double fdim(double x, double y)
```

**Parâmetros:**  $x$  e  $y$  – valores do tipo **double** que serão comparados.

**Retorno:** O maior valor entre  $x - y$  e zero.

**Exemplo:**

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

int main()
{
    double x = -3.2, y = -1.4;

    printf( "%f - %f %s maior do que 0\n", x, y,
            fdim(x, y) ? "e'" : "nao e'" );

    return 0;
}
```

*fmax()* (C99)

**Incluir:** <math.h>

**Descrição:** A função **fmax()** (C99) calcula o maior (i.e., o mais positivo) dentre os dois valores recebidos como argumentos.

**Protótipo:**

```
double fmax(double x, double y)
```

**Parâmetros:**  $x$  e  $y$  – valores do tipo **double** que serão comparados.

**Retorno:** Parâmetro de maior valor entre os dois parâmetros recebidos.

**Observação:** Compare esta função com **fmin()**.

**Exemplo:** Veja o exemplo de **fmin()**.

*fmin()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **fmin()** (C99) calcula o menor (i.e., o mais negativo) dentre os dois valores recebidos como argumentos.

**Protótipo:**

```
double fmin(double x, double y)
```

**Parâmetros:**  $x$  e  $y$  – valores do tipo **double** que serão comparados.

**Retorno:** O parâmetro de menor valor entre os dois parâmetros recebidos.

**Exemplo:**

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

int main()
{
    double x = 3.2, y = -1.4;

    printf("O maior valor entre %3.2f e %3.2f e' %3.2f\n",
           x, y, fmax(x, y));
```



```
printf("O menor valor entre %3.2f e %3.2f e' %3.2f\n",
      x, y, fmin(x, y));

return 0;
}
```

### 3.6.10 FUNÇÕES GAMA

A função gama é uma extensão da função fatorial para números de ponto flutuante e é usada comumente em Física Matemática. Como a função gama cresce muito rapidamente, muitas vezes, utiliza-se o logaritmo natural desta função no lugar dela. Esta seção apresenta duas funções declaradas em `<math.h>` que implementam essas operações.

*lgamma()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função **lgamma()** (C99) calcula o logaritmo natural do valor absoluto do resultado da aplicação da função gama sobre o argumento recebido.

**Protótipo:**

```
double lgamma(double x)
```

**Parâmetro:** `x` – valor do tipo **double** utilizado no cálculo.

**Retorno:** O logaritmo natural do valor absoluto do resultado da aplicação da função gama sobre o argumento recebido.

**Observação:** Ocorrerá um erro de domínio se  $x < 0$ .

**Exemplo:**

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

int main()
{
```

```

double x = -2.4;

printf( "O valor de lgamma(%3.2f) e' %f\n",
        x, lgamma(x) );

return 0;
}

```

### *tgamma()* (C99)

**Incluir:** <math.h>

**Descrição:** A função **tgamma()** (C99) calcula o valor resultante da aplicação da função gama sobre o argumento recebido.

**Protótipo:**

double tgamma(double x)
-------------------------

**Parâmetro:** x – valor do tipo **double** usado no cálculo.

**Retorno:** Valor resultante da aplicação da função gama sobre o argumento recebido.

**Observação:** Ocorrerá um erro de domínio se  $x < 0$ .

### **Exemplo:**

```

#include <stdio.h>
#include <math.h>

int main()
{
    double x = 3.14;

    printf( "A funcao gama de %3.2f e' %3.2f\n",
            x, tgamma(x) );

    return 0;
}

```

### 3.6.11 FUNÇÕES DE DIVISÃO

*fmod()*

**Incluir:** <math.h>

**Descrição:** A função **fmod()** calcula o resto da divisão de ponto flutuante do primeiro argumento pelo segundo argumento recebidos. Isto é, esta função calcula  $x \bmod y$ , onde  $x$  é o primeiro argumento e  $y$  é o segundo.

**Protótipo:**

```
double fmod(double x, double y)
```

**Parâmetros:**

- $x$  – valor a ser dividido.
- $y$  – divisor.

**Retorno:**  $x \bmod y$ , se  $x \neq 0$ ; zero, caso contrário.

**Observações:**

- Esta função calcula e retorna o valor de  $r$ , onde  $r = x - a*y$ , sendo  $a$  o maior quociente inteiro da divisão de  $x$  por  $y$  e tal que  $|a| \leq y$ .
- Se  $x$  e  $y$  têm sinais opostos, o valor retornado tem o mesmo sinal de  $x$ .

**Exemplo:**

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

int main(void)
{
    double x = 3.14159, y = 3.0;

    printf( "fmod(%3.5f, %3.2f) = %3.5f\n",
           x, y, fmod(x,y) );

    return 0;
}
```

*remainder()* (C99)**Incluir:** `<math.h>`

**Descrição:** A função **remainder()** (C99) calcula o resto da divisão do primeiro parâmetro pelo segundo parâmetro usando o padrão IEEE.

**Protótipo:**

```
double remainder(double x, double y)
```

**Parâmetros:**

- *x* – numerador da divisão.
- *y* – denominador da divisão.

**Retorno:** Resto da divisão do primeiro parâmetro pelo segundo parâmetro.

**Observação:** Esta função faz uma chamada `remquo(x, y, &tmp)` (v. a seguir), onde `tmp` é uma variável local à função **remainder()**.

**Exemplo:** Veja o exemplo de **remquo()**.

*remquo()* (C99)**Incluir:** `<math.h>`

**Descrição:** A função **remquo()** (C99) calcula o resto da divisão do primeiro parâmetro pelo segundo parâmetro usando o padrão IEEE e armazena o quociente da operação na variável apontada pelo terceiro parâmetro.

**Protótipo:**

```
double remquo(double x, double y, int *quociente)
```

**Parâmetros:**

- *x* – numerador da divisão.

- $y$  – denominador da divisão.
- `quociente` – endereço onde será armazenado o quociente de  $x$  por  $y$ .

**Retorno:** Resto da divisão do primeiro parâmetro pelo segundo parâmetro.

**Observações:**

- Esta função é semelhante a **`remainder()`**; a principal diferença é que **`remquo()`** armazena o quociente da divisão no terceiro parâmetro.
- Ocorrerá um erro de domínio se  $y$  for igual a 0.
- Consulte a descrição da função **`remainder()`**.

**Exemplo:** O programa a seguir demonstra o uso das funções **`remainder()`** e **`remquo()`**.

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

int main()
{
    double  x = 5.0, y = 3.0;
    int      quociente;

    printf( "Resto da divisao de %f por %f usando "
           "remainder: %f\n", x, y, remainder(x, y) );

    printf( "Resto da divisao de %f por %f "
           "usando remquo: %f\n", x, y,
           remquo(x, y, &quociente) );

    printf( "Quociente da divisao de %f por %f "
           "usando remquo: %d\n", x, y, quociente );

    return 0;
}
```

### 3.6.12 OUTRAS FUNÇÕES DECLARADAS EM <math.h>

Esta seção apresenta funções declaradas no cabeçalho <math.h> que não se encaixam em nenhuma das categorias anteriormente descritas.

*copysign()* (C99)

**Incluir:** <math.h>

**Descrição:** A função **copysign()** (C99) retorna um valor com a magnitude do primeiro argumento e o sinal do segundo argumento.

**Protótipo:**

```
double copysign(double x, double y)
```

**Parâmetro:**

- *x* – valor que provê a magnitude do resultado.
- *y* – valor que provê o sinal do resultado.

**Retorno:** A magnitude do primeiro argumento com o sinal do segundo argumento recebidos.

**Exemplo:**

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

int main()
{
    double x = -3.14, y = 2.5, z = -4.8;

    printf( "Valores originais: x = %3.2f, y = %3.2f\n",
           x, y );
    printf( "Valor de x com sinal de y = %3.2f\n",
           copysign(x, y) );

    printf( "\n\nValores originais: x = %3.2f, "
           "z = %3.2f\n", x, z );
```

```

printf( "Valor de x com sinal de z = %3.2f\n",
        copysign(x, z) );

return 0;
}

```

### *fabs()*

**Incluir:** <math.h>

**Descrição:** A função **fabs()** calcula o valor absoluto de um número do tipo **double**.

**Protótipo:**

```
double fabs(double d)
```

**Parâmetro:** d – valor do tipo **double** cujo valor absoluto é desejado.

**Retorno:** Valor absoluto do parâmetro recebido.

### **Exemplo:**

```

#include <stdio.h>
#include <math.h>

int main(void)
{
    return !printf("fabs(-0.2) = %3.2f\n", fabs(-0.2));
}

```

### *fma()* (C99)

**Incluir:** <math.h>

**Descrição:** A função **fma()** (C99) calcula e arredonda o valor da expressão  $x * y + z$ , onde x, y e z, todos valores do tipo **double**, são seus argumentos.

**Protótipo:**

```
double fma(double x, double y, double z)
```

**Parâmetros:** *x*, *y*, *z* – primeiro, segundo e terceiro operandos da expressão FMA.

**Retorno:**  $x * y + z$  (FMA).

**Observações:**

- Em Computação, FMA<sup>30</sup> é uma operação importante que calcula uma multiplicação e uma soma com arredondamento simples. Isto é, matematicamente, tem-se:

$$\text{FMA}(x, y, z) = x * y + z$$

- O uso da função **fma()** evita a ocorrência de erros de arredondamento ou *overflow* que poderiam surgir na avaliação da expressão equivalente.
- Quando esta operação é implementada em hardware, ela é mais rápida do que a execução das respectivas operações separadamente.

**Exemplo:**

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

int main()
{
    double x = 3.2, y = -1.4, z = 7.23;

    printf( "FMA de %3.2f, %3.2f e %3.2f: %3.2f\n",
           x, y, z, fma(x, y, z) );

    return 0;
}
```

*hypot()* (C99)

**Incluir:** `<math.h>`

---

<sup>30</sup> FMA significa *Fused Multiply-Add*, em inglês.



**Descrição:** A função **hypot()** (C99) retorna a raiz quadrada de  $x^2 + y^2$ , onde  $x$  e  $y$  são os argumentos recebidos pela função.

**Protótipo:**

```
double hypot(double x, double y)
```

**Parâmetros:**  $x$  e  $y$  – representam catetos de um triângulo retângulo.

**Exemplo:**

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

int main()
{
    double base = 3.0, altura = 4.0;

    printf( "A hipotenusa do triangulo com base = %3.2f"
           " e altura = %3.2f e' %3.2f\n",
           base, altura, hypot(base, altura) );

    return 0;
}
```

*modf()*

**Incluir:** <math.h>

**Descrição:** A função **modf()** calcula as partes inteira e fracionária de um valor do tipo **double**.

**Protótipo:**

```
double modf(double numero, double *inteira)
```

**Parâmetros:**

- **numero** – número de ponto flutuante do qual as partes inteira e fracionária serão encontradas.

- `inteira` – ponteiro para uma variável do tipo **double** que recebe a parte inteira do primeiro argumento.

**Retorno:** A parte fracionária do primeiro argumento.

**Observação:** A parte inteira do primeiro argumento é armazenada na variável cujo endereço é passado como segundo argumento.

### Exemplo:

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

int main(void)
{
    double inteira, frac, numero = 4.1234;

    frac = modf(numero, &inteira);

    printf("Parte fracionaria de %3.4f = %3.4f\n",
           numero, frac);
    printf("Parte inteira de %3.4f = %3.4f\n",
           numero, inteira);

    return 0;
}
```

### *nan()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A função `nan()` (C99) converte um *string* num valor do tipo **double** que representa NaN (v. **Seção 3.3.2**).

### Protótipo:

```
double nan(const char *str)
```

**Parâmetro:** `x` – *string* representando NaN.

**Retorno:** Valor do tipo **double** que representa NaN.

**Exemplo:**

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

int main()
{
    printf( "O resultado de 'nan(\"NAN\")' %s e' NaN\n",
            isnan(nan("32")) ? "" : "NAO" );

    return 0;
}
```

*signbit()* (C99)**Incluir:** <math.h>

**Descrição:** *signbit()* (C99) é uma macro que verifica se o bit de sinal de um valor de ponto flutuante (**float**, **double** ou **long double**) está ligado.

**Protótipo:**

int signbit(x)
----------------

**Parâmetro:** x – valor de qualquer tipo de ponto flutuante cujo sinal será verificado.

**Retorno:** Um valor diferente de zero se o bit de sinal do parâmetro fornecido estiver ligado; zero, caso contrário.

**Exemplo:**

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

int main()
{
    double x = -3.4, y = 7.44;

    printf( "O bit de sinal de %3.2f esta' %s\n",
            x, signbit(x) ? "ligado" : "desligado");
}
```

```

printf( "O bit de sinal de %3.2f esta' %s\n",
        y, signbit(y) ? "ligado" : "desligado");

return 0;
}

```

### 3.6.13 MACROS DE CLASSIFICAÇÃO

Esta seção descreve as macros de classificação de números de ponto flutuante preconizadas pelo padrão C99.

*fpclassify()* (C99)

**Incluir:** `<math.h>`

**Descrição:** `fpclassify()` (C99) é uma macro que classifica números de ponto flutuante.

**Protótipo:**

`int fpclassify(x)`

**Parâmetro:** `x` – valor de qualquer tipo de ponto flutuante que será classificado.

**Retorno** (constantes inteiras):

- **FP\_INFINITE**, se  $x = +\infty$  ou  $x = -\infty$ .
- **FP\_NAN**, se  $x = \text{NaN}$  (i.e., se  $x$  não for um número).
- **FP\_NORMAL**, se  $x$  for finito e normalizado.
- **FP\_SUBNORMAL**, se  $x$  for finito, mas não for normalizado.
- **FP\_ZERO**, se  $x = 0$ .

**Exemplo:**

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

```

#include <math.h>

int main()
{
    double x = 0.0, y = -2.0, z;
    int    classificacao;

    z = pow(x,y);

    classificacao = fpclassify(z);

    printf( "\nO resultado da avaliacao de "
           "pow(%3.1f, %3.1f) e' ", x, y );

    switch(classificacao) {
        case FP_INFINITE:
            printf("infinito");
            break;
        case FP_NAN:
            printf("NaN");
            break;
        case FP_NORMAL:
            printf("finito e normalizado");
            break;
        case FP_SUBNORMAL:
            printf("finito e nao normalizado");
            break;
        case FP_ZERO:
            printf("zero");
            break;
        default:
            printf("desconhecido");
    }

    putchar('\n');

    return 0;
}

```

Resultado da execução do programa quando compilado e executado no Windows:

O resultado da avaliação de `pow(0.0, -2.0)` é infinito

*isfinite()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A macro **isfinite()** (C99) verifica se um valor de ponto flutuante (**float**, **double** ou **long double**) é finito.

**Protótipo:**

```
int isfinite(x)
```

**Parâmetro:** `x` – valor de qualquer tipo de ponto flutuante que se deseja testar.

**Retorno:** Um valor diferente de zero se o parâmetro fornecido for finito; zero, caso contrário.

**Exemplo:** Veja o exemplo de **isunordered()**.

*isgreater()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A macro **isgreater()** (C99) verifica se um valor é maior do que outro; pelo menos um dos valores comparados deve ser de ponto flutuante.

**Protótipo:**

```
int isgreater(x, y)
```

**Parâmetros:** `x` e `y` – valores que serão testados; pelo menos um deles deve ser de algum tipo de ponto flutuante.

**Retorno:** 1, quando `x > y` e nem `x` nem `y` é NaN; zero, caso contrário.

**Observação:** O resultado obtido invocando esta macro como `isgreater(x, y)` é equivalente àquele obtido avaliando a expressão `x > y`, exceto pelo fato de a macro não causar o lançamento de exceção quando um dos valores envolvidos for NaN.

**Exemplo:** Veja o exemplo de `isunordered()`.

*isgreaterqual()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A macro `isgreaterqual()` (C99) verifica se um valor é maior do que ou igual ao outro; pelo menos um dos valores comparados deve ser de ponto flutuante.

**Protótipo:**

```
int isgreaterqual(x, y)
```

**Parâmetros:**  $x$  e  $y$  – valores que serão testados; pelo menos um deles deve ser de algum tipo de ponto flutuante.

**Retorno:** 1, quando  $x \geq y$  e nem  $x$  nem  $y$  é NaN; zero, caso contrário.

**Observação:** O resultado obtido com a invocação `isgreaterqual(x, y)` é equivalente àquele obtido avaliando a expressão  $x \geq y$ , exceto pelo fato de a macro não causar o lançamento de exceção quando um dos valores envolvidos for NaN.

**Exemplo:** Veja o exemplo de `isunordered()`.

*isinf()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A macro `isinf()` (C99) verifica se o parâmetro de ponto flutuante é  $+\infty$  ou  $-\infty$ .

**Protótipo:**

```
int isinf(x)
```

**Parâmetro:**  $x$  – valor de qualquer tipo de ponto flutuante.

**Retorno:** Um valor diferente de zero se o parâmetro fornecido for  $+\infty$  ou  $-\infty$ ; zero, caso contrário.

**Exemplo:** Veja o exemplo de `isunordered()`.

*isless()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A macro `isless()` (C99) verifica se um valor é menor do que ou igual a outro; pelo menos um dos valores comparados deve ser de ponto flutuante.

**Protótipo:**

```
int isless(x, y)
```

**Parâmetros:** `x` e `y` – valores que serão testados; pelo menos um deles deve ser de algum tipo de ponto flutuante.

**Retorno:** 1, quando  $x < y$  e nem `x` nem `y` é NaN; zero, caso contrário.

**Observação:** O resultado obtido invocando esta macro como `isless(x, y)` é equivalente àquele obtido avaliando a expressão  $x < y$ , exceto pelo fato de a macro não causar o lançamento de exceção quando um dos valores envolvidos for NaN.

**Exemplo:** Veja o exemplo de `isunordered()`.

*islessequal()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A macro `islessequal()` (C99) verifica se um valor é menor do que ou igual a outro; pelo menos um dos valores comparados deve ser de ponto flutuante.



**Protótipo:**

```
int islessequal(x, y)
```

**Parâmetros:**  $x$  e  $y$  – valores que serão testados; pelo menos um deles deve ser de algum tipo de ponto flutuante.

**Retorno:** 1, quando  $x \leq y$  e nem  $x$  nem  $y$  é NaN; zero, caso contrário.

**Observação:** O resultado obtido invocando esta macro como `islessequal(x, y)` é equivalente àquele obtido avaliando a expressão  $x \leq y$ , exceto pelo fato de a macro não causar o lançamento de exceção quando um dos valores envolvidos for NaN.

**Exemplo:** Veja o exemplo de `isunordered()`.

*islessgreater() (C99)*

**Incluir:** `<math.h>`

**Descrição:** A macro `islessgreater()` (C99) verifica se o primeiro parâmetro é menor do que o segundo, ou vice-versa; pelo menos um dos valores comparados deve ser de ponto flutuante.

**Protótipo:**

```
int islessgreater(x, y)
```

**Parâmetros:**  $x$  e  $y$  – valores que serão testados; pelo menos um deles deve ser de algum tipo de ponto flutuante.

**Retorno:** 1, quando  $x < y$  ou  $x > y$  e nem  $x$  nem  $y$  é NaN; zero, caso contrário.

**Observação:** O resultado obtido invocando esta macro como `islessgreater(x, y)` é equivalente àquele obtido avaliando a expressão  $x < y \ || \ x > y$ , exceto pelo fato de a macro não causar o lançamento de exceção quando um dos valores envolvidos for NaN.

**Exemplo:** Veja o exemplo de `isunordered()`.

*isnan()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A macro `isnan()` (C99) verifica se o argumento é NaN (v. **Seção 3.3.2**).

**Protótipo:**

```
int isnan(x)
```

**Parâmetro:** `x` – valor de qualquer tipo de ponto flutuante.

**Retorno:** Um valor diferente de zero se o parâmetro fornecido for NaN; zero, caso contrário.

**Exemplo:** Veja o exemplo de `isunordered()`.

*isnormal()* (C99)

**Incluir:** `<math.h>`

**Descrição:** A macro `isnormal()` (C99) verifica se o argumento é finito e normalizado.

**Protótipo:**

```
int isnormal(x)
```

**Parâmetro:** `x` – valor de qualquer tipo de ponto flutuante.

**Retorno:** Um valor diferente de zero se o parâmetro fornecido for finito e normalizado; zero, caso contrário.

**Exemplo:** Veja o exemplo de `isunordered()`.

*isunordered()* (C99)**Incluir:** <math.h>**Descrição:** A macro **isunordered()** (C99) verifica se algum dos argumentos é NaN; pelo menos um dos valores comparados deve ser de ponto flutuante.**Protótipo:**

<pre>int isunordered(x, y)</pre>
----------------------------------

**Parâmetros:** *x* e *y* – valores que serão testados; pelo menos um deles deve ser de algum tipo de ponto flutuante.**Retorno:** 1, quando *x* é NaN ou *y* é NaN; zero, caso contrário.**Observação:** A macro **isunordered()** testa se existe alguma relação de ordem entre dois valores de ponto flutuante. Ela resultará num valor diferente de zero se os números de ponto flutuante que ela recebe como argumentos forem desordenados ou zero caso contrário (v. **Seção 3.3.7**). (Esta descrição é equivalente àquela já apresentada.)**Exemplo:** O programa a seguir demonstra o uso das macros de classificação definidas em <math.h>.

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

int main()
{
    double x = 0, y = -2.0, z;

    z = pow(x, y);

    printf( "\nO resultado da avaliacao de "
           "pow(%3.1f, %3.1f) e' ", x, y );

    if (isfinite(z))
        printf("finito");
    else if (isinf(z))
```

```

        printf("infinito");
    else if (isnan(z))
        printf("NaN");
    else if (isnormal(z))
        printf("finito e normalizado");

    if (isnan(sqrt(-2)))
        printf( "\n0 resultado da avaliacao de "
                "sqrt(-2) e' NaN" );
    else
        printf( "\n0 resultado da avaliacao de "
                "sqrt(-2) NAO e' NaN" );

    if (isgreater(x, y))
        printf("\n%4.2f e' maior do que %4.2f", x, y);

    if (isgreaterequal(x, y))
        printf( "\n%4.2f e' maior do que ou igual a "
                "%4.2f", x, y );

    if (isless(x, y))
        printf("\n%4.2f e' menor do que %4.2f", x, y);

    if (islessequal(x, y))
        printf( "\n%4.2f e' menor do que ou igual a "
                "%4.2f", x, y );

    if (islessgreater(x, y))
        printf( "\nOu %4.2f e' maior do que %4.2f "
                "ou %4.2f e' maior do que %4.2f",
                x, y, y, x );

    if (isunordered(x, y))
        printf("\nOu %4.2f e' NaN ou %4.2f e' NaN", x, y);
    else
        printf("\nNem %4.2f nem %4.2f e' NaN", x, y);

    putchar('\n');

    return 0;
}

```

## 3.7 TRATAMENTO DE EXCEÇÕES E ARREDONDAMENTO: <fenv.h> (C99)

O cabeçalho <fenv.h> contém tipos, macros e funções que permitem testar e controlar exceções e modos de arredondamento de números de ponto flutuante.

### 3.7.1 TIPOS

*fenv\_t*

**Incluir:** <fenv.h>

**Descrição:** Uma variável do tipo **fenv\_t** é capaz de armazenar toda a configuração de ambiente de ponto flutuante. Esta configuração contém todos os sinalizadores de status e modos de controle (e.g., arredondamento) que a implementação dá suporte.

*fexcept\_t*

**Incluir:** <fenv.h>

**Descrição:** Uma variável do tipo **fexcept\_t** é capaz de representar todos os sinalizadores de exceções de ponto flutuante. Um sinalizador de exceção de ponto flutuante é um bit que é ligado quando ocorre uma condição de exceção em consequência de uma operação de ponto flutuante (v. **Seção 3.3.4**).

**Observação:** De acordo com o padrão C99, este tipo não precisa necessariamente ser inteiro. Em algumas implementações, uma variável deste tipo deve conter apenas sinalizadores; assim, neste caso, este tipo pode ser inteiro. Mas, outras implementações mais sofisticadas podem decidir armazenar não apenas sinalizadores como também informações adicionais sobre as respectivas exceções (e.g., o endereço da instrução que causou uma dada condição de exceção). Neste último caso, o tipo **fexcept\_t** não pode ser simplesmente definido como um tipo inteiro.

### 3.7.2 MACROS

A única macro requerida pelo padrão C99 é **FE\_DFL\_ENV**, mas este padrão sugere que implementações de C incorporem outras macros. Essas macros serão apresentadas a seguir.

*FE\_ALL\_EXCEPT***Incluir:** <fenv.h>

**Descrição:** A macro **FE\_ALL\_EXCEPT** resulta num valor inteiro positivo que representa a combinação por meio do operador | de todas as macros sinalizadoras de exceção suportadas pela implementação.

**Observação:** Esta macro será definida apenas se as funções declaradas em <fenv.h> puderem controlar exceções de ponto flutuante.

*FE\_DFL\_ENV***Incluir:** <fenv.h>

**Descrição:** A macro **FE\_DFL\_ENV** é expandida num ponteiro para uma variável do tipo **fenv\_t** que representa a configuração para controle de processamento de números de ponto flutuante quando o programa inicia. Em outras palavras, a macro **FE\_DFL\_ENV** representa a configuração padrão do ambiente de ponto flutuante.

*FE\_DIVBYZERO***Incluir:** <fenv.h>

**Descrição:** A macro **FE\_DIVBYZERO** resulta num valor inteiro positivo que representa um sinalizador de exceção de ponto flutuante correspondente à divisão por zero.

**Observação:** Esta macro será definida apenas se as funções declaradas em <fenv.h> puderem controlar exceções de ponto flutuante.

*FE\_DOWNWARD***Incluir:** <fenv.h>

**Descrição:** A macro **FE\_DOWNWARD** resulta num valor inteiro positivo que representa o modo de arredondamento para baixo (v. **Seção 3.3.5**). Este valor é aceito como argumento da função **fesetround()** e pode ser retornado pela função **fegetround()** (v. **Seção 3.7.3**).

**Observação:** Esta macro será definida apenas se as funções declaradas em `<fenv.h>` puderem controlar o modo de arredondamento.

### *FE\_INEXACT*

**Incluir:** `<fenv.h>`

**Descrição:** A macro **FE\_INEXACT** resulta num valor inteiro positivo que representa um sinalizador de exceção de ponto flutuante correspondente à inexatidão de resultados.

**Observação:** Esta macro será definida apenas se as funções declaradas em `<fenv.h>` puderem controlar exceções de ponto flutuante.

### *FE\_INVALID*

**Incluir:** `<fenv.h>`

**Descrição:** A macro **FE\_INVALID** resulta num valor inteiro positivo que representa um sinalizador de exceção de ponto flutuante correspondente à execução de uma operação inválida.

**Observação:** Esta macro será definida apenas se as funções declaradas em `<fenv.h>` puderem controlar exceções de ponto flutuante.

### *FE\_OVERFLOW*

**Incluir:** `<fenv.h>`

**Descrição:** A macro **FE\_OVERFLOW** resulta num valor inteiro positivo que representa um sinalizador de exceção de ponto flutuante correspondente à ocorrência de *overflow*.

**Observação:** Esta macro será definida apenas se as funções declaradas em `<fenv.h>` puderem controlar exceções de ponto flutuante.

*FE\_TONEAREST***Incluir:** <fenv.h>

**Descrição:** A macro **FE\_TONEAREST** resulta num valor inteiro que representa o modo de arredondamento para o mais próximo (v. **Seção 3.3.5**). Este valor é aceito como argumento da função **fesetround()** e pode ser retornado pela função **fegetround()** (v. **Seção 3.7.3**).

**Observação:** Esta macro será definida apenas se as funções declaradas em <fenv.h> puderem controlar o modo de arredondamento.

*FE\_TOWARDZERO***Incluir:** <fenv.h>

**Descrição:** A macro **FE\_TOWARDZERO** resulta num valor inteiro que representa o modo de arredondamento com aproximação de zero (v. **Seção 3.3.5**). Este valor é aceito como argumento da função **fesetround()** e pode ser retornado pela função **fegetround()** (v. **Seção 3.7.3**).

**Observação:** Esta macro será definida apenas se as funções declaradas em <fenv.h> puderem controlar o modo de arredondamento.

*FE\_UNDERFLOW***Incluir:** <fenv.h>

**Descrição:** A macro **FE\_UNDERFLOW** resulta num valor inteiro positivo que representa um sinalizador de exceção de ponto flutuante correspondente a uma condição de *underflow*.

**Observação:** Esta macro será definida apenas se as funções declaradas em <fenv.h> puderem controlar exceções de ponto flutuante.

*FE\_UPWARD***Incluir:** <fenv.h>



**Descrição:** A macro **FE\_UPWARD** resulta num valor inteiro que representa o modo de arredondamento para cima (v. **Seção 3.3.5**). Este valor é aceito como argumento da função **fesetround()** e pode ser retornado pela função **fegetround()** (v. **Seção 3.7.3**).

**Observação:** Esta macro será definida apenas se as funções declaradas em `<fenv.h>` puderem controlar o modo de arredondamento.

### 3.7.3 FUNÇÕES

*feclearexcept()* (C99)

**Incluir:** `<fenv.h>`

**Descrição:** A função **feclearexcept()** (C99) desliga as exceções de ponto flutuante representadas em seu argumento.

**Protótipo:**

```
int feclearexcept(int excecao)
```

**Parâmetro:** `excecao` – valor inteiro que representa as exceções a serem desligadas. Tipicamente, este argumento é representado por uma combinação, usando o operador sobre bits `|` (v. **Volume I**), de sinalizadores representados por macros [v. **Seção 3.7.2** e o exemplo da função **fetestexcept()**].

**Retorno:** Zero, se o argumento é zero ou se todas as exceções selecionadas são removidas; um valor diferente de zero, caso contrário.

**Exemplo:** Veja o exemplo da função **fetestexcept()**.

*fegetenv()* (C99)

**Incluir:** `<fenv.h>`

**Descrição:** A função **fegetenv()** (C99) armazena as configurações de ponto flutuante na variável cujo endereço é recebido como argumento.

**Protótipo:**

```
int fegetenv(fenv_t *ptrAmbiente)
```

**Parâmetro:** `ptrAmbiente` – ponteiro para uma variável do tipo **fenv\_t** onde as configurações de ponto flutuante serão armazenadas.

**Retorno:** Zero, se o armazenamento das configurações em `*ptrAmbiente` for bem sucedido; um valor diferente de zero, caso contrário.

**Exemplo:** Veja o exemplo da função **fesetenv()**.

*fegetexceptflag()* (C99)

**Incluir:** `<fenv.h>`

**Descrição:** A função **fegetexceptflag()** (C99) armazena na variável apontada por seu primeiro argumento uma representação de exceções de ponto flutuante.

**Protótipo:**

```
int fegetexceptflag(fexcept_t *ptrSinalizador, int excecao)
```

**Parâmetros:**

- `ptrSinalizador` – ponteiro para uma variável do tipo **fexcept\_t** que armazenará a representação de exceções de ponto flutuante.
- `excecao` – valor inteiro que representa exceções. Tipicamente, este argumento é representado por uma combinação, usando o operador sobre bits `|` (v. **Volume I**), de sinalizadores representados por macros [v. **Seção 3.7.2** e o exemplo da função **fetestexcept()**].

**Retorno:** Zero, se todas as exceções representadas por `excecao` forem armazenadas na variável apontada por `ptrSinalizador`; um valor diferente de zero, caso contrário.

**Observações:**

- Uma variável do tipo **fexcept\_t** não deve ser alterada diretamente, o que justifica a necessidade desta função.
- O valor armazenado em `*ptrSinalizador` não pode ser usado diretamente numa expressão para testar se um dado sinalizador está ligado.
- A única utilidade do valor armazenado em `*ptrSinalizador` deve ser a restauração dos sinalizadores através de uma chamada da função **fetestexcept()**.

**Exemplo:** Veja o exemplo da função **fetestexcept()**.

*fegetround()* (C99)

**Incluir:** `<fenv.h>`

**Descrição:** A função **fegetround()** (C99) retorna o modo corrente de arredondamento.

**Protótipo:**

```
int fegetround(void)
```

**Retorno:** O modo corrente de arredondamento, se a função for bem sucedida; um valor negativo, caso contrário.

**Observação:** O modo de arredondamento retornado por esta função corresponde ao valor de uma das macros: **FE\_DOWNWARD**, **FE\_UPWARD**, **FE\_TONEAREST** ou **FE\_TOWARDZERO** (v. Seção 3.7.2).

**Exemplo:** Veja o exemplo da função **fesetround()**.

*feholdexcept()* (C99)**Incluir:** <fenv.h>

**Descrição:** A função **feholdexcept()** (C99) armazena a configuração corrente do ambiente de ponto flutuante na variável cujo endereço é fornecido como argumento. Esta função também desliga todos os sinalizadores de exceções de ponto flutuante.

**Protótipo:**

```
int feholdexcept(fenv_t *ptrAmbiente)
```

**Parâmetro:** `ptrAmbiente` – ponteiro para uma variável do tipo **fenv\_t** onde será armazenada a configuração corrente do ambiente de ponto flutuante.

**Retorno:** Zero, quando a função for bem sucedida (conforme a descrição); um valor diferente de zero, caso contrário.

**Observações:**

- Após uma chamada desta função, qualquer condição de exceção que eventualmente ocorra após a execução de uma operação de ponto flutuante será ignorada.
- Compare esta função com **fegetenv()**.

**Exemplo:** Veja o exemplo de **feupdateenv()**.

*feraiseexcept()* (C99)**Incluir:** <fenv.h>

**Descrição:** A função **feraiseexcept()** (C99) liga os sinalizadores de exceção de ponto flutuante correspondentes àqueles especificados por seu argumento.

**Protótipo:**

```
int feraiseexcept(int excecao)
```

**Parâmetro:** `excecao` é um valor inteiro representando um sinalizador de exceção de ponto flutuante ou uma combinação de tais sinalizadores. Tipicamente, usa-se uma das macros apresentadas na **Seção 3.7.2** ou uma combinação de algumas destas macros por meio do operador `|`.

**Retorno:** Zero, quando o argumento é zero ou a função é bem sucedida (conforme a descrição); caso contrário, um valor diferente de zero.

**Observação:** Compare esta função com `fesetexceptflag()`.

**Exemplo:** O programa a seguir demonstra o uso das funções `fegetexceptflag()`, `fetestexcept()`, `feraiseexcept()`, `feclearexcept()` e `fesetexceptflag()`.

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

void ImprimeFlags(const char* textoInicial, int flags)
{
    if (flags & FE_INEXACT)
        printf("\n%s\n\tOcorreu um resultado inexato\n",
            textoInicial);

    if (flags & FE_UNDERFLOW)
        printf("\n%s\n\tOcorreu underflow\n",
            textoInicial);

    if (flags & FE_OVERFLOW)
        printf("\n%s\n\tOcorreu overflow\n",
            textoInicial);

    if (flags & FE_DIVBYZERO)
        printf("\n%s\n\tOcorreu uma divisao por zero\n",
            textoInicial);

    if (flags & FE_INVALID)
        printf("\n%s\n\tOcorreu uma operacao invalida\n",
            textoInicial);

    if (!(flags & FE_ALL_EXCEPT))
        printf("\n%s\n\tNao houve ocorrencia de excecao\n",
            textoInicial);
}
```

```

int main(void)
{
    fexcept_t flagsSalvas;
    int      excecao;

    /* Guardas o status dos sinalizadores */
    /* para restauração posterior */
    fetexceptflag(&flagsSalvas, FE_ALL_EXCEPT);

    /* Testa todos os sinalizadores de exceção */
    excecao = fetestexcept(FE_ALL_EXCEPT);
    ImprimeFlags("No inicio:", excecao);

    /* Provoca uma exceção do tipo divisão por zero */
    feraiseexcept(FE_DIVBYZERO);

    /* Testa se ocorreu algum tipo de exceção. */
    /* Após a chamada de fetestexcept(), pode-se */
    /* usar a variavel excecao para verificar se */
    /* ocorreu exeção de algum dos tipos */
    /* especificados como argumentos de */
    /* fetestexcept() usando o operador &, o que */
    /* é feito pela função ImprimeFlags(). */
    excecao = fetestexcept(FE_DIVBYZERO | FE_INEXACT);
    ImprimeFlags( "Apos feraiseexcept(FE_DIVBYZERO):",
                  excecao );

    /* Provoca uma exceção de */
    /* inexatidão e depois testa */
    feraiseexcept(FE_INEXACT);
    excecao = fetestexcept(FE_OVERFLOW | FE_INEXACT);
    ImprimeFlags( "Apos feraiseexcept(FE_INEXACT):",
                  excecao );

    /* Desliga todos os sinalizadores de exceção */
    feclearexcept(FE_ALL_EXCEPT);

    /* Testa todos os sinalizadores de exceção */
    excecao = fetestexcept(FE_ALL_EXCEPT);
    ImprimeFlags( "Apos desligar todas as excecoes:",
                  excecao );
}

```

```

        /* Provoca uma exceção de inexatidão */
        /* e depois as testa */
        feraiseexcept(FE_UNDERFLOW);
        excecao = fetestexcept(FE_ALL_EXCEPT);
        ImprimeFlags( "Apos feraiseexcept(FE_UNDERFLOW):",
                      excecao );

        /* Restaura o status de todos os */
        /* sinalizadores de exceções */
        fesetexceptflag(&flagsSalvas, FE_ALL_EXCEPT);

        /* Testa todos os sinalizadores de exceção */
        excecao = fetestexcept(FE_ALL_EXCEPT);
        ImprimeFlags("Apos restaura flags:", excecao);

    return 0;
}

```

### *fesetenv()* (C99)

**Incluir:** <fenv.h>

**Descrição:** A função **fesetenv()** (C99) restaura as configurações de controle de ponto flutuante armazenadas na variável cujo endereço é passado como argumento.

**Protótipo:**

```
int fesetenv(const fenv_t *ptrAmbiente)
```

**Parâmetro:** `ptrAmbiente` – ponteiro para uma variável do tipo **fenv\_t** que representa uma configuração de ambiente de ponto flutuante.

**Retorno:** Zero, se a função for bem sucedida (conforme a descrição acima); caso contrário, um valor diferente de zero.

**Observações:**

- Pode-se usar como parâmetro o endereço de uma variável cujo valor tenha sido obtido por meio de uma chamada de **fegetenv()** ou a macro **FE\_DFL\_ENV**, que representa a configuração de ambiente de ponto flutuante padrão (v. **Seção 3.7.2**).
- Consulte também a descrição da função **fegetenv()**.

**Exemplo:** O programa a seguir demonstra o uso das funções **fegetenv()** e **fesetenv()**, e da diretiva **#pragma STDC FENV\_ACCESS**.

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

/**
 *
 * A função a seguir altera o ambiente de ponto
 * flutuante para realizar algumas operações, mas
 * restaura o ambiente original antes de retornar
 *
 */
double UmaFuncao(double x, double y)
{
    /* Esta opção fica ativada até o */
    /* final da execução desta função */
#pragma STDC FENV_ACCESS ON

    double resultado = 0.0;
    fenv_t ambienteGuardado;

    /* Guarda o ambiente de ponto */
    /* flutuante original */
    fegetenv(&ambienteGuardado);

    /* Instala o ambiente de */
    /* ponto flutuante padrão */
    fesetenv(FE_DFL_ENV);

    /* Executa operações de ponto flutuante */
    /* ... */

    /* Antes de retornar, restaura */
    /* o ambiente original */
    fesetenv(&ambienteGuardado);

    return resultado;
}

int main()
```



```

{
    double resultado;

    /* Talvez o ambiente de ponto-flutuante seja */
    /* alterado no trecho de programa a seguir */
    /* ... */
    resultado = UmaFuncao(2.4, -0.5);

    /* Neste ponto, o ambiente de ponto-flutuante */
    /* é o mesmo de antes da chamada da função */
    /* ... */

    return 0;
}

```

### *fesetexceptflag()* (C99)

**Incluir:** <fenv.h>

**Descrição:** A função **fesetexceptflag()** (C99) restaura os sinalizadores de exceção representados em seu primeiro argumento e, então, liga os sinalizadores de exceção representados em seu segundo argumento.

#### **Protótipo:**

```

int fesetexceptflag( const fexcept_t *pSinalizador,
                    int excecao )

```

#### **Parâmetros:**

- **pSinalizador** – ponteiro para uma variável do tipo **fexcept\_t** contendo uma representação de exceções de ponto flutuante. Este valor deve ter sido obtido por meio de uma chamada anterior de **fegetexceptflag()**.
- **excecao** – valor inteiro representando um sinalizador de exceção de ponto flutuante ou uma combinação de sinalizadores de exceção. Tipicamente, usa-se uma das macros sinalizadoras de exceção apresentadas na **Seção 3.7.2** ou uma combinação de algumas dessas macros obtida por meio do operador |.

**Retorno:** Zero, se o segundo argumento for zero ou se todas as exceções representadas no segundo argumento forem ligadas; um valor diferente de zero, caso contrário.

### Observações:

- Todos os sinalizadores especificados no segundo argumento devem estar representados na variável cujo endereço é passado como primeiro argumento. Portanto, na chamada de **fegetexceptflag()** em que esta variável foi guardada, devem ter sido usados, pelo menos, todos os sinalizadores usados no segundo argumento da chamada de **fesetexceptflag()**.
- Consulte também a descrição de **fegetexceptflag()**.

**Exemplo:** O programa a seguir demonstra o uso das funções **fegetexceptflag()**, **feclearexcept()**, **fetestexcept()** e **fesetexceptflag()**.

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

/****
 *
 * Suponha que a função a seguir executa uma operação
 * de ponto flutuante que possa gerar uma exceção do
 * tipo representada pela flag FE_INVALID e que esta
 * função deva manter intacto o status da variável
 * que armazena todas as exceções de ponto flutuante
 *
 ****/
double UmaFuncao(double x, double y)
{
    int          flag;
    fexcept_t     flagsGuardadas;

    /* Armazena todas as flags de exceções de      */
    /* ponto flutuante na variável fegetexceptflag */
    fegetexceptflag(&flagsGuardadas, FE_ALL_EXCEPT);

    /* Desliga a flag que supostamente interessa */
    feclearexcept(FE_INVALID);
```

```

    /* Executa a operação que pode */
    /* gerar a suposta exceção      */
    /* ... */

    /* Armazena na variável flag a flag de interesse */
    /* (i.e., aquela que pode ter sido ligada) na    */
    /* operação representada acima por "... "        */
    flag = fetestexcept(FE_INVALID);

    /* Agora testa se ocorreu a provável exceção */
    if (flag & FE_INVALID) {
        /* Faz o devido tratamento */
        /* da situação de exceção */
        /* ... */
    }

    /* Antes de retornar, restaura todos */
    /* os sinalizadores de exceções de    */
    /* ponto flutuante armazenados na     */
    /* variável flagsGuardadas            */
    fesetexceptflag(&flagsGuardadas, FE_ALL_EXCEPT);

    /* Isto é apenas um programa educativo; por */
    /* isso, é retornado x + y. Na realidade,    */
    /* seria retornado o resultado da operação */
    /* implementada pela função.                */
    return x + y;
}

int main()
{
    double resultado;

    /* Executa operações que podem gerar */
    /* exceções de ponto flutuante        */
    /* ... */

    /* Talvez algum sinalizador de exceção de */
    /* ponto flutuante esteja ligado neste    */
    /* ponto do programa. A chamada da função */
    /* UmaFuncao() garante que o status dos   */
    /* sinalizadores será preservado.        */

```

```

    resultado = UmaFuncao(2.4, -0.5);

    return 0;
}

```

### *fesetround()* (C99)

**Incluir:** <fenv.h>

**Descrição:** A função **fesetround()** (C99) estabelece um novo modo de arredondamento representado por seu argumento.

**Protótipo:**

```
int fesetround(int modo)
```

**Parâmetro:** *modo* – inteiro que representa um modo de arredondamento. Tipicamente, usa-se uma das macros que representam modos de arredondamento apresentadas na **Seção 3.7.2**.

**Retorno:** Zero, se o modo de arredondamento for alterado de acordo com o valor do argumento; um valor diferente de zero, caso contrário.

**Observação:** Se o valor do argumento não for válido, o modo de arredondamento permanecerá inalterado.

**Exemplo:** O programa a seguir demonstra o uso de macros e das funções **fegetround()** e **fesetround()** para alteração do modo de arredondamento.

```

#include <stdio.h>
#include <fenv.h>
#include <assert.h>
#include <math.h>

void Arredonda(double arredondeMe)
{
    #pragma STDC FENV_ACCESS ON

    int arredondametoSalvo;

    /* Guarda o modo de arredondamento corrente */

```

```

arredondametoSalvo = fegetround();

if (fesetround(FE_DOWNWARD)) {
    printf( "Nao foi possivel alterar o modo de "
           "arredondamento" );
    return;
}

printf( "O valor de %3.2f arredondado usando lrint "
        "e' %ld\n", arredondeMe, lrint(arredondeMe) );
printf( "O valor de %3.2f arredondado usando "
        "lround e' %ld\n", arredondeMe,
        lround(arredondeMe) );

if (fesetround(FE_UPWARD)) {
    printf("Nao foi possivel alterar o modo de "
           "arredondamento");
    return;
}

printf( "O valor de %3.2f arredondado usando "
        "lrint e' %ld\n", arredondeMe,
        lrint(arredondeMe) );
printf( "O valor de %3.2f arredondado usando "
        "lround e' %ld\n", arredondeMe,
        lround(arredondeMe) );

    /* Restaura o modo de arredondamento original */
    fesetround(arredondametoSalvo);
}

int main()
{
    double x = 2.4;

    /* Executa algumas operações */
    /* ... */

    Arredonda(x);

    /* Executa outras operações com o mesmo */
    /* modo de arredondamento em vigor antes */
    /* de chamar Arredonda() */
}

```

```

    /* ... */

    return 0;
}

```

### *fetestexcept()* (C99)

**Incluir:** <fenv.h>

**Descrição:** A função **fetestexcept()** (C99) retorna a conjunção sobre bits (&) do sinalizador de exceção representado por seu argumento com os sinalizadores de exceção correntemente ligados.

**Protótipo:**

```
int fetestexcept(int excecao)
```

**Parâmetro:** *excecao* – valor inteiro representando um sinalizador de exceção de ponto flutuante ou uma combinação desses sinalizadores. Tipicamente, usa-se uma das macros sinalizadoras de exceção apresentadas na **Seção 3.7.2** ou uma combinação de algumas dessas macros obtida por meio do operador |.

**Retorno:** Um valor diferente de zero se o sinalizador de exceção de ponto flutuante representado por seu argumento estiver ligado; zero, caso contrário.

**Exemplo:** O programa a seguir demonstra o uso da função **fetestexcept()** e de macros usadas em tratamento de exceções de ponto flutuante.

```

#include <stdio.h>
#include <float.h>
#include <fenv.h>

int main()
{
    int    flags;
    double x;

    #pragma STDC FENV_ACCESS ON

```

```

x = 2.0/3.0; /* Causa uma exceção de inexatidão */

x = DBL_MIN / 5.0; /* Deve causar underflow */
/* Ao contrário do que muitos pensam, a */
/* instrução seguinte não causa o aborto */
/* do programa. Ela simplesmente liga o */
/* sinalizador de exceção de operação */
/* inválida e o resultado armazenado em */
/* x é indefinido. */
x = 0.0/0.0;
printf("%f", x);

/* Armazena todas em flags as exceções de */
/* ponto flutuante que se desejam testar */
flags = fetestexcept( FE_UNDERFLOW |
                     FE_INEXACT |
                     FE_INVALID );

/* Agora testa que tipo de exceção de ponto */
/* flutuante ocorreu nas operações acima */

if (flags & FE_UNDERFLOW)
    printf("\nOcorreu overflow numa das operacoes\n");

if (flags & FE_INEXACT)
    printf( "\nFoi executada uma operacao que "
           "resultou num valor inexato\n" );

if (flags & FE_INVALID)
    printf( "\nHouve uma tentativa de execucao de "
           "operacao invalida\n" );

return 0;
}

```

### *feupdateenv()* (C99)

**Incluir:** <fenv.h>

**Descrição:** Inicialmente, a função **feupdateenv()** (C99) guarda todos os sinalizadores de exceções de ponto flutuante correntes. Em seguida, atualiza a configuração do ambiente de ponto flutuante com o valor armazenado na variável cujo endereço é

fornecido como argumento. Finalmente, esta função liga os sinalizadores de exceção de ponto flutuante correspondentes àqueles guardados no início da operação.

### Protótipo:

```
int feupdateenv(const fenv_t *ptrAmbiente)
```

**Parâmetro:** `ptrAmbiente` – ponteiro para uma variável do tipo **fenv\_t** contendo uma configuração do ambiente de ponto flutuante.

**Retorno:** Zero, quando a função é bem sucedida (conforme a descrição anterior); um valor diferente de zero, caso contrário.

### Observações:

- A configuração a ser estabelecida, representada na variável cujo endereço é passado como argumento, é determinada pela macro **FE\_DFL\_ENV** ou por um valor armazenado numa variável por meio de uma chamada prévia de **fegetenv()** ou **fehldexcept()**. Qualquer outro valor de configuração resultará num comportamento indefinido da função.
- Chamar esta função é equivalente a executar as seguintes instruções:

```
int excecao = fetestexcept(FE_ALL_EXCEPT);
fesetenv(ambientePtr);
feraiseexcept(excecao);
```

**Exemplo:** O programa a seguir demonstra o uso das funções **fehldexcept()**, **feupdateenv()**, **feraiseexcept()** e **fetestexcept()**.

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

void ImprimeFlags(const char* textoInicial, int flags)
{
    if (flags & FE_INEXACT)
        printf("\n%s\n\tOcorreu um resultado inexato\n",
               textoInicial);

    if (flags & FE_UNDERFLOW)
        printf("\n%s\n\tOcorreu underflow\n",
               textoInicial);
}
```



```

if (flags & FE_OVERFLOW)
    printf("\n%s\n\tOcorreu overflow\n",
           textoInicial);

if (flags & FE_DIVBYZERO)
    printf("\n%s\n\tOcorreu uma divisao por zero\n",
           textoInicial);

if (flags & FE_INVALID)
    printf("\n%s\n\tOcorreu uma operacao invalida\n",
           textoInicial);

if (!(flags & FE_ALL_EXCEPT))
    printf("\n%s\n\tNao houve ocorrencia de excecao\n",
           textoInicial);
}

double OperacaoDePF(double x)
{
    fenv_t  ambientePF;
    int     excecao;

    /* O trecho a seguir demonstra que, como esta */
    /* função pode ter sido chamada depois de      */
    /* alguma flag ter sido ligada, se ocorrer     */
    /* alguma exceção aqui, não será possível      */
    /* saber se esta exceção surgiu nesta função  */
    /* ou se a mesma já havia ocorrido antes de   */
    /* esta função ser chamada.                    */
    excecao = fetestexcept(FE_ALL_EXCEPT);
    ImprimeFlags( "Na entrada da funcao OperacaoDePF:",
                  excecao );

    /* Aqui começa a situação real. Primeiro,     */
    /* salvam-se os sinalizadores e desligam-se    */
    /* os mesmos usando a função feholdexcept() */
    feholdexcept(&ambientePF);

    /* Agora são executadas operações de ponto */
    /* flutuante que podem dar origem a alguma */
    /* exceção. Aqui, isto é simulado usando    */
    /* a função feraiseexcept().                */

```

```

// ... (operações que podem causar exceção)
feraiseexcept(FE_INEXACT);

/* Numa situação real, as instruções a seguir */
/* seriam substituídas por instruções que testa */
/* se ocorreram exceções nas operações acima. */
excecao = fetestexcept(FE_ALL_EXCEPT);
ImprimeFlags("Na funcao OperacaoDePF, depois da "
             "ocorrenda de uma excecao:", excecao);

/* Agora as exceções de ponto flutuante que */
/* possam ter ocorrido seriam tratadas a seguir */
// ... (tratamento de eventuais exceções)

/* Restaura a configuração de ambiente */
/* guardada acima e liga os sinalizadores */
/* de exceções que ocorreram nesta função */
feupdateenv(&ambientePF);

excecao = fetestexcept(FE_ALL_EXCEPT);
ImprimeFlags( "Na funcao OperacaoDePF, antes "
             "de retornar:", excecao );

return x;
}

int main(void)
{
    int excecao;

    /* Suponha que no trecho de programa a seguir */
    /* possa ocorrer uma situação de exceção. */
    /* Aqui, nós simulamos isto usando a função */
    /* feraiseexcept(). */

    // ... (operações que podem causar exceção)
    feraiseexcept(FE_DIVBYZERO);

    excecao = fetestexcept(FE_ALL_EXCEPT);
    ImprimeFlags( "Na funcao main antes de chamar "
                 "OperacaoDePF:", excecao );

```

```

    /* Agora, chama a função OperacaoDePF() */
    /* que também pode provocar uma exceção. */
    OperacaoDePF(2.5);

    excecao = fetestexcept(FE_ALL_EXCEPT);
    ImprimeFlags( "Na funcao main depois de chamar "
                  "OperacaoDePF:", excecao );

    return 0;
}

```

### 3.8 EXERCÍCIOS DE REVISÃO

- Qual é o propósito geral do cabeçalho `<float.h>`?
- Qual é a expansão de cada uma das seguintes macros?
  - DBL\_DIG**
  - DBL\_MANT\_DIG**
  - DBL\_MAX**
  - DECIMAL\_DIG**
  - DBL\_MAX\_EXP**
  - DBL\_MIN**
- Qual é a utilidade da macro **DBL\_EPSILON**?
- (a) O que é um número de ponto flutuante normalizado? (b) O que é um número de ponto flutuante não normalizado?
- Como se pode determinar se um valor de ponto flutuante do tipo **float** é automaticamente promovido a **double**?
- Quais são os possíveis modos de arredondamento disponíveis em operações de ponto flutuante da biblioteca padrão de C?
- Como se pode determinar qual é o modo de arredondamento vigente?

8. Que opção do compilador gcc é utilizada para fazer ligação de programas que usam funções declaradas em `<math.h>`?

9. O que é **FMA**?

10. O que significa NaN?

11. (a) O que é um erro de domínio? (b) O que é um erro de intervalo? (c) Como as funções declarada no cabeçalho `<math.h>` podem indicar um erro de domínio?

12. Qual pode ser o efeito da seguinte diretiva **#pragma**?

```
#pragma STD FP_CONTRACT ON
```

13. Qual é o efeito de cada uma das diretivas **#pragma** a seguir?

(a) `#pragma STD FENV_ACCESS ON`

(b) `#pragma STD FENV_ACCESS OFF`

14. Descreva os tipos **double\_t** e **float\_t**.

15. Qual é a expansão de cada uma das seguintes macros?

(a) **FP\_FAST\_FMA**

(b) **FP\_NAN**

(c) **FP\_NORMAL**

16. (a) Qual é o significado da macro **HUGE\_VAL**? (b) Dê exemplo de uma operação que retorne este valor.

17. (a) Qual é o significado da macro **INFINITY**? (b) Dê exemplo de uma operação que resulte neste valor.

18. (a) Qual é o significado da macro **NAN**? (b) Dê exemplo de uma operação que resulte neste valor.

19. Descreva as seguintes funções declaradas em `<math.h>`:

(a) **ceil()**

(b) **floor()**

- (c) **rint()**
  - (d) **round()**
  - (e) **nearbyint()**
  - (f) **nextafter()**
  - (g) **nexttoward()**
  - (h) **trunc()**
20. (a) O que realiza a função **expm1()**? (b) O que justifica a existência desta função?
21. Descreva a função **fdim()** declarada em `<math.h>`.
22. Como as macros **math\_errhandling**, **MATH\_ERRNO** e **MATH\_ERREXCEPT** são utilizadas em conjunto?
23. Qual é a expansão de cada uma das seguintes macros?
- (a) **FE\_ALL\_EXCEPT**
  - (b) **FE\_DFL\_ENV**
  - (c) **FE\_DOWNWARD**
  - (d) **FE\_INEXACT**
  - (e) **FE\_TONEAREST**
  - (f) **FE\_TOWARDZERO**
  - (g) **FE\_UNDERFLOW**
  - (h) **FE\_UPWARD**
24. Apresente a interpretação de cada uma das seguintes macros:
- (a) **FE\_DIVBYZERO**
  - (b) **FE\_INVALID**
  - (c) **FE\_OVERFLOW**
25. Para que servem as macros de classificação definidas em `<math.h>`?

26. Qual é o propósito do cabeçalho `<fenv.h>`?
27. Descreva os tipos **fenv\_t** e **fexcept\_t** definidos em `<fenv.h>`.
28. (a) O que é uma exceção de ponto flutuante? (b) Apresente três exemplos de exceções de ponto flutuante.
29. O que é um sinalizador de exceção?
30. Descreva os seguintes modos de arredondamento:
  - (a) Arredondamento com afastamento de zero
  - (b) Arredondamento com aproximação de zero
  - (c) Arredondamento para o mais próximo
  - (d) Arredondamento para baixo
  - (e) Arredondamento para cima
31. Descreva como seria arredondado o valor  $3.45$  de modo que o resultado tenha uma casa decimal utilizando:
  - (a) Arredondamento com afastamento de zero
  - (b) Arredondamento com aproximação de zero
  - (c) Arredondamento para o mais próximo
  - (d) Arredondamento para baixo
  - (e) Arredondamento para cima
32. Suponha que se deseja arredondar o valor  $-2.56$  de modo que o resultado tenha apenas uma casa decimal. Qual seria o valor resultante se o arredondamento fosse:
  - (a) Com afastamento de zero
  - (b) Com aproximação de zero
  - (c) Para o mais próximo
  - (d) Para baixo
  - (e) Para cima

33. Que função pode ser utilizada para alterar o modo de arredondamento corrente?
34. Descreva o funcionamento de cada uma das seguintes funções declaradas em `<fenv.h>`.
- (a) **feclearexcept()**
  - (b) **fegetenv()**
  - (c) **fegetexceptflag()**
  - (d) **fegetround()**
  - (e) **feholdexcept()**
  - (f) **feraiseexcept()**
  - (g) **fesetenv()**
  - (h) **fesetexceptflag()**
  - (i) **fesetround()**
  - (j) **fetestexcept()**
  - (k) **feupdateenv()**

# *Capítulo 4*

---

*Números de ponto flutuante complexos e macros genéricas*



## 4.1 INTRODUÇÃO

Conforme foi exposto no **Capítulo 3**, os tipos de ponto flutuante primitivos de C são divididos em reais e complexos. Enquanto o enfoque daquele capítulo foram os tipos reais, o presente capítulo lida com os tipos complexos.

Além de apresentar uma breve introdução aos tipos complexos primitivos de C, este capítulo descreve ainda os seguintes cabeçalhos:

- `<complex.h>` – que provê suporte para operações básicas sobre números complexos representados em C.
- `<tgmath.h>` – que define macros genéricas capazes de decidir qual operação será executada de acordo com o tipo de parâmetro, real ou complexo, utilizado.

## 4.2 TIPOS PRIMITIVOS DE PONTO FLUTUANTE COMPLEXOS

Os tipos complexos primitivos de C, introduzidos pelo padrão C99, aparecem em negrito na **Figura 4-1**. Nesta figura, nota-se que existem seis tipos nesta categoria, mas o padrão C99 requer que uma implementação dê suporte apenas aos tipos: **float \_Complex**, **double \_Complex** e **long double \_Complex**. Os demais tipos complexos, de acordo com este padrão, são sugeridos (mas não são exigidos).

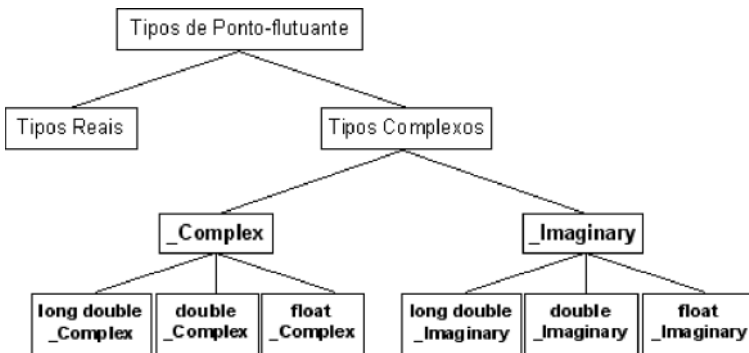


Figura 4-1: Tipos primitivos de ponto flutuante complexos.

Para possibilitar a escrita de constantes complexas, é necessário incluir o cabeçalho `<complex.h>` e utilizar a constante **\_Complex\_I** ou a constante **I**, que representam  $\sqrt{-1}$ . Por exemplo, a constante complexa:

```
-2.0 + 1.5*_Complex_I
```

é o mesmo que:

```
-2.0 + 1.5*I
```

Constantes complexas podem utilizar em suas partes real e imaginária os mesmos formatos de constantes de ponto flutuante vistos na **Seção 3.2**, incluindo os mesmos sufixos. Por exemplo, a constante:

```
-2.0f + 1.5f*I
```

é considerada como sendo do tipo **float \_Complex**.

Quando as partes real e imaginária de uma constante possuem tipos diferentes, ocorre conversão implícita no tipo de maior largura (v. **Capítulo 1** do **Volume I**). Por exemplo, a constante:

```
-2.0f + 1.5*I
```

é considerada como sendo do tipo **double \_Complex**, pois a parte imaginária é do tipo **double**.

Apesar de o padrão C99 incluir **\_Imaginary** como palavra-chave, a implementação de tipos imaginários puros é opcional. Portanto, não espere encontrar este tipo em qualquer implementação de C, mesmo que a implementação siga o padrão C99.

## 4.3 PRAGMA CX\_LIMITED\_RANGE

A diretiva **#pragma CX\_LIMITED\_RANGE** controla o comportamento de operações de multiplicação, divisão e módulo de números complexos. Os possíveis valores do parâmetro desta diretiva com seus respectivos efeitos são apresentados na **Tabela 4-1**.

PARÂMETRO	EFEITO
ON	<p>Operações de multiplicação, divisão e módulo são avaliadas conforme mostrado a seguir:</p> <ul style="list-style-type: none"> <li>• A expressão <math>(x + I*y) * (u + I*v)</math> é avaliada como: <math display="block">(x*u - y*v) + I*(y*u + x*v)</math> </li> <li>• A expressão <math>(x + I*y) / (u + I*v)</math> é avaliada como: <math display="block">((x*u + y*v) + I*(y*u - x*v)) / (u*u + v*v)</math> </li> <li>• A expressão <math> x + I*y </math> é avaliada como: <math display="block">\text{sqrt}(x*x + y*y)</math> </li> </ul>
OFF	Restaura o estado padrão original no qual as equivalências descritas não são permitidas.
DEFAULT	O mesmo que OFF.

Tabela 4-1: Possíveis valores do parâmetro da diretiva pragma CX\_LIMITED\_RANGE.

Levando em consideração os valores possíveis para o único parâmetro da diretiva **#pragma CX\_LIMITED\_RANGE**, existem três formas de utilização desta diretiva:

- #pragma STD CX\_LIMITED\_RANGE ON
- #pragma STD CX\_LIMITED\_RANGE OFF
- #pragma STD CX\_LIMITED\_RANGE DEFAULT

Se essa diretiva **#pragma** for utilizada dentro de um bloco, ela deverá preceder qualquer declaração ou instrução e seu efeito terá vigência até que seja invalidada por outra diretiva do mesmo tipo, ou até o final do referido bloco. Por outro lado, se essa diretiva pragma for utilizada fora de qualquer bloco, o efeito dela permanecerá até ser anulado por outra diretiva semelhante.

## 4.4 SUPORTE PARA NÚMEROS COMPLEXOS: <complex.h> (C99)

O cabeçalho <complex.h> define várias macros e funções que podem ser utilizadas com valores dos tipos complexos primitivos definidos pelo padrão C99.

Para entender muitas das operações sobre números complexos que serão descritas aqui, é necessário um conhecimento matemático prévio que vai além do conhecimento rudimentar sobre o assunto.

### 4.4.1 MACROS

As macros definidas em <complex.h> são apresentadas na **Tabela 4-2** junto com suas respectivas expansões.

MACRO	EXPANSÃO
<b>complex</b>	A palavra reservada <b>_Complex</b> .
<b>_Complex_I</b>	Uma expressão do tipo <code>const float _Complex</code> , cuja parte real é zero e cuja parte imaginária é 1.
<b>I</b>	<b>_Imaginary_I</b> , se a macro <b>imaginary</b> for definida; <b>_Complex_I</b> , caso contrário.
<b>imaginary</b>	A palavra reservada <b>_Imaginary</b> , se ela for definida pela implementação.
<b>_Imaginary_I</b>	A expressão <code>(const float _Imaginary) 1</code> , se esta macro for definida.

Tabela 4-2: Macros definidas em <complex.h>.

### 4.4.2 VISÃO GERAL DAS FUNÇÕES DECLARADAS EM <complex.h>

Cada função declarada no cabeçalho <complex.h> possui duas outras funções semelhantes (i.e., as três funções executam basicamente a mesma tarefa). Este fato é mostrado na **Tabela 4-3**, onde, em cada linha, aparecem três funções que executam exatamente a mesma operação. A única diferença entre estas funções é o tipo de dados sobre o qual cada uma delas atua. Funções que terminam em *f* (coluna do meio na tabela) recebem argumentos do tipo **float \_Complex** e retornam valores deste mesmo tipo ou do tipo real

correspondente, funções que terminam em *l* (terceira coluna na tabela) recebem argumentos do tipo **long double \_Complex** e retornam valores deste mesmo tipo ou do tipo real correspondente e, finalmente, as funções que aparecem na primeira coluna, que não terminam nem em *f*, nem em *l*, recebem argumentos do tipo **double \_Complex** e retornam valores deste mesmo tipo ou do tipo real correspondente<sup>31</sup>. Portanto, entender o funcionamento de apenas uma função em cada linha da **Tabela 4-3** é suficiente para entender o funcionamento das demais funções na mesma linha desta tabela.

FUNÇÕES QUE ATUAM SOBRE O TIPO...		
<b>double _Complex</b>	<b>float _Complex</b>	<b>long double _Complex</b>
<b>cabs()</b>	<b>cabsf()</b>	<b>cabsl()</b>
<b>cacos()</b>	<b>cacosf()</b>	<b>cacosl()</b>
<b>cacosh()</b>	<b>cacoshf()</b>	<b>cacoshl()</b>
<b>carg()</b>	<b>cargf()</b>	<b>cargl()</b>
<b>casin()</b>	<b>casinf()</b>	<b>casinl()</b>
<b>casinh()</b>	<b>casinhf()</b>	<b>casinhl()</b>
<b>catan()</b>	<b>catanf()</b>	<b>catanl()</b>
<b>catanh()</b>	<b>catanhf()</b>	<b>catanhl()</b>
<b>ccos()</b>	<b>ccosf()</b>	<b>ccosl()</b>
<b>ccosh()</b>	<b>ccoshf()</b>	<b>ccoshl()</b>
<b>cexp()</b>	<b>cexpf()</b>	<b>cexpl()</b>
<b>cimag()</b>	<b>cimagf()</b>	<b>cimagl()</b>
<b>clog()</b>	<b>clogf()</b>	<b>clogl()</b>
<b>conj()</b>	<b>conjf()</b>	<b>conjl()</b>
<b>cpow()</b>	<b>cpowf()</b>	<b>cpowl()</b>
<b>cproj()</b>	<b>cprojf()</b>	<b>cprojl()</b>
<b>creal()</b>	<b>crealf()</b>	<b>creall()</b>
<b>csin()</b>	<b>csinf()</b>	<b>csinl()</b>
<b>csinh()</b>	<b>csinhf()</b>	<b>csinhl()</b>
<b>csqrt()</b>	<b>csqrtf()</b>	<b>csqrtl()</b>
<b>ctan()</b>	<b>ctanf()</b>	<b>ctanl()</b>
<b>ctanh()</b>	<b>ctanhf()</b>	<b>ctanhl()</b>

Tabela 4-3: Funções correspondentes para os três tipos complexos.

31 Por exemplo, a função **cacos()** recebe um argumento do tipo **double \_Complex** e retorna um valor deste mesmo tipo. Por outro lado, a função **cabs()** recebe um argumento do tipo **double \_Complex** e retorna um valor do tipo **double**.

Nas subseções a seguir, serão apresentadas apenas as funções que aparecem na primeira coluna da **Tabela 4-3**. A extensão do conhecimento adquirido para as demais funções desta tabela é trivial. Estas funções são agrupadas por afinidade para facilidade de referência. Algumas funções descritas nas seções seguintes representam operações matemáticas que podem retornar valores múltiplos. Neste caso, tais funções retornam uma parte imaginária no intervalo  $(-\pi, \pi]$ , a não ser que haja informação explícita que contradiga isso.

Como ocorre com funções declaradas em `<math.h>`, o *linker* que acompanha o compilador gcc requer o uso da opção `-lm` para fazer ligações de um programa com funções declaradas em `<complex.h>` (v. **Seção 3.6**).

### 4.4.3 FUNÇÕES TRIGONOMÉTRICAS COMPLEXAS

*cacos()* (C99)

**Incluir:** `<complex.h>`

**Descrição:** A função **cacos()** (C99) calcula o arco cosseno do parâmetro fornecido.

**Protótipo:**

```
double _Complex cacos(double _Complex x)
```

**Parâmetro:** `x` – valor do tipo **double \_Complex** cujo arco cosseno será calculado.

**Retorno:** O arco cosseno do parâmetro fornecido.

**Exemplo:** Veja o exemplo de **ctan()**.

*casin()* (C99)

**Incluir:** `<complex.h>`

**Descrição:** A função **casin()** (C99) calcula o arco seno do parâmetro fornecido.

**Protótipo:**

```
double _Complex casin(double _Complex x)
```

**Parâmetro:** *x* – valor do tipo **double \_Complex** cujo arco seno será calculado.

**Retorno:** O arco seno do parâmetro fornecido.

**Exemplo:** Veja o exemplo de **ctan()**.

*catan()* (C99)

**Incluir:** <complex.h>

**Descrição:** A função **catan()** (C99) calcula o arco tangente do parâmetro fornecido.

**Protótipo:**

```
double _Complex catan(double _Complex x)
```

**Parâmetro:** *x* – valor do tipo **double \_Complex** cujo arco tangente será calculado.

**Retorno:** O arco tangente do parâmetro fornecido.

**Exemplo:** Veja o exemplo de **ctan()**.

*ccos()* (C99)

**Incluir:** <complex.h>

**Descrição:** A função **ccos()** (C99) calcula o cosseno do parâmetro fornecido.

**Protótipo:**

```
double _Complex ccos(double _Complex x)
```

**Parâmetro:**  $x$  – valor do tipo **double \_Complex** cujo cosseno será calculado.

**Retorno:** O cosseno do parâmetro fornecido.

**Exemplo:** Veja o exemplo de **ctan()**.

*csin()* (C99)

**Incluir:** `<complex.h>`

**Descrição:** A função **csin()** (C99) calcula o seno do parâmetro fornecido.

**Protótipo:**

```
double _Complex csin(double _Complex x)
```

**Parâmetro:**  $x$  – valor do tipo **double \_Complex** cujo seno será calculado.

**Retorno:** O seno do parâmetro fornecido.

**Exemplo:** Veja o exemplo de **ctan()**.

*ctan()* (C99)

**Incluir:** `<complex.h>`

**Descrição:** A função **ctan()** (C99) calcula a tangente do parâmetro fornecido.

**Protótipo:**

```
double _Complex ctan(double _Complex x)
```

**Parâmetro:**  $x$  – valor do tipo **double \_Complex** cuja tangente será calculada.

**Retorno:** A tangente do parâmetro fornecido.



**Exemplo:** O programa a seguir demonstra o uso das funções **cacos()**, **casin()**, **catan()**, **ccos()**, **csin()** e **ctan()**.

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

int main()
{
    double _Complex x = 2.0 -3.0*I,
            z;

    /* NOTA: As funções creal() e cimag() retornam, */
    /*      respectivamente, as partes real e      */
    /*      imaginária de um número complexo.      */

    z = cacos(x);
    printf( "Arco cosseno de %3.1f + %3.1f*i = "
            "%5.3f + %5.3f*i\n", creal(x), cimag(x),
            creal(z), cimag(z) );

    z = casin(x);
    printf( "Arco seno de %3.1f + %3.1f*i = "
            "%5.3f + %5.3f*i\n",
            creal(x), cimag(x), creal(z), cimag(z) );

    z = catan(x);
    printf( "Arco tangente de %3.1f + %3.1f*i = "
            "%5.3f + %5.3f*i\n", creal(x), cimag(x),
            creal(z), cimag(z) );

    z = ccos(x);
    printf( "Cosseno de %3.1f + %3.1f*i = "
            "%5.3f + %5.3f*i\n",
            creal(x), cimag(x), creal(z), cimag(z) );

    z = csin(x);
    printf( "Seno de %3.1f + %3.1f*i = %5.3f + %5.3f*i\n",
            creal(x), cimag(x), creal(z), cimag(z) );

    z = ctan(x);
    printf( "Tangente de %3.1f + %3.1f*i = "
            "%5.3f + %5.3f*i\n",
            creal(x), cimag(x), creal(z), cimag(z) );
```

```
    return 0;
}
```

#### 4.4.4 FUNÇÕES HIPERBÓLICAS COMPLEXAS

*cacosh()* (C99)

**Incluir:** <complex.h>

**Descrição:** A função **cacosh()** (C99) calcula o arco cosseno hiperbólico do parâmetro fornecido.

**Protótipo:**

```
double _Complex cacosh(double _Complex x)
```

**Parâmetro:** *x* – valor do tipo **double \_Complex** cujo arco cosseno hiperbólico será calculado.

**Retorno:** O arco cosseno hiperbólico do parâmetro fornecido.

**Exemplo:** Veja o exemplo de **ctanh()**.

*casinh()* (C99)

**Incluir:** <complex.h>

**Descrição:** A função **casinh()** (C99) calcula o arco seno hiperbólico do parâmetro fornecido.

**Protótipo:**

```
double _Complex casinh(double _Complex x)
```

**Parâmetro:** *x* – valor do tipo **double \_Complex** cujo arco seno hiperbólico será calculado.

**Retorno:** O arco seno hiperbólico do parâmetro fornecido.

**Exemplo:** Veja o exemplo de `ctanh()`.

*catanh()* (C99)

**Incluir:** `<complex.h>`

**Descrição:** A função `catanh()` (C99) calcula o arco tangente hiperbólica do parâmetro fornecido.

**Protótipo:**

```
double _Complex catanh(double _Complex x)
```

**Parâmetro:** `x` – valor do tipo `double _Complex` cujo arco tangente hiperbólica será calculado.

**Retorno:** O arco tangente hiperbólica do parâmetro fornecido.

**Exemplo:** Veja o exemplo de `ctanh()`.

*ccosh()* (C99)

**Incluir:** `<complex.h>`

**Descrição:** A função `ccosh()` (C99) calcula o cosseno hiperbólico do parâmetro fornecido.

**Protótipo:**

```
double _Complex ccosh(double _Complex x)
```

**Parâmetro:** `x` – valor do tipo `double _Complex` cujo cosseno hiperbólico será calculado.

**Retorno:** O cosseno hiperbólico do parâmetro fornecido.

**Exemplo:** Veja o exemplo de `ctanh()`.

*csinh()* (C99)

**Incluir:** `<complex.h>`

**Descrição:** A função `csinh()` (C99) calcula o seno hiperbólico do parâmetro fornecido.

**Protótipo:**

```
double _Complex csinh(double _Complex x)
```

**Parâmetro:** `x` – valor do tipo **`double _Complex`** cujo seno hiperbólico será calculado.

**Retorno:** O seno hiperbólico do parâmetro fornecido.

**Exemplo:** Veja o exemplo de `ctanh()`.

*ctanh()* (C99)

**Incluir:** `<complex.h>`

**Descrição:** A função `ctanh()` (C99) calcula a tangente hiperbólica do parâmetro fornecido.

**Protótipo:**

```
double _Complex ctanh(double _Complex x)
```

**Parâmetro:** `x` – valor do tipo **`double _Complex`** cuja tangente hiperbólica será calculada.

**Retorno:** A tangente hiperbólica do parâmetro fornecido.

**Exemplo:** O programa a seguir demonstra o uso das funções **cacosh()**, **casinh()**, **catanh()**, **ccosh()**, **csinh()** e **ctanh()**.

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

int main()
{
    double _Complex x = 2.0 -3.0*I,
            z;

    /* NOTA: As funções creal() e cimag() retornam, */
    /*      respectivamente, as partes real e      */
    /*      imaginária de um número complexo.      */

    z = cacosh(x);
    printf( "Arco cosseno hiperbolico de "
           "%3.1f + %3.1f*i = %5.3f + %5.3f*i\n",
           creal(x), cimag(x), creal(z), cimag(z) );

    z = casinh(x);
    printf( "Arco seno hiperbolico de "
           "%3.1f + %3.1f*i = %5.3f + %5.3f*i\n",
           creal(x), cimag(x), creal(z), cimag(z) );

    z = catanh(x);
    printf( "Arco tangente hiperbolica de "
           "%3.1f + %3.1f*i = %5.3f + %5.3f*i\n",
           creal(x), cimag(x), creal(z), cimag(z) );

    z = ccosh(x);
    printf( "Cosseno hiperbolico de %3.1f + %3.1f*i = "
           "%5.3f + %5.3f*i\n",
           creal(x), cimag(x), creal(z), cimag(z) );

    z = csinh(x);
    printf( "Seno hiperbolico de %3.1f + %3.1f*i = "
           "%5.3f + %5.3f*i\n",
           creal(x), cimag(x), creal(z), cimag(z) );

    z = ctanh(x);
    printf( "Tangente hiperbolica de %3.1f + %3.1f*i = "
           "%5.3f + %5.3f*i\n",
```

```

        creal(x), cimag(x), creal(z), cimag(z) );

    return 0;
}

```

## 4.4.5 FUNÇÕES EXPONENCIAIS E LOGARÍTMICAS COMPLEXAS

*cexp()* (C99)

**Incluir:** <complex.h>

**Descrição:** A função **cexp()** (C99) calcula a exponencial do parâmetro fornecido.

**Protótipo:**

```
double _Complex cexp(double _Complex x)
```

**Parâmetro:** *x* – valor do tipo **double \_Complex** cuja exponencial será calculada.

**Retorno:** A exponencial do parâmetro fornecido.

**Exemplo:** Veja o exemplo de **csqrt()**.

*clog()* (C99)

**Incluir:** <complex.h>

**Descrição:** A função **clog()** (C99) calcula o logaritmo do parâmetro fornecido.

**Protótipo:**

```
double _Complex clog(double _Complex x)
```

**Parâmetro:** *x* – valor do tipo **double \_Complex** cujo logaritmo será calculado.

**Retorno:** O logaritmo do parâmetro fornecido.

**Exemplo:** Veja o exemplo de **csqrt()**.

*cpow()* (C99)**Incluir:** <complex.h>**Descrição:** A função **cpow()** (C99) calcula o valor do primeiro parâmetro elevado ao valor do segundo parâmetro.**Protótipo:**

```
double _Complex cpow(double _Complex x, double _Complex y)
```

**Parâmetros:**

- **x** – valor do tipo **double \_Complex** que servirá como base no cálculo.
- **y** – valor do tipo **double \_Complex** que servirá como expoente no cálculo.

**Retorno:** O valor do primeiro parâmetro elevado ao valor do segundo parâmetro.**Exemplo:** Veja o exemplo de **csqrt()**.*csqrt()* (C99)**Incluir:** <complex.h>**Descrição:** A função **csqrt()** (C99) calcula a raiz quadrada do parâmetro fornecido.**Protótipo:**

```
double _Complex csqrt(double _Complex x)
```

**Parâmetro:** **x** – valor do tipo **double \_Complex** cuja raiz quadrada será calculada.**Retorno:** A raiz quadrada do parâmetro fornecido.**Exemplo:** O programa a seguir demonstra o uso das funções **cexp()**, **clog()**, **cpow()** e **csqrt()**.

```

#include <stdio.h>
#include <complex.h>

int main()
{
    double _Complex x = 2.0 -3.0*I,
           y = -1.5 + 2.3*I,
           z;

    /* NOTA: As funções creal() e cimag() retornam, */
    /*      respectivamente, as partes real e      */
    /*      imaginária de um número complexo.      */

    z = cexp(x);
    printf( "Exponencial de %3.1f + %3.1f*i = "
           "%5.3f + %5.3f*i\n", creal(x), cimag(x),
           creal(z), cimag(z) );

    z = clog(x);
    printf( "Logaritmo de %3.1f + %3.1f*i = "
           "%5.3f + %5.3f*i\n", creal(x), cimag(x),
           creal(z), cimag(z) );

    z = cpow(x, y);
    printf("%3.1f + %3.1f*i elevado a %3.1f + %3.1f*i = "
           " %5.3f + %5.3f*i\n", creal(x), cimag(x),
           creal(y), cimag(y), creal(z), cimag(z) );

    z = csqrt(x);
    printf( "Raiz quadrada de %3.1f + %3.1f*i = "
           "%5.3f + %5.3f*i\n", creal(x), cimag(x),
           creal(z), cimag(z) );

    return 0;
}

```

#### 4.4.6 OUTRAS FUNÇÕES DECLARADAS EM <complex.h>

Esta seção apresenta funções declaradas no cabeçalho <complex.h> que não se encaixam em nenhuma das categorias anteriormente descritas.



*cabs()* (C99)**Incluir:** <complex.h>**Descrição:** A função **cabs()** (C99) calcula o módulo do parâmetro fornecido.**Protótipo:**

```
double cabs(double _Complex x)
```

**Parâmetro:** *x* – valor do tipo **double \_Complex** cujo módulo será calculado.**Retorno:** O módulo do parâmetro fornecido.**Exemplo:** Veja o exemplo de **creal()**.*carg()* (C99)**Incluir:** <complex.h>**Descrição:** A função **carg()** (C99) calcula o ângulo de fase do parâmetro fornecido.**Protótipo:**

```
double carg(double _Complex x)
```

**Parâmetro:** *x* – valor do tipo **double \_Complex** cujo ângulo de fase será calculado.**Retorno:** O ângulo de fase do parâmetro fornecido.**Exemplo:** Veja o exemplo de **creal()**.*cimag()* (C99)**Incluir:** <complex.h>

**Descrição:** A função **cimag()** (C99) calcula a parte imaginária do parâmetro fornecido.

**Protótipo:**

```
double cimag(double _Complex x)
```

**Parâmetro:**  $x$  – valor do tipo **double \_Complex** cuja parte imaginária será determinada.

**Retorno:** A parte imaginária do parâmetro fornecido.

**Observação:** Consulte também a descrição da função **creal()**.

**Exemplo:** Veja o exemplo de **creal()**.

*conj()* (C99)

**Incluir:** `<complex.h>`

**Descrição:** A função **conj()** (C99) calcula o conjugado do parâmetro fornecido.

**Protótipo:**

```
double _Complex conj(double _Complex x)
```

**Parâmetro:**  $x$  – valor do tipo **double \_Complex** cujo conjugado será calculado.

**Retorno:** O conjugado do parâmetro fornecido.

**Exemplo:** Veja o exemplo de **creal()**.

*cproj()* (C99)

**Incluir:** `<complex.h>`

**Descrição:** A função **cproj()** (C99) calcula a projeção de seu parâmetro na esfera de Riemann.

**Protótipo:**

```
double _Complex cproj(double _Complex x)
```

**Parâmetro:** *x* – valor do tipo **double \_Complex** cuja projeção na esfera de Riemann é calculada.

**Retorno:** A projeção na esfera de Riemann do parâmetro fornecido.

**Exemplo:** Veja o exemplo de **creal()**.

*creal()* (C99)

**Incluir:** <complex.h>

**Descrição:** A função **creal()** (C99) calcula a parte real do parâmetro fornecido.

**Protótipo:**

```
double creal(double _Complex x)
```

**Parâmetro:** *x* – valor do tipo **double \_Complex** cuja parte real será determinada.

**Retorno:** A parte real do parâmetro fornecido.

**Observação:** Consulte também a descrição da função **cimag()**.

**Exemplo:** O programa a seguir demonstra o uso das funções **cabs()**, **carg()**, **conj()**, **cproj()**, **creal()** e **cimag()**.

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

int main()
{
    double _Complex x = 2.0 - 3.0*I,
              y = -1.5 + 2.3*I,
              z;
```

```

printf( "Modulo de %3.1f + %3.1f*i = %5.3f\n",
        creal(x), cimag(x), cabs(x) );

printf( "Angulo de fase de %3.1f + %3.1f*i = "
        "%5.3f\n", creal(x), cimag(x), carg(x) );

z = conj(x);
printf( "Conjugado de %3.1f + %3.1f*i = "
        "%5.3f + %5.3f*i\n", creal(x), cimag(x),
        creal(z), cimag(z) );

z = cproj(x);
printf( "Projecao de %3.1f + %3.1f*i na esfera de "
        "Riemann = %5.3f + %5.3f*i\n",
        creal(x), cimag(x), creal(z), cimag(z) );

return 0;
}

```

## 4.5 MACROS ARITMÉTICAS GENÉRICAS: <tgmath.h> (C99)

O cabeçalho <tgmath.h> define várias **macros genéricas** utilizadas em operações matemáticas com números de ponto flutuante reais e complexos. No contexto corrente, uma macro genérica é uma macro com argumentos que, independentemente do tipo de dado (i.e., real ou complexo) dos parâmetros com os quais ela é invocada, é capaz de decidir corretamente o tipo de operação a ser efetuada. Por exemplo, uma das macros definidas neste arquivo é **cos()**, que calcula o cosseno de um número de ponto flutuante real ou complexo. Quando esta macro recebe um argumento de algum tipo primitivo de ponto flutuante real (**float**, **double** ou **long double**), ela simplesmente chama a função [**cos()**, **cosf()** ou **cosl()**], declarada em <math.h>, que executa a devida operação; por outro lado, quando esta macro recebe um argumento de algum tipo primitivo complexo (**float \_Complex**, **double \_Complex**, ou **long double \_Complex**), ela chama a função [**ccos()**, **ccosf()** ou **ccosl()**], declarada em <complex.h>, que executa a operação adequada. Todos os componentes do cabeçalho <tgmath.h> são macros com argumentos.

Examinando-se as funções declaradas em <math.h> (**Seção 3.6**) e <complex.h> (**Seção 4.4**), pode-se constatar que, de acordo com o padrão C99, cada função possui duas outras funções correspondentes que se diferenciam apenas pelo fato de

atuarem sobre tipos de dados diferentes. Ou seja, as funções cujos nomes terminam em *f* atuam sobre o tipo **float** ou **float \_Complex**, as funções cujos nomes terminam em *l* atuam sobre o tipo **long double** ou **long double \_Complex**, e as funções que não terminam em *f* ou *l* atuam sobre o tipo **double** ou **double \_Complex**<sup>32</sup>.

O cabeçalho `<tgmath.h>` inclui `<math.h>` e `<complex.h>`, porque, na realidade, o arquivo `<tgmath.h>` não acrescenta nenhuma nova operação à biblioteca padrão de C. No arquivo `<tgmath.h>`, existem apenas macros genéricas cujo único objetivo é decidir, com base nos tipos dos parâmetros com os quais uma tal macro é invocada, qual função declarada em `<math.h>` ou `<complex.h>` a macro deve chamar. Mais especificamente, estas macros aplicam as seguintes regras de decisão (em ordem de aplicação):

- Se algum dos argumentos utilizados na invocação da macro possui tipo **long double**, a macro chama a versão da função correspondente para o tipo **long double**.
- Caso contrário, se algum dos argumentos utilizados na invocação da macro possui tipo **double** ou algum tipo inteiro, a macro chama a versão da função correspondente para o tipo **double**.
- Se nenhum dos casos anteriores for aplicado, a macro chamará a versão da função correspondente para o tipo **float**.

Além dessas regras, para cada função com nome sem sufixo (*f* ou *l*) declarada no cabeçalho `<math.h>`, para a qual existe uma função correspondente declarada em `<complex.h>` com um nome semelhante começando com *c*, a macro genérica correspondente a estas duas funções tem o mesmo nome da função declarada em `<math.h>`. As macros genéricas associadas a estas funções são apresentadas na **Tabela 4-4**.

FUNÇÃO DECLARADA EM <code>&lt;math.h&gt;</code>	FUNÇÃO DECLARADA EM <code>&lt;complex.h&gt;</code>	MACRO GENÉRICA
<b>acos()</b>	<b>cacos()</b>	<b>acos()</b>
<b>asin()</b>	<b>casin()</b>	<b>asin()</b>
<b>atan()</b>	<b>catan()</b>	<b>atan()</b>
<b>acosh()</b>	<b>cacosh()</b>	<b>acosh()</b>

32 A única exceção a esta regra é a função **modf()**, cuja versão **double** tem este mesmo nome, enquanto que a correspondente versão para **float** é **modff()**.

FUNÇÃO DECLARADA EM <math.h>	FUNÇÃO DECLARADA EM <complex.h>	MACRO GENÉRICA
<b>asinh()</b>	<b>casinh()</b>	<b>asinh()</b>
<b>atanh()</b>	<b>catanh()</b>	<b>atanh()</b>
<b>cos()</b>	<b>ccos()</b>	<b>cos()</b>
<b>sin()</b>	<b>csin()</b>	<b>sin()</b>
<b>tan()</b>	<b>ctan()</b>	<b>tan()</b>
<b>cosh()</b>	<b>ccosh()</b>	<b>cosh()</b>
<b>sinh()</b>	<b>csinh()</b>	<b>sinh()</b>
<b>tanh()</b>	<b>ctanh()</b>	<b>tanh()</b>
<b>exp()</b>	<b>cexp()</b>	<b>exp()</b>
<b>log()</b>	<b>clog()</b>	<b>log()</b>
<b>pow()</b>	<b>cpow()</b>	<b>pow()</b>
<b>sqrt()</b>	<b>csqrt()</b>	<b>sqrt()</b>
<b>fabs()</b>	<b>cabs()</b>	<b>fabs()</b>

Tabela 4-4: Funções declaradas em &lt;math.h&gt; e &lt;complex.h&gt; com respectivas macros genéricas.

Quando um dos argumentos de uma macro genérica apresentada na **Tabela 4-4** é complexo, a macro invoca uma função complexa (i.e., declarada em <complex.h>) seguindo as regras enunciadas anteriormente. Caso contrário, a macro chama uma função declarada em <math.h>, seguindo as mesmas regras.

As funções declaradas em <math.h> apresentadas na **Tabela 4-5** não possuem contrapartida em <complex.h>. Para cada uma destas funções, existe uma macro genérica com o mesmo nome da função. Se todos os argumentos usados na invocação de uma dessas macros forem de tipos inteiros ou de ponto flutuante reais, a função correspondente será chamada, seguindo as regras enunciadas anteriormente. Caso contrário (i.e., se algum argumento for complexo), o resultado será indefinido.

<b>atan2()</b>	<b>fma()</b>	<b>llround()</b>	<b>remainder()</b>
<b>cbrt()</b>	<b>fmax()</b>	<b>log10()</b>	<b>remquo()</b>
<b>ceil()</b>	<b>fmin()</b>	<b>log1p()</b>	<b>rint()</b>
<b>copysign()</b>	<b>fmod()</b>	<b>log2()</b>	<b>round()</b>
<b>erf()</b>	<b>frexp()</b>	<b>logb()</b>	<b>scalbn()</b>
<b>erfc()</b>	<b>hypot()</b>	<b>lrint()</b>	<b>scalbln()</b>
<b>exp2()</b>	<b>ilogb()</b>	<b>lround()</b>	<b>tgamma()</b>
<b>expm1()</b>	<b>ldexp()</b>	<b>nearbyint()</b>	<b>trunc()</b>
<b>fdim()</b>	<b>lgamma()</b>	<b>nextafter()</b>	
<b>floor()</b>	<b>llrint()</b>	<b>nexttoward()</b>	

Tabela 4-5: Funções declaradas em <math.h> sem correspondência em <complex.h>.

As funções declaradas em <complex.h> apresentadas na **Tabela 4-6** não possuem contrapartida em <math.h>. Para cada uma destas funções, existe uma macro genérica com o mesmo nome da função. Se os argumentos usados na invocação de uma destas macros for de algum tipo de ponto flutuante real ou complexo, a função complexa correspondente será chamada, seguindo as regras enunciadas anteriormente<sup>33</sup>.

<b>carg()</b>
<b>cimag()</b>
<b>conj()</b>
<b>cproj()</b>
<b>creal()</b>

Tabela 4-6: Funções declaradas em <complex.h> sem correspondência em <math.h>.

**Exemplo:** O programa apresentado a seguir ilustra o uso das macros genéricas definidas no cabeçalho <tgmath.h><sup>34</sup>.

33 Neste caso, se o tipo do argumento for real, ele será convertido no tipo complexo correspondente tendo a parte imaginária igual a zero.

34 O *linker* que acompanha o compilador gcc requer o uso da opção `-lm` para fazer ligações de programas com funções declaradas em <tgmath.h> (v. **Seção 3.6**).

```

#include <stdio.h>
#include <tgmath.h>

#define IMPRIME(x) printf("%s = %f\n", #x, x)
#define IMPRIME_LD(x) printf("%s = %Lf\n", #x, x)

#define IMPRIME_C(x) printf("%s = %f + %f*i\n", \
                             #x, creal(x), cimag(x))
#define IMPRIME_CLD(x) printf("%s = %Lf + %Lf*i\n", \
                               #x, creal(x), cimag(x))

int main()
{
    int                i;
    float              f;
    double             d;
    long double        ld;
    float _Complex      fc;
    double _Complex     dc;
    long double _Complex ldc;

    /****** Chama: *****/
    IMPRIME(exp(i));      /* exp() em <math.h> */
    IMPRIME(asinh(f));    /* asinhf() em <math.h> */
    IMPRIME(sin(d));      /* sin() em <math.h> */
    IMPRIME_LD(atan(ld)); /* atanl() em <math.h> */
    IMPRIME_C(log(fc));   /* clogf() em <complex.h> */
    IMPRIME_C(sqrt(dc));  /* csqrt() em <complex.h> */
    IMPRIME_CLD(pow(ldc,f)); /* cpowl() em <complex.h> */
    /******

    /* Chama remainder() em <math.h> */
    IMPRIME(remainder(i, i));

    /* Chama nextafter() em <math.h> */
    IMPRIME(nextafter(d, f));

    /* Chama nexttowardf() em <math.h> */
    IMPRIME(nexttowardf(f, ld));

    /* Chama copysignl() em <math.h> */
    IMPRIME_LD(copysign(i, ld));

```



```

                                /***** Chama: *****/
IMPRIME( carg(i));             /* carg() em <complex.h> */
IMPRIME_C(cproj(f));           /* cprojf() em <complex.h> */
IMPRIME(creal(d));             /* creal() em <complex.h> */
IMPRIME_LD(cimag(ld));         /* cimagl() em <complex.h> */
IMPRIME(cabs(fc));             /* cabsf() em <complex.h> */
IMPRIME( carg(dc));            /* carg() em <complex.h> */
IMPRIME_CLD(cproj(ldc));       /* cprojl() em <complex.h> */
                                /*****

/* As chamadas a seguir produzem resultados */
/* indefinidos (problemas ã vista!) */
IMPRIME_C(ceil(fc));
IMPRIME_C(rint(dc));
IMPRIME_CLD(fmax(ldc, ld));

return 0;
}

```

Note no exemplo apresentado que as três últimas invocações de macros genéricas produzirão um resultado indefinido que depende de implementação porque as três funções [`ceil()`, `rint()` e `fmax()`] correspondentes a essas macros são declaradas em `<math.h>` e não possuem contrapartida em `<complex.h>`; por isso, essas macros não poderiam ter sido invocadas com parâmetros de tipo complexo.

## 4.6 EXERCÍCIOS DE REVISÃO

1. Como se compõe uma constante de ponto flutuante complexa em C?
2. (a) Quais são os tipos complexos primitivos requeridos pelo padrão C99? (b) Que tipos complexos primitivos são sugeridos, mas não são requeridos pelo padrão C99?
3. Descreva o significado de cada uma das macros a seguir:
  - (a) **complex**
  - (b) **\_Complex\_I**
  - (c) **I**

- (d) **imaginary**
- (e) **\_Imaginary\_I**

4. Qual é o efeito de cada diretiva **#pragma** a seguir?

- (a) `#pragma STD CX_LIMITED_RANGE ON`
- (b) `#pragma STD CX_LIMITED_RANGE OFF`

5. (a) Que sufixo possui as funções que atuam sobre valores do tipo **float \_Complex**? (b) Que sufixo possui as funções que atuam sobre valores do tipo **double \_Complex**? (c) Que sufixo possui as funções que atuam sobre valores do tipo **long double \_Complex**?

6. Descreva cada uma das seguintes funções declaradas em `<complex.h>`:

- (a) **cabs()**
- (b) **carg()**
- (c) **cimag()**
- (d) **conj()**
- (e) **cproj()**

7. Qual é o propósito do cabeçalho `<tgmath.h>`?

8. O que é uma macro genérica?

9. Por que o cabeçalho `<tgmath.h>` inclui os arquivos `<math.h>` e `<complex.h>`?

10. Que regras as macros definidas em `<tgmath.h>` utilizam para decidir qual função será chamada?

11. Em que situações a invocação de uma macro genérica produz um resultado indefinido?

# *Capítulo 5*

---

*Localização e datação*

## 5.1 INTRODUÇÃO

Este capítulo apresenta conceitos fundamentais de localização de programas e explora os seguintes cabeçalhos da biblioteca padrão de C:

- `<locale.h>` – que dá suporte à localização de programas.
- `<time.h>` – cujos componentes auxiliam o programador a lidar com datas e horários.

### 5.1.1 LOCALIDADES

Uma **localidade** especifica um conjunto de convenções para apresentação externa de dados. Com esse propósito, uma localidade provê informações sobre:

- Como caracteres multibytes devem ser interpretados (v. **Capítulo 8**).
- Como categorizar caracteres no código de caracteres corrente (v. **Seções 6.2 e 8.6**).
- A ordenação de caracteres (colação) usada na língua e no código de caracteres selecionados (v. **Seções 6.4 e 7.7**).
- Formatação de números e valores monetários (v. **Seção 5.2**).
- Formatação de datas e horas (v. **Seção 5.3**).

As informações presentes numa localidade são agrupadas em **categorias**, cada uma das quais concentrada num aspecto de formatação de dados. Isto permite que formatos de apresentação de dados em categorias diferentes sejam alterados de modo independente.. Por exemplo, pode-se alterar o formato de apresentação de data e hora independentemente do formato de apresentação de valores numéricos.

**Localizar** um programa significa adaptá-lo às características dos usuários de uma determinada região ou cultura. Idealmente, um **programa localizado** permite que seus usuários selecionem localidades durante sua execução.

Na prática, é virtualmente impossível especificar uma localidade para um país ou região independentemente de linguagem, de modo que as localidades são relacionadas principalmente a linguagens e não a culturas locais em si. Entretanto, com frequência, a seleção de localidades é apresentada ao usuário como uma questão de escolha de país ou região. Deve-se ainda salientar que o fato de duas localidades usarem a mesma

língua não implica que os mesmos formatos de informação sejam usados. Por exemplo, Inglaterra e Estados Unidos têm inglês como língua nativa, mas representam datas de modos diferentes.

## 5.1.2 IMPLEMENTAÇÃO DE LOCALIDADES EM SISTEMAS OPERACIONAIS DA FAMÍLIA UNIX

Em sistemas operacionais da família Unix, localidades são implementadas em arquivos de texto cujos formatos dependem do sistema operacional específico. **Nomes de localidades** (i.e., nomes de arquivos contendo informações de localidade) são construídos com componentes para linguagem (e.g., *pt*), sistema de escrita (e.g., *Han*), país ou território (e.g., *br*) e variante. O caractere subscrito ('\_') é usado para separar os componentes. Apenas o primeiro componente (i.e., aquele usado para a linguagem), constituído por duas ou três letras, é obrigatório. Comumente, localidades são identificadas por apenas um código de linguagem ou um código de linguagem seguido por um código de país (e.g., *pt\_BR*). Também é comum incluir um nome correspondente à codificação de caracteres utilizada, como, por exemplo, *pt\_BR.ISO8859-1*.

Resumindo, nomes de localidades usualmente são compostos de:

- Nome da linguagem, obedecendo ao padrão ISO 639-1 (e.g., *pt* representando português), seguido de sublinha e do nome do país de acordo com o padrão ISO 3166-1 (por exemplo, *BR* representando Brasil).
- Nome opcional de uma codificação de caracteres (e.g., *ISO8859-1*, representando a codificação popularmente conhecida como Latim-1); quando este nome de codificação está presente, ele é separado da parte inicial do nome da localidade por ponto.
- Outros caracteres qualificadores opcionais (e.g., *x*).

Num sistema que utilize estas regras, a localidade que representa português do Brasil, aparece como *pt\_BR.ISO-8859-1* ou *pt\_BR.utf8*.

Tipicamente, uma localidade em sistemas operacionais da família Unix contém as seguintes categorias, especificadas pelo padrão POSIX<sup>35</sup>:

---

<sup>35</sup> POSIX é um conjunto de padrões que objetivam definir uma interface padrão de sistema operacional baseada no sistema Unix. A denominação é derivada de *Portable Operating System Interface*, que pode ser traduzida como *Interface Portável entre Sistemas Operacionais*.

- `LC_CTYPE` – que contém informações sobre classificação de caracteres e conversões entre maiúsculas e minúsculas.
- `LC_COLLATE` – que define a ordem de colação de caracteres.
- `LC_MONETARY` – que especifica formatação de valores monetários.
- `LC_NUMERIC` – que especifica formatação de valores numéricos (não monetários).
- `LC_TIME` – que define formatos de data e hora.
- `LC_MESSAGES` – que especifica formatos de mensagens informativas e de diagnóstico, e respostas interativas.

Com exceção da categoria `LC_MESSAGES`, todas as categorias usadas por sistemas da família Unix possuem correspondentes em C. Além disso, a localidade padrão desses sistemas corresponde à localidade "C", que é a localidade padrão da linguagem C.

Em sistemas operacionais da família Unix, pode-se configurar qualquer destas categorias POSIX usando-se o comando `export`. Por exemplo:

```
$ export LC_ALL=br
```

Ou, usando o comando `setenv`:

```
$ setenv LC_ALL br
```

Pode-se consultar a lista de localidades disponíveis usando-se o comando:

```
$ locale -a
```

A lista resultante deste último comando pode conter uma mistura de nomes de linguagens primárias (e.g., "fr") com especificador de país (e.g., "fr\_FR") e nomes de localidade contendo o nome de uma codificação (e.g., "fr\_FR.ISO8859-1"). Para algumas linguagens, pode ser que não haja nenhuma localidade geral como, por exemplo, "br" ou "en".

Pode-se consultar o nome da codificação de caracteres na localidade corrente por meio do comando:

```
$ locale charmap
```

Se desejar obter uma lista contendo todas as codificações de caracteres instaladas, utilize o comando:

```
$ locale -m
```

Em distribuições de Linux, usualmente os arquivos de localidade encontram-se no diretório:

```
/usr/lib/locale/
```

### 5.1.3 LOCALIZAÇÃO E INTERNACIONALIZAÇÃO

**Internacionalização** (ou **globalização**) é um conceito relacionado com localização, mas estes conceitos não devem ser confundidos. Isto é, internacionalização é o processo de tornar programas portáteis entre linguagens e culturas diferentes, enquanto localização consiste em adaptar programas para linguagens ou culturas específicas. Assim, um programa internacionalizado é aquele que permite ser localizado com um mínimo de esforço (idealmente, sem necessidade de reescrita).

### 5.1.4 A BASE DE DADOS CLDR E A BIBLIOTECA ICU

**CDRL**<sup>36</sup> é um projeto do consórcio Unicode que visa disponibilizar uma base de dados de localidades pronta para uso em programas. Os dados providos pela base de dados CLDR podem ser usados por programas quando eles geram menus, mensagens de diagnóstico, relatórios, etc. A base de dados CLDR usa um formato baseado em XML contendo muitas informações de localidade, dentre as quais:

- Nomes de linguagens e scripts
- Nomes de países, territórios e continentes
- Nomes de calendários e fusos horários
- Formatos diferentes para data e hora
- Formatos de números decimais e valores monetários
- Regras de transliteração entre scripts e de colação específicas para cada linguagem

Para maiores informações sobre a base de dados CLDR consulte a página principal do projeto no endereço: <http://www.unicode.org/cldr/>.

---

<sup>36</sup> CDRL é um acrônimo derivado de *Common Locale Data Repository*.

Uma importante biblioteca que auxilia a internacionalização e localização de programas é a biblioteca ICU<sup>37</sup>. Esta biblioteca contém componentes que podem ser usados para várias finalidades, tais como: processamento de texto Unicode, conversões entre esquemas de codificações de caracteres, busca e ordenação de *strings* sensíveis a localidade (v. **Capítulo 7**), operações com datas e horas em diversos tipos de calendários e fusos horários, processamento de informações sobre localidade, etc.

A versão desta biblioteca destinada às linguagens C e C++ é denominada ICU4C e é recomendada a qualquer desenvolvedor envolvido em projetos de localização ou internacionalização. A biblioteca ICU encontra-se no site do projeto: <http://www.icu-project.org/>. É grátis!

## 5.2 LOCALIZAÇÃO DE PROGRAMAS: <locale.h>

O objetivo do cabeçalho <locale.h> é prover componentes que facilitem a localização de programas. No presente contexto, uma localidade refere-se a um conjunto de propriedades específicas de uma determinada cultura ou região (v. **Seção 5.1**).

### 5.2.1 ESTRUTURAS lconv

Uma estrutura **lconv** contém campos que descrevem como formatar valores numéricos monetários e não monetários. Esses campos são do tipo **char** – quando representam inteiros não negativos – ou **char \*** – quando representam *strings*. Um valor igual a **CHAR\_MAX** atribuído a um campo do tipo **char** ou um *string* vazio ("") atribuído a um campo do tipo **char \*** indicam que o respectivo campo não é suportado pela localidade corrente. Os campos de uma estrutura **lconv** devem ser modificados apenas por chamadas da função **setlocale()** (v. **Seção 5.2.3**).

A **Tabela 5-1** apresenta os campos de uma estrutura **lconv** acompanhados de uma breve descrição<sup>38</sup>.

CAMPO	TIPO	DESCRIÇÃO
currency_symbol	char *	Símbolo monetário usado na localidade corrente.

37 O acrônimo ICU é derivado de *International Components for Unicode*.

38 Os campos desta estrutura podem ser implementados em qualquer ordem, e não necessariamente naquele que aparece na **Tabela 5-1**.



CAMPO	TIPO	DESCRIÇÃO
<code>int_curr_symbol</code>	<b>char *</b>	<ul style="list-style-type: none"> <li>• Símbolo monetário internacional utilizado na localidade corrente de acordo com o padrão ISO 4217.</li> <li>• <i>String</i> consistindo em três caracteres que identificam o símbolo seguidos de um caractere de separação<sup>38a</sup>.</li> </ul>
<code>decimal_point</code>	<b>char *</b>	<ul style="list-style-type: none"> <li>• <i>String</i> contendo o separador de casas decimais usado com valores não monetários.</li> <li>• Este <i>string</i> não pode ser vazio.</li> </ul>
<code>mon_decimal_point</code>	<b>char *</b>	Similar a <code>decimal_point</code> , mas é usado para formatar valores monetários.
<code>frac_digits</code>	<b>char</b>	Número de dígitos apresentados após o ponto decimal em valores monetários formatados localmente.
<code>int_frac_digits</code>	<b>char</b>	Idêntico a <code>frac_digits</code> , mas é usado com valores monetários formatados internacionalmente.
<code>grouping</code>	<b>char *</b>	<ul style="list-style-type: none"> <li>• Especifica o tamanho de cada agrupamento de dígitos.</li> <li>• Não é utilizado com valores monetários.</li> <li>• A <b>Tabela 5-2</b> apresenta os possíveis valores.</li> </ul>
<code>mon_grouping</code>	<b>char *</b>	Idêntico a <code>grouping</code> , mas é utilizado com valores monetários (v. <b>Tabela 5-2</b> ).
<code>thousands_sep</code>	<b>char *</b>	<i>String</i> utilizado para separar agrupamentos de dígitos à esquerda das casas decimais para valores não monetários.
<code>mon_thousands_sep</code>	<b>char *</b>	Idêntico a <code>thousands_sep</code> , mas é usado com valores monetários.

38a Cada país possui dois símbolos monetários: um usado internamente no país (e.g., R\$) e outro usado internacionalmente (e.g., BRL). O símbolo monetário internacional (i.e., o conteúdo do campo `int_curr_symbol` de uma estrutura **lconv**) normalmente segue o padrão ISO 4217 e consiste de três caracteres seguidos de um caractere de separação. Assim, quando se imprime `int_curr_symbol` não se deve usar mais nenhum espaço adicional.

CAMPO	TIPO	DESCRIÇÃO
n_cs_precedes	char	Indica se o símbolo monetário precede a ou segue um valor monetário negativo <sup>38b</sup> : <ul style="list-style-type: none"> <li>• 1 determina que o símbolo monetário precede ao valor.</li> <li>• 0 determina que o símbolo monetário sucede ao valor.</li> </ul>
p_cs_precedes	char	Idêntico a n_cs_precedes, mas é usado com valores monetários não negativos.
int_n_cs_precedes (C99)	char	Idêntico a n_cs_precedes, mas é usado com valores monetários formatados internacionalmente.
int_p_cs_precedes (C99)	char	Idêntico a p_cs_precedes, mas é usado com valores monetários formatados internacionalmente.
n_sep_by_space	char	Indica se o símbolo monetário é separado ou não de um valor monetário negativo por um espaço: <ul style="list-style-type: none"> <li>• 1 significa que o símbolo monetário será separado do valor por um espaço em branco.</li> <li>• 0 significa que não haverá nenhum espaço de separação.</li> <li>• 2 (C99) indica que um espaço separa o símbolo do sinal adjacente, ou um espaço separa o sinal não adjacente do valor.</li> </ul>
p_sep_by_space	char	Idêntico a n_sep_by_space, mas é usado com valores monetários não negativos.

38b Símbolos monetários variam não apenas em termos dos caracteres usados mas também na posição relativa ao valor onde são colocados. Isto é, símbolos monetários podem ser colocados antes, depois ou no interior do valor.

CAMPO	TIPO	DESCRIÇÃO
<code>int_n_sep_by_space (C99)</code>	<b>char</b>	<ul style="list-style-type: none"> <li>• Idêntico a <code>n_sep_by_space</code>, mas é usado com valores monetários formatados internacionalmente.</li> <li>• O espaço é definido por <code>int_curr_symbol[3]</code>.</li> </ul>
<code>int_p_sep_by_space (C99)</code>	<b>char</b>	<ul style="list-style-type: none"> <li>• Idêntico a <code>p_sep_by_space</code>, mas é usado com valores monetários formatados internacionalmente.</li> <li>• O espaço é definido por <code>int_curr_symbol[3]</code>.</li> </ul>
<code>n_sign_posn</code>	<b>char</b>	Especifica formatos para valores monetários negativos (v. <b>Tabela 5-3</b> ).
<code>p_sign_posn</code>	<b>char</b>	Idêntico a <code>n_sign_posn</code> , mas é usado com valores monetários não negativos (v. <b>Tabela 5-3</b> ).
<code>int_n_sign_posn (C99)</code>	<b>char</b>	Idêntico a <code>n_sign_posn</code> , mas é usado com valores monetários internacionais.
<code>int_p_sign_posn (C99)</code>	<b>char</b>	Idêntico a <code>p_sign_posn</code> , mas é usado com valores monetários internacionais.
<code>negative_sign</code>	<b>char *</b>	<i>String</i> que assinala valores monetários negativos.
<code>positive_sign</code>	<b>char *</b>	<i>String</i> que assinala valores monetários não negativos.

Tabela 5-1: Campos de estruturas `lconv`.

Os campos `decimal_point`, `grouping` e `thousands_sep` da estrutura **`lconv`** são afetados pela categoria de localidade **`LC_NUMERIC`**, enquanto os demais campos são afetados pela categoria **`LC_MONETARY`** (v. **Seção 5.2.2**).

Os campos `grouping` e `mon_grouping` são interpretados conforme mostra a **Tabela 5-2**.

CONTEÚDO DO ARRAY	INTERPRETAÇÃO
String vazio ("")	Não haverá agrupamento.
Elemento com valor <b><code>CHAR_MAX</code></b>	Encerra qualquer agrupamento posterior (i.e., nenhum agrupamento adicional é efetuado).

CONTEÚDO DO ARRAY	INTERPRETAÇÃO
Caractere '\0'	O elemento anterior é usado indefinidamente para os dígitos restantes.
Outro valor	Indica quantos dígitos constituem o agrupamento corrente. O próximo elemento determina o tamanho do próximo agrupamento de dígitos antes do agrupamento corrente.

Tabela 5-2: Campos `grouping` e `mon_grouping` de estruturas `lconv`.

Os agrupamentos especificados pelos campos `grouping` e `mon_grouping` são considerados da direita para a esquerda, como mostram os seguintes exemplos:

- O array `{3, 2, CHAR_MAX}` representa (da direita para a esquerda) um agrupamento de três dígitos, seguido de um agrupamento de dois dígitos, seguido dos dígitos restantes não agrupados (e.g., 9876.54.321).
- O *string* `"\3"` significa que o número será dividido em grupos repetidos de três dígitos (e.g., 987.654.321).
- O *string* `"\4\3\2"` significa que o número será dividido, da direita para a esquerda, num grupo de quatro dígitos, seguido de um grupo de três dígitos, seguido por grupos repetidos de dois dígitos (e.g., 12.34.98.765.4321).

A **Tabela 5-3** mostra como os valores dos campos `p_sign_posn` e `n_sign_posn` de uma estrutura `lconv` são interpretados.

VALOR	INTERPRETAÇÃO	EXEMPLOS
0	Parênteses envolvem o valor e o símbolo monetário.	(R\$2,00) (2,00R\$)
1	O <i>string</i> de sinal precede ao valor e ao símbolo monetário.	−R\$2,00 −2,00R\$
2	O <i>string</i> de sinal sucede ao símbolo monetário.	R\$2,00− 2,00R\$−
3	O <i>string</i> de sinal imediatamente precede ao símbolo monetário.	−R\$2,00 2,00−R\$
4	O <i>string</i> de sinal imediatamente sucede ao símbolo monetário.	R\$−2,00 2,00R\$−

Tabela 5-3: Campos `p_sign_posn` e `n_sign_posn` de estruturas `lconv`.

A **Tabela 5-4** apresenta os valores assumidos pelos campos de uma estrutura **lconv** na localidade padrão "C" e numa localidade nacional (e.g., "pt\_BR.utf8" no Linux).

CAMPO	LOCALIDADE "C"	LOCALIDADE NACIONAL
currency_symbol	" "	"R\$"
int_curr_symbol	" "	"BRL"
decimal_point	". "	", "
mon_decimal_point	" "	", "
frac_digits	<b>CHAR_MAX</b>	2
int_frac_digits	<b>CHAR_MAX</b>	2
grouping	" "	"\3"
mon_grouping	" "	"\3"
thousands_sep	" "	". "
mon_thousands_sep	" "	". "
n_cs_precedes	<b>CHAR_MAX</b>	1
p_cs_precedes	<b>CHAR_MAX</b>	1
int_n_cs_precedes	<b>CHAR_MAX</b>	1
int_p_cs_precedes	<b>CHAR_MAX</b>	1
n_sep_by_space	<b>CHAR_MAX</b>	0
p_sep_by_space	<b>CHAR_MAX</b>	0
int_n_sep_by_space	<b>CHAR_MAX</b>	0
int_p_sep_by_space	<b>CHAR_MAX</b>	0
n_sign_posn	<b>CHAR_MAX</b>	4
p_sign_posn	<b>CHAR_MAX</b>	4
int_n_sign_posn	<b>CHAR_MAX</b>	4
int_p_sign_posn	<b>CHAR_MAX</b>	4
negative_sign	" "	"_ "
positive_sign	" "	"+"

Tabela 5-4: Campos de estruturas lconv em localidades padrão e nacional

Note na **Tabela 5-4** que, na localidade "C", os valores **CHAR\_MAX** de alguns campos do tipo **char** (e.g., o campo `frac_digits`) e os valores " " de alguns campos do tipo **char** \* (e.g., o campo `negative_sign`) indicam que os respectivos campos são indefinidos nessa localidade.

## 5.2.2 MACROS

Em C, bem como em outras linguagens e sistemas operacionais, informações de localidade são divididas em categorias (v. **Seção 5.1.1**). As categorias de localidade presentes na biblioteca padrão de C são representadas pelas macros: **LC\_ALL**, **LC\_COLLATE**, **LC\_CTYPE**, **LC\_MONETARY**, **LC\_NUMERIC** e **LC\_TIME**. Cada uma destas macros tem a mesma interpretação da correspondente categoria especificada pelo padrão POSIX apresentada na **Seção 5.1.1**.

O fato de informações sobre localidade serem divididas em categorias é importante porque pode ser que um determinado programa necessite alterar configurações de localidade para apenas uma categoria. Por exemplo, um programa que lida com valores monetários em culturas diversas pode alterar a categoria **LC\_MONETARY** sem alterar as demais categorias de localidade sempre que precisar.

Usando-se a função **setlocale()** (v. **Seção 5.2.3**), pode-se atribuir uma localidade a todas as categorias, representadas pela macro **LC\_ALL**, ou para uma categoria específica, representada por uma das demais macros apresentadas anteriormente. Cada categoria de localidade afeta o comportamento de algumas funções da biblioteca padrão de C, como mostra a **Tabela 5-5**.

CATEGORIA	AFETA...
<b>LC_ALL</b>	Todas as funções mencionadas nesta tabela.
<b>LC_COLLATE</b>	As funções <b>strcoll()</b> , <b>strxfrm()</b> (v. <b>Seção 6.4</b> ), <b>wscoll()</b> e <b>wcsxfrm()</b> (v. <b>Seção 8.5.5</b> ).
<b>LC_CTYPE</b>	As funções que classificam e transformam caracteres, com exceção das funções <b>isdigit()</b> , <b>isxdigit()</b> , <b>iswdigit()</b> e <b>iswxdigit()</b> (v. <b>Seções 6.2 e 8.6</b> ).
<b>LC_MONETARY</b>	Os formatos monetários retornados pela função <b>localeconv()</b> .
<b>LC_NUMERIC</b>	Funções que precisam interpretar o caractere utilizado como ponto decimal; i.e funções das famílias <b>scanf</b> e <b>printf</b> (v. <b>Seções 10.7.6 e 10.7.7</b> ), e funções de conversão de <i>strings</i> em números (v. <b>Seções 6.5 e 8.5.6</b> ).
<b>LC_TIME</b>	Funções que formatam datas e horas: <b>strftime()</b> , <b>mktime()</b> (v. <b>Seção 5.3.3</b> ) e <b>wcsftime()</b> (v. <b>Seção 8.5.5</b> ).

Tabela 5-5: Funções da biblioteca padrão afetadas por categorias de localidade

### 5.2.3 FUNÇÕES

*localeconv()*

**Incluir:** <locale.h>

**Descrição:** A função **localeconv()** fornece informações sobre formatos numéricos e monetários da localidade corrente.

**Protótipo:**

```
struct lconv *localeconv(void)
```

**Retorno:** Um ponteiro para uma estrutura **lconv** preenchida com valores apropriados de localização.

**Observações:**

- Os valores da estrutura retornada por esta função serão modificados sempre que uma chamada de **setlocale()** (v. adiante) modificar a localidade atribuída à categoria **LC\_MONETARY** ou **LC\_NUMERIC**.
- A estrutura cujo endereço é retornado pela função **localeconv()** não deve ter seu conteúdo alterado diretamente, mas ela pode ser alterada por uma chamada subsequente de **localeconv()** ou **setlocale()**. O padrão ISO de C garante que nenhuma outra função da biblioteca padrão altere o valor desta estrutura.
- Consulte também a descrição da função **setlocale()**.

**Exemplo:**

```
#include<locale.h>
#include<stdio.h>
#include<limits.h> /* Define CHAR_MAX */

/****
 *
 * Função ImprimeAgrupamento(): Imprime o conteúdo do
 *                               campo grouping ou
 *                               mon_grouping de uma
 *                               estrutura lconv
 *
```

```

* Argumentos: ar (entrada) - array contendo o agrupamento
*
* Retorno: Nada
*
* Observação: Nem sempre o agrupamento (ar) é um string.
*             Por isso, esta função é necessária.
*
****/
void ImprimeAgrupamento(char *ar)
{
    while (*ar != '\0' && *ar != CHAR_MAX)
        printf("%d ", *ar++);

    printf("\n");
}

int main(void)
{
    struct lconv *ptrLocal;
    char          *localidadeCorrente;

    localidadeCorrente = setlocale(LC_ALL, "");

    if (!localidadeCorrente) {
        printf("Erro na chamada da funcao setlocale()");
        return 1;
    }

    printf( "\nA localidade corrente e': %s\n",
            localidadeCorrente );

    ptrLocal = localeconv();

    printf( "\nSeparador de Casa Decimal:      %s\n",
            ptrLocal->decimal_point );
    printf( "Separador de Milhares:            %s\n",
            ptrLocal->thousands_sep );
    printf("Agrupamento de digitos:          ");
    ImprimeAgrupamento(ptrLocal->grouping);

    printf("\n***** Convencoes Monetarias *****\n\n");

    printf( "Simbolo Monetario:                  %s\n",

```



```

        ptrLocal->currency_symbol );
printf( "Simbolo Monetario Internacional:  %s\n",
        ptrLocal->int_curr_symbol);
printf( "Separador de Casa Decimal:        %s\n",
        ptrLocal->mon_decimal_point );
printf( "Separador de Milhares:            %s\n",
        ptrLocal->mon_thousands_sep );
printf("Agrupamento Monetario:            ");
ImprimeAgrupamento(ptrLocal->mon_grouping);
printf( "Sinal Positivo:                   %s\n",
        ptrLocal->positive_sign );
printf( "Sinal Negativo:                   %s\n",
        ptrLocal->negative_sign );
printf( "Casas Decimais:                   %i\n",
        ptrLocal->frac_digits );
printf( "Casas Decimais (Internacional):    %i\n",
        ptrLocal->int_frac_digits );
printf( "Sinal Positivo Precede Valor:      %i\n",
        ptrLocal->p_cs_precedes );
printf( "Sinal Positivo Separado p espaco:  %i\n",
        ptrLocal->p_sep_by_space );
printf( "Sinal Negativo Precede Valor:      %i\n",
        ptrLocal->n_cs_precedes );
printf( "Sinal Negativo Separado p espaco:  %i\n",
        ptrLocal->n_sep_by_space );
printf( "Sinal Positivo Posn:               %i\n",
        ptrLocal->p_sign_posn );
printf( "Sinal Negativo Posn:               %i\n",
        ptrLocal->n_sign_posn );

return 0;
}

```

*setlocale()*

**Incluir:** <locale.h>

**Descrição:** A função **setlocale()** seleciona ou consulta uma localidade associada a uma categoria representada por uma das macros apresentadas na **Seção 5.2.2**.

**Protótipo:**

<pre>char *setlocale(int categoria, const char *local)</pre>
--

**Parâmetros:**

- *categoria* – categoria a ser modificada ou consultada. Os valores possíveis para este argumento são as macros apresentadas na **Seção 5.2.2**.
- *local* – *string* correspondente a uma localidade que será atribuída à categoria representada pelo primeiro argumento:
  - O *string* "C" corresponde à localidade padrão.
  - Se o *string* vazio ("") for usado, a localidade de uso corrente pelo sistema operacional será utilizada.
  - Se a macro **NULL** for usada, a função apenas consultará o valor corrente da localidade estabelecida para a categoria representada pelo primeiro argumento.
  - Outros valores para este parâmetro (i.e., nomes de localidades) dependem do sistema operacional usado.

**Retorno:**

- Um *string* que representa a nova localidade para a categoria especificada, se o segundo argumento não for **NULL** e a seleção de localidade obtiver êxito.
- Um *string* que representa a localidade associada à categoria especificada, se o segundo argumento for **NULL** e a consulta obtiver êxito.
- **NULL**, se a seleção ou consulta de localidade não obtiver êxito.

**Observações:**

- Quando **setlocale()** não é capaz de selecionar uma nova localidade, a localidade corrente não é afetada.
- Um *string* retornado por **setlocale()** pode ser usado como segundo parâmetro desta função quando ela é chamada novamente. Isto permite guardar e restaurar localidades.
- O *string* retornado por **setlocale()** pode ser sobrescrito por uma chamada subsequente desta função. Assim, se o programador desejar guardar este *string* para uso posterior, ele deverá fazer uma cópia do *string*.

- Quando um nome de localidade inválido (i.e., desconhecido pelo sistema) é utilizado numa chamada da função **setlocale()**, ela retorna **NULL**.
- De acordo com o padrão ISO, qualquer programa escrito em C é iniciado usando a localidade "C". Para usar a localidade especificada pelo sistema operacional, o programador deve chamar a função **setlocale()** como:

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

- Usar a localidade especificada pelo usuário no sistema operacional, em vez de especificar alguma localidade explicitamente por nome, é sempre a opção mais portátil, visto que diferentes sistemas utilizam diferentes esquemas para lidar com localidades. Por exemplo, a localidade brasileira é representada por "Portuguese\_Brazil.1252" no sistema Windows XP, por "Portuguese\_Brazil.850" no sistema Windows 2000 e por "pt\_BR.utf8" no sistema Linux.
- Em compiladores e sistemas operacionais que suportam apenas a localidade padrão "C", a função **setlocale()** não tem efeito nenhum.

### Exemplo:

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

#define LINUX_LOCALE "pt_BR.utf8"
#define WIN_LOCALE   "Portuguese_Brazil.850"

int main(void)
{
    char *localidade;

    /* Retorna um string que especifica */
    /* a localidade corrente             */
    localidade = setlocale(LC_COLLATE, NULL);

    if (localidade)
        printf( "A localidade corrente e': %s\n",
                localidade );

    /* Funciona com Linux Ubuntu 8.10 */
    localidade = setlocale(LC_ALL, LINUX_LOCALE);
```

```

if (localidade)
    printf( "A localidade foi alterada para: %s\n",
           localidade );
else
    printf("Nao foi possivel alterar a localidade\n");

return 0;
}

```

### 5.3 DATAS E HORAS: <time.h>

O cabeçalho <time.h> provê componentes que auxiliam o programador a lidar com datas e horas.

Tempo em C é medido como o número de segundos decorridos desde um instante específico denominado **época**. Usualmente, este instante coincide com o início da (assim denominada) **era Unix**, definida como 00:00:00 do dia primeiro de janeiro de 1970. Assim, instantes que ocorrem após essa referência (e.g., 8:00 do dia 21 de dezembro de 2001) são considerados positivos, enquanto que instantes que ocorrem antes dessa referência (e.g., 5:10:20 do dia 21 de dezembro de 1969) são considerados negativos.

A medição de tempo em C, assim como em Unix, leva em consideração o fato de cada dia possuir exatamente 86400 segundos<sup>39</sup>. Deste modo, o instante 00:00:00 do dia 21 de dezembro de 1971, que ocorreu 719 dias após o instante de referência, é representado pelo número inteiro de segundos:

$$719 * 86400 = 62121600$$

Como outro exemplo de cálculo de tempo em C, considere como seria representado o instante 05:15:10 do dia 31 de dezembro de 1969, que ocorreu 366 dias antes do início da era Unix:

$$-366 * 86400 + 5 * 86400 + 15 * 60 + 10 = -31189500 \text{ segundos}$$

---

<sup>39</sup> Surpreendentemente para alguns, o número de segundos em um dia não é constante, mas oscila de acordo com alterações no movimento de rotação da terra. Uma discussão detalhada sobre o assunto está obviamente bem além do escopo deste livro, mas é importante lembrar que, devido a este fato, as medições de tempo, tanto em C, quanto no sistema Unix, não são muito precisas. Este rigor só é importante quando a contagem de tempo requerer precisão de segundos e a contagem for regressiva, pois não há como prever exatamente que a rotação da terra vá sofrer ou não uma alteração em tempo futuro. Consulte uma bibliografia adequada se este tipo de precisão for realmente desejada.

### 5.3.1 TIPOS

*clock\_t*

**Incluir:** `<time.h>`

**Descrição:** **clock\_t** é o tipo aritmético do valor retornado pela função **clock()** e que representa um intervalo de tempo de processamento da CPU. Em geral, este tipo é definido como **unsigned long**.

*size\_t*

**Incluir:** `<time.h>`, `<string.h>`, `<wchar.h>`, `<stdlib.h>` ou `<std-def.h>`

**Descrição:** **size\_t** é um tipo inteiro, sem sinal, que representa o tipo do valor resultante da aplicação do operador **sizeof** (v. **Volume I**). Este tipo é definido em vários cabeçalhos (v. **Seção 1.7.1**).

*time\_t*

**Incluir:** `<time.h>`

**Descrição:** **time\_t** é o tipo aritmético do valor retornado pelas funções **time()** e **mktime()**, e que representa data e hora.

**Observação:** É interessante notar que, se o tipo **time\_t** for inteiro e ocupar 32 bits, o intervalo de tempo coberto em tal implementação será de 136 anos. Neste caso, a data mais antiga, que pode ser representada, é 13 de dezembro de 1901, e, a mais recente, 18 de janeiro de 2038.

*tm*

**Incluir:** `<time.h>` ou `<wchar.h>`

**Descrição:** Uma estrutura do tipo **struct tm** contém membros que podem armazenar várias informações sobre um dado instante de tempo. O tipo **struct tm** é assim declarado:

```
struct tm {
    int tm_sec;
    int tm_min;
```

```

int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};

```

A **Tabela 5-6** a seguir apresenta uma descrição de cada campo de uma estrutura **tm**, bem como seus possíveis valores.

CAMPO	DESCRIÇÃO	VALORES
tm_sec	Número de segundos decorridos desde o início do minuto.	0 – 60
tm_min	Número de minutos decorridos desde o início da hora.	0 – 59
tm_hour	Número de horas desde a meia-noite do respectivo dia.	0 – 23
tm_mday	Dia do mês.	1 – 31
tm_mon	Número de meses desde janeiro.	0 – 11
tm_year	Número de anos desde 1900.	0 –
tm_wday	Número de dias desde domingo.	0 – 6
tm_yday	Número de dias desde 1º de janeiro.	0 – 365
tm_isdst	Sinalizador de uso de horário de verão (DST <sup>40</sup> ).	<ul style="list-style-type: none"> <li>• 0 – não está em uso</li> <li>• Positivo – está em uso</li> <li>• Negativo – indefinido</li> </ul>

Tabela 5-6: Descrição dos campos de uma estrutura tm.

**Observações:**

- Note que **tm** é um rótulo de estrutura e não um tipo (v. **Capítulo 9** do **Volume I**).
- Este rótulo também é declarado em `<wchar.h>` (v. **Seção 8.5.1**).

---

<sup>40</sup> DST significa, em inglês, Daylight Savings Time, o que, em português do Brasil, é equivalente a *horário de verão*.

## 5.3.2 MACROS

### *NULL*

**Incluir:** `<time.h>`, `<string.h>`, `<wchar.h>`, `<stdlib.h>` ou `<std-def.h>`

**Descrição:** A macro **NULL** representa um ponteiro nulo. Esta macro é definida em vários cabeçalhos (v. **Seção 1.7.2**).

### *CLOCKS\_PER\_SEC*

**Incluir:** `<time.h>`

**Descrição:** A macro **CLOCKS\_PER\_SEC** expande-se no número de tiques retornados pela função **clock()** em um segundo.

### **Exemplo:**

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

int main(void)
{
    clock_t inicio, fim;
    int      n;

    /* Marca o início do tempo */
    inicio = clock();

    /* Pedir para o usuário introduzir um número */
    printf("Digite um numero inteiro: ");
    scanf("%d", &n);

    /* Encerra a contagem de tempo */
    fim = clock();

    /* Exibe o tempo */
    printf("Voce levou %f segundos para escrever esse numero.",
          (float) (fim - inicio) / CLOCKS_PER_SEC);
```

```
    return 0;
}
```

### 5.3.3 FUNÇÕES

*asctime()*

**Incluir:** <time.h>

**Descrição:** A função **asctime()** converte a data e a hora armazenadas numa estrutura do tipo **struct tm** num *string*.

**Protótipo:**

```
char *asctime(const struct tm *p)
```

**Parâmetros:** *p* – Um ponteiro para uma estrutura do tipo **struct tm** adequadamente iniciada.

**Retorno:** Um *string* de comprimento 25 contendo a data e a hora num formato como o seguinte:

```
"Tue Dez 21 20:15:00 1999\n"
```

**Observações:**

- Pode-se iniciar o parâmetro desta função utilizando-se **localtime()** (v. exemplo adiante).
- O *string* retornado por **asctime()** é uma variável estática que é sobrescrita cada vez que esta função é chamada. Portanto, se necessário, deve-se fazer uma cópia deste *string* para uso futuro.

**Exemplo:** Veja exemplo da função **localtime()**.

*clock()*

**Incluir:** <time.h>



**Descrição:** A função **clock()** fornece o número de tiques (tempo de CPU) decorridos desde que o programa começou a ser executado.

**Protótipo:**

`clock_t clock(void)`

**Retorno:** Tiques de CPU desde o início de execução do programa; -1, se este valor não puder ser calculado.

**Observação:** Para determinar o intervalo de tempo decorrido em segundos, divida o valor retornado por esta função pela macro **CLOCKS\_PER\_SEC** (v. **Seção 5.3.2**).

**Exemplo:**

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

void Dorme(double tempoDeSono)
{
    double t1, t0;

    if (tempoDeSono <= 0)
        return;

    t0 = (double) clock() / CLOCKS_PER_SEC;

    printf("O programa esta' dormindo\n");

    do {
        t1 = (double) clock() / CLOCKS_PER_SEC;
        printf("%f\n", t1 - t0); /* Apenas para teste */
    } while ((t1 - t0) < tempoDeSono);
}

int main(void)
{
    double t1, t0 = (double) clock() / CLOCKS_PER_SEC;

    Dorme(4.5); /* Dorme durante 4.5 segundos */
}
```

```

t1 = (double) clock() / CLOCKS_PER_SEC;

printf("O programa dormiu %f segundos\n", t1 - t0);

return 0;
}

```

## *ctime()*

**Incluir:** <time.h>

**Descrição:** A função **ctime()** converte em *string* a data e a hora armazenadas na variável cujo endereço é recebido como argumento.

**Protótipo:**

```
char *ctime(const time_t *pSegundos)
```

**Parâmetros:** *pSegundos* – ponteiro para uma variável do tipo **time\_t** apropriadamente iniciada [por exemplo, usando **time()**].

**Retorno:** *String* formatado como, por exemplo:

```
"Tue Dez 21 15:17:00 1999\n"
```

**Observações:**

- O *string* retornado por essa função tem o mesmo formato que aquele retornado pela função **asctime()**. Esta coincidência não é à toa, pois uma chamada dessa função, como **ctime(&t)**, onde *t* é uma variável do tipo **time\_t**, é equivalente à chamada de **asctime()**:

```
asctime(localtime(&t));
```

- O *string* retornado por **ctime()** é uma variável estática que é sobrescrita cada vez que esta função ou **asctime()** é chamada.
- Consulte também a descrição de **asctime()**.

**Exemplo:**

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

```
#include <time.h>

int main(void)
{
    time_t t;

    time(&t);
    printf("Data e hora correntes: %s\n", ctime(&t));
    return 0;
}
```

### *difftime()*

**Incluir:** <time.h>

**Descrição:** A função **difftime()** calcula o intervalo de tempo, em segundos, decorrido entre dois instantes especificados.

**Protótipo:**

```
double difftime(time_t tempoInicial, time_t tempoFinal)
```

**Parâmetros:**

- tempoInicial – instante inicial.
- tempoFinal – instante final.

**Retorno:** Intervalo de tempo, em segundos, entre os instantes inicial e final.

**Exemplo:**

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

int main(void)
{
    time_t t0 = time(NULL);
    time_t t1;

    printf( "\nAguarde alguns instantes e depois "
           "digite alguma tecla " );
```

```

getchar();

t1 = time(NULL);

printf( "\nVoce aguardou %5.2f segundos\n",
        difftime(t1, t0) );

return 0;
}

```

### *gmtime()*

**Incluir:** <time.h>

**Descrição:** A função **gmtime()** converte data e hora em GMT<sup>41</sup>.

**Protótipo:**

```
struct tm *gmtime(const time_t *t)
```

**Parâmetro:** *t* – ponteiro para uma variável do tipo **time\_t** representando um instante.

**Retorno:** Um ponteiro para uma estrutura do tipo **struct tm**. Esta estrutura é uma variável estática que é sobrescrita a cada chamada de **gmtime()**.

### **Observações:**

- Muito frequentemente, o valor utilizado como parâmetro desta função é a data e a hora correntes.
- Consulte também as descrições de **time()**, **localtime()** e **asctime()**.

### **Exemplo:**

```

#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t t;

```

---

<sup>41</sup> GMT é derivado de *Greenwich Mean Time*.

```

struct tm *gmt, *localT;

t = time(NULL); /* Obtém a hora atual */

/* Obtém e imprime a data e a */
/* hora no fuso horário local */
localT = localtime(&t);
printf("\nData e hora locais:\t%s", asctime(localT));

/* Obtém e imprime a data e a */
/* hora no fuso horário GMT */
gmt = gmtime(&t);
printf("Data e hora GMT:\t%s", asctime(gmt));

return 0;
}

```

### *localtime()*

**Incluir:** <time.h>

**Descrição:** A função **localtime()** converte o argumento recebido do tipo **time\_t** numa estrutura do tipo **struct tm** e retorna um ponteiro para o resultado da conversão.

**Protótipo:**

```
struct tm *localtime(const time_t *dataEHora)
```

**Parâmetros:** **dataEHora** – ponteiro para uma variável do tipo **time\_t**.

**Retorno:** Ponteiro para uma estrutura do tipo **struct tm** resultante da conversão.

**Observações:**

- Essa função retorna um ponteiro para uma estrutura que é uma variável estática e é sobrescrita a cada chamada da função.
- Consulte também as descrições de **time()** e **gmtime()**.

**Exemplo:** O programa a seguir demonstra o uso das funções **localtime()** e **asctime()**.

```

#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t hora;
    struct tm *tmHora;

    /* Obtém a data e a hora atuais */
    hora = time(NULL);

    /* Converte a data e a hora numa estrutura tm */
    tmHora = localtime(&hora);

    printf("Data e hora correntes: %s", asctime(tmHora));

    return 0;
}

```

### *mktime()*

**Incluir:** <time.h>

**Descrição:** A função **mktime()** calcula o intervalo de tempo decorrido desde a época usada pela implementação até o instante especificado pela estrutura do tipo **struct tm** recebida como argumento.

**Protótipo:**

`time_t mktime(struct tm *t)`

**Parâmetros:** **t** – ponteiro para uma estrutura **tm**.

**Retorno:** O número de segundos decorridos desde a época usada pela implementação até o instante especificado como argumento, se a função obtém êxito; -1, caso contrário.

**Observações:**

- Se os campos da estrutura recebida como argumento não tiverem valores apropriados, eles serão ajustados.

- A data e a hora são formatadas de acordo com a localidade correntemente atribuída à categoria **LC\_TIME**.

### Exemplo:

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

/* Macro utilizada para          */
/* verificar entrada de dados */
#define ASSEGURA( x, msg ) if (!(x) ) {\
                                printf("\n%s", #msg "\n"); \
                                exit(1); \
                                }

char *diaDaSemana[] = {
    "Domingo", "Segunda", "Terça",
    "Quarta", "Quinta", "Sexta",
    "Sabado", "Invalido"
};

int main(void)
{
    struct tm t;
    int ano, mes, dia, teste;

    ano = mes = dia = 0;

    /* Introduz dia, mes e ano */
    printf("\nAno a partir de 1970: ");
    teste = scanf("%d", &ano);
    ASSEGURA(teste && (ano >= 1970), Entrada incorreta);

    printf("Mes: ");
    teste = scanf("%d", &mes);
    ASSEGURA( teste && (mes >= 1) && (mes <= 12),
        Entrada incorreta);

    printf("Dia: ");
    teste = scanf("%d", &dia);
    ASSEGURA( teste && (dia >= 1) && (dia <= 31),
        Entrada incorreta);
```

```

        /* Preenche a estrutura t com os dados */
        t.tm_year = ano - 1900;
        t.tm_mon  = mes - 1;
        t.tm_mday = dia;
        t.tm_hour = 0;
        t.tm_min  = 0;
        t.tm_sec  = 1;
        t.tm_isdst = -1;

        /* Preenche o dia da semana */
        /* da estrutura t com mktime */
        if (mktime(&t) == -1)
            t.tm_wday = 7;

        /* Imprime o dia da semana */
        printf( "\nO dia da semana e' %s\n",
                diaDaSemana[t.tm_wday] );

        return 0;
    }

```

Exemplo de execução do programa anterior:

*Ano a partir de 1970: 2007*

*Mes: 12*

*Dia: 25*

*O dia da semana e' Terca*

*strftime()*

**Incluir:** <time.h>

**Descrição:** A função **strftime()** converte num *string* a data e a hora recebidas como argumento, de acordo com um formato especificado e limitando o número de caracteres armazenados.



**Protótipo:**

```
size_t strftime( char *restrict s, size_t max,
                const char *restrict formato,
                const struct tm *restrict t )
```

**Parâmetros:**

- `s` – array que receberá o *string* contendo a data e a hora formatadas.
- `max` – número máximo de caracteres armazenados em `s`.
- `formato` – *string* de formatação (v. **Apêndice C**).
- `t` – ponteiro para uma estrutura **tm** contendo informação sobre data e hora.

**Retorno:** O número de caracteres armazenados no array, quando a função obtém êxito; zero, se ocorrer algum erro ou se o número de caracteres requeridos é maior do que `max`.

**Observações:**

- Todos os caracteres que não constituem especificadores de formato no argumento `formato` são copiados para o *string* resultante sem modificação.
- Todos os especificadores de formato que podem ser usados no *string* de formatação (terceiro argumento) desta função são apresentados no **Apêndice C**.

**Exemplo:**

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

int main()
{
    time_t      agora;
    struct tm *tLocal;
    char        strHora[80] = "";

    time(&agora);
    tLocal = localtime(&agora);
```

```

    if (strftime(strHora, 78,
                "Data: %a, %d %b %Y %T %z\n", tLocal))
        printf("\n%s", strHora);
    else
        return 1;

    return 0;
}

```

Resultado de uma execução do programa anterior no Windows XP:

*Data: Wed, 27 May 2009 Hora oficial do Brasil*

*time()*

**Incluir:** <time.h>

**Descrição:** A função **time()** fornece o número de segundos decorridos desde a época de referência medido em UTC<sup>42</sup>.

**Protótipo:**

```
time_t time(time_t *segundos)
```

**Parâmetros:** segundos – ponteiro para uma variável do tipo **time\_t** que, se não for **NULL**, também receberá o valor retornado.

**Retorno:** O número de segundos decorridos desde a época de referência.

**Observações:**

- Usualmente, a época de referência utilizada é 00:00:00 (GMT) do dia 1º de janeiro de 1970.
- Frequentemente, o argumento da função **time()** é **NULL**, como mostra o próximo exemplo.
- Consulte também as descrições de **time()**, **localtime()** e **strftime()**.

---

<sup>42</sup> A sigla UTC significa *Tempo Universal Coordenado* e não é derivada de palavras em nenhuma língua específica.

**Exemplo:** O programa a seguir demonstra o uso das funções **time()**, **localtime()** e **strftime()**.

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

int main(void)
{
    struct tm      *agora;
    time_t         segundos;
    char           str[80];

    time(&segundos);
    agora = localtime(&segundos);
    strftime( str, 80,
              "Passam %M minutos das %I horas (%z) "
              "%A, %B %d 20%y", agora );
    printf("%s\n", str);

    strftime(str, 80, "%Ex", agora);

    printf("%s\n", str);

    return 0;
}
```

## 5.4 EXERCÍCIOS DE REVISÃO

1. (a) O que é localização de programas? (b) Qual é a diferença entre localização e internacionalização de programas? (c) No contexto de localização de programas, o que é uma localidade?
2. Qual é a utilidade prática da base de dados CLDR?
3. Descubra e descreva como se alteram informações de localidade nos sistemas operacionais (a) Linux e (b) Windows.
4. Como se podem listar todas as localidades disponíveis no sistema Linux?
5. Descreva em linhas gerais o cabeçalho `<locale.h>`.
6. Para que serve uma estrutura do tipo **struct lconv**?

7. Por que localidades são divididas em categorias?
8. (a) Descreva as categorias de localidade representadas pelas macros a seguir.  
(b) Que funções da biblioteca padrão de C são afetadas por cada uma destas categorias de localidade?
  - (i) **LC\_ALL**
  - (ii) **LC\_COLLATE**
  - (iii) **LC\_CTYPE**
  - (iv) **LC\_MONETARY**
  - (v) **LC\_NUMERIC**
  - (vi) **LC\_TIME**
9. Descreva em linhas gerais a função **localeconv()** e apresente uma chamada típica desta função.
10. Qual é o efeito de cada uma das seguintes chamadas de **setlocale()**?
  - (a) `setlocale(LC_ALL, "C");`
  - (b) `setlocale(LC_ALL, "");`
  - (c) `setlocale(LC_ALL, NULL);`
11. Qual é o valor retornado em cada chamada de **setlocale()** do exercício anterior?
12. (a) No contexto de medição de tempo em programação, o que significa *época*?  
(b) O que é *era Unix*?
13. Que valores uma variável do tipo **clock\_t** é capaz de conter?
14. O que deve armazenar uma variável do tipo **time\_t**?
15. Descreva em linhas gerais os campos de uma estrutura do tipo **struct tm**.
16. Para que serve a macro **CLOCKS\_PER\_SEC**?
17. Escreva um esboço de programa que, logo antes de encerrar, informa ao usuário o intervalo de tempo em segundos durante o qual esteve em execução.

18. Descreva cada uma das seguintes funções declaradas em `<time.h>`:

(a) **asctime()**

(b) **clock()**

(c) **ctime()**

(d) **difftime()**

(e) **gmtime()**

(f) **localtime()**

(g) **mktime()**

(h) **strftime()**

(i) **time()**

# *Capítulo 6*

---

*Caracteres e strings monobytes*

## 6.1 INTRODUÇÃO

Este capítulo é dedicado, principalmente, ao estudo de caracteres monobytes e *strings* constituídos por caracteres monobytes. Isto é, cada caractere explorado neste capítulo pode ser representado num único byte. Nos dois próximos capítulos, o estudo explorado aqui é estendido para abranger caracteres que ocupam mais de um byte.

Neste capítulo, os cabeçalhos `<ctype.h>` e `<string.h>` serão examinados. Os componentes do primeiro deles são dedicados à classificação e transformação entre maiúsculas e minúsculas de caracteres. O segundo cabeçalho provê suporte básico para processamento de blocos e *strings* monobytes em C.

Antes da apresentação desses cabeçalhos, serão apresentados alguns conceitos fundamentais sobre caracteres e *strings* em C.

### 6.1.1 CONJUNTO BÁSICO DE CARACTERES

No tocante aos caracteres usados pela linguagem C, o padrão ISO de C especifica que deve haver dois conjuntos de caracteres:

- **Conjunto de caracteres fonte**, contendo caracteres com os quais os programas são escritos. Este conjunto de caracteres é definido pela implementação.
- **Conjunto de caracteres de execução**, contendo caracteres usados no ambiente de execução de programas. Este conjunto de caracteres também é definido pela implementação, mas pode, usualmente, ser alterado por meio de uma mudança de localidade (v. **Capítulo 5**).

Os caracteres que devem fazer parte desses conjuntos são referidos coletivamente pelo padrão de C como **conjunto básico de caracteres**. Cada conjunto básico de caracteres deve conter um subconjunto mínimo de 96 caracteres especificado pelo padrão de C e o valor inteiro atribuído a cada caractere deve caber em um byte<sup>43</sup>. Devem fazer parte do conjunto básico de caracteres os caracteres que aparecem na **Tabela 6-1**.

---

<sup>43</sup> Lembre-se de que nem sempre um byte significa 8 bits, mas a exigência do padrão de C implica que um byte não pode conter menos de 7 bits (caso contrário, não seria possível armazenar os 96 caracteres do conjunto básico de caracteres).

CATEGORIA	CARACTERES
Letras maiúsculas	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Letras minúsculas	a b c d e f g h i j k l m n o p q r s t u v w x y z
Dígitos	0 1 2 3 4 5 6 7 8 9
Caracteres gráficos	! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ _ {   } ~
Espaços em branco	espaço (' '), tabulação horizontal ('\t'), tabulação vertical ('\v'), quebra de linha ('\n') e quebra de página ('\f')

Tabela 6-1: Conjunto básico de caracteres de C.

O padrão ISO de C também especifica que os dígitos de 0 a 9 devem ser codificados de forma contígua. Assim, tem-se, por exemplo, que '5' + 1 é sempre igual a '6'. Mas, esta contiguidade não é requerida para outros caracteres. Por exemplo, não é requerido pelo padrão de C que 'A' + 1 seja igual a 'B'. Neste último exemplo, como o padrão especifica que 'A' é menor do que 'B', isto implica que é permitido haver algum outro caractere entre 'A' e 'B'<sup>44</sup>.

Além dos caracteres enumerados na **Tabela 6-1**, o conjunto de caracteres de execução deve incluir ainda os seguintes caracteres não imprimíveis:

- Caractere nulo – '\0'
- Caractere de retorno – '\r'
- Caractere de alerta – '\a'
- *Backspace* – '\b'

Dentre estes últimos caracteres, o caractere nulo merece destaque especial por ser intensamente usado como terminal de *strings* em C.

O conjunto básico de caracteres preconizado pelo padrão ISO de C, acrescido de caracteres próprios de uma implementação de C, é denominado **conjunto estendido de caracteres**. Tipicamente, este conjunto de caracteres contém um número bem maior de caracteres do que o conjunto básico de caracteres e pode requerer que cada

<sup>44</sup> De fato, o antigo código de caracteres EBCDIC, ainda utilizado em mainframes da IBM, contém espaços vazios entre caracteres que representam letras.



caractere seja representado em mais de um byte. Neste capítulo, serão discutidos apenas conjuntos de caracteres nos quais cada caractere ocupa apenas um byte. O **Capítulo 7** discute conjuntos de caracteres que requerem mais de um byte para representar cada caractere.

## 6.1.2 CÓDIGOS DE CARACTERES

Um **código de caracteres**, ou **conjunto de caracteres codificados**, faz um mapeamento entre um conjunto de caracteres e um conjunto de inteiros. Cada inteiro que identifica unicamente um caractere num código de caracteres é denominado **ponto de código** e é caracteristicamente apresentado em formato hexadecimal. Como exemplos de código de caracteres bem conhecidos têm-se: ASCII, EBCDIC e ISO 8859-1.

O código ASCII é um dos mais antigos e conhecidos códigos de caracteres ainda em uso. É comum ouvir falar em código ASCII de 8 bits, mas, na realidade, a especificação original do código ASCII é de 7 bits; i.e., qualquer ponto de código (no intervalo de 0 a 127) cabe em 7 bits. Comumente, nos dias atuais, os pontos de código ASCII são representados por bytes de 8 bits, mas, neste caso, apenas 7 bits são usados; i.e., o bit mais significativo não faz parte do valor representado.

Outro antigo código de caracteres é EBCDIC que, apesar de conter todos os caracteres que constam no código ASCII, não tem pontos de código coincidentes com este último código. Além disso, os pontos de código de EBCDIC correspondentes às letras de A a Z não são consecutivos<sup>45</sup>.

Apesar de o padrão ISO de C não especificar um código de caracteres, a maioria dos conjuntos de caracteres em uso atualmente pode ser considerada extensão do código ASCII; i.e., eles preservam o significado dos pontos de código de 0 a 127 e acrescentam alguns mais. Exemplos de tais códigos de caracteres são aqueles especificados pelo padrão ISO 8859 e apresentados na **Tabela 6-2**.

---

45 A ordenação dos pontos de código corresponde à ordem alfabética usual, apesar de existirem lacunas entre alguns pontos de código correspondentes a letras. Por isso, neste caso, não é correto concluir que um ponto de código corresponde a uma letra se ele simplesmente estiver entre os pontos de código das letras de A a Z.

<b>NOME OFICIAL</b>	<b>NOME POPULAR</b>	<b>LINGUAGENS ABRANGIDAS</b>
ISO 8859-1	Latim 1	Linguagens da Europa Ocidental
ISO 8859-2	Latim 2	Linguagens da Europa Oriental
ISO 8859-3	Latim 3	Africâner, catalão, holandês, inglês, esperanto, alemão, italiano, maltês, espanhol e turco
ISO 8859-4	Latim 4	Dinamarquês, inglês, estoniano, finlandês, alemão, groenlandês, lituano e linguagens bálticas
ISO 8859-5	Latim-Cirílico	Búlgaro, bielo-russo, inglês, macedônio, russo, servo-croata e ucraniano
ISO 8859-6	Latim-Árabe	Árabe
ISO 8859-7	Latim-Grego	Grego
ISO 8859-8	Latim-Hebreu	Hebreu
ISO 8859-9	Latim 5	Dinamarquês, holandês, inglês, finlandês, francês, alemão, irlandês, italiano, norueguês, português, espanhol, sueco e turco
ISO 8859-10	Latim 6	Dinamarquês, inglês, finlandês, alemão, groenlandês, sueco e algumas outras linguagens bálticas e nórdicas

Tabela 6-2: Códigos de caracteres especificados pelo padrão ISO 8859.

A **Figura 6-1** apresenta o código de caracteres ISO 8859-1, popularmente conhecido como Latim 1.

		Primeiro dígito hexadecimal															
Segundo dígito hexadecimal		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0			SP	0	@	P	'	p			NBSP	°	À	Ð	à	ð
	1			!	1	A	Q	a	q			i	±	Á	Ñ	á	ñ
	2			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
	3			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
	4			\$	4	D	T	d	t			¤	'	Ä	Ô	ä	ô
	5			%	5	E	U	e	u			¥	μ	Å	Õ	å	õ
	6			&	6	F	V	f	v				¶	Æ	Ö	æ	ö
	7			'	7	G	W	g	w			§	·	Ç	×	ç	÷
	8			(	8	H	X	h	x			"	,	È	Ø	è	ø
	9			)	9	I	Y	i	y			©	'	É	Ù	é	ù
	A			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
	B			+	;	K	[	k	{			<	>	Ë	Û	ë	û
	C			,	<	L	\	l				¬	¼	Ì	Ü	ì	ü
	D			-	=	M	]	m	}			SHY	½	Í	Ý	í	ý
	E			.	>	N	^	n	~			®	¾	Î	Þ	î	þ
	F			/	?	O	_	o				—	¿	Ï	ß	ï	ÿ

Figura 6-1: Código de caracteres ISO 8859-1 (Latim 1).

### 6.1.3 PÁGINAS DE CÓDIGO

Com a disseminação do uso dos microcomputadores, foram criadas **páginas de código**, que eram códigos de caracteres semelhantes aos mais modernos especificados pelo padrão ISO 8859-X. Essas páginas de código foram inventadas por IBM e Microsoft no início da década de 1980 para uso com o sistema operacional DOS. Elas eram identificadas por um confuso sistema de numeração. Por exemplo, a página de código CP860 foi elaborada para ser usada com a língua portuguesa. Havia dezenas de páginas de código criadas para contemplar diversas línguas e scripts e, curiosamente, algumas línguas tinham mais de uma página de código associada. Por exemplo, as páginas de código CP737 e CP869 foram ambas criadas para a língua grega.

Mais adiante, a Microsoft criou sua própria família de páginas de código que seriam denominadas **páginas de código ANSI**<sup>46</sup>. Conceitualmente, não há diferença notável entre as antigas páginas de código usadas com DOS e as mais recentes usadas com Windows. Novamente, as páginas de código ANSI apresentam denominações confusas e com um agravante: uma página de código ANSI podia ser conhecida por várias denominações. Por exemplo, as denominações *1252*, *windows-1252* e *WinLatin1* todas referem-se a uma mesma página de código<sup>47</sup>. Com isso, a Microsoft deu sua contribuição para a confusão já reinante à época.

### 6.1.4 PROBLEMAS COM CARACTERES MONOBYTES

Em alguns programas multilíngues, é suficiente usar algum código de caracteres do padrão ISO 8859 (e.g., um programa que lida ao mesmo tempo com inglês, espanhol e português). Mas, para outros programas, não há um único código de caracteres monobytes que, sozinho, atenda a suas necessidades de uso de vários scripts e linguagens simultâneos (e.g., um programa que lide concomitantemente com inglês, grego e cirílico).

O mesmo problema aflige as páginas de código. Por exemplo, as línguas portuguesa e russa usam páginas de códigos que, não apenas possuem repertórios de caracteres diferentes, como também associam caracteres diferentes a um mesmo inteiro maior do que 128. Além disso, em todos os códigos de caracteres apresentados até aqui, cada caractere pode ser representado em apenas um byte. Estes códigos de caracteres não satisfazem, por exemplo, as necessidades de linguagens asiáticas que, por usarem um número muito grande de símbolos, precisam de mais um byte para representar cada caractere.

Para resolver os problemas associados a antigos códigos de caracteres que usam apenas 8 bits foram desenvolvidos os padrões de codificação de caracteres Unicode e ISO 10646, que usam mais de 8 bits para representar caracteres. Os padrões Unicode e ISO/IEC 10646 definem o **Conjunto de Caracteres Universal (UCS)**, que é um código de caracteres que permite o uso de caracteres de todas as linguagens, incluindo caracteres artificiais (e.g., aqueles usados em Música e Matemática) e aqueles usados em linguagens asiáticas. Estes modernos padrões serão examinados no **Capítulo 7**.

---

46 Apesar da denominação, o instituto de padrões ANSI nunca referendou essas páginas de código.

47 Ainda pior é que esta página de código é frequentemente confundida com o código de caracteres especificado pelo padrão ISO 8859-1. Muitas vezes, este engano causa problemas em navegadores da web e programas de e-mail.

Para que um programa em C seja capaz de manipular caracteres que ocupam mais de um byte, existem duas soluções básicas: (1) o uso de caracteres extensos e (2) o uso de caracteres multibyte. Estas duas formas de representação de caracteres serão estudadas nos **Capítulos 7 e 8**.

### 6.1.5 CARACTERES CONSTANTES

Existem três formas básicas de representação de caracteres constantes em C:

- **Escrevendo-se o caractere entre apóstrofos** (e.g., 'A '). Apesar de esta forma de representação ser a mais legível e portátil, ela é factível apenas quando o caractere possui representação gráfica e o programador possui meios para introduzi-lo (e.g., uma configuração adequada de teclado).
- **Por meio de uma sequência de escape**, que consiste no caractere \ (barra invertida) seguido de um caractere com significado especial, sendo ambos colocados entre apóstrofos. Sequências de escape devem ser usadas para representar certos caracteres que não possuem representação gráfica (e.g., quebra de linha) ou que têm significados especiais em C (e.g., apóstrofo). Esta forma de representação de caracteres também é portátil, apesar de ser menos legível do que a forma de representação anterior. A **Tabela 6-3** apresenta as sequências de escape que podem ser utilizados num programa escrito em C.

SEQUÊNCIA DE ESCAPE	DESCRIÇÃO
\a	Campainha (alerta)
\b	<i>Backspace</i>
\t	Tabulação horizontal
\v	Tabulação vertical
\n	Quebra de linha
\f	Quebra de página
\r	Retorno de carro
\e	Escape
\0	Caractere nulo
\\	Barra invertida
\?	Interrogação
\'	Apóstrofo
\"	Aspas

Tabela 6-3: Sequências de escape e seus significados.

- **Informando qual é o valor do inteiro associado ao caractere.** Um caractere escrito nesta forma de representação deve ser colocado entre apóstrofes, e seu valor deve ser precedido por barra invertida e ser escrito em um dos seguintes formatos:
  - **Octal** – de acordo com o que foi apresentado na **Seção 2.2**, mas não precisa começar com zero (e.g., `'\132'`).
  - **Hexadecimal** – de acordo com o que foi apresentado na **Seção 2.2** (e.g., `'\x5A'`).
  - **Caractere universal (C99)** – o valor do caractere é escrito começando com `u` (ou `U`) seguido de um número hexadecimal com quatro ou oito dígitos que corresponde ao valor do caractere no código de caracteres especificado pelo padrão ISO/IEC 10646 (e.g., `'\u005A'`).

Qualquer caractere pode ser representado por esta última forma de representação. No entanto, ela nem é portátil, nem legível, e deve ser utilizada apenas como último recurso; i.e., quando nenhum dos outros dois formatos anteriores pode ser usado.

Qualquer que seja o formato utilizado, o compilador interpreta-o como o valor inteiro correspondente ao caractere no código de caracteres ora utilizado. Por exemplo, se o código de caracteres utilizado for ASCII, o compilador interpretará qualquer dos caracteres constante `'A'`, `'\101'`, `'\x41'` ou `'\u0041'` como 65, que é o valor corresponde ao caractere 'A' no código ASCII. Neste exemplo, evidentemente, a melhor maneira de representação do caractere A é `'A'`.

## 6.1.6 STRINGS CONSTANTES

Um **string constante** consiste em uma sequência de caracteres constantes. Em C, um *string* constante pode conter caracteres constantes em qualquer dos formatos apresentados na **Seção 6.1.5** e deve ser envolvido entre aspas. A seguir são apresentados alguns exemplos de *strings* constantes:

```
"casa"
"Dia\tMes\tAno\n"
"\x62\x6F\x6C\x61"
```

*Strings* constantes separados por espaços em branco são automaticamente concatenados pelo compilador. Isto é útil quando se tem um *string* constante muito grande e se deseja escrevê-lo em duas linhas. Por exemplo:

```
printf( "E' melhor dividir strings constantes muito "
        "grandes em varias linhas." )
```

Os dois *strings* constantes do último exemplo serão concatenados pelo compilador para formar um *string* constante:

```
printf( "E' melhor dividir strings constantes muito grandes
em varias linhas." )
```

## 6.2 CLASSIFICAÇÃO E TRANSFORMAÇÃO DE CARACTERES: <ctype.h>

O cabeçalho <ctype.h> contém alusões de funções utilizadas para classificar caracteres (por exemplo, testando se um caractere é letra ou dígito) e de duas funções que fazem transformações entre letras maiúsculas e minúsculas.

Com exceção das funções **isdigit()** e **isxdigit()**, todas as demais funções declaradas em <ctype.h> são afetadas pelo valor corrente da localidade associada à categoria **LC\_CTYPE** (v. **Seção 5.2**). Além disso, qualquer função declarada neste cabeçalho pode ser definida como macro.

### 6.2.1 FUNÇÕES DE CLASSIFICAÇÃO DE CARACTERES

*isalnum()*

**Incluir:** <ctype.h>

**Descrição:** A função **isalnum()** testa se um dado caractere é alfanumérico (i.e., letra ou dígito).

**Protótipo:**

```
int isalnum(int caractere)
```

**Parâmetro:** *caractere* – o caractere a ser testado.

**Retorno:** Um valor diferente de zero se o caractere for alfanumérico, ou zero, caso contrário.

**Observação:** Consulte também **isalpha()** e **isdigit()**.

**Exemplo:** Veja o exemplo da função **isxdigit()**.

*isalpha()*

**Incluir:** `<ctype.h>`

**Descrição:** A função **isalpha()** testa se um dado caractere é uma letra (i.e., a–z, A–Z ou outra dependendo de localidade).

**Protótipo:**

```
int isalpha(int caractere)
```

**Parâmetro:** `caractere` – o caractere a ser testado.

**Retorno:** Um valor diferente de zero se o caractere for letra, ou zero, caso contrário.

**Observação:** Consulte também **isalnum()**.

**Exemplo:**

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

/****
 * Função EhStringAlfabetico(): verifica se um string
 *                               contém apenas letras
 *
 * Argumentos: str (entrada) - o string a ser checado
 *
 * Retorno: 1, se o string for alfabético;
 *          0, em caso contrário
 ****/
int EhStringAlfabetico(const char *str)
{
    /* O laço while é encerrado quando */
    /* um caractere que não é letra    */
    /* é encontrado (inclusive '\0')   */
    while (*str != '\0' && isalpha(*str))
        str++;
    return (*str == '\0');
}
```



```

while (isalpha(*str++))
    ;

    /* Faz 'str' apontar para o caractere */
    /* que causou o encerramento do laço */
    --str;

return !*str;
}

int main(void)
{
    char *s = "bola";

    printf( "O string \"%s\" %s alfabetico\n", s,
            EhStringAlfabetico(s) ? "e'" : "nao e'" );

    return 0;
}

```

### *isblank()* (C99)

**Incluir:** <ctype.h>

**Descrição:** A função **isblank()** (C99) testa se o caractere recebido como argumento é um espaço em branco que separa palavras numa linha.

**Protótipo:**

<pre>int isblank(int caractere)</pre>
---------------------------------------

**Parâmetro:** `caractere` – o caractere a ser testado.

**Retorno:** Um valor diferente de zero se caractere for um espaço em branco que separa palavras numa linha ou zero se não for o caso.

**Observação:** Compare esta função com **isspace()**.

**Exemplo:**

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

int main(void)
{
    printf("O caractere '\\n' %s espaco em branco\n",
           isblank('\\n') ? "e" : "nao e");
    printf("O caractere '$' %s espaco em branco\n",
           isblank('$') ? "e" : "nao e");
    printf("O caractere '\\t' %s espaco em branco\n",
           isblank('\\t') ? "e" : "nao e");
    printf("O caractere ' ' %s espaco em branco\n",
           isblank(' ') ? "e" : "nao e");

    return 0;
}
```

Resultado do programa quando compilado e executado no Linux:

```
O caractere '\\n' nao e' espaco em branco
O caractere '$' nao e' espaco em branco
O caractere '\\t' e' espaco em branco
O caractere ' ' e' espaco em branco
```

*isctrl()*

**Incluir:** <ctype.h>

**Descrição:** A função **isctrl()** testa se um dado caractere é um caractere de controle ou [DELETE].

**Protótipo:**

```
int isctrl(int caractere)
```

**Parâmetro:** caractere – o caractere a ser testado.

**Retorno:** Um valor diferente de zero, se o caractere for um caractere de controle, ou zero, se não for o caso.

**Observação:** Consulte também **isblank()** e **isspace()**.

**Exemplo:** Veja o exemplo da função **isspace()**.

*isdigit()*

**Incluir:** `<ctype.h>`

**Descrição:** A função **isdigit()** testa se um dado caractere é um dígito (0–9).

**Protótipo:**

```
int isdigit(int caractere)
```

**Parâmetro:** `caractere` – o caractere a ser testado.

**Retorno:** Um valor diferente de zero, se o caractere for um dígito, ou zero, caso contrário.

**Observações:**

- Esta função *não* é afetada pelo valor corrente de localidade atribuído à categoria **LC\_CTYPE**.
- Consulte também **isalnum()** e **isxdigit()**.

**Exemplo:** Veja o exemplo da função **isxdigit()**.

*isgraph()*

**Incluir:** `<ctype.h>`

**Descrição:** A função **isgraph()** testa se um dado caractere pode ser impresso (exceto espaço em branco).

**Protótipo:**

```
int isgraph(int caractere)
```

**Parâmetro:** `caractere` – o caractere a ser testado.

**Retorno:** Um valor diferente de zero se o caractere pode ser impresso (exceto espaço em branco) ou zero, caso contrário.

**Exemplo:** Veja o exemplo da função **ispunct()**.

*islower()*

**Incluir:** `<ctype.h>`

**Descrição:** A função **islower()** testa se um dado caractere é uma letra minúscula.

**Protótipo:**

```
int islower(int caractere)
```

**Parâmetro:** `caractere` – o caractere a ser testado.

**Retorno:** Um valor diferente de zero se o caractere for uma letra minúscula; zero, se não for o caso.

**Observação:** Consulte também **isupper()**.

**Exemplo:** Veja o exemplo da função **toupper()**.

*isprint()*

**Incluir:** `<ctype.h>`

**Descrição:** A função **isprint()** testa se um dado caractere pode ser impresso (incluindo espaços em branco).

**Protótipo:**

```
int isprint(int caractere)
```

**Parâmetro:** `caractere` – o caractere a ser testado.

**Retorno:** Um valor diferente de zero se o caractere pode ser impresso; zero, caso contrário.

**Observação:** Consulte também **isgraph()**.

**Exemplo:** Veja o exemplo da função **ispunct()**.

*ispunct()*

**Incluir:** `<ctype.h>`

**Descrição:** A função **ispunct()** testa se um dado caractere é um símbolo de pontuação.

**Protótipo:**

```
int ispunct(int Caractere)
```

**Parâmetro:** `caractere` – o caractere a ser testado.

**Retorno:** Um valor diferente de zero se o caractere for um símbolo de pontuação ou zero se não for o caso.

**Exemplo:** O programa a seguir demonstra o uso das funções **isgraph()**, **isprint()** e **ispunct()**.

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

int main()
{
    /* Uso de isgraph() */
```

```

printf( "\nO caractere '\\n' %s caractere grafico\n",
        isgraph('\\n') ? "e" : "nao e");
printf( "O caractere '$' %s caractere grafico\n",
        isgraph('$') ? "e" : "nao e");
printf( "O caractere '\\a' %s caractere grafico\n",
        isgraph('\\a') ? "e" : "nao e");

/* Uso de isprint() */
printf( "\nO caractere '\\n' %s caractere imprimivel\n",
        isprint('\\n') ? "e" : "nao e");
printf( "O caractere '$' %s caractere imprimivel\n",
        isprint('$') ? "e" : "nao e");
printf( "O caractere '\\a' %s caractere imprimivel\n",
        isprint('\\a') ? "e" : "nao e");

/* Uso de ispunct() */
printf( "\nO caractere '?' %s caractere de pontuacao\n",
        ispunct('?') ? "e" : "nao e");
printf( "O caractere 'B' %s caractere de pontuacao\n",
        ispunct('B') ? "e" : "nao e");
printf( "O caractere ':' %s caractere de pontuacao\n",
        ispunct(':') ? "e" : "nao e");

return 0;
}

```

## *isspace()*

**Incluir:** <ctype.h>

**Descrição:** A função **isspace()** testa se um dado caractere é um espaço em branco qualquer.

**Protótipo:**

```
int isspace(int caractere)
```

**Parâmetro:** *caractere* – o caractere a ser testado.

**Retorno:** Um valor diferente de zero se o caractere for um espaço em branco qualquer ou zero, caso contrário.

**Observações:**

- Compare esta função com **isblank()**.
- A **Tabela 6-4** apresenta os espaços em branco mais comuns.

CARACTERE	DESCRIÇÃO
' '	Caractere de espaço
'\f'	Caractere de quebra de página ( <i>form feed</i> )
'\n'	Caractere de quebra de linha
'\r'	Caractere de retorno ( <i>return</i> )
'\t'	Caractere de tabulação horizontal
'\v'	Caractere de tabulação vertical

Tabela 6-4: Caracteres interpretados como espaços em branco.

**Exemplo:** O programa a seguir demonstra o uso das funções **iscntrl()** e **isspace()**.

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

int main()
{
    /* Uso de iscntrl() */
    printf("\nO caractere '\\n' %s caractere de "
           "controle\n",iscntrl('\\n') ? "e" : "nao e");
    printf("O caractere '$' %s caractere de controle\n",
           iscntrl('$') ? "e" : "nao e");
    printf("O caractere '#' %s caractere de controle\n",
           iscntrl('#') ? "e" : "nao e");

    /* Uso de isspace() */
    printf("O caractere '\\n' %s espaco em branco\n",
           isspace('\\n') ? "e" : "nao e");
    printf("O caractere '\\t' %s espaco em branco\n",
           isspace('\\t') ? "e" : "nao e");
    printf("O caractere '#' %s espaco em branco\n",
           isspace('#') ? "e" : "nao e");
}
```

```
    return 0;  
}
```

### *isupper()*

**Incluir:** <ctype.h>

**Descrição:** A função **isupper()** testa se um dado caractere é uma letra maiúscula.

**Protótipo:**

```
int isupper(int caractere)
```

**Parâmetro:** `caractere` – o caractere a ser testado.

**Retorno:** Um valor diferente de zero se o caractere for uma letra maiúscula; zero, caso contrário.

**Observação:** Para testar se um caractere é letra minúscula, use **islower()**.

**Exemplo:** Veja o exemplo da função **toupper()**.

### *isxdigit()*

**Incluir:** <ctype.h>

**Descrição:** A função **isxdigit()** testa se um dado caractere é um dígito hexadecimal válido (a–f, A–F, 0–9).

**Protótipo:**

```
int isxdigit(int caractere)
```

**Parâmetro:** `caractere` – O caractere a ser testado.

**Retorno:** Um valor diferente de zero se o caractere for um dígito hexadecimal válido ou zero se não for.



**Observações:**

- Esta função *não* é afetada pelo valor corrente de localidade atribuído à categoria **LC\_CTYPE**.
- Consulte também **isalnum()** e **isdigit()**.

**Exemplo:** O programa a seguir demonstra o uso das funções **isalnum()**, **isdigit()** e **isxdigit()**.

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

int main()
{
    /* Uso de isalnum() */
    printf( "\nO caractere 'z' %s letra nem digito\n",
            isalnum('z') ? "e" : "nao e" );
    printf( "O caractere '3' %s letra nem digito\n",
            isalnum('3') ? "e" : "nao e" );
    printf( "O caractere '@' %s letra nem digito\n",
            isalnum('@') ? "e" : "nao e" );

    /* Uso de isdigit() */
    printf( "\nO caractere '5' %s digito\n",
            isdigit('5') ? "e" : "nao e" );
    printf( "O caractere '$' %s digito\n",
            isdigit('$') ? "e" : "nao e" );

    /* Uso de isxdigit() */
    printf( "\nO caractere 'z' %s digito hexadecimal\n",
            isxdigit('z') ? "e" : "nao e" );
    printf( "O caractere '3' %s digito hexadecimal\n",
            isxdigit('3') ? "e" : "nao e" );
    printf( "O caractere 'C' %s digito hexadecimal\n",
            isxdigit('C') ? "e" : "nao e" );

    return 0;
}
```

## 6.2.2 FUNÇÕES DE TRANSFORMAÇÃO DE CARACTERES

*tolower()*

**Incluir:** `<ctype.h>`

**Descrição:** A função **tolower()** converte uma letra maiúscula em minúscula.

**Protótipo:**

```
int tolower(int caractere)
```

**Parâmetro:** `caractere` – caractere a ser convertido.

**Retorno:** O caractere convertido em letra minúscula, se ele for uma letra maiúscula; o próprio caractere sem modificação, caso contrário.

**Observação:** Consulte também **toupper()**.

**Exemplo:** Veja o exemplo da função **toupper()**.

*toupper()*

**Incluir:** `<ctype.h>`

**Descrição:** A função **toupper()** converte uma letra minúscula em maiúscula.

**Protótipo:**

```
int toupper(int caractere)
```

**Parâmetro:** `caractere` – caractere a ser convertido.

**Retorno:** O caractere convertido em letra maiúscula, se ele for uma letra minúscula; o próprio caractere sem modificação, caso contrário.

**Observação:** Consulte também **tolower()**.

**Exemplo:** O programa a seguir demonstra o uso das funções **islower()**, **isupper()**, **tolower()** e **toupper()**.

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

int main()
{
    /* Uso de islower() */
    printf( "\nO caractere 'u' %s minuscula\n",
            islower('u') ? "e" : "nao e" );
    printf( "O caractere 'L' %s minuscula\n",
            islower('L') ? "e" : "nao e" );
    printf( "O caractere '$' %s minuscula\n",
            islower('$') ? "e" : "nao e" );

    /* Uso de isupper() */
    printf( "\nO caractere 'u' %s maiuscula\n",
            isupper('u') ? "e" : "nao e" );
    printf( "O caractere 'L' %s maiuscula\n",
            isupper('L') ? "e" : "nao e" );
    printf( "O caractere '$' %s maiuscula\n",
            isupper('$') ? "e" : "nao e" );

    /* Uso de tolower() */
    printf( "\nO caractere 'u' convertido em minuscula"
            " e' %c\n", tolower('u') );
    printf( "O caractere 'L' convertido em minuscula"
            " e' %c\n", tolower('L') );
    printf( "O caractere '$' convertido em minuscula"
            " e' %c\n", tolower('$') );

    /* Uso de toupper() */
    printf( "\nO caractere 'u' convertido em maiuscula"
            " e' %c\n", toupper('u') );
    printf( "O caractere 'L' convertido em maiuscula"
            " e' %c\n", toupper('L') );
    printf( "O caractere '$' convertido em maiuscula"
            " e' %c\n", toupper('$') );

    return 0;
}
```

## 6.3 PROCESSAMENTO DE STRINGS E BLOCOS: <string.h>

No cabeçalho <string.h>, encontram-se componentes para o processamento de *strings* e arrays de bytes (blocos de memória), que serão descritos a seguir.

### 6.3.1 TIPO `size_t`

**Incluir:** <time.h>, <string.h>, <wchar.h>, <stdlib.h> ou <std-def.h>

**Descrição:** `size_t` é um tipo inteiro sem sinal definido em vários cabeçalhos (v. **Seção 1.7.1**).

### 6.3.2 MACRO `NULL`

**Incluir:** <time.h>, <string.h>, <wchar.h>, <stdlib.h> ou <std-def.h>

**Descrição:** A macro `NULL` representa um ponteiro nulo e é definida em vários cabeçalhos (v. **Seção 1.7.2**).

### 6.3.3 FUNÇÕES DE PROCESSAMENTO DE BLOCOS

As funções apresentadas nesta seção atuam sobre blocos arbitrários de memória (i.e., arrays de bytes) e não necessariamente sobre *strings*. Isto é, embora um *string* seja um array de caracteres e caractere (monobyte) seja sinônimo de byte em termos de armazenamento, os blocos manipulados pelas funções apresentadas nesta seção não precisam conter o caractere '`\0`'. Por isso, essas funções incluem um parâmetro do tipo `size_t` que limita o número de bytes processados em cada bloco.

*memchr()*

**Incluir:** <string.h>

**Descrição:** A função `memchr()` procura a primeira ocorrência de um caractere num bloco, limitando a busca ao número de bytes recebido como parâmetro.

**Protótipo:**

```
void *memchr( const void *bloco, int caractere, size_t n )
```

**Parâmetros:**

- `bloco` – o bloco onde será feita a busca.
- `caractere` – o caractere (convertido em **int**) a ser procurado.
- `n` – número máximo de caracteres a serem pesquisados no bloco.

**Retorno:** Um ponteiro para o local onde o caractere for encontrado ou **NULL**, se ele não for encontrado.

**Observação:** Compare esta função com **strchr()**.

**Exemplo:**

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

int main()
{
    char *s = "Apenas um string";
    char *p;

    p = memchr(s, 'r', strlen(s));

    printf("\nString original: \"%s\\\"", s);
    printf( "\nString a partir do caractere 'r':"
           " \"%s\\\"\\n\", p );

    return 0;
}
```

*memcmp()*

**Incluir:** <string.h>

**Descrição:** A função **memcmp()** compara um número limitado de bytes de dois blocos de memória.

**Protótipo:**

```
int memcmp( const void *bloco1,
            const void *bloco2, size_t n )
```

**Parâmetros:**

- `bloco1` – ponteiro para o primeiro bloco de memória.
- `bloco2` – ponteiro para o segundo bloco de memória.
- `n` – número de bytes a serem comparados.

**Retorno:**

- Zero se os blocos são iguais.
- Um valor menor do que zero se `bloco1` é menor do que `bloco2`.
- Um valor maior do que zero se `bloco1` é maior do que `bloco2`.

**Observações:**

- Os blocos são comparados byte a byte; i.e., o primeiro byte do primeiro bloco é comparado com primeiro byte do segundo bloco, o segundo byte do primeiro bloco é comparado com segundo byte do segundo bloco, e assim por diante.
- Se todos os `n` bytes comparados forem iguais, os blocos serão considerados iguais.
- Se dois bytes comparados forem diferentes, o bloco contendo o byte de menor valor será considerado menor.
- Normalmente, não há interesse em saber se um bloco é maior ou menor do que outro; i.e., tipicamente esta função é usada para determinar se dois blocos são iguais ou diferentes (v. exemplo a seguir).
- Compare esta função com **strcmp()** e **strncmp()**.

**Exemplo:**

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

typedef struct {
    char nome[30];
    char endereco[20];
    char telefone[10];
} tContato;

int main()
{
    tContato e1 = {"Manoel", "Rua X", "88882424"};
    tContato e2 = {"Manoel", "Rua X", "88882424"};
    tContato e3 = {"Joaquim", "Rua Y", "99994242"};

    printf( "\nAs estruturas 'e1' e 'e2' %ssao iguais\n",
            memcmp(&e1, &e2, sizeof(tContato)) ? "NAO " : "" );

    printf( "As estruturas 'e1' e 'e3' %ssao iguais\n",
            memcmp(&e1, &e3, sizeof(tContato)) ? "NAO " : "" );

    return 0;
}
```

*memcpy()***Incluir:** <string.h>

**Descrição:** A função **memcpy()** copia um número de bytes de um bloco de memória para outro.

**Protótipo:**

```
void *memcpy( void *restrict destino,
              const void *restrict origem,
              size_t n )
```

**Parâmetros:**

- destino – ponteiro para o bloco de destino.
- origem – ponteiro para o bloco de origem.
- n – número de bytes que serão copiados.

**Retorno:** Um ponteiro para o bloco destino.

**Observações:**

- O bloco de destino deve ter espaço suficiente para conter o resultado da operação.
- Se houver sobreposição dos blocos, o resultado será indefinido.
- Compare esta função com **memmove()**.

**Exemplo:**

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

typedef struct {
    char nome[30];
    char endereco[20];
    char telefone[10];
} tContato;

int main()
{
    tContato umaPessoa = {"Manoel", "Rua X", "88882424"};
    tContato pessoaCopia;

    memcpy(&pessoaCopia, &umaPessoa, sizeof(umaPessoa));

    printf("\nDados copiados:\n");
    printf("\n\tNome: %s", pessoaCopia.nome);
    printf("\n\tEndereco: %s", pessoaCopia.endereco);
    printf("\n\tTelefone: %s\n", pessoaCopia.telefone);

    return 0;
}
```



*memmove()***Incluir:** <string.h>**Descrição:** A função **memmove()** copia um número de bytes de um bloco de memória para outro, permitindo sobreposição dos blocos de origem e destino.**Protótipo:**

```
void * memmove(void *destino, const void *origem, size_t n)
```

**Parâmetros:**

- destino – ponteiro para o bloco de destino.
- origem – ponteiro para o bloco de origem.
- n – número de bytes que serão copiados.

**Retorno:** Um ponteiro para o bloco destino.**Observação:** Compare esta função com **memcpy()**.**Exemplo:**

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

int main()
{
    char s[30] = "Apenas um string";

    printf( "\nString ANTES da chamada de memmove(): "
           "\"%s\"\n", s );
    memmove(s + 7, s, 10);

    printf( "String DEPOIS da chamada de memmove(): "
           "\"%s\"\n", s );

    return 0;
}
```

*memset()***Incluir:** <string.h>**Descrição:** A função **memset()** preenche um número limitado de bytes de um bloco com o valor de um caractere.**Protótipo:**

<pre>void *memset(void *bloco, int i, size_t n)</pre>
---

**Parâmetros:**

- **bloco** – bloco de memória a ser preenchido.
- **i** – caractere convertido em **int** com o qual o bloco será preenchido.
- **n** – número de bytes que serão preenchidos.

**Retorno:** Ponteiro para o bloco.**Exemplo:**

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

typedef struct {
    char nome[30];
    char endereco[20];
    char telefone[10];
} tContato;

int main()
{
    tContato pessoa = {"Manoel", "Rua X", "88882424"};

    printf("\nEstrutura ANTES de chamar memset():\n");
    printf("\n\tNome: %s", pessoa.nome);
    printf("\n\tEndereco: %s", pessoa.endereco);
    printf("\n\tTelefone: %s\n", pessoa.telefone);

    memset(&pessoa, '*', sizeof(tContato));
```

```

    /* É preciso transformar os arrays */
    /* novamente em strings para poder */
    /* usar printf(). */
    pessoa.nome[29] = pessoa.endereco[19] =
        pessoa.telefone[9] = '\0';

    printf("\nEstrutura DEPOIS de chamar memset():\n");
    printf("\n\tNome: %s", pessoa.nome);
    printf("\n\tEndereco: %s", pessoa.endereco);
    printf("\n\tTelefone: %s\n", pessoa.telefone);

    return 0;
}

```

### 6.3.4 FUNÇÕES DE PROCESSAMENTO DE STRINGS

Esta seção apresenta a maioria das funções de processamento de *strings* encontradas no cabeçalho `<string.h>`. Por tratar-se de um assunto mais complexo, as funções envolvidas em comparação de *strings* declaradas neste cabeçalho serão apresentadas numa seção específica (**Seção 6.4**).

*strcat()*

**Incluir:** `<string.h>`

**Descrição:** A função **strcat()** concatena uma cópia de um *string* ao final de outro *string*.

**Protótipo:**

```

char *strcat( char *restrict destino,
              const char *restrict origem )

```

**Parâmetros:**

- *destino* – array contendo o *string* que será acrescido de origem.

- *origem* – *string* que será acrescentado a *destino*.

**Retorno:** Ponteiro para o *string* destino.

**Observações:**

- O comprimento do *string* resultante será:  

$$\text{strlen}(\text{destino}) + \text{strlen}(\text{origem})$$
- Ambos os *strings* podem ter comprimento zero.
- Certifique-se de que o array *destino* tenha espaço suficiente para conter o *string* concatenado.
- Compare esta função com **strncat()**.

**Exemplo:**

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

int main(void)
{
    char nome[30]="Manoel ";
    char sobrenome[]="Silva";

    printf("\nNome: \"%s\"\n", nome);
    printf("Sobrenome: \"%s\"\n", sobrenome);

    strcat(nome,sobrenome);

    printf("Nome completo: \"%s\"\n", nome);

    return 0;
}
```

*strchr()*

**Incluir:** <string.h>

**Descrição:** A função **strchr()** procura a primeira ocorrência de um caractere num *string*.

**Protótipo:**

```
char *strchr(const char *string, int caractere)
```

**Parâmetros:**

- *string* – *string* a ser pesquisado.
- *caractere* – caractere a ser procurado.

**Retorno:** Ponteiro para a primeira ocorrência do caractere no *string*, se o caractere for encontrado; **NULL**, caso contrário.

**Observações:**

- A função considera o caractere terminal de *string* ( `'\0'` ) como parte do *string*. Assim, `strchr(str, 0)` retorna um ponteiro para o caractere terminal do *string* `str`.
- Consulte também **strrchr()**.

**Exemplo:**

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

int main(void)
{
    char str[]="Isto e' um teste";

    printf("\nString original: \"%s\"\n", str);
    printf( "String apos a primeira ocorrencia de 'e':"
           "\"%s\"\n", strchr(str,'e') );

    return 0;
}
```

*strcpy()*

**Incluir:** <string.h>

**Descrição:** A função **strcpy()** copia o conteúdo de um *string* num array de caracteres.

**Protótipo:**

```
char *strcpy( char *restrict destino,
              const char *restrict origem )
```

**Parâmetros:**

- destino – array de caracteres que receberá a cópia do *string*.
- origem – *string* que é copiado para destino.

**Retorno:** Um ponteiro para o início do array destino.

**Observações:**

- Certifique-se de que o array que receberá a cópia seja suficientemente grande para conter o *string* copiado.
- Consulte também **strncpy()** e **memcpy()**.

**Exemplo:**

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

/****
 *
 * Função CopiaFinal(): copia os caracteres finais de um
 *                      string
 *
 * Argumentos: copia (saída) - array que receberá os
 *                      caracteres copiados
 *                      origem (entrada) - string que fornecerá
 *                      os caracteres
 *                      nCaracteres (entrada) - no. de caracteres
 *                      que serão copiados
 *
 * Retorno: ponteiro para o início do string 'copia'.
 *
```

```

****/

char *CopiaFinal( char *copia, const char *origem,
                  int nCaracteres )
{
    const char *ptr = origem;

    if ( nCaracteres <= 0 ) {
        *copia = '\0';
        return copia;
    }

    while ( *ptr++ )
        ; /* VAZIO */

    if (ptr - nCaracteres - 1 > origem)
        origem = ptr - nCaracteres - 1;

    return strcpy(copia, origem);
}

int main()
{
    char ar[50];
    char str[] = "String que fornecera caracteres";

    printf("\nString de origem: \"%s\"\n", str);

    printf( "String que recebeu caracteres: \"%s\"\n",
            CopiaFinal(ar, str, 10) );

    return 0;
}

```

### *strcspn()*

**Incluir:** <string.h>

**Descrição:** A função **strcspn()** examina um *string* até encontrar algum caractere presente em outro *string*.

**Protótipo:**

```
size_t strcspn(const char *string1, const char *string2)
```

**Parâmetros:**

- `string1` – *string* a ser analisado.
- `string2` – *string* contendo os caracteres que serão procurados.

**Retorno:** O número de caracteres examinados em `string1` antes de encontrar o primeiro caractere de `string2`<sup>48</sup>.

**Observações:**

- Curiosidade: o nome desta função é derivado de *string character span* em inglês, que significa que a função calcula a distância do início do primeiro *string* até o primeiro caractere do segundo *string* encontrado.
- O caractere terminal de *string* não é levado em consideração.
- Consulte também `strspn()`, `strchr()` e `strrchr()`.

**Exemplo:**

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

main()
{
    char    *str1 = "Isto e' um teste";
    char    *str2 = "AHuh";
    size_t  intersecao;

    intersecao = strcspn(str1, str2);

    printf( "\nOs strings \"%s\" e \"%s\" se "
           "interceptam no caractere '%c'\n",
           str1, str2, str1[intersecao] );
}
```

---

<sup>48</sup> Consequentemente, se nenhum caractere de `string2` for encontrado em `string1`, a função `strcspn()` retorna o comprimento de `string1`.



```
    return 0;
}
```

### *strerror()*

**Incluir:** <string.h>

**Descrição:** A função **strerror()** retorna um *string* descrevendo o erro associado a um valor inteiro.

**Protótipo:**

<pre>char *strerror(int nErro)</pre>
--------------------------------------

**Parâmetro:** `nErro` – inteiro que representa alguma condição de erro.

**Retorno:** Um ponteiro para um *string* contendo uma mensagem de erro ou indicando que não existe mensagem de erro associada a `nErro`.

**Observação:** A mensagem de erro retornada é armazenada num array de duração fixa que é sobrescrito a cada chamada de **strerror()**.

### **Exemplo:**

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

int main()
{
    int i;

    /* Enumera erros com códigos entre 0 e 4 */
    for (i = 0; i <= 4; ++i)
        printf("Erro No. %d: %s\n", i, strerror(i));

    return 0;
}
```

*strlen()*

**Incluir:** <string.h>

**Descrição:** A função **strlen()** calcula o comprimento de um *string*.

**Protótipo:**

```
size_t strlen(const char *string)
```

**Parâmetros:** *string* – *string* cujo comprimento será calculado.

**Retorno:** O número de caracteres visíveis no *string* (i.e., o caractere terminal '`\0`' não é incluído na contagem).

**Exemplo:**

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

/****
 *
 * Função FinalDeString(): retorna um ponteiro para
 *                        o caractere '\0', que
 *                        termina um string
 *
 * Argumentos: string (entrada) - o string
 *
 * Retorno: ponteiro para o caractere terminal '\0'
 *
 ****/

char *FinalDeString( const char *string )
{
    return string + strlen(string);
}

int main(void)
{
    char str[] = "String que sera' invertido";
    char *p;
```

```

printf("\nString original: \"%s\"", str);

printf("\nString invertido: \"");

    /* Faz p apontar para o último */
    /* caractere do string          */
p = FinalDeString(str) - 1;

    /* Imprime os caracteres de trás para frente */
while (p >= str)
    putchar(*p--);

    /* Retoques finais */
putchar('\n');
putchar('\n');

return 0;
}

```

### *strncat()*

**Incluir:** <string.h>

**Descrição:** A função **strncat()** concatena um número limitado de caracteres ao final de um *string*.

**Protótipo:**

```

char *strncat( char *restrict destino,
               const char *restrict origem,
               size_t n )

```

**Parâmetros:**

- *destino* – array contendo o *string* que será acrescido de *n* caracteres obtidos de *origem*.
- *origem* – *string* de onde serão obtidos os caracteres a ser acrescentados ao *string* armazenado em *destino*.

- $n$  – número máximo de caracteres do *string* origem que serão concatenados a destino.

**Retorno:** Um ponteiro para o *string* destino.

**Observações:**

- O *string* de destino pode ter comprimento zero.
- No máximo, o comprimento do *string* resultante será:  

$$\text{strlen}(\text{destino}) + n.$$
- Certifique-se de que o array contendo o *string* destino tenha espaço suficiente para conter o *string* concatenado.
- Consulte também **strcat()**.

**Exemplo:**

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

int main()
{
    char destino[80] = "Isto e'";
    char *origem = " um teste apenas.";

    printf("\nString original: \n\t \"%s\"\n", destino);
    printf( "\nString que contem os caracteres a ser "
            "concatenados: \n\t \"%s\"\n",
            origem );

    strncat(destino, origem, 9);

    printf( "\nString original apos ser concatenado "
            "usando strncat(destino, origem, 9): "
            "\n\t \"%s\"\n", destino );

    return 0;
}
```

*strncpy()***Incluir:** <string.h>**Descrição:** A função **strncpy()** copia um número limitado de caracteres de um *string* para um array.**Protótipo:**

```
char *strncpy( char *restrict destino,
               const char *restrict origem,
               size_t n )
```

**Parâmetros:**

- destino – array que receberá os caracteres copiados.
- origem – *string* que terá caracteres copiados para destino.
- n – número máximo de caracteres que serão copiados.

**Retorno:** Um ponteiro para o array destino.**Observações:**

- Se n for maior do que ou igual ao comprimento do *string* origem, o resultado da cópia não será um *string* (i.e., o caractere '\0' não será copiado).
- Se n for menor do que o comprimento do *string* origem, a função copia os n primeiros caracteres deste *string* para o array destino e acrescenta caracteres '\0' ao array, de modo que, ao final, n caracteres terão sido copiados.
- Se houver sobreposição do array destino com o *string* origem, o resultado será indefinido.
- Certifique-se de que o array destino tenha espaço suficiente para conter os caracteres copiados.
- Consulte também **strcpy()**.

**Exemplo:**

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

int main()
{
    char destino[80];

    strncpy(destino, "Pequeno", 20);
    printf( "\nString 'destino' apos chamada de "
           "strncpy(destino, \"Pequeno\", 20):"
           "\n\t \"%s\"", destino );

    strncpy(destino, "Um string bem grande", 10);

    printf( "\nString 'destino' apos chamada de "
           "strncpy(destino, \"Um string bem grande\", "
           " 10):\n\t \"%s\"", destino );

    return 0;
}
```

*strpbrk()***Incluir:** <string.h>

**Descrição:** A função **strpbrk()** procura num *string* a primeira ocorrência de qualquer caractere presente noutra *string*.

**Protótipo:**

```
char *strpbrk(const char *string1, const char *string2)
```

**Parâmetros:**

- *string1* – *string* no qual será efetuada a busca.
- *string2* – *string* contendo os caracteres que serão procurados em *string1*.

**Retorno:** Um ponteiro para o primeiro caractere de *string2* encontrado em *string1* ou **NULL**, se nenhum caractere de *string2* for encontrado em *string1*.

**Observações:**

- Curiosidade: o nome desta função é derivado de *string pointer break* em inglês, o que, convenha-se, não é muito representativo.
- Compare esta função com **strchr()**, **strrchr()**, **strcspn()** e **strspn()**.

**Exemplo:**

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

int main()
{
    char *str1 = "Apenas um string";
    char *str2 = "Teste";
    char *ocorrenciaPtr;

    ocorrenciaPtr = strpbrk(str1, str2);

    if (ocorrenciaPtr)
        printf( "\nDentre os caracteres em \"%s\", '%c'"
               " e' o primeiro caractere \na aparecer em"
               " \"%s\"\n", str2, *ocorrenciaPtr, str1 );
    else
        printf( "Nao existe caractere em \"%s\" que "
               "apareca em \"%s\"\n", str2, str1 );

    return 0;
}
```

**strrchr()**

**Incluir:** <string.h>

**Descrição:** A função **strrchr()** procura a última ocorrência de um caractere em um *string*.

**Protótipo:**

```
char *strrchr(const char *string, int caractere)
```

**Parâmetros:**

- *string* – *string* a ser pesquisado.
- *caractere* – caractere a ser procurado.

**Retorno:** Ponteiro para a última ocorrência de *caractere* encontrado em *string*; **NULL**, se o caractere não for encontrado.

**Observações:**

- A função considera o caractere terminal '`\0`' como parte do *string*.
- Compare esta função com **strchr()**.

**Exemplo:**

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

int main()
{
    char *str = "Apenas um string";
    char c = 's';

    printf( "\nRestante de \"%s\" começando com a "
           "última \nocorrência do caractere \'%c\': "
           "\"%s\"\n", str, c, strrchr(str, c) );

    return 0;
}
```

*strspn()*

**Incluir:** <string.h>



**Descrição:** A função **strspn()** procura o segmento inicial de um *string* que contenha todos os caracteres de outro *string*.

**Protótipo:**

```
size_t strspn(const char *string1, const char *string2)
```

**Parâmetros:**

- *string1* – *string* a ser pesquisado.
- *string2* – *string* contendo caracteres que podem estar contidos em *string1*.

**Retorno:** O comprimento do segmento inicial contido em *string1* que contém todos os caracteres de *string2*.

**Observações:**

- Curiosidade: o nome desta função é derivado de *string span* em inglês, cujo significado é vago e ambíguo.
- Os caracteres não precisam aparecer em *string1* na mesma sequência em que aparecem em *string2*.
- O caractere terminal de *string* não é levado em consideração.
- Compare esta função com **strstr()**.
- Consulte também **strespn()** e **strpbrk()**.

**Exemplo:**

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

main()
{
    char *str1 = "ABCDEFGHJIJH";
    char *str2 = "DCFAB";
    int    posicao;

    posicao = strspn(str1, str2);
```

```

    printf( "\nPrimeiro caractere de \"%s\" que "
           "nao esta em \"%s\": '%c'\n",
           str1, str2, str1[posicao] );

    return 0;
}

strstr()

```

**Incluir:** <string.h>

**Descrição:** A função **strstr()** procura a primeira ocorrência de um *string* em outro *string*.

**Protótipo:**

```
char *strstr(const char *string1, const char *string2)
```

**Parâmetros:**

- *string1* – *string* que será pesquisado.
- *string2* – *string* que será procurado em *string1*.

**Retorno:** Um ponteiro para a primeira ocorrência de *string2* em *string1*; **NULL**, se *string2* não ocorrer em *string1*.

**Observações:**

- Os caracteres devem aparecer em *string1* na mesma sequência em que aparecem em *string2*.
- Compare esta função com **strspn()**.
- Consulte também **strcspn()** e **strpbrk()**.

**Exemplo:**

```

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

int main()
{
    char *str1 = "abcdefabcdef";

```

```

char *str2 = "def";

printf( "\nPorcao do string \"%s\" começando"
        "\ncom o string \"%s\": \"%s\"\n",
        str1, str2, strstr(str1, str2) );

return 0;
}

```

### *strtok()*

**Incluir:** <string.h>

**Descrição:** A função **strtok()** divide um *string* em partes (*tokens*), sendo estas partes separadas por um dos caracteres separadores especificados.

### **Protótipo:**

```

char *strtok( char *restrict string,
              const char *restrict separadores )

```

### **Parâmetros:**

- *string* – *string* a ser dividido em partes.
- *separadores* – *string* contendo separadores de partes.

**Retorno:** Ponteiro para uma parte de *string* se esta for encontrada; **NULL**, se nenhuma parte for encontrada.

### **Observações:**

- A primeira chamada de **strtok()** retorna um ponteiro para o primeiro *token* encontrado no *string*, e um caractere terminal '`\0`' é colocado no *string* seguindo imediatamente a parte retornada. Chamadas subsequentes da função com **NULL** como primeiro argumento retornarão as partes seguintes até que não haja nenhuma parte remanescente no *string* (v. exemplo a seguir).
- Esta função modifica o *string* passado como argumento. Portanto, se necessário, faça uma cópia do mesmo antes de chamar a função.
- O *string separadores* pode ser diferente em duas chamadas sucessivas.

- Não utilize **strtok()** como argumento de uma função que também utilize **strtok()**, pois isto poderá acarretar numa recursão infinita.

### Exemplo:

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

int main()
{
    char *token;
    int i = 0;
    char string[] = "String com 4 tokens.";
    char separadores[] = " ."; /* Note o espaço */
                                /* em branco      */

    printf( "\nString a ser separado em tokens:"
           "\n\t\"%s\"\n", string );
    printf("\nOs tokens sao:\n\n");

    token = strtok(string, separadores);

    while (token) {
        printf("\tToken %d: \"%s\"\n", ++i, token);
        token = strtok(NULL, separadores);
    }

    printf( "\n\nString original alterado por strtok():"
           "\n\t\"%s\"\n", string );

    return 0;
}
```

Quando compilado e executado no Windows, o programa do último exemplo produz a seguinte saída:

```
String a ser separado em tokens:
    "String com 4 tokens."
```

```
Os tokens sao:
    Token 1: "String"
```

```
Token 2: "com"
Token 3: "4"
Token 4: "tokens"
```

```
String original alterado por strtok():
"String"
```

## 6.4 INTRODUÇÃO À COLAÇÃO DE STRINGS

**Colaço** é a ordenação de caracteres de uma dada linguagem e é responsável pelo processo de arranjar *strings* numa dada ordem, como, por exemplo, em listas telefônicas e dicionários. Frequentemente, uma ordem de colaço é denominada **ordenação alfabética**, apesar de envolver qualquer categoria de caracteres e não apenas letras.

A ordenação de caracteres inerente dos pontos de código de um código de caracteres não é conveniente para colaço. Isto é, nenhum mapeamento entre caracteres e inteiros fornece subsídios suficientes para uso em colaço em qualquer linguagem natural. Nem mesmo os pontos de código providos pelo padrão Unicode, tipicamente aceito como o código de caracteres mais completo, correspondem a uma colaço aceitável em nenhuma língua natural. Isso ocorre porque, na maioria das localidades, as convenções lexicográficas de ordenação são diferentes da ordenação obtida comparando-se pontos de código de caracteres.

Mesmo quando a língua inglesa é utilizada, às vezes obtêm-se resultados estranhos quando se comparam valores numéricos atribuídos a caracteres objetivando obter ordenação alfabética<sup>49</sup>. Por exemplo, quando este tipo de comparação é utilizado, o *string* "ZEBRA" aparece antes do *string* "anta", pois letras maiúsculas aparecem antes de letras minúsculas em todos os códigos de caracteres usados na prática. Assim, para se obter um resultado que independa da distinção entre maiúsculas e minúsculas, esta ordenação vai requerer que as letras maiúsculas sejam transformadas em minúsculas, ou vice-versa, antes de serem comparadas [v. exemplo da função **stricmp()** mais adiante].

É importante salientar que colaço é a ordenação esperada de caracteres numa linguagem utilizada numa cultura ou localidade particular (v. **Seção 5.1.1**). Colaço é definida por um conjunto de **regras de colaço** que especificam como *strings* devem ser comparados e ordenados de modo a atender as expectativas de certa cultura. Regras de colaço especificam, entre outras coisas, como lidar com letras maiúsculas e

---

<sup>49</sup> Este tipo de ordenação é de modo jocoso chamada de ordenação *ASCIIbética*.

minúsculas, e com caracteres acentuados. Um conjunto de regras de colação constitui um **algoritmo de colação** (v. **Seção 7.7**).

### 6.4.1 COLAÇÃO VERSUS ORDENAÇÃO

O conceito de colação está intimamente associado ao conceito de ordenação no sentido de que a existência de uma ordem de colação é pré-requisito para a ordenação de um conjunto de *strings*. Entretanto, um algoritmo de colação não deve ser confundido com um algoritmo de ordenação.

Um algoritmo de colação [e.g., aquele implementado pela função **strxfrm()**] define a ordem na qual os caracteres são comparados, enquanto um algoritmo de ordenação [e.g., aquele implementado na função **qsort()** – v. **Seção 12.2.6**] coloca uma lista de itens (que não são necessariamente *strings*) numa determinada ordem.

No caso de *strings*, um algoritmo de colação especifica como dois *strings* devem ser comparados, enquanto um algoritmo de ordenação serve para colocar listas de *strings* na ordem especificada. Portanto, ordenação e colação de *strings* são relacionadas entre si porque todo algoritmo de ordenação de *strings* requer o auxílio de um algoritmo de colação. Por exemplo, considere o protótipo da função **qsort()** apresentada na **Seção 12.2.6**:

```
void qsort( void *array, size_t nElementos,
           size_t tamanhoDoElemento,
           int (*compara) (const void *, const void *))
```

A função **qsort()** implementa um algoritmo de ordenação bem conhecido (denominado *quicksort*). O primeiro argumento desta função é um array contendo os itens a ser ordenados, enquanto o terceiro argumento é um ponteiro para uma função que informa como cada dois itens devem ser comparados. Portanto, a função cujo endereço é passado como argumento para **qsort()** deve seguir algum algoritmo de colação. Diferentes ordenações dos mesmos itens podem ser obtidas utilizando-se funções que implementam diferentes algoritmos de colação.

## 6.4.2 FUNÇÕES DE COLAÇÃO DE STRINGS

A biblioteca padrão de C contém várias funções de colação declaradas no cabeçalho `<string.h>`. Algumas dessas funções são bastante rústicas e têm pouca utilidade prática na ordenação de *strings*, como mostra a **Tabela 6-5** a seguir<sup>50</sup>.

FUNÇÃO	COMPARA CARACTERES USANDO...
<b>memcmp()</b>	Valores dos bytes.
<b>strcmp()</b>	Inteiros que representam os caracteres.
<b>strncmp()</b>	Inteiros que representam os caracteres.
<b>strcoll()</b>	Sequências de bytes que representam a ordem de colação definida na localidade atribuída à categoria <b>LC_COLLATE</b> .
<b>strxfrm()</b>	Esta função não compara caracteres, mas cria sequências de bytes que, quando comparadas, resultam na ordem de colação definida na localidade atribuída à categoria <b>LC_COLLATE</b> .

Tabela 6-5: Funções de colação declaradas em `<string.h>`.

Como as funções **memcmp()**, **strcmp()** e **strncmp()** usam a ordenação imposta pelos pontos de código do código de caracteres em uso, elas não são convenientes para ordenação alfabética em nenhuma linguagem, conforme foi argumentado na seção anterior. Portanto, para obter-se uma ordenação de acordo com a ordem de colação de uma localidade, deve-se usar **strcoll()** para comparar *strings* ou **strxfrm()** para transformar *strings* de modo que eles possam ser ordenados corretamente por **memcmp()**, **strcmp()** ou **strncmp()**.

A categoria de localidade que afeta as funções **strcoll()** e **strxfrm()** é **LC\_COLLATE** (v. **Seção 5.2.2**). Na localidade padrão "C", não há diferença entre usar **strcoll()** ou **strcmp()**, e **strxfrm()** tem praticamente o mesmo efeito de **strcmp()**.

As funções **strcoll()** e **strxfrm()** fazem um mapeamento entre os *strings* recebidos como argumentos e sequências de bytes que correspondem às posições relativas destes *strings* na ordem de colação determinada pela localidade corrente. Então, as sequências de bytes resultantes podem ser comparadas, byte a byte, para refletir a ordem esperada. A função **strcoll()** armazena esses mapeamentos em arrays locais de

<sup>50</sup> Além das funções apresentadas na **Tabela 6-5**, existem funções de colação correspondentes para strings extensos declaradas em `<wchar.h>` que serão apresentadas no **Capítulo 8**. Tudo aquilo que foi afirmado nesta seção com relação às funções de colação declaradas em `<string.h>` poderá ser estendido para funções correspondentes declaradas em `<wchar.h>`.

duração automática que existem apenas enquanto esta função os utiliza para compará-los. Por outro lado, a função **strxfrm()** armazena o mapeamento de um *string* num array recebido como argumento. Assim, esta última função é conveniente quando se deseja comparar um *string* diversas vezes, pois, armazenando-se o resultado do mapeamento, pode-se economizar tempo.

Com o resultado do mapeamento obtido com a função **strxfrm()**, pode-se, a princípio, chamar **memcmp()**, **strcmp()** ou **strncmp()**, que não fazem nenhum tipo de mapeamento. Entretanto, neste caso, não é prático usar **memcmp()** ou **strncmp()** porque é necessário especificar o número de bytes que devem ser comparados.

Em resumo, quando se tem um conjunto de *strings* que precisam ser comparados diversas vezes, pode-se adotar o seguinte procedimento:

1. Gere um mapeamento (**chave de ordenação**) para cada elemento da lista de *strings* a ser ordenada usando a função **strxfrm()**.
2. Compare os *strings* usando suas respectivas chaves de ordenação; esta comparação pode ser feita usando-se **strcmp()**.
3. Armazene as chaves de ordenação junto com os respectivos *strings*, de modo que não seja mais preciso recriá-las cada vez que forem necessárias.

As funções declaradas no cabeçalho `<string.h>` que podem ser usadas em colação, já descritas anteriormente de maneira sucinta, serão apresentadas integralmente a seguir.

*strcmp()*

**Incluir:** `<string.h>`

**Descrição:** A função **strcmp()** compara dois *strings* e indica se eles são iguais ou se um deles é menor do que o outro. A comparação começa com o primeiro caractere de cada *string* e continua até que caracteres correspondentes difiram ou até que o final de um dos *strings* seja atingido.

**Protótipo:**

```
int strcmp(const char *string1, const char *string2)
```

**Parâmetros:** `string1` e `string2` – *strings* que serão comparados.



**Retorno:**

- Zero se os *strings* são iguais.
- Um valor menor do que zero se `string1` é menor do que `string2`.
- Um valor maior do que zero se `string1` é maior do que `string2`.

**Observações:**

- Os *strings* são comparados caractere a caractere; i.e., o primeiro caractere do primeiro *string* é comparado com primeiro caractere do segundo *string*, o segundo caractere do primeiro *string* é comparado com segundo caractere do segundo *string*, e assim por diante.
- Se todos os caracteres comparados forem iguais, os *strings* serão considerados iguais.
- Se dois caracteres comparados forem diferentes, o *string* contendo o caractere de menor valor será considerado menor.
- A função **strcmp()** compara valores inteiros associados aos caracteres que compõem os dois *strings*, de modo que dois *strings* serão iguais se, e somente se, contiverem exatamente os mesmos caracteres. Assim, esta função não é muito útil em ordenação ou busca aproximada de *strings*.

**Exemplo:**

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

int main(void)
{
    char str1[] = "Zebra";
    char str2[] = "abelha";
    int comparacao;

    comparacao = strcmp(str1, str2);

    if (comparacao < 0)
        printf( "\n\"%s\" precede \"%s\"\n",
                str1, str2 );
    else if (comparacao > 0)
```

```

        printf( "\n\"%s\" sucede \"%s\"\\n",
                str1, str2 );
    else
        printf( "\n\"%s\" e \"%s\" sao iguais\\n",
                str1, str2 );

    return 0;
}

```

Quando o programa anterior é executado, ele apresenta o seguinte resultado no meio de saída padrão:

```
"Zebra" precede "abelha"
```

Muito provavelmente este resultado não satisfaz a expectativa de um usuário comum. Conforme já visto, isto ocorre porque a função **strcmp()** simplesmente compara os valores inteiros associados aos caracteres que compõem os dois *strings* recebidos como argumentos. Como o *string* "Zebra" começa com letra maiúscula e letras maiúsculas antecedem às letras minúsculas no código de caracteres utilizado, o resultado obtido é facilmente justificado.

Uma forma de superar o problema causado por diferenciação indevida entre letras minúsculas e maiúsculas é criar cópias dos *strings* contendo letras minúsculas no lugar das respectivas letras maiúsculas (ou vice-versa) e comparar essas cópias em vez dos *strings* originais. Essa técnica é demonstrada no exemplo a seguir.

### Exemplo:

```

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

/****
*
* Função StringMinusculo(): cria uma cópia de um
*                          string substituindo
*                          letras maiúsculas por
*                          minúsculas
*
* Argumentos: s (entrada) - o string a ser convertido

```

```

*
* Retorno: cópia do string recebido como argumento
*           com letras maiúsculas substituídas por
*           minúsculas
*
****/

char *StringMinusculo(const char *s)
{
    char *p = malloc(strlen(s) + 1);
    char *pAux = p;

    if (!p)
        return NULL;

    while (*p++ = tolower(*s++))
        ;

    return pAux;
}

int main(void)
{
    char str1[] = "Zebra";
    char str2[] = "abelha";
    char *strMinusculo1, *strMinusculo2;
    int comparacao;

    strMinusculo1 = StringMinusculo(str1);
    strMinusculo2 = StringMinusculo(str2);

    if (!strMinusculo1 || !strMinusculo2) {
        printf( "\nNao foi possivel criar copias"
                "minúsculas dos strings\n" );
        return 1;
    }

    comparacao = strcmp(strMinusculo1, strMinusculo2);

    if (comparacao < 0)
        printf( "\n\"%s\" precede \"%s\"\n",
                str1, str2 );
    else if (comparacao > 0)

```

```

        printf( "\n\"%s\" sucede \"%s\"\\n",
                str1, str2 );
    else
        printf( "\n\"%s\" e \"%s\" sao iguais\\n",
                str1, str2 );

    free(strMinusculo1);
    free(strMinusculo2);

    return 0;
}

```

Esse programa produz como resultado:

```
"Zebra" sucede "abelha"
```

### *strncmp()*

**Incluir:** <string.h>

**Descrição:** A função **strncmp()** compara um número limitado de caracteres de dois *strings* e indica se eles são iguais ou se um deles é menor do que o outro. A comparação começa com o primeiro caractere de cada *string* e continua até que caracteres correspondentes difiram, até que o número limite de caracteres tenha sido atingido ou até que o final de um dos *string* seja atingido.

### **Protótipo:**

```
int strncmp( const char *string1, const char *string2,
             size_t n )
```

### **Parâmetros:**

- *string1* – primeiro *string* a ser comparado.
- *string2* – segundo *string* a ser comparado.
- *n* – número máximo de caracteres que serão comparados.

### **Retorno:**

- Zero se os *strings* são iguais.

- Um valor menor do que zero se `string1` é menor do que `string2`.
- Um valor maior do que zero se `string1` é maior do que `string2`.

#### Observações:

- Esta função é similar a **`strcmp()`**, com a diferença de que a função **`strncmp()`** limita o número de caracteres comparados.
- Consulte também **`strcmp()`**.

#### Exemplo:

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

int main(void)
{
    char string1[] = "Isto eh um teste";
    char string2[] = "Isto nao eh um teste";
    int comparacao;

    comparacao = strncmp(string1, string2, 5);

    printf("\nComparando ate' o quinto caractere, ");

    if (comparacao < 0)
        printf( "'%s' eh menor do '%s'\n",
                string1, string2 );
    else if (comparacao > 0)
        printf( "'%s' eh maior do '%s'\n",
                string1, string2 );
    else
        printf("os strings sao iguais\n");

    return 0;
}
```

#### *strcoll()*

**Incluir:** `<string.h>`

**Descrição:** A função **strcoll()** compara dois *strings* usando a sequência de ordenação da localidade correntemente atribuída à categoria **LC\_COLLATE** e indica se eles são iguais ou se um deles é menor do que o outro. A comparação começa com o primeiro caractere de cada *string* e continua até que caracteres correspondentes difiram ou até que o final de cada *string* seja atingido.

**Protótipo:**

```
int strcoll(const char *string1, const char *string2)
```

**Parâmetros:** *string1* e *string2* – *strings* que serão comparados.

**Retorno:**

- Zero se os *strings* são iguais.
- Um valor menor do que zero se *string1* é menor do que *string2*.
- Um valor maior do que zero se *string1* é maior do que *string2*.

**Observações:**

- Esta função depende da localidade correntemente atribuída à categoria **LC\_COLLATE**, mas é equivalente a **strcmp()** quando a sequência de colação padrão especificada pela localidade "C" é usada.
- Consulte também **strcmp()** e **strxfrm()**.

**Exemplo:** O programa a seguir demonstra o uso da função **strcoll()**<sup>51</sup>.

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

int main(void)
{
    char str1[] = "Zebra";
    char str2[] = "abelha";
    int comparacao;
    char *localidade;
```

---

<sup>51</sup> Compare este exemplo com aquele apresentado para a função **strcmp()**.

```

    /* Usa a localidade corrente */
    localidade = setlocale(LC_COLLATE, "");

    if (!localidade) {
        printf("\nNao foi possivel alterar localidade\n");
        return 1;
    }

    comparacao = strcoll(str1, str2);

    if (comparacao < 0)
        printf( "\n\"%s\" precede \"%s\"\n",
                str1, str2 );
    else if (comparacao > 0)
        printf( "\n\"%s\" sucede \"%s\"\n",
                str1, str2 );
    else
        printf( "\n\"%s\" e \"%s\" sao iguais\n",
                str1, str2 );

    return 0;
}

```

Quando compilado e executado no Windows, esse programa produz o seguinte resultado:

```
"Zebra" sucede "abelha"
```

### *strxfrm()*

**Incluir:** <string.h>

**Descrição:** A função **strxfrm()** cria e copia num array uma sequência de bytes que representa um *string* mapeado de acordo com a ordem de colação definida pela localidade correntemente atribuída à categoria **LC\_COLLATE**. O número de bytes copiados no array é limitado.

**Protótipo:**

```
size_t strxfrm( char *restrict destino,
               const char *restrict origem,
               size_t n )
```

**Parâmetros:**

- destino – array que receberá o resultado do mapeamento.
- origem – *string* a ser mapeado.
- n – número máximo de bytes que serão copiados no array.

**Retorno:** O número de bytes resultantes do mapeamento do *string*, sem incluir o caractere terminal '\0'.

**Observações:**

- Se o valor retornado por esta função for maior do que n, o conteúdo do *string* armazenado em destino será indeterminado. Neste caso, o programador deve chamar novamente a função, mas, desta vez, passando-lhe um array de maior capacidade.
- O número de bytes resultantes do mapeamento pode ser maior, menor ou igual àquele no *string* original.
- Quando o valor do terceiro argumento é zero, pode-se usar o valor retornado pela função para dimensionar o tamanho do array que deve conter o *string* transformado, como mostra o trecho de programa a seguir:

```
int    nBytes;
char *container = NULL;
...
/* Chama strxfrm() apenas para determinar */
/* o tamanho do array que conterà o      */
/* resultado da transformação do string   */
nBytes = strxfrm(umString, container, 0);

/* Aloca o array do tamanho adequado */
container = malloc(nBytes + 1);

/* Chama novamente strxfrm(); desta vez, */
/* para armazenar o resultado da operação */
```



```
/* no array apontado por container */
nBytes = strxfrm(umString, container, nBytes + 1);
```

- Quando a localidade corrente é "C", esta função difere de **strncpy()** pelo fato de não preencher o array com caracteres nulos quando o *string* tem comprimento menor do que *n*.

### Exemplo:

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

int main(void)
{
    char  str1[] = "Zebra";
    char  str2[] = "abelha";
    char  strColl1[50];
    char  strColl2[50];
    int    comparacao;
    char  *localidade;

    printf("\n** Usando a localidade padrao (\"C\") **");

    strxfrm(strColl1, str1, 50);
    strxfrm(strColl2, str2, 50);

    comparacao = strcmp(strColl1, strColl2);

    if (comparacao < 0)
        printf( "\n\"%s\" precede \"%s\"\n",
                str1, str2 );
    else if (comparacao > 0)
        printf( "\n\"%s\" sucede \"%s\"\n",
                str1, str2 );
    else
        printf( "\n\"%s\" e \"%s\" sao iguais\n",
                str1, str2 );

    /* Usa a localidade corrente do SO */
    localidade = setlocale(LC_COLLATE, "");
```

```

    if (!localidade) {
        printf("\nNao foi possivel alterar localidade\n");
        return 1;
    }

    printf("\n** Usando a localidade do SO **");

    strxfrm(strColl1, str1, 50);
    strxfrm(strColl2, str2, 50);

    comparacao = strcmp(strColl1, strColl2);

    if (comparacao < 0)
        printf( "\n\"%s\" precede \"%s\"\n",
                str1, str2 );
    else if (comparacao > 0)
        printf( "\n\"%s\" sucede \"%s\"\n",
                str1, str2 );
    else
        printf( "\n\"%s\" e \"%s\" sao iguais\n",
                str1, str2 );

    return 0;
}

```

O programa apresentado do último exemplo produz o seguinte resultado:

```

** Usando a localidade padrao ("C") **
"Zebra" precede "abelha"

** Usando a localidade do SO **
"Zebra" sucede "abelha"

```

## 6.5 FUNÇÕES DE CONVERSÃO DE STRINGS EM NÚMEROS

As funções discutidas nesta seção são todas declaradas no cabeçalho `<stdlib.h>`, que é explorado em detalhes no **Capítulo 12**, e foram incluídas aqui por proximidade contextual.

### 6.5.1 CONVERSÕES DE STRINGS EM NÚMEROS INTEIROS

As funções descritas nesta seção são divididas em duas categorias; em cada uma delas, a única diferença entre as funções é o tipo de retorno.

*atoi(), atol() e atoll() (C99)*

**Incluir:** `<stdlib.h>`

**Descrição:** As funções **atoi()**, **atol()** e **atoll()** convertem *strings* numéricos (i.e., constituídos por dígitos e sinal) em valores dos tipo **int**, **long** e **long long**, respectivamente.

**Protótipos:**

```
int atoi(const char *string)
```

```
long atol(const char *string)
```

```
long long atoll(const char *string)
```

**Parâmetro:** *string* – *string* que se deseja converter em inteiro.

**Retorno:** Valor inteiro (**int**, **long** ou **long long**, dependendo da função) representando a conversão do *string*; zero, se o *string* não puder ser convertido.

**Observações:**

- O número pode conter um sinal.
- A conversão prossegue até que o primeiro caractere inválido seja encontrado.
- Se houver *overflow*, o resultado será indefinido.
- Essas funções não utilizam necessariamente a variável **errno** para indicar erros de conversão (mas podem fazê-lo, dependendo da implementação), o que torna praticamente impossível detectar tais erros.
- Em vez de usar essas funções, dê preferência ao uso das funções **strtol()** e **strtoll()**. Por exemplo, substitua chamadas de **atoi()**, como:

```
atoi(s)
```

por chamadas de **strtol()**, como:

```
(int)strtol(s, 0, 10)
```

**Exemplo:** O programa a seguir demonstra o uso da função **atoi()**.

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

int main()
{
    char *str = "1234";
    int    i;

    i = atoi(str);

    printf( "Convertendo o string \"%s\" para int"
           " o resultado e': %d\n", str, i );

    return 0;
}
```

**Exemplo:** O programa a seguir demonstra o uso das funções **atol()** e **atoll()**.

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

int main()
{
    char      *str = "-92233720368547758";
    long      umLong;
    long long umLongLong;

    umLong = atol("-92233720368547758");
    umLongLong = atoll("-92233720368547758");

    printf( "\nString \"%s\" convertido para long:"
           "\n\t%ld\n", str, umLong );
    printf( "\nString \"%s\" convertido para long long:"
           "\n\t%lld\n\n", str, umLongLong );
}
```



```
unsigned long long strtoull( const char *restrict string,
                           char **restrict final,
                           int base )
```

**Parâmetros:**

- *string* – *string* a ser convertido.
- *final* – se este parâmetro não for **NULL**, ao final da conversão ele estará apontando para a porção do *string* que não foi convertida.
- *base* – base do número a ser convertido.

**Retorno:** O inteiro convertido, se não ocorrer erro; zero, caso contrário.

**Observações:**

- O parâmetro *base* especifica qual base é usada e deve ser 0 ou de 2 a 36. Se este parâmetro for igual a zero, a base do número será determinada pelo seu formato. Isto é, se o número começar com 0x ou 0X, ele será considerado hexadecimal (base 16); se ele começar com zero, a base octal será assumida; em outros casos, a base será considerada decimal.
- Se *base* for igual a 1, menor que zero ou maior que 36, este valor será considerado inválido. Qualquer valor inválido deste parâmetro faz com que o retorno seja zero e *final* aponte para o início de *string*, o que permite checar se uma conversão foi bem sucedida ou não.
- Apenas os dígitos e letras correspondentes à base fornecida são reconhecidos durante a conversão.
- A conversão prossegue até que o primeiro caractere inválido seja encontrado.
- Compare essas funções com as funções **atoi()**, **atol()** e **atoll()**.

**Exemplo:** O programa a seguir demonstra o uso das funções **strtoul()** e **strtoull()**.

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

int main()
{
```

```

long          li;
unsigned long ul;
char          *string = "1234567abc", *resto;

li = strtol(string, &resto, 0);
ul = strtoul(string, &resto, 0);

printf( "\nString original: \"%s\"\n", string);
printf( "\nValor convertido para long int: "
        "%ld\n", li );
printf( "\nValor convertido para unsigned long: "
        "%lu\n", ul );
printf( "\nResto do string original que nao "
        "foi convertido: \"%s\"\n", resto );

return 0;
}

```

## 6.5.2 CONVERSÕES DE STRINGS EM NÚMEROS DE PONTO FLUTUANTE REAIS

*atof()*

**Incluir:** <stdlib.h>

**Descrição:** A função **atof()** converte um *string* num valor do tipo **double**.

**Protótipo:**

```
double atof(const char *string)
```

**Parâmetro:** *string* – o *string* que será convertido.

**Retorno:** Um valor do tipo **double** resultante da conversão do *string*, se esta conversão for possível; zero, caso contrário.

**Observações:**

- Se houver espaços em branco no início do *string*, eles serão saltados. Caracteres são considerados espaços em branco de acordo com a função **isspace()** (v. Seção 6.2.1).

- Um *string* válido pode incluir:
  - Sinal
  - Ponto decimal de acordo com a localidade corrente
  - e ou E (notação científica)
  - +INF ou -INF, representando, respectivamente,  $+\infty$  ou  $-\infty$
  - +NAN ou -NAN representando um valor não numérico (v. **Seção 3.3**)
  - 0x ou 0X mais uma sequência de dígitos hexadecimais (C99)
- A conversão prossegue até que o primeiro caractere inválido seja encontrado.
- Se o valor resultante da conversão não puder ser representado, o comportamento da função **atof()** será indefinido.
- Existem duas diferenças fundamentais entre **atof()** e **strtod()** (v. mais adiante):
  - A função **strtod()** tem um parâmetro a mais que indica o local no *string* onde a conversão é encerrada.
  - A função **strtod()** usa a variável **errno** para indicar a ocorrência de erro de conversão. Implementações da função **atof()** podem fazê-la agir do mesmo modo, mas, de acordo com o padrão de C, não existe esta obrigatoriedade.
- Devido à provável impossibilidade de detectar-se erro de conversão, deve-se dar preferência ao uso de **strtod()**, em vez de **atof()**.

### Exemplo:

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

int main()
{
    char    *str = "2.54cm";
    double  d;

    d = atof(str);
```



```

printf( "Convertendo o string \"%s\" para "
        "double obtem-se: %3.2f\n", str, d );

return 0;
}

```

### *strtod()*, *strtof()* (C99), *strtold()* (C99)

**Incluir:** <stdlib.h>

**Descrição:** As funções **strtod()**, **strtof()** e **strtold()** convertem *strings* em números de ponto flutuante dos tipos **double**, **float** e **long double**, respectivamente.

#### **Protótipos:**

```
double strtod( const char *restrict string,
               char **restrict final )
```

```
float strtof( const char *restrict string,
              char **restrict final )
```

```
long double strtold( const char *restrict string,
                     char **restrict final )
```

#### **Parâmetros:**

- *string* – *string* a ser convertido.
- *final* – se este parâmetro não for **NULL**, ao final da conversão ele estará apontando para a porção do *string* que não foi convertida.

#### **Retorno:**

- Se a conversão for possível e representável no tipo de retorno da função, o número de ponto flutuante convertido.
- Zero, se nenhuma conversão for possível (v. **Observações** adiante).

- Se o número resultante da conversão for excessivamente grande para ser representado no tipo de retorno da função (i.e., se ocorrer *overflow*), ela retornará **HUGE\_VAL** [**strtod()**], **HUGE\_VALF** [**strtof()**] ou **HUGE\_VALL** [**strtold()**] (v. **Seção 3.6.2**).
- Se o número resultante da conversão for excessivamente pequeno para ser representado no tipo de retorno da função (i.e., se ocorrer *underflow*), ela retornará um valor cuja magnitude é o menor valor maior do que zero que pode ser representado no tipo de retorno da função.

### Observações:

- Se houver espaços em branco no início do *string*, eles serão saltados. A verificação de espaços em branco é feita de acordo com a função **isspace()** (v. **Seção 6.2.1**).
- Um *string* válido pode incluir:
  - Sinal
  - Ponto decimal de acordo com a localidade corrente
  - *e* ou *E* (notação científica)
  - $+\infty$  ou  $-\infty$ , representando, respectivamente,  $+\infty$  ou  $-\infty$
  - $+\text{NAN}$  ou  $-\text{NAN}$  representando um valor não numérico (v. **Seção 3.3**)
  - $0x$  ou  $0X$  mais uma sequência de dígitos hexadecimais (C99)
- Se nenhuma conversão for possível, essas funções retornarão zero, que, obviamente, é um valor válido de qualquer tipo de ponto flutuante. Em algumas implementações, essas funções atribuem um valor diferente de zero à variável **errno** (v. **Seção 11.5**), o que permite distinguir um valor válido de um erro de conversão testando-se o valor da variável **errno**. Mas, mesmo quando este não é o caso, pode-se verificar se ocorreu erro de conversão checando-se o parâmetro *final*. Isto é, se, ao retorno da função, este parâmetro estiver apontando para o início do *string* original, pode-se concluir que não houve nenhuma conversão.
- Quando ocorre uma condição de *overflow*, essas funções atribuem **ERANGE** à variável **errno** (v. **Seção 11.5**). Entretanto, diferentemente do que ocorre quando elas retornam zero, testar a variável **errno** torna-se desnecessário, visto que, neste caso, os valores retornados não representam valores válidos.

- Compare essas funções com **atof()**.

**Exemplo:** O programa a seguir demonstra o uso da função **strtod()**.

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

int main()
{
    double d;
    char    *str = "51.2 + alguma coisa", *final;

    d = strtod(str, &final );

    printf( "O string \"%s\" foi convertido para %f\n",
           str, d );
    printf( "O string \"%s\" NAO fez parte da "
           "conversao\n", final );

    return 0;
}
```

### 6.5.3 OUTRAS FUNÇÕES DE CONVERSÃO DE STRINGS EM NÚMEROS

Existem várias outras funções que convertem *strings* em números:

- As funções **strtoimax()**, **strtoumax()**, **wcstoimax()** e **wcstoumax()** declaradas em `<inttypes.h>` (v. **Seção 2.5.3**).
- Funções de conversão de *strings* extensos em números declaradas em `<wchar.h>` (v. **Seção 8.5.6**).
- Funções de formatação em memória declaradas em `<stdio.h>` (v. **Seção 10.7.8**) e `<wchar.h>` (v. **Seção 10.8**).

## 6.6 EXERCÍCIOS DE REVISÃO

1. Descreva os seguintes conceitos:
  - (a) Conjunto de caracteres fonte
  - (b) Conjunto de caracteres de execução
  - (c) Conjunto básico de caracteres
  - (d) Conjunto estendido de caracteres
  - (e) Código de caracteres
  - (f) Conjunto codificado de caracteres
  - (g) Ponto de código
2. Qual é o propósito geral do cabeçalho `<ctype.h>`?
3. (a) Que categorias de localidade afetam funções declaradas em `<ctype.h>`?  
(b) Quais são as funções afetadas?
4. Qual é o problema com a seguinte definição de macro que tenta ter a mesma funcionalidade da função **toupper()** discutida na **Seção 6.2.2**?  

```
#define toupper(c) ((c) + 'A' - 'a')
```
5. Que função declarada no cabeçalho `<ctype.h>` você utilizaria num programa para testar se um caractere é classificado como:
  - (a) Alfanumérico
  - (b) Letra
  - (c) Dígito
  - (d) Caractere de controle
  - (e) Caractere imprimível
  - (f) Letra maiúscula
  - (g) Símbolo de pontuação
6. Qual é a diferença entre as funções **isspace()** e **isblank()**?

7. Que função declarada em `<ctype.h>` é usada para converter letras maiúsculas em minúsculas?
8. O que têm em comum todas as funções de processamento de blocos declaradas em `<string.h>`?
9. Em que diferem as funções **memcpy()** e **strcpy()**?
10. Qual é a diferença entre as funções **memcpy()** e **strcmp()**?
11. O que distingue as funções **memcpy()** e **memmove()**?
12. Descreva o funcionamento da função **memchr()**.
13. (a) O que é colação? (b) Qual é a diferença entre este conceito e ordenação? (c) Qual é a relação entre colação e ordenação?
14. A função **strcmp()** é útil em ordenação de *strings*? Explique.
15. Quais funções declaradas em `<string.h>` podem ser usadas em ordenação de *strings* e como elas devem ser usadas?
16. As funções declaradas em `<string.h>` a seguir possuem semelhanças e diferenças fundamentais. Enumere estas semelhanças e diferenças.
  - (a) **strchr()**
  - (b) **strcspn()**
  - (c) **strpbrk()**
  - (d) **strrchr()**
  - (e) **strspn()**
  - (f) **strstr()**
17. Descreva o funcionamento da função **strtok()**.
18. (a) Como você poderia implementar a função **strlen()** usando apenas uma chamada da função **strchr()**? (b) Você seria capaz de fazer o mesmo usando apenas uma chamada de **strrchr()**?
19. A função **strcpy()** sempre retorna um ponteiro para um *string*. Pode-se afirmar o mesmo com relação à função semelhante **strncpy()**?

20. Que cuidado se deve tomar para evitar corrupção de memória quando se chama a função **strcat()**?
21. Qual é a utilidade prática da função **strerror()**?
22. (a) Que categoria de localidade afeta as funções **strcoll()** e **strxfrm()**? (b) Esta categoria de localidade afeta a função **strcmp()**?
23. (a) Por que se deve evitar o uso da função **atoi()**? (b) Que função deve ser usada preferencialmente no lugar de **atoi()**?
24. (a) Qual é o significado do parâmetro `base` no protótipo das funções **strtol()**, **strtoll()**, **strtoul()** e **strtoull()**? (b) Idem para o parâmetro `final`.
25. Por que se deve dar preferência ao uso da função **strtod()** em vez da função **atof()**?

# *Capítulo 7*

---

*Caracteres extensos e multibytes I:  
conceitos*

## 7.1 INTRODUÇÃO

Conforme discutido na **Seção 6.1.4**, caracteres monobytes não são capazes de representar todos os caracteres necessários em muitos programas. A linguagem C provê duas maneiras de lidar com caracteres que precisam ser representados em mais de um byte: caracteres multibytes — que são sequências de bytes de comprimentos variáveis — e caracteres extensos — que são valores do tipo **wchar\_t**.

Caracteres multibytes são usados principalmente em armazenamento externo ou em transmissão de dados e têm como objetivo economizar espaço, mas este tipo de representação não é conveniente para processamento interno. Por outro lado, caracteres extensos são tão fáceis de processar quanto caracteres monobytes, mas podem causar grande desperdício de memória.

Este capítulo apresenta conceitos importantes relacionados a caracteres extensos e multibytes e faz uma breve introdução aos mais modernos códigos de caracteres usados atualmente: Unicode e ISO 10646. Finalmente, descreve-se o algoritmo de colação Unicode, que representa um exemplo da abordagem mais aceita atualmente de tratamento de colação.

Antes de prosseguir, é relevante a apresentação dos seguintes conceitos que são importantes para melhor entendimento do material exposto neste capítulo:

- Um **grafema** é uma unidade abstrata que representa um símbolo de um sistema de escrita (e.g., letras, dígitos, símbolos de pontuação, etc.).
- Um **script** é o mesmo que um sistema de escrita e consiste em um conjunto de grafemas que pode ser usado por diversas línguas (e.g, o script romano é usado por várias línguas: português, inglês, espanhol, etc.).

## 7.2 CARACTERES E STRINGS EXTENSOS

**Caracteres extensos** são caracteres que ocupam mais de um byte (tipicamente, ocupam dois ou quatro bytes). Estes caracteres são valores do tipo derivado **wchar\_t**, encontrado na biblioteca padrão de C. Um **string extenso** é um *string* composto de caracteres extensos (v. **Capítulo 8**).

O número de bytes usados para representar um caractere extenso pode variar de uma implementação para outra, mas é sempre fixo numa dada implementação. Caracteres extensos são tão fáceis de processar quanto caracteres monobytes, porque cada



caractere ocupa sempre o mesmo espaço. Entretanto, o uso de caracteres extensos apresenta algumas desvantagens, que serão discutidas na **Seção 7.4**.

## 7.3 CARACTERES E STRINGS MULTIBYTES

Um **caractere multibyte** ocupa entre 1 e  $n$  bytes, de forma que a denominação *multibyte* é um tanto confusa, considerando que caracteres extensos também ocupam *múltiplos* bytes. A diferença entre estas duas formas de representação de caracteres é que, enquanto todos os caracteres extensos numa dada implementação ocupam um número fixo de bytes, nem todos os caracteres multibytes ocupam o mesmo número de bytes<sup>52</sup>. Isto é, uma codificação multibyte usa sequências de bytes de tamanhos variados para representar pontos de códigos que não podem ser representados num único byte.

Codificações de caracteres multibytes foram desenvolvidas com três objetivos:

1. Aumentar o número de caracteres que podem ser representados.
2. Manter a compatibilidade com programas existentes que lidam com caracteres monobytes.
3. Economizar memória.

Para permitir que mais de 256 caracteres possam ser representados, a escolha óbvia seria utilizar dois ou mais bytes para representar cada caractere. Mas, se dois bytes, por exemplo, forem utilizados para representar cada caractere, não há compatibilidade com programas já existentes que utilizam caracteres representados num único byte. Como um dos objetivos de uma codificação multibyte é manter compatibilidade com programas já existentes que processam caracteres monobytes, alguns caracteres devem ser representados em apenas um byte, enquanto outros são representados em múltiplos bytes.

Como um caractere multibyte ocupa de 1 a  $n$  bytes, com  $n$  podendo até ser maior do que o número de bytes utilizados numa representação equivalente com caracteres extensos, o uso de caracteres multibytes só será vantajoso se o número de caracteres que requerem poucos bytes for bem maior do que aqueles que requerem o número máximo de bytes.

---

<sup>52</sup> Uma melhor denominação para caracteres multibytes poderia ser *caracteres com um número variável de bytes*, mas a simplicidade da denominação *caracteres multibytes* prevalece.

Um caractere nulo é um byte com todos os bits iguais a zero e não pode fazer parte de uma sequência de bytes de um caractere multibyte, a não ser que ele seja o único byte da sequência. Caracteres nulos são usados para *terminar strings* de caracteres multibytes, do mesmo modo que ocorre com *strings* de caracteres monobytes.

Trechos de um programa responsáveis por entrada e saída precisam ter conhecimento da codificação de caracteres utilizada, e, neste aspecto, o uso de caracteres multibytes apresenta um problema potencial. Por exemplo, suponha que se leiam quatro bytes de um arquivo contendo caracteres multibytes. Como se pode dizer se estes bytes representam quatro caracteres de um byte cada um, dois caracteres de dois bytes cada um, um caractere de um byte e outro de três bytes, etc.? Assim, para que seja possível resolver esse dilema, duas formas de representação de caracteres foram propostas: (1) representação de caracteres multibytes **com estado** (v. **Seção 7.3.1**) e (2) representação de caracteres multibytes **sem estado** (v. **Seção 7.3.2**).

### 7.3.1 CODIFICAÇÕES MULTIBYTES COM ESTADO

Uma codificação de **caracteres multibytes com estado** possui sequências de bytes que alteram a interpretação dos bytes seguintes num *string* ou *stream*. Uma tal sequência de bytes é denominada **estado de mudança** ou **caractere de mudança de estado**. Um caractere de mudança de estado afeta o modo como os próximos caracteres são interpretados até que outro caractere de mudança de estado seja encontrado.

Para facilitar o entendimento, pode-se fazer uma analogia entre uma codificação com estado e o uso da tecla [SHIFT] de um teclado: enquanto esta tecla está ativada, o pressionamento de algumas teclas produz resultados diferentes daqueles obtidos quando as mesmas teclas são pressionadas sem o uso de [SHIFT]. Nesta analogia, há dois estados de mudança que correspondem ao pressionamento e não-pressionamento da tecla [SHIFT].

O **estado de mudança inicial** é aquele no qual todos os caracteres têm a interpretação considerada padrão. No estado de mudança inicial, se o valor de um byte casa com o inteiro atribuído a algum caractere do conjunto de caracteres básico de C (v. **Seção 6.1.1**), este byte representa o respectivo caractere.

Um exemplo de codificação de caracteres com estado é a codificação ISO 2022-JP, utilizada com a língua japonesa. Nesta codificação, o estado de mudança de três bytes<sup>53</sup> \x1B\$B faz com que cada sequência de dois bytes seguinte seja interpretada

---

<sup>53</sup> Se você tem dificuldade em identificar os três bytes, eles são, em notação de caracteres constantes: '\x1B', '\$' e 'B'.

como um caractere, enquanto o estado de mudança de três bytes<sup>54</sup> \x1B (B faz com que cada byte (único) a seguir seja interpretado como um caractere.

Codificações de caracteres multibytes com estado são complicadas de usar num programa, porque o programa deve sempre levar em consideração o estado corrente. Por exemplo, quando se analisa um *string* de caracteres multibytes com estado a partir de um dado ponto, pode ser necessário retroceder no *string* para determinar como interpretar a sequência de bytes corrente. Ou seja, para interpretar um dado byte num local arbitrário de um *string*, é necessário voltar para alguma posição anterior do *string* para consultar o último caractere de mudança de estado.

Algumas vezes, sequências de mudanças podem usar até seis bytes, de modo que mudanças frequentes de estado num *string* podem requerer mais bytes para representar sequências de mudanças do que para representar os próprios caracteres. Portanto, codificações com estado são ineficientes tanto para processamento quanto para armazenamento interno.

### 7.3.2 CODIFICAÇÕES MULTIBYTES SEM ESTADO

Numa codificação de **caracteres multibytes sem estado**, o formato dos bytes que compõem cada caractere indica o comprimento da própria sequência de bytes que representa o caractere. Isto é, numa codificação multibyte sem estado, cada sequência de bytes que representa um caractere segue uma sintaxe bem definida que indica como a própria sequência deve ser interpretada. Uma codificação de multibyte sem estado notável é a codificação UTF-8, que será discutida na **Seção 7.6.1**.

Apesar de serem um pouco mais complicadas do que as representações de caracteres extensos, as representações de caracteres multibytes sem estado são bem mais simples do que do que as representações de caracteres multibytes que usam estados. A maioria das codificações multibytes modernas segue este modelo.

## 7.4 CARACTERES EXTENSOS VERSUS CARACTERES MULTIBYTES

O tamanho de um caractere extenso deve ser suficiente para armazenar o maior ponto de código do conjunto de caracteres extensos usado. Tipicamente, este requisito

---

<sup>54</sup> Os três bytes, em notação de caracteres constantes, são: '\x1B', '(' e 'B'.

implica o uso de inteiros de 32 bits e, consequentemente, um grande desperdício de memória, pois armazenar caracteres do padrão Unicode ou ISO 10646 requer apenas 21 bits (v. **Seção 7.5**). Além disso, o subconjunto de caracteres mais frequentemente utilizado destes padrões requer apenas 16 bits. Portanto, armazenar caracteres em 32 bits pode causar um grande desperdício de memória.

Outra desvantagem decorrente do uso de caracteres extensos é que eles são afetados por ordenação de bytes (v. **Seção 13.3**). Além dessas desvantagens, o tamanho do tipo `wchar_t`, usado para representar caracteres extensos, é dependente de implementação.

Caracteres multibytes constituem um modo eficiente para transferir caracteres entre um programa e um meio de entrada ou saída. Entretanto, dentro de um programa, é mais fácil e eficiente manipular caracteres de um mesmo tamanho e formato, como é o caso de caracteres extensos.

## 7.5 CÓDIGOS DE CARACTERES EXTENSOS

A maioria dos programas e linguagens de programação lida com caracteres que refletem as convenções da língua inglesa. Entretanto, a maioria das pessoas não fala inglês como linguagem nativa e, com a disseminação do uso de computadores, é cada vez mais importante permitir que os usuários os usem em suas próprias línguas e culturas. Um passo crucial nessa direção foi o surgimento dos padrões Unicode e ISO 10646 que visam abranger todos os caracteres usados em todas as línguas faladas no mundo. Os códigos de caracteres especificados por estes padrões requerem mais de um byte para representar os pontos de código dos caracteres que constam em seus repertórios. Estes códigos de caracteres são aqui denominados **códigos de caracteres extensos**.

### 7.5.1 UNICODE

O padrão **Unicode** especifica um conjunto de caracteres, atualmente com cerca de 100.000 caracteres, e várias codificações. Este conjunto de caracteres é aberto no sentido de que novos acréscimos possam sempre ocorrer, mas os acréscimos mais recentes ao repertório de caracteres Unicode referem-se a caracteres raramente usados.

O modelo de codificação seguido por Unicode compreende quatro níveis, apresentados resumidamente na **Tabela 7-1**.

NÍVEL DE CODIFICAÇÃO	BREVE DESCRIÇÃO	EXEMPLO
1. Repertório de caracteres abstratos	Os caracteres são enumerados e descritos.	ç <sup>55</sup>
2. Conjunto de caracteres codificados	Os caracteres têm pontos de código atribuídos.	U+00E7
3. Forma de codificação de caracteres	Pontos de código são mapeados em unidades de código.	000000E7
4. Esquema de codificação de caracteres	Unidades de código são mapeadas em sequências de bytes.	0xC3 0xA7 [UTF-8]

Tabela 7-1: Modelo de codificação Unicode.

Na notação U+xxxx, utilizada para representar pontos de código (Nível 2), xxxx é o ponto de código em formato hexadecimal.

O terceiro nível do modelo de codificação Unicode (i.e., a forma de codificação) também é conhecido como **formato de armazenamento**. Este nível mapeia pontos de código em sequências de inteiros de tamanho específico chamadas **unidades de código**. O tamanho das unidades de código pode variar entre as diversas formas de codificação de caracteres (e.g., 8, 16 e 32 bits), mas, uma vez definida a forma de codificação, este tamanho é uniforme.

O mapeamento de unidades de código em sequências de bytes (Nível 4) não é imediato, mesmo quando o tamanho destas sequências é igual ao tamanho das unidades de código. Por exemplo, se um ponto de código for representado em unidades de código pelos bytes B<sub>1</sub>B<sub>2</sub>B<sub>3</sub>B<sub>4</sub>, num dado esquema de codificação (e.g., UTF-32LE), ele pode ser representado por B<sub>4</sub>B<sub>3</sub>B<sub>2</sub>B<sub>1</sub> para refletir uma ordenação diferente de bytes (v. **Seção 7.6**).

O padrão Unicode é compatível tanto com o código ASCII quanto com a parte inicial do repertório de caracteres do padrão ISO 8859-1 (v. **Seção 6.1.2**). Isto é, os caracteres coincidentes nestes conjuntos de caracteres têm pontos de código também coincidentes em Unicode. Por exemplo, o caractere 'A' tem ponto de código 0x0041 tanto no código ASCII quanto no código ISO 8859-1 quanto em Unicode<sup>56</sup>.

55 Esta letra recebe como denominação oficial: LATIN SMALL LETTER C WITH CEDILA; i.e., letra latina cedilha minúscula.

56 Note, entretanto, que não há coincidência entre pontos de código de caracteres do padrão Unicode e de outras codificações 8859-X do padrão ISO 8859, mesmo porque isso seria impossível para a maioria dos caracteres dessas codificações.

Em Unicode, alguns caracteres (e.g., letras acentuadas usadas em português) podem ser considerados como um único caractere ou compostos de dois caracteres. Por exemplo, a letra *á* pode ser considerada como um único caractere (com ponto de código U+00E1) ou uma combinação dos caracteres *a* (U+0061) e acento agudo (U+00B4). Neste exemplo, a letra *a* é denominada **caractere base** e o acento agudo é denominado **marca diacrítica**.

Com o objetivo de facilitar o uso de seu vasto conjunto de caracteres, o padrão Unicode divide-o em **planos**. Quando se sabe que todos os caracteres pertencem a um mesmo plano, pode-se omitir o número que identifica o plano e considerar apenas os números que identificam as linhas e colunas no plano. Por exemplo, o plano **BMP**<sup>57</sup> contém 65536 caracteres usados na maioria das linguagens faladas no mundo e cada caractere pode ser identificado por um número de 16 bits. Muitos caracteres estão ausentes do plano BMP, mas estes caracteres fazem parte de scripts específicos (e.g., caracteres Han) e notações científicas, e são raramente usados no cotidiano.

Para simplificar a busca de caracteres, o padrão Unicode agrupa pontos de código por script ou função em subconjuntos de caracteres denominados **blocos**. Por exemplo, caracteres romanos fazem parte de um bloco, símbolos matemáticos fazem parte de outro bloco, etc. A **Tabela 7-2** mostra alguns scripts com respectivos intervalos de pontos de código presentes no padrão Unicode<sup>58</sup>.

SCRIPT	PONTO DE CÓDIGO ENTRE...
Árabe	U+0600 – U+06FF
Latim básico	U+0000 – U+007F
Bengali	U+0980 – U+09FF
Ideogramas unificados CJK	U+4E00 – U+9FAF
Cirílico	U+0400 – U+04FF
Grego	U+0370 – U+03FF
Hebreu	U+0590 – U+05FF
Hirakana	U+3040 – U+309F
Tailandês	U+0E00 – U+0E7F

Tabela 7-2: Alguns scripts e respectivos intervalos de pontos de código em Unicode

<sup>57</sup> A sigla BMP é derivada de *Basic Multilingual Plane*, em inglês; i.e., plano multilingual básico.

<sup>58</sup> Esta tabela foi obtida a partir de dados coletados no site do consórcio Unicode na internet: <http://unicode.org/charts/>.

## 7.5.2 ISO 10646

O padrão **ISO 10646**, também conhecido como **UCS**<sup>59</sup>, define um valor de 21 bits para, virtualmente, cada caractere existente no mundo. Desde 1991, Unicode e ISO 10646 têm repertórios de caracteres e respectivos pontos de código coincidentes e mantidos em sincronia. Além disso, em quase todos os aspectos práticos de programação, ISO 10646 é similar a Unicode (v. **Seção 7.5.3**).

O padrão ISO 10646 é dividido em duas partes: ISO 10646-1 — que define a arquitetura básica e padroniza a codificação no plano BMP — e ISO 10646-2, que padroniza a codificação em outros 16 planos.

## 7.5.3 DIFERENÇAS ENTRE UNICODE E ISO 10646

Os padrões Unicode e ISO 10646 constituem as formas mais modernas de codificação de caracteres. A principal diferença entre ISO 10646 e Unicode é que Unicode acrescenta algumas regras sobre como os caracteres devem ser usados, o que está além do escopo do padrão ISO 10646. Estas regras servem, por exemplo, como referência para profissionais que implementam sistemas tipográficos. Unicode também especifica algoritmos para apresentação de caracteres, manipulação de textos bidirecionais (e.g., para hebreu e árabe) e comparação de *strings*. Por outro lado, o padrão ISO 10646 não vai muito além de ser um conjunto de caracteres codificados.

Outra diferença entre os padrões Unicode e ISO 10646 é que Unicode não possui divisões, como ocorre com o padrão ISO; i.e., Unicode é publicado como um padrão único.

## 7.6 ESQUEMAS DE CODIFICAÇÃO DE CARACTERES

No padrão Unicode, um **esquema de codificação de caracteres** (nível 4 na **Tabela 7-1, Seção 7.5.1**) é um mapeamento entre unidades de código (nível 3 na **Tabela 7-1, Seção 7.5.1**) e sequências de bytes. Na prática, uma unidade de código consiste em um byte (8 bits), dois bytes (16 bits) ou quatro bytes (32 bits)<sup>60</sup>.

---

<sup>59</sup> A sigla UCS é derivada da expressão em inglês: *Universal Coded Character Set* ou, simplesmente, *Universal Character Set*.

<sup>60</sup> Para simplificar, aqui se utiliza *byte* com o mesmo significado de *octeto*. Este último termo corresponde exatamente ao que é especificado nos padrões Unicode e ISO 10646.

Como o intervalo de pontos de código usado pelos padrões Unicode e ISO 10646 é muito extenso, existem diferentes esquemas de codificações. Alguns esquemas de codificação permitem um processamento bem simples, mas desperdiçam muita memória, enquanto outros esquemas permitem armazenamento mais eficiente, mas requerem processamento mais complexo.

O padrão Unicode possui três formas de codificação<sup>61</sup>: UTF-32, UTF-16 e UTF-8, e sete esquemas de codificação: UTF-8, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE e UTF-32LE. Os números seguindo *UTF*- nestes esquemas de codificação indicam os tamanhos (em bits) das unidades de código utilizadas pelos esquemas.

Os esquemas de codificação especificados pelo padrão ISO 10646 começam com *UCS*<sup>62</sup>: UCS-2 e UCS-4. Neste caso, o número que acompanha *UCS* representa o número de bytes usados para representar cada caractere no respectivo esquema.

Todos os esquemas de codificação que usam unidades de código com mais de um byte apresentam problemas de portabilidade causados por ordenação de bytes (v. **Seção 13.3**). Para evitar interpretação equivocada de caracteres devido às duas ordenações de bytes possíveis, convencionou-se usar um inteiro de 16 bits no início de cada arquivo de texto que usa UTF-16 ou UTF-32. Este inteiro, que é denominado **marca de ordenação de bytes** ou **BOM**<sup>63</sup>, terá como valor 0xFEFF, se a ordenação usada por um arquivo for *big-endian*, ou 0xFFFE, se a ordenação usada for *little-endian*.

A maioria dos sistemas que aderiram ao padrão Unicode usa UTF-8 ou UTF-16. O esquema de codificação de caracteres UTF-8 notabiliza-se por suportar todo o conjunto de caracteres Unicode (ISO 10646), além de ser compatível com a codificação ASCII. Esses e outros relevantes esquemas de codificação serão explorados nas seções a seguir, mas existem outros esquemas de codificação (e.g., UTF-7) que, por serem menos usados na prática, não serão discutidos aqui.

## 7.6.1 UTF-8

O esquema de codificação **UTF-8** utiliza de uma a quatro unidades de código (bytes) para representar cada ponto de código do padrão Unicode ou ISO 10646<sup>64</sup>.

61 O acrônimo *UTF* é derivado de *Unicode Transformation Format*.

62 O acrônimo *UCS* é derivado de *Universal Character Set*.

63 O acrônimo *BOM* é derivado de *Byte Order Mark*.

64 Frequentemente, encontram-se definições de UTF-8 que indicam que este esquema de codificação pode utilizar até oito bytes, mas estas definições não se restringem ao espaço de codificação de caracteres de Unicode.



Este esquema de codificação não apresenta problema de ordenação de bytes e é o esquema mais utilizado na prática atualmente, em especial na internet e em bancos de dados.

UTF-8 usa sequências de unidades de código de tamanhos variados (i.e., UTF-8 é uma codificação multibyte). Os caracteres cujos pontos de código estão entre 0 e 127 não sofrem modificação; i.e., eles são armazenados em bytes que representam seus próprios pontos de código. Isto torna este esquema de codificação compatível com o código ASCII. Caracteres com pontos de código maiores do que 127 são mapeados em sequências de bytes conforme mostra a **Tabela 7-3**.

PONTO DE CÓDIGO	REPRESENTAÇÃO EM UTF-8
0x00000000 – 0x0000007F	0xxxxxxx
0x00000080 – 0x000007FF	110xxxxx 10xxxxxx
0x00000800 – 0x0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
0x00010000 – 0x001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
0x00200000 – 0x03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
0x04000000 – 0x7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Tabela 7-3: Esquema de codificação de caracteres UTF-8.

Na **Tabela 7-3**, cada byte representado como 10xxxxxx é um **byte de continuação** e os bits xxxxxx são preenchidos com os respectivos bits da representação binária do ponto de código do caractere. A menor sequência de bytes que representa um dado caractere é usada. Note que quanto maior for o ponto de código do caractere, mais bytes serão necessários para representá-lo.

Conforme pode-se observar na **Tabela 7-3**, caracteres codificados em UTF-8 podem usar sequências de até seis bytes. Entretanto, como o ponto de código de um caractere Unicode (ou ISO 10646) pode, no máximo, ocupar três bytes, o maior número de bytes usados por um caractere codificado em UTF-8 é quatro<sup>65</sup>.

65 Explicando melhor, qualquer caractere do repertório Unicode ou ISO 10646 ocupa, no máximo, 21 bits. Portanto, o maior ponto de código que um caractere Unicode (ou ISO 10646) poderia ter é 0x1FFFFF ou 2097151<sub>10</sub>. No entanto, o maior valor especificado por estes padrões é 0x10FFFF que corresponde a 1.114.112 caracteres. No momento da escrita deste livro, existem menos de 100.000 caracteres catalogados no repertório Unicode ou ISO 10646.

**Exemplos:**

- O caractere Unicode © tem ponto de código 0x00A9, que tem a seguinte representação binária:

```
00000000 10101001
```

Portanto, de acordo com a segunda linha da **Tabela 7-3**, este caractere é codificado em UTF-8 pela sequência de bytes:

```
11000010 10101001
```

- O caractere Unicode ≠ tem ponto de código 0x2260, que tem a seguinte representação binária:

```
00100010 01100000
```

Assim, de acordo com a terceira linha da **Tabela 7-3**, o caractere ≠ é codificado em UTF-8 como:

```
11100010 10001001 10100000
```

- Os caracteres na *string* "Eu♥JP" têm, respectivamente, os seguintes pontos de código em Unicode:

```
0x0045 0x0075 0x2665 0x004A 0x0050 0x0000
```

Estes pontos de código seriam codificados em UTF-8 como:

```
0x45 0x75 0xE2 0x99 0xA5 0x4A 0x50 0x00
```

As seguintes propriedades do esquema de codificação UTF-8 merecem destaque:

- Se um byte tem valor entre 0x0 e 0x7F, ele, isoladamente, representa um caractere e este caractere tem exatamente este ponto de código em ASCII. Isto significa que, quando um byte representa um caractere ASCII, seu bit mais significativo é 0.
- Bytes que fazem parte de caracteres que não são ASCII estão no intervalo de 0x80 a 0x9F e têm 1 como bit mais significativo.
- Não há bytes válidos em UTF-8 com valores entre 0xA0 e 0xFF. Isto significa que, entre os 256 possíveis valores de um byte, 95 não podem ser usados. Este fato torna a codificação UTF-8 relativamente ineficiente.
- O primeiro byte de uma sequência multibyte está sempre no intervalo de 0xC0 a 0xF7 e indica o tamanho da sequência. Por exemplo, se o valor de um byte de uma sequência multibyte estiver entre 0xE0 e 0xEF, trata-se do primeiro byte de uma sequência de três bytes.

- Os bytes que sucedem ao primeiro byte de uma sequência multibyte estão no intervalo de 0x80 a 0xBF.
- Os bytes com valores 0xFE e 0xFF nunca são usados em codificação UTF-8.

Devido às propriedades descritas aqui, considerando um byte arbitrário no interior de um *string* e sabendo-se que a codificação usada é UTF-8, pode-se determinar se o byte é válido ou inválido, se ele sozinho representa um caractere, se ele é o primeiro caractere ou se aparece depois do primeiro caractere de uma sequência multibyte<sup>66</sup>.

Dentre as vantagens apresentadas pelo uso do esquema de codificação UTF-8 podem ser citadas:

- Qualquer caractere Unicode pode ser representado usando-se, no máximo, quatro bytes, e os caracteres mais comuns (i.e., aqueles que pertencem ao plano BMP) requerem menos de quatro bytes.
- Muitos programas escritos em C que usam *strings* de caracteres monobytes continuam funcionando, pois UTF-8 é compatível com ASCII. Ou seja, arquivos e *strings* que contêm apenas caracteres ASCII possuem a mesma representação quando codificados em ASCII e UTF-8.
- Um byte corrompido ou ausente numa transmissão de dados afeta apenas um caractere. Ou seja, devido às propriedades mencionadas anteriormente, é fácil encontrar o início da próxima sequência multibyte examinando-se apenas alguns poucos bytes.
- UTF-8 não apresenta problema de ordenação de bytes, como ocorre, por exemplo, com as codificações UTF-16 e UTF-32.

A principal desvantagem de UTF-8 é que o processamento de caracteres se torna mais complexo, pois não há correspondência entre caracteres e bytes. Outra desvantagem é que *strings* contendo caracteres codificados em UTF-8 não são indexáveis; i.e., não é possível acessar diretamente o n-ésimo caractere num tal *string*.

## 7.6.2 UTF-16, UTF-16BE E UTF-16LE

O esquema de codificação **UTF-16** utiliza uma ou duas unidades de código de 16 bits para representar cada ponto de código. Como os caracteres mais comumente usados têm pontos de código que cabem em 16 bits, normalmente apenas uma unidade

---

<sup>66</sup> Entretanto, numa sequência com mais de dois bytes, não é possível dizer se um determinado byte examinado isoladamente é, por exemplo, o segundo ou o terceiro byte da sequência.

de código é usada. O sistema Windows XP e a linguagem de programação Java usam este esquema de codificação.

Quando se utilizam unidades de código de 16 ou 32 bytes, a interpretação depende da ordem na qual os bytes são dispostos (v. **Seção 13.3**). Portanto, este esquema de codificação apresenta problema de ordenação de bytes e, por isso, não é portátil<sup>67</sup>. Entretanto, este esquema de codificação possui duas versões portáteis: **UTF-16BE**, para bytes ordenados a partir do byte mais significativo, e **UTF-16LE**, para bytes ordenados a partir do byte menos significativo<sup>68</sup>.

Caracteres com valores acima de 0xFFFF são representados em UTF-16 por meio de **pares substitutos**. Isto é, alguns valores são reservados para uso como **substituto alto** (inicial) e outros como **substituto baixo** (final) num par de unidades de código. Um substituto alto consiste em uma unidade de código no intervalo de 0xD800 a 0xDBFF e um substituto baixo é uma unidade de código no intervalo de 0xDC00 a 0xDFFF. Juntos, eles são capazes de representar pontos de código que estão fora do plano BMP; i.e., no intervalo de U+10000 a U+10FFFF. Um substituto alto que não possui um substituto baixo correspondente, ou vice-versa, não faz sentido.

Embora um par substituto requiera 32 bits para ser representado, raramente o uso de pares substitutos faz-se necessário, pois eles representam caracteres pouco utilizados. Assim, o uso de espaço pelo esquema de codificação UTF-16 é relativamente eficiente.

Como uma implementação aderente ao padrão Unicode não precisa dar suporte para todos os caracteres deste padrão, é permitido que sejam ignorados todos os caracteres fora do plano BMP. Mesmo assim, a implementação deve reconhecer quando encontra um par substituto, mesmo quando não é capaz de lidar com o caractere representado por ele.

### 7.6.3 UTF-32, UTF-32BE E UTF-32LE

**UTF-32** é o esquema de codificação Unicode mais simples e utiliza uma unidade de código de 32 bits para representar cada ponto de código. Aparentemente, este é o melhor esquema, já que cada ponto de código é mapeado em exatamente uma unidade

---

67 O termo *ordenação de bytes* refere-se à posição relativa dos bytes numa unidade de código de dois ou quatro bytes. Existem dois tipos de ordenação de bytes: do byte mais significativo para o menos significativo – denominada **big-endian** – e do byte menos significativo para o mais significativo – denominada **little-endian**. A ordenação de bytes torna-se problemática se houver transferência de arquivos entre computadores que lidem com ordenações diferentes. Maiores detalhes sobre o assunto são apresentados na **Seção 13.2**.

68 Os sufixos **LE** e **BE** são derivados, respectivamente, de *Little Endian* e *Big Endian* (v. **Seção 13.2**).

de código. Entretanto, conforme já foi discutido, o uso de unidades de código de 32 bits pode gerar um grande desperdício de memória, pois o maior ponto de código Unicode cabe em 21 bits. Assim, usando este esquema de codificação, pelo menos 11 bits de qualquer unidade de código são sempre zeros.

No esquema de codificação UTF-32, se uma unidade de código estiver corrompida, ela não afetará outros dados de um *string* ou *stream*, pois cada unidade de código corresponde a um único caractere que independe de outras unidades de código. Outra vantagem de UTF-32 é que o processamento de unidades de 32 bits é muito eficiente em computadores atuais.

Apesar dessas vantagens, o esquema de codificação UTF-32 apresenta problema de ordenação de bytes. Por causa disso, existem duas versões deste esquema: **UTF-32BE**, para ordenação a partir do byte mais significativo, e **UTF-32LE**, para ordenação a partir do byte menos significativo. Estes esquemas de codificação são raramente utilizados devido ao desperdício de memória.

## 7.6.4 UCS-2

O esquema de codificação **UCS-2** é similar a UTF-16, pois ambos utilizam unidades de código de 16 bits. Mas, UCS-2 usa sempre uma única unidade de código (16 bits) para representar caracteres, enquanto UTF-16 pode usar uma ou duas unidades de código, dependendo do caractere representado. Portanto, UCS-2 é limitado ao plano BMP, enquanto que UTF-16 é capaz de representar quaisquer caracteres Unicode.

O esquema de codificação UCS-2 faz parte do padrão ISO 10646, mas não faz parte do padrão Unicode.

## 7.6.5 UCS-4

**UCS-4** é um esquema de codificação exclusivo do padrão ISO 10646 e é equivalente à codificação UTF-32 do padrão Unicode.

## 7.6.6 ESCOLHA DE UM ESQUEMA DE CODIFICAÇÃO

A escolha do esquema de codificação mais adequado para um programa deve levar em consideração as seguintes observações sobre os esquemas de codificação mais comuns:

- UTF-8 é conveniente quando um programa ou sistema operacional requer que caracteres sejam tratados em unidades de 8 bits. Isso ocorre, por exemplo, quando programas antigos estão sendo atualizados.
- UTF-16 é bastante usado por programas desenvolvidos para Windows devido ao suporte oferecido por este sistema operacional<sup>69</sup>.
- Em média, tanto UTF-8 quanto UTF-16 oferecem vantagens em termos de armazenamento com relação a UTF-32.
- Em particular, UTF-8 é mais conveniente do que UTF-16, embora não seja assim quando se envolvem línguas asiáticas, pois, neste caso, UTF-8 ocupa em média três bytes por ponto de código.
- Quando muitos caracteres com pontos de código maiores do que 0xFFFF são necessários, é melhor adotar UTF-32 do que UTF-16, pois UTF-32 oferece um processamento mais eficiente do que UTF-16 quando esta codificação usa pares substitutos.

## 7.7 COLAÇÃO AVANÇADA

Esta seção é relativamente longa, opcional e não é necessária para dominar o uso de funções de colação da biblioteca padrão de C. Contudo, esta seção provê um entendimento mais profundo deste tópico complexo e muitas vezes negligenciado em livros sobre programação. O conhecimento provido nesta seção é necessário se o leitor deseja implementar funções de colação mais sofisticadas e adequadas para uma situação específica.

### 7.7.1 COLAÇÃO E LOCALIDADE

Ordem de colação é dependente de cultura e pode variar numa mesma linguagem. Mais precisamente, do ponto de vista de programação, colação é dependente de localidade (v. **Capítulo 5**). Isto é, colação, assim como outros aspectos de localidade,

---

<sup>69</sup> Versões mais antigas do sistema Windows (i.e., Windows 95, 98 e Me) não dão suporte para Unicode ou UTF-16. Além disso, o suporte provido varia de acordo com a versão de Windows. Mas, nem mesmo versões mais recentes oferecem suporte para pares substitutos.

depende da cultura dos usuários. Por exemplo, na língua alemã predominante na Alemanha, a letra *ö* segue imediatamente a letra *o*, enquanto que, na língua alemã usada na Suécia, a letra *ö* é a última letra do alfabeto. Além disso, diferentes princípios de colação podem ser aplicados a diferentes contextos. Por exemplo, o princípio de ordenação aplicado às palavras de um dicionário pode ser diferente daquele aplicada aos nomes numa lista telefônica.

Apesar de pontos de código possuírem uma ordem natural, esta ordem não é conveniente como base para ordenação de caracteres, conforme já foi discutido na **Seção 6.4**. Além disso, seria impossível atribuir pontos de código de tal modo que a ordenação resultante correspondesse à ordem de colação de caracteres, pois, entre outras razões, diferentes linguagens ou culturas podem ordenar os mesmos caracteres de formas diferentes<sup>70</sup>.

Como ordenação alfabética varia entre linguagens diferentes, um mesmo conjunto de *strings* pode ser ordenado de maneiras diferentes dependendo da linguagem considerada. Além disso, ordenação de *strings* não depende da linguagem original de cada *string* em si, mas da localidade na qual eles são lidos. Por exemplo, um usuário brasileiro deseja ler uma lista de nomes alemães dispostos de acordo com a ordenação tradicionalmente utilizada em português, e não na ordenação usada em alemão.

A abordagem de colação de *strings* a ser apresentada a seguir é baseada no uso de um algoritmo que recebe como entrada uma tabela que informa como ordenar caracteres numa determinada linguagem. Nesta abordagem, cada caractere é mapeado num conjunto de pesos inteiros específicos para uma dada linguagem, e a ordenação de *strings* utiliza estes pesos em vez de pontos de código.

## 7.7.2 COLAÇÃO EM MÚLTIPLOS NÍVEIS

A abordagem mais moderna usada em colação é baseada em **níveis** (ou **pesos**) **múltiplos**. Nesta abordagem, são necessários, pelo menos, dois pesos, representados por valores inteiros, para cada caractere. O **peso primário**, mais forte, deve ter o

---

<sup>70</sup> Apesar dessas restrições, se o conjunto de caracteres usado for limitado a alguns pequenos subconjuntos do repertório Unicode, pontos de código podem servir como base de ordenação. Por exemplo, se o conjunto de caracteres for restrito às letras latinas de A a Z (ou de a a z), a ordenação pode ser feita comparando-se apenas os pontos de código, pois a ordenação destes corresponde à ordenação dos respectivos caracteres. Também, se letras maiúsculas e minúsculas forem usadas ao mesmo tempo, o problema pode ser facilmente resolvido, conforme mostrado na **Seção 6.4.2**.

mesmo valor para todas as versões de uma mesma letra. Por exemplo, as letras *A*, *a*, *á*, *ã* e todas as outras versões de *a* têm o mesmo peso primário. O **peso secundário**, menos importante, tem valores diferentes para diferentes versões de uma mesma letra. Por exemplo, as letras *A*, *a*, *á*, *ã* e todas as outras versões de *a* têm diferentes pesos secundários.

Nesta abordagem de pesos múltiplos, comparam-se dois *strings* caractere a caractere usando-se inicialmente os pesos primários das letras correspondentes. Se for encontrada alguma diferença entre estes pesos a comparação é encerrada e a ordem dos *strings* é determinada pela respectiva ordem definida por estes pesos. Por outro lado, se, durante a comparação, dois caracteres têm o mesmo peso primário, utiliza-se o peso secundário como critério de desempate. Utilizando esta abordagem, uma diferença primária que aparece no final dos *strings* supera uma diferença secundária que apareça no início dele. Por exemplo, "Bota" segue "bola", mesmo que *B* anteceda a *b*, porque a diferença entre *t* e *l* ocorre nos pesos primários destas letras, enquanto a diferença entre *b* e *B* ocorre em seus pesos secundários. Verifique que, utilizando esta técnica de comparação de *strings*, obtém-se a seguinte ordenação de *strings*:

```
BOLA
Bola
bola
BOTA
Bota
bota
```

Para abranger várias linguagens, assim como levar em consideração outros símbolos (e não apenas letras), são necessários mais de dois níveis (pesos) de comparação. Em línguas contendo caracteres acentuados, como português, a diferenciação em termos de acentuação é feita pelos pesos secundários, enquanto a diferenciação entre maiúsculas e minúsculas é feita por pesos terciários. Por exemplo, as letras *A*, *a*, *á*, *ã* e todas as outras versões de *a* têm o mesmo peso primário; as letras *a*, *á*, *ã*, *à* e *â* têm pesos secundários diferentes; as letras *a* e *A* têm os mesmos pesos primário e secundário, mas diferentes pesos terciários.

Caracteres de separação (e.g., hífen) são frequentemente utilizados em *strings* que precisam ser ordenados, de modo que tais caracteres também devem ter pesos atribuídos para que se obtenham ordenações que correspondam às expectativas. Mas, não se pode, por exemplo, atribuir ao hífen um peso maior de que o peso de *j* para que "vicejar" apareça antes de "vice-reitor", pois, se fosse assim, resultados inesperados seriam obtidos quando se comparassem outros *strings* (e.g., "vice-che-



fe" apareceria depois de "vicejar"). A melhor solução, neste caso, é não atribuir ao hífen nenhum peso primário. Isto é, caracteres separadores devem ser levados em consideração pelo algoritmo de colação apenas a partir do nível secundário.

Esta última consideração é implementada definindo-se um valor nulo que indique que o respectivo peso é ignorável. Assim, se um caractere com peso ignorável for encontrado num dos *strings* comparados, ele é saltado; i.e., passa-se para o caractere seguinte como se aquele caractere não existisse. Usando-se esta estratégia, na comparação dos *strings* "vicejar" e "vice-reitor", o quinto caractere do primeiro *string* (j) é comparado com o sexto caractere do segundo *string* (r).

No caso de caracteres separadores, não faz sentido ter pesos ignoráveis nos segundo e terceiro níveis, mas não ignorável no primeiro nível. Assim, quando um caractere é ignorável num nível, ele deve também ser ignorável em todos os outros níveis mais importantes. Por exemplo, se um caractere é ignorável no segundo nível, ele também deve ser ignorável no primeiro nível, mas pode ter peso significativo no terceiro nível.

Resumindo o que foi exposto até aqui, o modelo de colação mais aceite atualmente envolve comparações em quatro níveis (ou pesos):

- **Nível primário** – Este nível é utilizado para denotar diferenças entre caracteres básicos (i.e., sem levar em consideração se uma letra é acentuada/não acentuada ou maiúscula/minúscula). Esta é a diferença mais forte e é utilizada, por exemplo, para separar seções de um dicionário.
- **Nível secundário** – Neste nível, acentuação gráfica é levada em consideração. Mas, este nível só é utilizado se não houver diferença primária entre os caracteres envolvidos.
- **Nível terciário** – Este nível diferencia letras maiúsculas e minúsculas, assim como variantes de um mesmo caractere (e.g., *a* e *á*), mas é usado apenas se os caracteres em consideração forem iguais nos níveis primário e secundário.
- **Nível quaternário** – Este nível serve para distinguir caracteres de separação e pontuação que são ignorados nos três níveis anteriores. Frequentemente, utiliza-se o próprio ponto de código do caractere como peso neste nível.

## 7.7.3 CASOS ESPECIAIS DE COLAÇÃO

### *Acentuação francesa*

Um caso especial de ordenação de *strings*, denominado **ordenação de acentos**

**franceses**, ocorre com palavras acentuadas na língua francesa<sup>71</sup>. Neste caso, *strings* que diferem apenas pelo uso de letras acentuadas são ordenados de trás para frente. Por exemplo, as palavras francesas a seguir:

cote  
coté  
côte  
côté

estariam perfeitamente organizadas de acordo com uma ordenação convencional. Em francês, entretanto, estas palavras são ordenadas assim:

cote  
côte  
coté  
côté

Uma discussão sobre como tratar esse caso especial de ordenação está além do escopo do presente texto, mas o leitor interessado pode consultar as referências bibliográficas ao final do livro.

### *Caracteres com contração*

Uma letra acentuada pode ser interpretada de duas maneiras equivalentes: (1) como um caractere único (e.g., *ã*) ou (2) como uma combinação de dois caracteres (e.g., a combinação de *a* com til). Um algoritmo de colação deve tratar as duas interpretações da mesma maneira. Por exemplo, em português, a letra *ã* deve aparecer entre as letras *a* e *b*, quer ela seja interpretada como um único caractere, quer ela seja interpretada como a letra *a* seguida de til. No último caso, essa equivalência é obtida mapeando-se os dois caracteres num único conjunto de pesos que faz com que a combinação seja ordenada de modo diferente de qualquer dos dois caracteres considerados isoladamente. Estes dois caracteres constituem uma **sequência de caracteres de contração** e, para o propósito de comparação de *strings*, tal sequência é considerada um **agrupamento de grafemas**; i.e, uma sequência de caracteres que se comportam como um único caractere em comparações de *strings*.

Além de caracteres acentuados, outro exemplo notável de sequências de caracteres de contração são os dígrafos *ch* e *ll*, que são considerados caracteres isolados em espanhol. O dígrafo *ch* é ordenado entre *c* e *d* e o dígrafo *ll* é ordenado entre *l* e *m*.

---

71 A mesma situação ocorre em algumas outras linguagens (e.g., albanês).

### *Caracteres com expansão*

Um **caractere com expansão** é o oposto de uma sequência de caracteres com contração. Isto é, um único caractere pode ser mapeado em mais de um conjunto de pesos. Por exemplo, o caractere *æ* pode ser considerado equivalente a *a* seguido de *e*. Neste caso, o caractere *æ* é mapeado em dois conjuntos de pesos: um correspondente ao caractere *a* e outro correspondente ao caractere *e*.

### *Colação aproximada*

Frequentemente, usam-se busca e ordenação de *strings* insensíveis ao uso de maiúsculas e minúsculas ou acentuação. Por exemplo, neste tipo de operação, os *strings* "Água" e "agua" são considerados iguais.

## 7.7.4 ALGORITMO DE COLAÇÃO UNICODE

O consórcio Unicode publica um padrão denominado **Algoritmo de Colação Unicode (UCA<sup>72</sup>)**. Este algoritmo recebe como entrada um *string* de caracteres Unicode e uma **tabela de elementos de colação** e tem como saída uma sequência de bytes, denominada **chave de ordenação**. Esta sequência de bytes pode ser comparada, byte a byte, com outra resultante da aplicação do mesmo algoritmo em outro *string*.

A tabela de elementos de colação deve levar em consideração informações sobre localidade (v. **Seção 5.2**), de modo que se possa obter, ao final do processo de comparação, a ordenação de *strings* esperada. Assim, *strings* podem ser ordenados de modos diferentes usando-se diferentes tabelas de elementos de colação. Para levar em consideração a ocorrência de caracteres expansíveis e contraíveis, além de caracteres comuns, é necessária uma tabela de elementos de colação capaz de fazer mapeamentos um-para-um, muitos-para-um e muitos-para-muitos.

Cada **elemento de colação** que aparece numa tabela de elementos de colação consiste em uma sequência de três ou mais pesos inteiros, e a ordem com que os pesos aparecem corresponde aos seus níveis. Isto é, o primeiro valor inteiro corresponde ao peso primário (mais forte), o segundo valor corresponde ao peso secundário, e assim por diante, de modo que o último valor corresponda ao peso mais fraco (i.e., o peso de menor influência na ordenação). A ordenação dos elementos de colação é feita

---

72 O acrônimo UCA é derivado de *Unicode Collation Algorithm*.

da seguinte maneira: se os pesos primários forem diferentes, a ordem dos elementos será definida pela ordem deste peso; se esses pesos forem iguais, a ordem dos pesos secundários será usada, e assim por diante. Um peso com valor 0000 significa que o elemento de colação é ignorável neste nível de peso.

Usando-se apenas três níveis de pesos, dois *strings* podem ser considerados iguais sem que sejam literalmente iguais. Assim, o algoritmo UCA inclui um quarto nível de comparação. Neste nível, o peso atribuído a um caractere é exatamente seu ponto de código. Assim, se dois caracteres são idênticos até o terceiro nível de comparação, seu ponto de código é utilizado como critério de desempate.

Caracteres de controle e formatação são totalmente ignorados na ordenação padrão utilizada pelo algoritmo UCA. Excetuam-se desta regra os caracteres de quebra de linha ('\\n') e tabulação horizontal ('\\t'), que são considerados espaços em branco.

### *Elementos de colação variáveis*

Alguns caracteres, tais como caracteres de espaço, pontuação e muitos símbolos, possuem elementos de colação variáveis. Um **elemento de colação variável** tem pesos predefinidos, mas também pode usar **pesos alternativos**. Existem quatro opções para tratamento de caracteres com pesos alternativos, que recebem as seguintes denominações: não ignorável, ignorável, alterado e alterado-reduzido.

Quando um caractere é considerado **não ignorável**, os mesmos pesos associados a ele na tabela de elementos de colação são utilizados, como no caso de elementos de colação que não são variáveis. Por exemplo, a seguinte lista está ordenada de acordo com este critério<sup>73</sup>:

```
e-mail [Hífen: U+002D]
e-mail [Travessão: U+2014]
eleitor
email
exercício
```

Quando a opção utilizada é **ignorável**, os pesos primário, secundário e terciário do caractere são considerados iguais a zero, e seu peso quaternário continua sendo igual ao seu ponto de código. Esta escolha faz com que o caractere seja totalmente ignorado

---

<sup>73</sup> Foram acrescentados comentários entre colchetes para ajudar o leitor a distinguir hífen de travessão, já que esta diferença pode não ser prontamente visível. Também, do ponto de vista de estilo de redação, não é recomendável o uso de travessão nesta situação. Mas o propósito aqui é apenas demonstrar como ele seria ordenado, se fosse o caso, nesta situação.

se a comparação prosseguir até o terceiro nível de pesos. Se a comparação continuar além deste nível, o ponto de código do caractere (peso quaternário) será usado como critério de desempate. A mesma lista de palavras apresentada anteriormente seria ordenada da maneira a seguir de acordo com esta opção:

```
eleitor
e-mail [Hífen]
email
e-mail [Travessão]
exercício
```

Neste último exemplo, as três variantes de *email* são consideradas iguais até o terceiro nível de comparação, pois, até este nível, o hífen e o travessão são invisíveis ao algoritmo de comparação. No quarto nível de comparação, os pontos de código são usados para resultar na ordenação final dos três *strings*<sup>74</sup>.

Na opção **alterado**, os três primeiros pesos de um caractere com peso alternativo recebem o valor 0x0000, como no caso ignorável, mas seu peso primário original passa a ser seu peso quaternário. Neste caso, qualquer caractere que não tenha pesos variáveis recebe o maior valor possível como peso quaternário (i.e., 0xFFFF). O uso desta opção resulta numa ordenação mais intuitiva quando *strings* são considerados iguais até o terceiro nível de comparação porque se assegura de que todos os *strings* quase iguais, cada um dos quais contendo um caractere ignorável numa mesma posição, aparecem juntos na ordenação final, em vez de separados como na ordenação do último exemplo. Utilizando-se esta opção, os mesmos *strings* dos exemplos anteriores seriam ordenados como:

```
eleitor
e-mail [Hífen]
e-mail [Travessão]
email
exercício
```

No último exemplo, as três variantes de *email* são consideradas iguais até o terceiro nível de comparação, como ocorreu no exemplo anterior. Mas, agora, no quarto nível de comparação, os pesos (i.e., pontos de código) do hífen e do travessão são comparados com o peso de valor 0xFFFF atribuído à letra *m*. Assim, as duas versões de *email* com caracteres ignoráveis aparecem juntas.

---

<sup>74</sup> Os caracteres comparados no quarto nível e seus respectivos pontos de código são hífen (U+002D), travessão (U+2014) e letra *m* (U+006D). Assim, tem-se a seguinte ordem: o hífen precede à letra *m* que precede ao travessão.

A opção **alterado-reduzido** é muito parecida com a opção **alterado**. A diferença é que, na opção **alterado-reduzido**, os caracteres que não têm pesos alternativos recebem `0x0001` como pesos quaternários (em vez de `0xFFFF`). Esta alteração faz com que os *strings* com caracteres ignoráveis continuem sendo ordenados juntos, como na opção **alterado**, mas sejam ordenados depois de *strings* sem caracteres ignoráveis considerados iguais até o terceiro nível de comparação. Por exemplo, a mesma lista de *strings* dos exemplos anteriores seria agora ordenada como:

```
eleitor
email
e-mail [Hífen]
e-mail [Travessão]
exercício
```

### *Chaves de ordenação*

Seguindo o algoritmo UCA, a chave de ordenação criada para cada *string* a ser comparado consiste em uma sequência de pesos. Para obtê-la, examinam-se todos os caracteres do *string* e copiam-se todas as chaves primárias para a chave de ordenação; se um caractere for ignorável (e.g., hífen), seu peso não será copiado para a chave; um caractere expansível (e.g., *æ*) tem mais de um peso acrescentado à chave e caracteres que se contraem (e.g., *ch* em espanhol) têm apenas um peso acrescentado à chave. Depois que todos os pesos primários são acrescentados à chave de ordenação, acrescenta-se a ela um valor de separação (v. adiante) e repete-se o processo para os pesos secundário e terciário de cada caractere que compõe o *string*.

Após a criação das chaves de ordenação de dois *strings*, o processo de ordenação deles resume-se a comparar os pesos constantes nas duas chaves nas respectivas ordens. A comparação de dois pesos é feita do mesmo modo que se comparam números inteiros.

O **valor de separação** que é acrescentado a uma chave de ordenação entre a sequência de pesos primários e a sequência de pesos secundários e entre estes e a sequência de pesos terciários deve ter um valor menor do que o menor de todos os pesos envolvidos. Tipicamente, o separador de pesos é escolhido como sendo zero. Isto faz com que *strings* mais curtos precedam a *strings* mais longos quando os caracteres do menor *string* coincidem com o início de um *string* maior (e.g., "manga" precede à "mangaba"). Quando se deseja o efeito contrário (e.g., "mangaba" precedendo à "manga"), utiliza-se um valor maior do que o valor de qualquer peso.

Um conjunto de pesos que faz parte de uma chave de ordenação é denominado **elemento de ordenação** ou **elemento de colação**. Na prática, usualmente, um elemento de colação é obtido por meio da concatenação, num inteiro de 32 bits, de todos os pesos de cada caractere (ou sequência de caracteres de contração)<sup>75</sup>. Utilizando esta abordagem, uma chave de ordenação pode ser obtida por meio de uma única passagem pelo *string*. Por outro lado, o algoritmo de ordenação pode fazer mais de uma passagem, comparando, em cada uma delas, os pesos primários, secundários ou terciários. Em cada passagem apenas um conjunto de pesos torna-se visível por meio de uma operação de mascaramento (v. **Capítulo 13 – Volume I**).

Uma consideração de natureza prática é que, usualmente, não é necessária a criação de chaves de ordenação antes da comparação de dois *strings*. Isto é, os elementos de colação dos *strings* podem ser criados e imediatamente comparados. Assim, estes elementos de colação só são criados até que seja encontrada uma diferença entre os eles.

### *Tabela DUCET*

O algoritmo UCA apresenta uma tabela de elementos colação padrão denominada **DUCET**<sup>76</sup>. Esta tabela pode ser usada como tal ou pode servir como base para a definição de uma tabela de colação para uma necessidade específica.

A ordenação padrão especificada pela tabela DUCET é inserida num arquivo-texto denominado `allkeys.txt`, que pode ser facilmente encontrado no site do consórcio Unicode na internet (<http://www.unicode.org>).

O arquivo `allkeys.txt` contém uma lista de mapeamentos de pontos de código em sequências de pesos. Cada mapeamento consiste em um ponto de código ou uma sequência de pontos de código, seguido por ponto e vírgula e por uma ou mais sequências de pesos. Cada mapeamento ocupa, no arquivo, uma linha que termina com um breve comentário, começando com `#`, que descreve o respectivo caractere sendo mapeado. Cada ponto de código é apresentado como uma sequência de quatro dígitos hexadecimais<sup>77</sup>. Cada elemento de colação (i.e., sequência de pesos) é apresentado por

---

<sup>75</sup> Esta concatenação de pesos poderá ser compactada num único inteiro de 32 bits se os pesos utilizarem os intervalos sugeridos pelo padrão Unicode. Na realidade, usando-se estes intervalos, apenas 24 bits serão necessários para armazenar cada conjunto de pesos: 16 bits para o peso primário, 4 bits para o peso secundário e 4 bits para o peso terciário. O peso quaternário sugerido pelo padrão Unicode não precisa ser armazenado porque ele corresponde exatamente ao ponto de código do caractere. A compactação dos pesos num único inteiro pode ser obtida usando-se operações de mascaramento (v. **Capítulo 13 – Volume I**).

<sup>76</sup> DUCET é uma abreviação derivada de *Default Unicode Collation Element Table*.

<sup>77</sup> O padrão Unicode não utiliza a notação da linguagem C para representar valores hexadecimais; i.e., em Unicode, estes valores não começam com `0x`.

quatro valores hexadecimais de quatro dígitos separados por ponto e envolvidos por colchetes. Estes valores representam respectivamente os pesos primário, secundário, terciário e quaternário do respectivo caractere sendo mapeado. Por exemplo, a linha do arquivo `allkeys.txt` a seguir:

```
0061 ; [.0861.0020.0002.0061] # LATIN SMALL LETTER A
```

faz o mapeamento do caractere Unicode, cujo ponto de código é U+0061, com um único elemento de colação contendo os pesos 0x0861, 0x0020, 0x0002 e 0x0061. A linha termina com um comentário informando que o caractere ora mapeado é a letra *a* minúscula. Note que o valor do peso quaternário (0x0061) coincide com o ponto de código do caractere.

Se o conteúdo entre colchetes começar com ponto, o elemento de colação será considerado normal. Mas, se em vez de ponto, houver um asterisco, o elemento de colação será variável. Elementos de colação variáveis têm pesos definidos, mas uma dada implementação do algoritmo UCA pode usar pesos alternativos, conforme já foi discutido antes. Por exemplo, a seguinte linha na tabela DUCET:

```
002D ; [*0221.0020.0002.002D] # HYPHEN-MINUS
```

indica que o hífen é um elemento de colação variável cujo peso primário é 0x0221. Este peso faz com que esse caractere seja ordenado antes de qualquer letra. Assim, por exemplo, o *string* "pré-natal" seria ordenado antes do *string* "prédio", uma vez que o peso primário da letra *d* é 0x1182. Ainda considerando o mesmo exemplo, se o peso primário do hífen fosse ignorado, usando como peso alternativo 0x0000, seria obtida a ordenação usualmente encontrada em dicionários.

Sequências de contração e expansão são incluídas no arquivo `allkeys.txt` para assegurar que formas de normalização diferentes de um mesmo caractere ou sequência de caracteres sejam tratadas da mesma maneira.

Uma entrada de caractere expansível no arquivo `allkeys.txt` é semelhante ao seguinte<sup>78</sup>:

```
2474 ; [*027A.0020.0004.2474][.0A0C.0020.0004.2474]
      [*027B.0020.001F.2474] # PARENTHESIZED DIGIT ONE; COMPATSEQ
```

Esta entrada informa que o caractere (1), com ponto de código U+2474, é ordenado como se fosse composto de abre-parênteses, seguido do número um e terminado com fecha-parênteses (i.e., tal qual o formato do caractere sugere). Os pesos deste

---

<sup>78</sup> Algumas vezes, um comentário termina com um rótulo informativo que tem como objetivo identificar a respectiva entrada na tabela. Por exemplo, o rótulo *COMPATSEQ* identifica um caractere associado a uma sequência de elementos de colação (i.e., um caractere expansível).



caractere são os mesmos dos referidos caracteres da composição, exceto a partir do terceiro nível de cada elemento de colação, o que assegura que o caractere (1) não seja o mesmo que o *string* " (1) " composto de três caracteres.

Os valores dos pesos na tabela DUCET não são normativos; i.e, quaisquer pesos podem ser utilizados, desde que eles façam com que os *strings* sejam ordenados na mesma ordem que seriam se utilizassem os pesos propostos nesta tabela.

### *Normalização*

Em Unicode, muitos caracteres e sequências de caracteres possuem múltiplas representações. Um dos requisitos para aderência ao padrão Unicode é que todas as representações de um caractere sejam tratadas da mesma maneira. Por exemplo, a letra *ü* pode ser representada como:

```
U+00FC # LATIN SMALL LETTER U WITH DIAERESIS
```

Ou como a combinação de caracteres:

```
U+0075 # LATIN SMALL LETTER U
```

```
U+0308 # COMBINING DIAERESIS
```

Assim, um algoritmo de busca ou ordenação que siga o padrão Unicode deve considerar U+00FC e a combinação de U+0075 com U+0308 como iguais. Para obter este efeito, os *strings* a ser comparados são convertidos numa **forma normalizada**.

Tipicamente, utiliza-se a **Forma Normalizada D**, na qual cada caractere é totalmente decomposto na representação equivalente com o maior número de componentes<sup>79</sup>. Uma razão para a escolha desta forma normalizada, em detrimento de outras sugeridas pelo padrão Unicode, é que, usualmente, é mais simples tratar acentos isoladamente como caracteres, que são ignoráveis no primeiro nível de comparação, do que como parte integrante de outras letras. Usando-se esta última opção, pode ser preciso considerar mais caracteres como expansíveis (v. **Seção 7.7.3**).

No algoritmo UCA, o processo de normalização é efetuado durante o processo de mapeamento de *strings* em elementos de colação.

---

<sup>79</sup> Em alguns scripts, há letras com um número muito maior de representações equivalentes do que o exemplo apresentado.

## *Resumo do algoritmo UCA*

Em resumo, o algoritmo UCA apresenta as seguintes características:

- Quatro níveis de pesos são usados, com os pesos no quarto nível correspondendo a pontos de código.
- É requerido suporte para caracteres ignoráveis, caracteres com sequências de contração, caracteres com expansão e uso de pesos variáveis.
- É sugerido suporte para ordenação de acentos franceses.
- É requerido suporte para o uso de pesos alternativos.
- Uma ordenação padrão (DUCET) é especificada para todos os caracteres do repertório Unicode. Na ausência de alterações desta ordenação, todas as implementações do algoritmo devem ordenar qualquer conjunto de *strings* da mesma maneira.
- Para qualquer alteração feita na ordenação padrão, qualquer implementação do algoritmo deve apresentar o mesmo resultado para um mesmo conjunto de *strings*.

A comparação de *strings* de acordo com o algoritmo UCA pode ser resumida nos seguintes passos:

1. **Decomposição e reordenação de caracteres.** Neste passo, os *strings* são colocados na Forma Normalizada D e, se necessário, reordenados<sup>80</sup>.
2. **Mapeamento de *strings* normalizados e reordenados em elementos de colação.** Este passo pode envolver o mapeamento de caracteres em múltiplos elementos de colação, mapeamento de sequências de caracteres num único elemento de colação ou mapeamento de sequências de caracteres em sequências de elementos de colação.
3. **Criação das chaves de ordenação.** Aqui, os pesos são reorganizados de modo que todos os pesos primários apareçam primeiro, seguidos por um valor de separação, seguido de todos os pesos secundários, e assim por diante<sup>81</sup>.

---

80 Reordenação é um processo necessário em algumas línguas asiáticas. Maiores considerações sobre este assunto específico estão além do escopo deste livro e podem ser encontradas na bibliografia apresentada ao final deste volume.

81 Se ordenação francesa de caracteres acentuados for desejada, a inversão de pesos secundários para produzir a ordem desejada será feita neste passo.

- 4. Comparação das chaves de ordenação.** Esta comparação é realizada naturalmente byte a byte.

### 7.7.5 BUSCA

Além de ordenação, outra operação importante na qual colação é essencial é a busca. Quando se faz uma operação de busca na qual se tenta encontrar um casamento exato entre *strings*, a comparação pode ser feita utilizando pontos de código. Por outro lado, quando se buscam *strings* equivalentes (e.g., quando se ignoram distinções entre maiúsculas e minúsculas), o uso da abordagem delineada pelo algoritmo UCA é mais apropriado. Isto é, neste caso, em vez de comparar pontos de código, constroem-se chaves de ordenação e comparam-se os respectivos elementos de colação. A exatidão do casamento entre os *strings* depende dos níveis em que ocorrem as comparações.

Utilizando essa última abordagem, para ignorar, por exemplo, diferenças entre maiúsculas e minúsculas, basta não levar em consideração os pesos terciários das chaves de ordenação. Por outro lado, se for desejado ignorar as diferenças produzidas por acentuação, ignoram-se os pesos secundários.

Uma complicação no processo de busca descrito até aqui é que, se for encontrado um casamento entre elementos de colação dos dois *strings*, ele só é considerada bem sucedido se começar e terminar nos limites de um agrupamento de grafemas. Por exemplo, se o *string* "café" estiver presente no texto em sua forma decomposta (i.e., se a letra *é* aparecer como a letra *e* seguida de acento agudo), "cafe" casará com a porção inicial de "café". Mas, se diferenças causadas por acentuação forem importantes, "cafe" não deve casar com "café" porque o possível casamento termina no meio de um agrupamento de grafema; i.e., o casamento inclui a letra *e*, mas não inclui o acento do *string* "café".

### 7.7.6 EXEMPLO

Para ilustrar o processo de comparação de *strings*, considere como exemplo a comparação dos *strings* "Maca" e "maçã". Este processo começa com os *strings* em seus estados originais e o quadro a seguir apresenta-os com seus respectivos pontos de código.

Maca	0x004D	0x0061	0x0063	0x0061
maçã	0x006D	0x0061	0x00E7	0x00E3

O primeiro passo no processo de comparação consiste em mapear os caracteres dos dois *strings* em suas respectivas formas normalizadas. O resultado é apresentado no quadro a seguir<sup>82</sup>:

Maca	0x004D	0x0061	0x0063	0x0061		
maçã	0x006D	0x0061	0x0063	0x0327	0x0061	0x0303

Em seguida, os pontos de código são mapeados em elementos de colação, como mostra o quadro a seguir<sup>83</sup>:

Maca	[0x1291.0x0020.0x0008.0x004D]
	[0x1141.0x0020.0x0002.0x0061]
	[0x116F.0x0020.0x0002.0x0063]
	[0x1141.0x0020.0x0002.0x0061]
maçã	[0x1291.0x0020.0x0002.0x006D]
	[0x1141.0x0020.0x0002.0x0061]
	[0x116F.0x0020.0x0002.0x0063]
	[0x0000.0x0056.0x0002.0x0327]
	[0x1141.0x0020.0x0002.0x0061]
	[0x0000.0x004E.0x0002.0x0303]

Então, os pesos de cada caractere são reunidos, os valores ignoráveis são descartados e os separadores de pesos são inseridos entre os conjuntos de pesos<sup>84</sup>. O resultado é apresentado no quadro a seguir:

<sup>82</sup> Observe que apenas as letras ç e ã são expandidas na passagem para a Forma Normalizada D.

<sup>83</sup> Os pesos foram obtidos da tabela DUCET e são separados por pontos.

<sup>84</sup> O valor utilizado para separadores de pesos foi 0x0000.

Maca	0x1291	0x1141	0x116F	0x1141	0x0000		
	0x0020	0x0020	0x0020	0x0020	0x0000		
	0x0008	0x0002	0x0002	0x0002	0x0000		
	0x004D	0x0061	0x0063	0x0061			
maçã	0x1291	0x1141	0x116F	0x1141	0x0000		
	0x0020	0x0020	0x0020	0x0056	0x0020	0x004E	0x0000
	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0000
	0x006D	0x0061	0x0063	0x0327	0x0061	0x0303	

Finalmente, os elementos de colação são comparados um a um. Os dois primeiros elementos de colação que apresentam diferença são apresentados em **negrito** e **sublinhados** no quadro a seguir.

Maca	0x1291	0x1141	0x116F	0x1141	0x0000		
	0x0020	0x0020	0x0020	<b><u>0x0020</u></b>	0x0000		
	0x0008	0x0002	0x0002	0x0002	0x0000		
	0x004D	0x0061	0x0063	0x0061			
maçã	0x1291	0x1141	0x116F	0x1141	0x0000		
	0x0020	0x0020	0x0020	<b><u>0x0056</u></b>	0x0020	0x004E	0x0000
	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0000
	0x006D	0x0061	0x0063	0x0327	0x0061	0x0303	

Conforme mostra o último quadro, os primeiros elementos de colação diferentes são `0x0020`, no *string* "Maca", e `0x0056`, no *string* "maçã". Como, evidentemente, `0x0020` é menor do que `0x0056`, tem-se que "Maca" deve preceder "maçã".

Concluindo o exemplo, note que os dois *strings* são considerados iguais no nível primário de comparação e que, devido à normalização, o *string* "maçã" tem dois conjuntos de pesos a mais que o *string* "Maca" no início do processo. Mas, os pesos primários dos caracteres til e cedilha são ignoráveis no primeiro nível, de modo que os dois *strings* passam a ter o mesmo conjunto de pesos primários. A primeira diferença encontrada (i.e., os elementos de colação `0x0020` e `0x0056`) correspondem, respectivamente, aos pesos secundários do segundo *a* de "Maca" e do cedilha de "maçã". A diferença entre a letra maiúscula inicial de "Maca" e a letra minúscula inicial de "maçã" só aparece em seus respectivos pesos terciários; isto é, *M* tem peso

terciário  $0 \times 0008$ , enquanto  $m$  tem peso terciário  $0 \times 0002$ . Se os *strings* fossem essencialmente os mesmos e diferissem apenas no uso de maiúsculas e minúsculas, esta seria a primeira diferença encontrada, e o *string*, que começasse com letra maiúscula, precederia ao outro.

O processo de comparação em quatro passos apresentado no exemplo anterior tem propósito ilustrativo e didático. Na prática, estes passos são intercalados de modo que uma diferença primária no primeiro caractere de cada *string* seja encontrada logo no início do processo em vez de após executar muitas tarefas que serão, enfim, desnecessárias.

## 7.8 EXERCÍCIOS DE REVISÃO

1. Defina: (a) caractere extenso e (b) caractere multibyte.
2. Defina os seguintes conceitos:
  - (a) Grafema
  - (b) Script
  - (c) Plano Unicode
  - (d) BMP
3. Compare caractere extenso e caractere multibyte apontando vantagens e desvantagens obtidas com o uso de cada um destes tipos de caracteres.
4. O que é uma codificação multibyte com estado?
5. O que é um código de caracteres extensos?
6. (a) Descreva o padrão Unicode. (b) Pode-se afirmar que Unicode é simplesmente um código de caracteres (como, por exemplo, ASCII)?
7. Quais são as principais diferenças entre os padrões Unicode e ISO 10646?
8. Descreva os níveis de codificação especificados pelo padrão Unicode.
9. (a) O que é unidade de código? (b) O que é esquema de codificação de caracteres?
10. Como caracteres são representados usando o esquema de codificação UTF-8?

11. (a) Por que os esquemas de codificação UTF-16 e UTF-32 apresentam problemas de portabilidade intrínsecos? (b) Por que UTF-8 não apresenta o mesmo problema de portabilidade?
12. Como o caractere £ que tem ponto de código U+20A4 é representado em UTF-8?
13. O que são pares substitutos no esquema de codificação UTF-16?
14. O que é uma marca de ordenação de bytes e quando ela é necessária?
15. Quando o uso de cada um dos seguintes esquemas de codificação de caracteres é vantajoso?
  - (a) UTF-8
  - (b) UTF-16
  - (c) UTF-32
16. (a) Os esquemas de codificação UCS-2 e UTF-16 são equivalentes? (b) E os esquemas de codificação UCS-4 e UTF-32?
17. Compare os esquemas de codificação UTF-8, UTF-16 e UTF-32 em termos de tempo de processamento.
18. Em que situações é mais eficiente em termos de uso de espaço usar (a) UTF-8 e (b) UTF-16?
19. Baseado no que foi exposto a respeito de esquemas de codificações de caracteres, explique por que, alguma vez, um navegador da web não é capaz de apresentar corretamente caracteres acentuados da língua portuguesa.
20. Descreva o processo de colação em níveis.
21. (a) Descreva o algoritmo de colação Unicode. (b) Este algoritmo faz parte do padrão Unicode?
22. (a) O que é contração de caracteres? (b) O que é expansão de caractere?
23. O que é um caractere com pesos alternativos de acordo com o algoritmo de colação Unicode?
24. Defina os seguintes conceitos: (a) chave de ordenação e (b) elemento de colação.
25. O que é a tabela DUCET?

# *Capítulo 8*

---

*Caracteres extensos e multibytes II:  
suporte*



## 8.1 INTRODUÇÃO

Este capítulo expõe as facilidades encontradas na biblioteca padrão de C para processamento de caracteres e *strings* multibytes e extensos. Conforme visto no **Capítulo 7**, o uso de caracteres e *strings* multibytes num programa oferece vantagens e desvantagens, e o mesmo ocorre com relação a caracteres e *strings* extensos. Portanto, conversões entre caracteres e *strings* multibytes e extensos são algumas das mais importantes operações providas pela biblioteca padrão de C. Funções que executam estas operações ou auxiliam suas execuções são declaradas em dois cabeçalhos discutidos parcialmente neste capítulo: `<stdlib.h>` e `<wchar.h>`. Estas funções são apresentadas resumidamente na **Tabela 8-1**.

FUNÇÃO	DESCRIÇÃO SUCINTA	CABEÇALHO
<b>mblen()</b>	Calcula o número de bytes de um caractere multibyte.	<code>&lt;stdlib.h&gt;</code>
<b>mbrlen()</b>	Calcula o número de bytes de um caractere multibyte, <i>permitindo reinício</i> .	<code>&lt;wchar.h&gt;</code>
<b>mbtowl()</b>	Retorna o caractere extenso correspondente a um caractere multibyte.	<code>&lt;stdlib.h&gt;</code>
<b>mbrtowl()</b>	Retorna o caractere extenso correspondente a um caractere multibyte, <i>permitindo reinício</i> .	<code>&lt;wchar.h&gt;</code>
<b>mbstowcs()</b>	Converte um <i>string</i> multibyte num <i>string</i> extenso.	<code>&lt;stdlib.h&gt;</code>
<b>mbsrtowcs()</b>	Converte um <i>string</i> multibyte num <i>string</i> extenso, <i>permitindo reinício</i> .	<code>&lt;wchar.h&gt;</code>
<b>wctomb()</b>	Retorna o caractere multibyte correspondente a um caractere extenso.	<code>&lt;stdlib.h&gt;</code>
<b>wcrtomb()</b>	Retorna o caractere multibyte correspondente a um caractere extenso, <i>permitindo reinício</i> .	<code>&lt;wchar.h&gt;</code>
<b>wcstombs()</b>	Converte um <i>string</i> extenso num <i>string</i> multibyte.	<code>&lt;stdlib.h&gt;</code>
<b>wcsrtombs()</b>	Converte um <i>string</i> extenso num <i>string</i> multibyte, <i>permitindo reinício</i> .	<code>&lt;wchar.h&gt;</code>

Tabela 8-1: Funções usadas em conversões entre caracteres multibytes e extensos.

Observe na **Tabela 8-1** que cada função apresentada declarada no cabeçalho `<stdlib.h>` possui uma função correspondente declarada em `<wchar.h>`. Sintaticamente, cada função em `<wchar.h>` diferencia-se de sua correspondente em `<stdlib.h>` por apresentar, no interior de seu nome, uma letra *r* que indica que a respectiva função é **reiniciável**. Funções reiniciáveis são indicadas quando se lida com caracteres multibytes com estado (v. **Seção 7.3.1**). Caso contrário, pode ser mais eficiente utilizar funções não reiniciáveis.

O cabeçalho `<wchar.h>` é exclusivamente dedicado ao tratamento de caracteres e *strings* multibytes e extensos, enquanto `<stdlib.h>` apresenta uma miscelânea de componentes utilizados com diversas finalidades (v. **Capítulo 12**). Além das funções apresentadas na **Tabela 8-1**, o cabeçalho `<wchar.h>` contém outros componentes que dão suporte ao processamento de caracteres extensos e multibytes.

Este capítulo também descreve o cabeçalho `<wctype.h>` que provê suporte para classificação e transformação de caracteres extensos e que é semelhante ao cabeçalho `<ctype.h>` discutido no **Capítulo 6**.

Antes de prosseguir com a completa apresentação dos componentes dos cabeçalhos mencionados, as próximas duas seções apresentam, de modo conciso, parte dos conceitos apresentados em detalhes no **Capítulo 7** e como eles são implementados em C. O objetivo destas seções é dotar o leitor do conhecimento mínimo necessário para um bom entendimento dos referidos componentes que serão apresentados no restante do capítulo.

## 8.2 CONCEITOS E TERMINOLOGIAS

Esta seção apresenta importantes observações sobre conceitos e terminologias usados neste capítulo. Estas observações, que aparecem com frequência nas definições dos componentes da biblioteca padrão de C discutidos neste capítulo, são resumidas na **Tabela 8-2**. O leitor pode usar esta tabela como referência para facilitar o entendimento destes componentes.

CONCEITO	DEFINIÇÃO
<b>Caractere</b> ou <b>caractere monobyte</b>	Caractere representado num único byte, como aqueles discutidos no <b>Volume I</b> e no <b>Capítulo 6</b> .
<b>String</b> ou <b>string monobyte</b>	<i>String</i> de caracteres monobytes, como aqueles discutidos no <b>Volume I</b> e no <b>Capítulo 6</b> .

CONCEITO	DEFINIÇÃO
<b>Caractere extenso</b>	Caractere do tipo <b>wchar_t</b> , cuja representação utiliza um número fixo de $n$ bytes, onde $n > 1$ .
<b>String extenso</b>	<i>String</i> de caracteres extensos.
<b>Caractere multibyte</b>	Caractere cuja representação utiliza um número variável de $n$ bytes, onde $n \geq 1$ e $n \leq \mathbf{MB\_CUR\_MAX}$ .
<b>String multibyte</b>	<i>String</i> de caracteres multibytes.
<b>Código de caracteres</b>	Um conjunto de caracteres e um mapeamento entre cada caractere do conjunto e um inteiro não negativo.
<b>Ponto de código</b>	Inteiro não negativo associado a um caractere num código de caracteres.
<b>Esquema de codificação de caracteres</b>	Representação interna de pontos de código de caracteres no computador. Existem três tipos básicos de esquemas de codificação: monobyte, extenso e multibyte.
<b>Codificação de caracteres</b>	O mesmo que esquema de codificação de caracteres.
<b>Estado de mudança</b>	Valor que indica como serão interpretados os próximos caracteres de um <i>string</i> ou <i>stream</i> numa codificação multibyte com estado.
<b>Variável de estado</b>	Variável que armazena um estado de mudança.
<b>Mudança de estado</b>	Alteração de valor de uma variável de estado com a consequente mudança de interpretação dos próximos caracteres de uma sequência de caracteres multibytes com estado.
<b>Estado inicial</b>	Numa codificação multibyte com estado, é um estado de mudança especial no qual um caractere é representado em apenas um byte.

CONCEITO	DEFINIÇÃO
<b>Estado de conversão</b>	Funções de conversão entre caracteres extensos e multibytes usam uma variável local de duração fixa do tipo <b>mbstate_t</b> ou um parâmetro que é um ponteiro para uma variável externa deste tipo. Esta variável, local ou não, armazena o estado de mudança usado correntemente na conversão e é denominada estado de conversão. Quando uma função de conversão encontra um estado de mudança enquanto examina um ou mais caracteres multibytes, ela atualiza o estado de conversão. Do mesmo modo, quando uma função converte um caractere extenso em caractere multibyte e a representação deste último requer uma mudança de estado, o estado de conversão também é atualizado.

Tabela 8-2: Conceitos comuns em processamento de caracteres extensos e multibytes.

Os mnemônicos usados para facilitar a interpretação de identificadores associados a funções de manipulação de caracteres multibytes e extensos são um tanto difíceis de entender. A **Tabela 8-3** a seguir tem como objetivo facilitar a identificação destas funções a partir de seus nomes.

COMPONENTE DO NOME	POSIÇÃO NO NOME	SIGNIFICADO	EXEMPLO
<b>b</b>	Início ou final	Caractere monobyte	<b>btowc()</b> [converte um <i>caractere monobyte</i> em caractere extenso]
<b>mb</b>	Início ou final	Caractere multibyte	<b>mblen()</b> [calcula o comprimento de um <i>caractere multibyte</i> ]
<b>mbs</b>	Início ou final	<i>String</i> multibyte	<b>mbstowcs()</b> [converte um <i>string multibyte</i> em <i>string</i> extenso]
<b>r</b>	Seguindo <i>mb</i> ou <i>mbs</i>	Reiniciável	<b>mbrtowc()</b> [converte um caractere multibyte em caractere extenso de modo <i>reiniciável</i> ]

COMPONENTE DO NOME	POSIÇÃO NO NOME	SIGNIFICADO	EXEMPLO
<b>to</b>	Meio	Conversão	<b>mbstowcs()</b> [converte um <i>string</i> multibyte em <i>string</i> extenso]
<b>wc</b>	Início ou final	Caractere extenso	<b>btowc()</b> [converte um caractere monobyte em <i>caractere extenso</i> ]
<b>wcs</b>	Início ou final	<i>String</i> extenso	<b>mbstowcs()</b> [converte um <i>string</i> multibyte em <i>string</i> extenso]
<b>wmem</b>	Início	Array de elementos do tipo <b>wchar_t</b>	<b>wmemcpy()</b> [copia caracteres extensos num <i>array</i> de elementos do tipo <b>wchar_t</b> ]

Tabela 8-3: Componentes de nomes de funções de processamento de caracteres multibytes e extensos.

## 8.3 IMPLEMENTAÇÕES DE CARACTERES EXTENSOS E MULTIBYTES EM C

### 8.3.1 CARACTERES E STRINGS EXTENSOS EM C

Num programa em C, um **caractere extenso constante** tem sintaxe semelhante à sintaxe de um caractere monobyte, mas é precedido por **L**. Isto é, caracteres extensos constantes são do tipo **wchar\_t** e podem ser escritos (sempre com o prefixo **L**) usando-se qualquer formato usado para representar caracteres monobytes (v. **Seção 6.1.5**). Exemplos de caracteres extensos constantes válidos:

```
wchar_t carExtenso1 = L'A';
wchar_t carExtenso2 = L'\n';
wchar_t carExtenso3 = L'\x5A';
```

Um ***string* extenso constante** é um array de elementos do tipo **wchar\_t** contendo o caractere extenso nulo (**L'\0'**) ao final. Um ***string* extenso constante** tem sintaxe semelhante àquela usada com *strings* de caracteres monobytes (v. **Seção 6.1.6**), mas é precedido por **L**. Exemplos de *strings* extensos constantes válidos:

```
wchar_t *strExtenso = L"string extenso";
wchar_t *pExtenso = L"extenso";
wchar_t arExtenso[] = L"string extenso";
```

O *string* extenso constante L"abc" é interpretado como o array de caracteres extensos: {L'a', L'b', L'c', L'\0'}. Sem o uso de L, que indica que o *string* é extenso, o mesmo *string* seria interpretado como *string* monobyte: {'a', 'b', 'c', '\0'}.

De acordo com o padrão C99, a macro **\_\_STDC\_ISO\_10646\_\_** deverá ser definida apenas quando o tipo **wchar\_t** for capaz de representar qualquer ponto de código do repertório de caracteres especificado pelo padrão ISO 10646 (ou Unicode)<sup>85</sup>.

**Exemplo:** O programa a seguir demonstra o uso da macro **\_\_STDC\_ISO\_10646\_\_**.

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

int main(void)
{
    printf("sizeof(wchar_t) = %u\n", sizeof(wchar_t));

    #ifdef __STDC_ISO_10646__
        printf("A macro __STDC_ISO_10646__ e' definida\n");
    #else
        printf("A macro __STDC_ISO_10646__ NAO e' definida\n");
    #endif

    return 0;
}
```

Quando executado no Linux, o programa apresenta o seguinte resultado:

```
sizeof(wchar_t) = 4
A macro __STDC_ISO_10646__ e' definida
```

---

<sup>85</sup> Na prática, isto significa que o tipo **wchar\_t** deve ter largura igual a 32 bits para que esta macro seja definida, pois, como foi visto no **Capítulo 7**, são necessários, no máximo, 21 bits para representar qualquer ponto de código definido pelos padrões Unicode e ISO 10646.

Quando executado no Windows XP, o resultado apresentado pelo último programa é:

```
sizeof(wchar_t) = 2
A macro __STDC_ISO_10646__ NAO e' definida
```

**Caracteres universais** são caracteres que fazem parte do repertório de caracteres do padrão ISO/IEC 10646, que, conforme discutido na **Seção 7.5**, é idêntico ao repertório Unicode. De acordo com o padrão C99, caracteres universais podem fazer parte de caracteres e *strings* constantes, identificadores ou comentários<sup>86</sup>.

As notações UCN<sup>87</sup> são utilizadas para representar caracteres que não fazem parte do conjunto básico de caracteres de uma implementação de C (v. **Capítulo 6**). A notação UCN ' \Unnnn ' é usada quando o caractere ora representado faz parte do plano BMP, enquanto a notação ' \Unnnnnnnnn ' é usada quando o caractere está fora do plano BMP (v. **Seção 7.5**).

Um editor de texto capaz de interpretar caracteres do repertório do padrão ISO 10646 escritos usando a sintaxe UCN (e.g., L ' \u00E7 ' ) livraria o programador de ter que escrever caracteres dessa maneira. Em tal editor, quando o programador digitasse, por exemplo, Ç como caractere extenso constante, seria armazenado o valor inteiro correspondente a este caractere de modo transparente para o programador. Infelizmente, tais editores não têm ainda uso muito difundido.

### 8.3.2 CARACTERES E STRINGS MULTIBYTES EM C

Caracteres multibytes não possuem tipo predefinido ou derivado em C, mas um caractere multibyte pode ser implementado como um array de elementos do tipo **char**, visto que ele consiste simplesmente em uma sequência de bytes.

Quando um programa usa uma codificação multibyte com estado (v. **Seção 7.3.1**), caracteres de mudança de estado podem ser armazenados em variáveis do tipo **mbstate\_t** definido em `<wchar.h>` (v. **Seção 8.5.1**).

A macro **MB\_LEN\_MAX**, definida em `<limits.h>` (v. **Seção 2.3**), especifica o comprimento da maior sequência de bytes que compõem um caractere multibyte em qualquer localidade, enquanto a macro **MB\_CUR\_MAX**, definida em `<stdlib.h>` (v. **Seção 12.2.2**), especifica o comprimento da maior sequência de bytes que compõem um caractere multibyte na localidade corrente. Ao contrário da macro **MB\_LEN\_MAX**, a macro **MB\_CUR\_MAX** não precisa ser uma constante.

<sup>86</sup> O uso de caracteres universais em identificadores ou comentários é discutido no **Volume I**.

<sup>87</sup> O acrônimo UCN é derivado de *Universal Character Names*; i.e., *Nomes de Caracteres Universais*, em português.

Para que as funções da biblioteca padrão de C que lidam com caracteres multibytes possam interpretar caracteres multibytes que utilizam um tipo específico de codificação, é necessário que elas sejam informadas sobre o tipo de codificação com o qual estão lidando. Esta informação é provida pelas categorias de localidade **LC\_CTYPE** e **LC\_COLLATE**. Isto significa que, para processar caracteres multibytes em mais de uma forma de codificação, é necessário alterar convenientemente o valor atribuído a estas categorias de localidade chamando **setlocale()** (v. **Seção 5.2**).

A maioria das funções para processamento de *strings* extensos (v. **Seção 8.5**) não possui nenhuma função correspondente para *strings* multibytes na biblioteca padrão de C. Mas, algumas operações envolvendo *strings* multibytes podem ser efetuadas por meio de funções declaradas no cabeçalho `<string.h>` quando estas funções não precisam ter conhecimento se cada byte processado representa um único caractere monobyte ou faz parte de um caractere multibyte. Por exemplo, as funções **strcpy()** e **strcat()** funcionam tão bem com *strings* monobytes quanto com *strings* multibytes porque o único caractere que estas funções precisam interpretar é o caractere terminal de *string*, que tem a mesma representação em *strings* multibytes e monobytes. Em outras palavras, estas funções apenas processam sequências de bytes até encontrar o primeiro byte nulo e não precisam saber quantos bytes compõem um determinado caractere. Do mesmo modo, as funções de entrada e saída formatada usadas para leitura e impressão de *strings* monobytes [e.g., **printf()**] declaradas em `<stdio.h>` (v. **Capítulo 10**) também podem ser usadas com *strings* multibytes, conforme mostram diversos exemplos ao longo do presente capítulo.

Por outro lado, algumas funções declaradas no cabeçalho `<string.h>` podem não funcionar corretamente quando utilizadas com *strings* multibytes. Por exemplo, a função **strchr()**, que procura a última ocorrência de um caractere em um *string*, pode não funcionar com *strings* multibytes mesmo que o caractere procurado seja representado por um único byte. Isto é, algumas funções, como **strchr()**, pressupõem que um caractere é representado por um único byte e, portanto, podem não funcionar satisfatoriamente com *strings* multibytes. Neste caso, uma solução é implementar uma função que realize a operação desejada. Mas, a solução mais cômoda é mesmo converter os *strings* multibytes em *strings* extensos usando uma das funções declaradas em `<stdlib.h>` (v. **Seção 8.4**) ou `<wchar.h>` (v. **Seção 8.5.3**) e, então, utilizar uma das funções apresentadas na **Seção 8.5** para processar os *strings* extensos resultantes da conversão.



## 8.4 CONVERSÕES ENTRE CARACTERES E STRINGS EXTENSOS E MULTIBYTES: <stdlib.h>

### 8.4.1 PRELIMINARES

No cabeçalho <stdlib.h>, são declaradas funções para conversão entre caracteres multibytes e caracteres extensos. Se uma codificação multibyte com estado estiver sendo usada (v. **Seção 7.3.1**), a conversão de um caractere multibyte em caractere extenso depende do estado corrente de mudança. De modo análogo, a conversão de um caractere extenso em caractere multibyte pode requerer a inserção de um caractere de mudança de estado apropriado no caractere multibyte.

Quando a codificação multibyte utiliza estados de mudança, as funções **mblen()**, **mbtowc()** e **wcsrtombs()**, descritas adiante, devem manter atualizado o estado de mudança corrente à medida que exploram um *string*. Estas funções usam uma variável local para acompanhar o estado de mudança corrente. Portanto, para que estas funções funcionem adequadamente, as seguintes regras devem ser seguidas<sup>88</sup>:

- Antes de começar a processar um *string*, deve-se chamar uma dessas funções com **NULL** como endereço do caractere multibyte [e.g, `mblen(NULL, 0);`] para iniciar o estado de mudança com seu valor inicial padrão.
- Deve-se evitar retroceder no processamento de um *string*.
- Deve-se evitar processar *strings* diferentes simultaneamente por uma mesma função.

Todas as funções de conversão entre caracteres multibytes e extensos declaradas em <stdlib.h> usam uma variável local para acompanhar o estado de mudança corrente, o que faz com que elas não possam ser usadas para executar mais de uma conversão concomitantemente. Portanto, quando se usam estas funções de conversão, deve-se converter completamente um *string* antes de começar a conversão de outro *string*. Apesar disso, o padrão ISO de C garante que nenhuma outra função da biblioteca padrão chame estas funções, de modo que os estados armazenados por essas funções possam ser alterados apenas por chamadas feitas pelo próprio programador que as usa.

---

<sup>88</sup> Se a codificação multibyte usada for sem estado, você não precisará preocupar-se com essas questões.

As funções reiniciáveis declaradas em `<wchar.h>` e descritas na **Seção 8.5.3** superam algumas das limitações descritas anteriormente a respeito das funções correspondentes declaradas em `<stdlib.h>`.

Deve-se notar que as conversões efetuadas por funções de conversão entre caracteres (ou *strings*) extensos e multibytes [e.g., **mbstowcs()** e **wcstombs()**] dependem do valor corrente da macro **LC\_TYPE**, pois pontos de código maiores que 127 representam caracteres diferentes dependendo da configuração corrente de localidade.

## 8.4.2 FUNÇÕES DE CONVERSÃO ENTRE CARACTERES E STRINGS EXTENSOS E MULTIBYTES I

*mblen()*

**Incluir:** `<stdlib.h>`

**Descrição:** A função **mblen()** determina o número de bytes num caractere multibyte limitando o número máximo de bytes examinados.

**Protótipo:**

```
int mblen(const char *caractereMB, size_t n)
```

**Parâmetros:**

- `caractereMB` – ponteiro para o primeiro byte de um caractere multibyte.
- `n` – número máximo de bytes que serão examinados.

**Retorno:**

- O número de bytes no caractere multibyte, se `caractereMB` aponta para um caractere multibyte válido não nulo.
- Zero, se `caractereMB` aponta para um caractere nulo (`'\0'`).
- `-1`, se `caractereMB` não aponta para um caractere multibyte válido ou aponta para o início de um caractere multibyte composto por um número de bytes maior do que `n`.

- Se `caractereMB` for **NULL**:
  - Zero, se a codificação for sem estado.
  - Um valor diferente de zero, se a codificação for com estado.

**Observações:**

- Esta função é afetada pelo valor corrente de **LC\_CTYPE**.
- Quando o primeiro argumento é **NULL**, esta função é equivalente a **mbtowc()**.

**Exemplo:** Veja o exemplo da função **wctomb()**.

*mbstowcs()*

**Incluir:** `<stdlib.h>`

**Descrição:** A função **mbstowcs()** converte um *string* multibyte num *string* extenso e armazena o resultado num array de caracteres extensos, limitando o número de caracteres extensos armazenados.

**Protótipo:**

```
size_t mbstowcs( wchar_t *restrict arExtenso,
                  const char *restrict strMB,
                  size_t n )
```

**Parâmetros:**

- `arExtenso` – array de elementos do tipo **wchar\_t** que receberá os caracteres extensos resultantes da conversão.
- `strMB` – *string* de caracteres multibytes que será convertido.
- `n` – número máximo de caracteres extensos armazenados no array.

**Retorno:** Número de caracteres convertidos ou `(size_t) -1` se ocorrer algum erro (i.e., se for encontrado um caractere multibyte inválido).

**Observações:**

- A função **mbstowcs()** termina o *string* de caracteres extensos resultante da conversão com o caractere terminal extenso `L'\0'` apenas se ela ainda não tiver armazenado o número máximo de caracteres extensos especificado pelo terceiro argumento. Se o valor de retorno for igual ao valor do terceiro argumento, o caractere terminal `L'\0'` não será incluído no array (primeiro argumento) e, portanto, não constituirá um *string* extenso.
- Consulte também **mblen()** e **mbtowc()**.

**Exemplo:** O programa a seguir converte um *string* multibyte num *string* extenso na localidade corrente.

```
#include <locale.h>
#include <stdlib.h>
#include <wchar.h>
#include <stdio.h>

#define LOCALE_BR    "pt_BR.utf8" /* Linux Ubuntu 8.10 */

/****
 *
 * Função MultibyteParaExtenso(): converte um string
 *                               multibyte na localidade
 *                               corrente em caracteres
 *                               extensos
 *
 * Argumentos: str (entrada) - o string multibyte que
 *                               será convertido
 *
 * Retorno: ponteiro para o string convertido ou
 *          NULL se a conversão não for possível
 *
 * Nota: O string a ser convertido deve ser compatível
 *       com a localidade corrente.
 *
 ****/
wchar_t *MultibyteParaExtenso(char * str)
```

```

{
    wchar_t *ptr;
    size_t   nCars;

    /* Usa-se NULL como primeiro argumento */
    /* para calcular o espaço necessário */
    nCars = mbstowcs(NULL, str, 0);

    /* Se a função mbstowcs() retorna (size_t)-1 */
    /* é porque ocorreu um erro na codificação. */
    if (nCars == (size_t) -1)
        return NULL;

#ifdef TESTE
    wprintf( L"\nNumero de caracteres extensos no "
             "resultado: %d\n", nCars );
#endif

    /* Aloca o espaço necessário. O retorno */
    /* de mbstowcs() não inclui o caractere */
    /* terminal L'\0'. */
    ptr = malloc((nCars + 1)*sizeof(wchar_t));

    if (!ptr) /* Espaço não foi alocado */
        return NULL;

    /* Copia o resultado da conversão */
    /* no espaço alocado */
    mbstowcs(ptr, str, nCars);

    ptr[nCars] = L'\0'; /* Termina o string extenso */

    /* Deve-se liberar o espaço apontado por */
    /* ptr quando ele não for mais necessário */
    return ptr;
}

int main()
{
    char      *localidade;
    wchar_t   *str;

    localidade = setlocale(LC_ALL, LOCALE_BR);

```

```

    if (!localidade) {
        wprintf(L"\nNao pode alterar a localidade\n");
        return 1;
    }

    wprintf( L"\nA localidade corrente e': %s",
            localidade );

    /* O string é "Ação" em UCN */
    str = MultibyteParaExtenso("A\u00E7\u00E3o");

    if (str) {
        wprintf(L"String extenso: %ls\n", str);
        free(str);
    } else {
        fprintf( stderr,
                "\nOcorreu um erro de codificacao\n" );
        return 1;
    }

    wprintf(L"\n-----\n");

    localidade = setlocale(LC_ALL, "C");

    if (!localidade) {
        wprintf(L"\nNao pode alterar a localidade\n");
        return 1;
    }

    wprintf( L"\nA localidade corrente e': %s\n",
            localidade );

    /* O string é "Ação" em UCN */
    str = MultibyteParaExtenso("A\u00E7\u00E3o");

    if (str) {
        wprintf(L"String extenso: %ls\n", str);
        free(str);
    } else {
        fprintf( stderr,
                "Ocorreu um erro de codificacao\n" );
        return 1;
    }
}

```

```
    return 0;
}
```

Como resultado de uma execução do último programa tem-se<sup>89</sup>:

```
A localidade corrente e': pt_BR.utf8
Numero de caracteres extensos no resultado: 4
String extenso: Ação
```

```
-----
```

```
A localidade corrente e': C
Ocorreu um erro de codificacao
```

### *mbtowc()*

**Incluir:** <stdlib.h>

**Descrição:** A função **mbtowc()** determina o caractere extenso correspondente a um caractere multibyte limitando o número de bytes examinados.

### **Protótipo:**

```
int mbtowc( wchar_t *restrict carExtenso,
            const char *restrict carMB,
            size_t n )
```

### **Parâmetros:**

- **carExtenso** – endereço da variável que receberá o caractere extenso resultante da conversão.
- **carMB** – ponteiro para o caractere multibyte que será convertido.
- **n** – número máximo de bytes que serão examinados no caractere multibyte apontado por **carMB**.

---

<sup>89</sup> O programa foi compilado e executado no sistema Linux Ubuntu 8.10. As opções usadas com o compilador gcc foram `-std=c99` e `-DTESTE`.

**Retorno:**

- O número de bytes do caractere multibyte que foram realmente convertidos, excetuando-se os casos a seguir.
- -1, quando `carMB` aponta para um caractere multibyte inválido.
- Zero, quando `carMB` aponta para o caractere nulo (`'\0'`).
- Se o segundo argumento for **NULL**:
  - Zero, se a codificação corrente for sem estado.
  - Um valor diferente de zero, se a codificação corrente for com estado.

**Observações:**

- O valor de retorno nunca ultrapassa o valor de **MB\_CUR\_MAX** ou `n`.
- Consulte também `mblen()` e `mbtowcs()`.

**Exemplo:**

```
#include <locale.h>
#include <stdlib.h>
#include <wchar.h>
#include <stdio.h>

#define LOCALE_BR  "pt_BR.utf8" /* Linux */

int main()
{
    char      *localidade;
    wchar_t   carExtenso;
    int       n;

    localidade = setlocale(LC_ALL, LOCALE_BR);

    if (!localidade) {
        wprintf(L"\nNao pude alterar a localidade\n");
        return 1;
    }

    wprintf( L"\nA localidade corrente e': %s",
            localidade);
```



```

wprintf( L"\nA codificacao corrente e' %s estado\n",
          mbtowc(NULL, NULL, 0) ? "com" : "sem");

/* \u00E7 é 'ç' em UCN */
n = mbtowc(&carExtenso, "\u00E7", MB_CUR_MAX);

if (-1 != n)
    wprintf( L"\nCaractere extenso convertido: %lc\n",
             carExtenso );
else
    fprintf( stderr, "\nO caractere e' invalido na "
              "localidade corrente\n");

wprintf(L"\n-----\n");

localidade = setlocale(LC_ALL, "C");

if (!localidade) {
    wprintf(L"\nNao pude alterar a localidade\n");
    return 1;
}

wprintf( L"\nA localidade corrente e': %s\n",
          localidade );

wprintf( L"\nA codificacao corrente e' %s estado\n",
          mbtowc(NULL, NULL, 0) ? "com" : "sem");

/* \u00E7 é 'ç' em UCN */
n = mbtowc(&carExtenso, "\u00E7", MB_CUR_MAX);

if (-1 != n)
    wprintf( L"\nCaractere extenso convertido: %lc\n",
             carExtenso );
else
    fprintf( stderr, "\nO caractere e' invalido na "
              "localidade corrente\n");

return 0;
}

```

Resultado da execução do último programa no sistema Linux:

```
A localidade corrente e': pt_BR.utf8
A codificacao corrente e' sem estado
Caractere extenso convertido: ç
```

```
-----
```

```
A localidade corrente e': C
A codificacao corrente e' sem estado
O caractere e' invalido na localidade corrente
```

*wcstombs()*

**Incluir:** <stdlib.h>

**Descrição:** A função **wcstombs()** converte um *string* extenso num *string* multi-byte. O processo termina quando um caractere extenso nulo ou inválido é encontrado ou é atingido o número máximo de bytes armazenados no array que recebe o resultado.

**Protótipo:**

```
size_t wcstombs( char *restrict destino,
                  const wchar_t *carExtensos,
                  size_t n )
```

**Parâmetros:**

- *destino* – array que receberá os caracteres multibytes resultantes da conversão.
- *carExtensos* – array contendo os caracteres extensos que serão convertidos.
- *n* – número máximo de bytes armazenados em *destino*.

**Retorno:** Número de bytes armazenados no array de caracteres multibytes, sem incluir o caractere nulo (se for o caso); `(size_t)-1`, se ocorrer algum erro de conversão.

**Observações:**

- Se o número máximo de bytes for processado antes de um caractere extenso nulo ser encontrado, o array de caracteres multibytes não receberá um caractere nulo (e, portanto, não será um *string*).
- A conversão executada em cada caractere extenso é a mesma efetuada pela função **wctomb()**.
- Esta função é afetada pelo valor corrente de **LC\_CTYPE**.
- Frequentemente se usa **NULL** como primeiro argumento para calcular o espaço necessário para conter o *string* multibyte (v. exemplo).
- Consulte também **wctomb()**.

**Exemplo:** O programa a seguir converte um *string* extenso num *string* multibyte compatível com uma dada localidade.

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

#define LOCALE_BR "pt_BR.utf8" /* Linux Ubuntu 8.1 */

/*****
 *
 * Função ExtensoParaMultibyte(): converte um string
 *                               extenso num string
 *                               multibyte compatível
 *                               com uma dada localidade
 *
 * Argumentos: str (entrada) - o string extenso que será
 *                               convertido
 *               local (entrada) - localidade na qual será
 *                               feita a conversão
 *
 * Retorno: ponteiro para o string convertido ou NULL se
 *          a conversão não for possível
 *
 *****/
```

```

*
* NOTA: Deve-se liberar o espaço alocado para conter o
*       string quando este não for mais necessário.
****/

char *ExtensoParaMultibyte( const wchar_t *str,
                           const char *local )
{
    char    *ptr;
    size_t   s;
    char    *p, *localCorrente, *localidade;

    /* Guarda a localidade corrente */
    p = setlocale(LC_ALL, "");
    localCorrente = malloc(strlen(p) + 1);
    strcpy(localCorrente, p);

    /* Tenta alterar a localidade corrente */
    localidade = setlocale(LC_ALL, local);

    if (!localidade) {
        fprintf( stderr,
                "\nNao pode alterar a localidade\n" );
        free(localCorrente);
        return NULL;
    }

#ifdef TESTE
    printf("\nA localidade corrente e': %s", localidade);
#endif

    /* Usa NULL como primeiro argumento */
    /* para calcular o espaço necessário */
    /* para conter o string multibyte */
    s = wcstombs(NULL, str, 0);

    /* Se a função mbstowcs() retorna (size_t) -1 */
    /* significa que há caracteres que não podem */
    /* ser convertidos para a localidade corrente */
    if (s == -1) {
        fprintf( stderr,
                "\nA conversao nao e' possivel "
                "nesta localidade\n" );
    }
}

```

```

        free(localCorrente);
        return NULL;
    }

    /* Aloca o espaço necessário */
    if (!(ptr = malloc(s + 1))) {
        fprintf( stderr,
            "\nNao foi possivel alocar o espaco "
            "necessario para conter o string\n" );
        free(localCorrente);
        return NULL;
    }

    /* Copia o resultado da          */
    /* conversão no espaço alocado */
    wcstombs(ptr, str, s);

    ptr[s] = '\0'; /* Termina o string multibyte */

    /* Restaura a localidade corrente */
    localidade = setlocale(LC_ALL, localCorrente);

    if (!localidade)
        fprintf( stderr,
            "\nNao foi possivel restaurar "
            "a localidade corrente\n" );

    free(localCorrente); /* Não é mais necessária */

#ifdef TESTE
    printf("\nString convertido: %s\n", ptr);
#endif

    return ptr;
}

int main()
{
    char *localidade;
    char *str, *p;

    str = ExtensoParaMultibyte(L"Eu\u2665JP", LOCALE_BR);

```

```

if (!str) {
    fprintf(stderr, "Ocorreu um erro de codificacao");
    return 1;
}

/* Não há problema em usar strlen() com um      */
/* string multibyte, mas, neste caso, o valor */
/* retornado é o número de bytes no string e */
/* não o número de caracteres.                */
printf( "\nNumero de bytes no string multibyte: "
        "%d\n", strlen(str) );

/* Imprime cada byte do string multibyte */
/* em formato hexadecimal                */

printf("Bytes que compoem o string multibyte: ");

/* É preciso guardar um ponteiro para */
/* o início do string para que seja */
/* possível liberar o espaço          */
p = str;

while(*str)
    printf("%#x ", (unsigned char)*str++);

putchar('\n');

free(p); /* Não se pode fazer free(str) aqui!!! */

return 0;
}

```

Resultado de uma execução do último programa no sistema Linux<sup>90</sup>:

```

A localidade corrente e': pt_BR.utf8
String convertido: Eu♥JP

```

```

Numero de bytes no string multibyte: 7
Bytes que compoem o string multibyte: 0x45 0x75 0xe2 0x99 0xa5
0x4a 0x50

```

---

<sup>90</sup> As opções usadas com o compilador gcc foram `-std=c99`, `-g` e `-DTESTE`.

*wctomb()*

**Incluir:** <stdlib.h>

**Descrição:** A função **wctomb()** converte um caractere extenso num caractere multibyte.

**Protótipo:**

```
int wctomb(char *destino, const wchar_t carExtenso)
```

**Parâmetros:**

- **destino** – array que receberá os caracteres multibytes resultantes da conversão.
- **carExtenso** – caractere extenso a ser convertido.

**Retorno:**

- -1, se **destino** não é **NULL** e **carExtenso** não corresponde a um caractere multibyte válido.
- Se **destino** for **NULL**:
  - Um valor diferente de zero, se a codificação corrente for com estado.
  - Zero, caso se trate de uma codificação sem estado.
- Em outros casos, o número de bytes no caractere multibyte correspondente a **carExtenso**.

**Observações:**

- O número máximo de caracteres armazenados no array **destino** é dado pela macro **MB\_CUR\_MAX** (definida em <stdlib.h>).
- O valor retornado por esta função nunca é maior do que **MB\_CUR\_MAX**.
- Se a função **wctomb()** receber **NULL** como primeiro argumento, ela se comportará como **mblen()**.
- Se **carExtenso** for um caractere extenso nulo (**L'\0'**), a função armazenará no array **destino** um caractere nulo, precedido por uma sequência de bytes correspondente ao estado inicial de mudança.

- Esta função é afetada pelo valor corrente de **LC\_CTYPE**.
- Consulte também as funções **mblen()** e **wctombs()**.

**Exemplo:** O programa a seguir demonstra o uso das funções **mblen()** e **wctomb()**.

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define LOCALE_BR "pt_BR.utf8" /* Linux */

int main(void)
{
    char    *localidade;
    char    cm[MB_LEN_MAX + 1];
    wchar_t ce = L'\u263A'; /* White Smiling Face */
    int      n;

    localidade = setlocale(LC_ALL, LOCALE_BR);

    if (!localidade) {
        fprintf( stderr, "Nao foi possivel "
                  "alterar a localidade\n" );
        return 1;
    }

    printf("A localidade %s e' %s estado\n", localidade,
           mblen(NULL, MB_CUR_MAX) ? "COM" : "SEM" );

    n = wctomb(cm, ce);
    printf("Numero de caracteres convertidos: %d\n", n);

    n = mblen(cm, MB_CUR_MAX);

    /* Transforma o caractere multibyte num */
    /* string multibyte com um caractere    */
    cm[n] = '\0';

    printf("Caractere multibyte: %s\n", cm);
```



```
printf("Tamanho do caractere multibyte: %d\n", n);

return 0;
}
```

Resultado de uma execução do último programa no Linux<sup>91</sup>:

```
A localidade pt_BR.utf8 e' SEM estado
Numero de caracteres convertidos: 3
Caractere multibyte: @
Tamanho do caractere multibyte: 3
```

## 8.5 SUPORTE PARA CARACTERES MULTIBYTES E EXTENSOS: <wchar.h>

O cabeçalho <wchar.h> provê suporte para operações de entrada e saída e de processamento de *strings* extensos, além de funções equivalentes àquelas declaradas no cabeçalho <stdlib.h> para conversão entre caracteres multibytes e extensos (v. **Seção 8.4**). As funções declaradas em <wchar.h> podem ser divididas em quatro categorias:

- Funções de conversões entre caracteres extensos e caracteres multibytes (v. **Seção 8.5.3**).
- Funções de processamento de arrays de caracteres extensos (v. **Seção 8.5.4**).
- Funções de processamento de *strings* extensos (v. **Seção 8.5.5**).
- Funções de conversões de *strings* extensos em números (v. **Seção 8.5.6**).
- Funções de entrada e saída de caracteres e *strings* extensos (v. **Seção 10.8**).

As subseções a seguir descrevem em detalhes os componentes do cabeçalho <wchar.h>, com exceção daqueles dedicados a entrada e saída de caracteres extensos que serão explorados no **Capítulo 10**.

---

91 As opções usadas com o compilador gcc foram `-std=c99` e `-DTESTE`.

## 8.5.1 TIPOS

*mbstate\_t*

**Incluir:** `<wchar.h>`

**Descrição:** **mbstate\_t** é o tipo de variáveis que representam estados de mudança em codificações multibytes com estado (v. **Seção 7.3.1**).

Antes do primeiro uso, uma variável do tipo **mbstate\_t** deve ser iniciada com o estado inicial. Como não existe nenhuma função específica para atribuir um valor a uma variável do tipo **mbstate\_t**, tipicamente, uma variável deste tipo é iniciada como:

```
mbstate_t estado;
memset(&estado, 0, sizeof(estado));
```

A maioria das funções de conversão entre caracteres e *strings* multibytes e extensos (v. **Seção 8.5.3**) usa um parâmetro que é um ponteiro para uma variável do tipo **mbstate\_t**. Tal variável é mantida atualizada com o estado corrente de mudança pelas funções de conversão que a usam. Se um programa não usa nenhuma codificação com estado, como a codificação JIS usada com a língua japonesa, não é necessário definir nenhuma variável deste tipo. Neste caso, simplesmente, usa-se **NULL** como valor do parâmetro que deveria apontar para uma variável deste tipo.

*size\_t*

**Incluir:** `<time.h>`, `<string.h>`, `<wchar.h>`, `<stdlib.h>` ou `<std-def.h>`

**Descrição:** **size\_t** é um tipo inteiro sem sinal definido em vários cabeçalhos (v. **Seção 1.7.1**). Este tipo é declarado no cabeçalho `<wchar.h>` porque ele é usado como tipo de retorno e de argumentos de diversas funções [e.g., **mbrlen()**] declaradas neste cabeçalho.

*tm*

**Incluir:** `<time.h>` ou `<wchar.h>`

**Descrição:** Uma estrutura do tipo **struct tm** contém membros que descrevem várias propriedades de data e hora e é mostrada em detalhes na apresentação do cabe-

çalho `<time.h>` (v. **Seção 5.3.1**). Este rótulo de estrutura é declarada no cabeçalho `<wchar.h>` porque é necessário para declaração de um dos parâmetros da função `wcsftime()` (v. **Seção 8.5.4**).

*wchar\_t*

**Incluir:** `<wchar.h>`, `<stdlib.h>` ou `<stddef.h>`

**Descrição:** O tipo `wchar_t` serve para representar caracteres extensos. A largura deste tipo é dependente de implementação. Este tipo também é definido em `<stdlib.h>` (v. **Seção 12.2.1**) e `<stddef.h>` (v. **Seção 12.3.1**).

*wctype\_t*

**Incluir:** `<wchar.h>` ou `<wctype.h>`

**Descrição:** O tipo `wctype_t` representa valores associados a classificações de caracteres extensos que são específicas de localidade e é o tipo do valor retornado pela função `wctype()` (v. **Seção 8.6.3**). Este tipo também é definido em `<wctype.h>` (v. **Seção 8.6.1**).

*wint\_t*

**Incluir:** `<wchar.h>` ou `<wctype.h>`

**Descrição:** O tipo `wint_t` é usado para representar caracteres extensos quando eles são passados como parâmetros ou retornados por funções. Desse modo, pode-se dizer que o tipo `wint_t` está para `int` assim como `wchar_t` está para `char`. Este tipo também é definido em `<wctype.h>` (v. **Seção 8.6.1**).

## 8.5.2 MACROS

*NULL*

**Incluir:** `<time.h>`, `<string.h>`, `<wchar.h>`, `<stdlib.h>` ou `<stddef.h>`

**Descrição:** A macro **NULL** representa um ponteiro nulo e é definida em vários cabeçalhos (v. **Seção 1.7.2**).

## *WCHAR\_MAX*

**Incluir:** `<wchar.h>`

**Descrição:** A macro **WCHAR\_MAX** é expandida no maior valor que pode ser representado numa variável do tipo **wchar\_t**. Esta macro também é definida em `<stdint.h>` (v. **Seção 2.4.2**).

## *WCHAR\_MIN*

**Incluir:** `<wchar.h>`

**Descrição:** A macro **WCHAR\_MIN** é expandida no menor valor que pode ser representado numa variável do tipo **wchar\_t**. Esta macro também é definida em `<stdint.h>` (v. **Seção 2.4.2**).

## *WEOF*

**Incluir:** `<wchar.h>` ou `<wctype.h>`

**Descrição:** A macro **WEOF**, do tipo **wint\_t**, é usada para indicar o final de um *stream* com orientação extensa ou uma condição de erro. O padrão C99 garante que o valor de **WEOF** seja diferente do valor de qualquer caractere extenso, mas, diferentemente do que ocorre com a macro similar **EOF**, o valor de **WEOF** pode ser positivo.

Esta macro também é definida em `<wctype.h>` (v. **Seção 8.6.2**).

### 8.5.3 FUNÇÕES DE CONVERSÃO ENTRE CARACTERES E STRINGS EXTENSOS E MULTIBYTES II

As funções declaradas em `<wchar.h>` que possuem equivalentes em `<stdlib.h>` possuem uma letra *r* no meio de seus nomes, que denota *reiniciável* (v. **Tabela 8-1** na **Seção 8.1**). Funções reiniciáveis declaradas em `<wchar.h>` recebem um argumento a mais do que suas respectivas contrapartes em `<stdlib.h>`. Este argumento adicional é um ponteiro para uma variável do tipo **mbstate\_t** (v. **Seção 8.5.1**) que representa o estado corrente de conversão usado na interpretação de caracteres multibytes com estado. Estas funções são mais úteis quando se utiliza este tipo de codificação, de maneira que, se este não for o caso, é melhor usar funções não reiniciáveis (declaradas em `<stdlib.h>`).

*btowc()*

**Incluir:** `<wchar.h>`

**Descrição:** A função **btowc()** converte um caractere multibyte que ocupa apenas um byte num caractere extenso.

**Protótipo:**

```
wint_t btowc(int c)
```

**Parâmetro:** *c* – um valor inteiro interpretado como caractere multibyte de um byte no estado inicial (se a codificação corrente for com estado).

**Retorno:**

- O caractere recebido como parâmetro convertido num caractere extenso, se a conversão for possível.
- **WEOF**, se *c* for **EOF** ou se a conversão não for possível.

**Observações:**

- Esta função não utiliza nenhuma informação sobre estado de mudança. Daí a restrição de o caractere multibyte ser interpretado em seu estado inicial ou a codificação ser sem estado.

- Esta função pode ser usada para converter caracteres monobytes em caracteres extensos.

### Exemplo:

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

#define MAIOR_CARACTERE    255
#define PORTUGUES_BRASIL "pt_BR.utf8" /* Linux */

int main(void)
{
    wint_t ce;    /* Armazena um caractere extenso */
    int    cmb;   /* Armazena um caractere multibyte */

    if (!setlocale(LC_ALL, PORTUGUES_BRASIL)) {
        printf("Nao foi possivel alterar a localidade\n");
        return 1;
    }

    for (cmb = 0; cmb <= MAIOR_CARACTERE; ++cmb) {
        ce = btowc(cmb);

        if (ce == WEOF)
            printf( "%#04x nao e' um caractere multibyte "
                    "de um byte valido\n", cmb );
        else
            printf( "Caractere extenso correspondente a "
                    "%#04x: %#06x\n", cmb, ce );
    }

    ce = btowc EOF);

    printf("O caractere multibyte e' EOF.\n");
    printf( "Representacao do caractere extenso WEOF: "
            "%#06x\n", ce );

    return 0;
}
```

O resultado da execução deste último programa no Linux é o seguinte:

```
Caractere extenso correspondente a 0000: 000000
Caractere extenso correspondente a 0x01: 0x0001
Caractere extenso correspondente a 0x02: 0x0002
Caractere extenso correspondente a 0x03: 0x0003
... [Trecho omitido similar ao antecedente]
Caractere extenso correspondente a 0x7f: 0x007f
0x80 nao e' um caractere multibyte de um byte valido
0x81 nao e' um caractere multibyte de um byte valido
0x82 nao e' um caractere multibyte de um byte valido
... [Trecho omitido similar ao antecedente]
O caractere multibyte e' EOF.
Representacao do caractere extenso WEOF: 0xffffffff
```

## *mbrlen()*

**Incluir:** <wchar.h>

**Descrição:** A função **mbrlen()** determina o número de bytes de um caractere multibyte, podendo reiniciar no meio de um caractere multibyte.

### **Protótipo:**

```
size_t mbrlen( const char *restrict s, size_t nMax,
               mbstate_t *restrict pEstado )
```

### **Parâmetros:**

- *s* – ponteiro para um array contendo o caractere multibyte a ser examinado.
- *nMax* – número máximo de bytes que serão examinados.
- *pEstado* – ponteiro para o estado corrente de mudança; se este valor for **NULL**, será usado um valor armazenado numa variável local de duração fixa.

### **Retorno:**

- O número de bytes que constituem o caractere multibyte, se os próximos *nMax* ou menos bytes constituem um caractere multibyte válido.
- Zero, se o próximo caractere multibyte completo é um caractere nulo.

- `(size_t)-1`, se `s` não aponta para um caractere multibyte válido.
- `(size_t)-2`, se, depois de examinar `nMax` bytes, o estado de mudança (i.e., `*pEstado`) indica que um caractere multibyte inválido ou incompleto foi encontrado.
- `(size_t)-3`, se nenhum byte adicional é necessário para completar o próximo caractere multibyte.

### Observações:

- Esta função pode ser implementada como:

```
return mbrtowc( NULL, s, nMax,
                pEstado ? pEstado : &estadoLocal );
```

onde `estadoLocal` é uma variável local de duração fixa que armazena o estado de mudança.

- O valor da variável apontada por `pEstado` pode ser atualizado pela função para refletir um novo estado de mudança.
- Esta função é afetada pelo valor da localidade correntemente atribuída à categoria **LC\_CTYPE**.
- Esta função armazena o valor **EILSEQ**<sup>92</sup> na variável global **errno** (v. **Capítulo 11**) quando `s` não aponta para um caractere multibyte válido.

### Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <locale.h>

#define LOCALE_BR "pt_BR.utf8" /* Linux */

int main( void )
{
    size_t      nCar = 0, comp = 0;
    mbstate_t   estado;
```

---

92 EILSEQ significa *sequência (multibyte) ilegal* (v. Seção 11.5.1).



```

char          *localidade, *aux;
const char *str = "A vida s\u00F3 pode ser entendida"
                  " olhando-se para tr\u00E1s, mas "
                  "s\u00F3 pode ser vivida olhando-"
                  "se para a frente. (An\u00F4nimo)";

localidade = setlocale(LC_ALL, LOCALE_BR);

if (!localidade)
    printf("Nao foi possivel alterar a localidade.");

    /* Inicia estado de conversão */
memset(&estado, 0, sizeof(estado));

aux = str;

while ((comp = mbrlen(str, MB_CUR_MAX, &estado)) != 0
        && comp != (size_t)-1 && comp != (size_t)-2
        && comp != (size_t)-3) {
    str += comp;
    nCar++;
}

printf("%s\nComprimento do string: %d\n", aux, nCar);

return 0;
}

```

#### Resultado do programa quando executado no Linux:

```

A vida só pode ser entendida olhando-se para trás, mas só pode
ser vivida olhando-se para a frente. (Anônimo)
Comprimento do string: 109

```

**Exemplo:** Conforme foi afirmado anteriormente, quando um programa não usa codificações com estado, ele não precisa usar variáveis do tipo **mbstate\_t** para acompanhar mudanças de estado, como foi feito no último programa. Assim, este programa poderia ser implementado de modo mais simples como mostrado a seguir.

```

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

```

```

#include <wchar.h>
#include <locale.h>

#define LOCALE_BR "pt_BR.utf8" /* Linux */

int main(void)
{
    size_t      nCar = 0, comp = 0;
    char        *localidade, *aux;
    const char *str = "A vida s\u00F3 pode ser entendida"
                      " olhando-se para tr\u00EAs, mas "
                      "s\u00F3 pode ser vivida olhando-"
                      "se para a frente. (An\u00F4nimo)";

    localidade = setlocale(LC_ALL, LOCALE_BR);

    if (!localidade)
        printf("Nao foi possivel alterar a localidade.");

    aux = str;

    /* Note o uso de NULL como terceiro */
    /* argumento de mbrlen() */
    while ((comp = mbrlen(str, MB_CUR_MAX, NULL)) != 0 &&
           comp != (size_t)-1 && comp != (size_t)-2 &&
           comp != (size_t)-3) {
        str += comp;
        nCar++;
    }

    printf("%s\nComprimento do string: %d\n", aux, nCar);

    return 0;
}

```

O resultado produzido por este último programa é o mesmo daquele decorrente do programa anterior, pois os dois programas usam uma codificação UTF-8, que é uma codificação sem estado.

*mbrtowc()***Incluir:** <wchar.h>**Descrição:** A função **mbrtowc()** converte um caractere multibyte num caractere extenso.**Protótipo:**

```
size_t mbrtowc( wchar_t *restrict carExtenso,
               const char *restrict s, size_t nMax,
               mbstate_t *restrict pEstado )
```

**Parâmetros:**

- *carExtenso* – ponteiro para uma variável do tipo **wchar\_t** onde o caractere extenso resultante da conversão será armazenado ou **NULL** (v. adiante).
- *s* – ponteiro para a sequência de bytes a ser convertida (supostamente um caractere multibyte).
- *nMax* – número máximo de bytes que serão convertidos.
- *pEstado* – ponteiro para o estado de mudança do caractere multibyte ou **NULL**.

**Retorno:**

- Zero, se *s* for **NULL** ou apontar para um caractere nulo.
- O número de bytes utilizados na conversão do caractere multibyte, se, no máximo, os próximos *nMax* bytes constituírem um caractere multibyte válido.
- $(\text{size\_t}) - 1$ , se a função encontrar algum erro de codificação (i.e., *s* não aponta para um caractere multibyte válido).
- $(\text{size\_t}) - 2$ , se os próximos *nMax* bytes fizerem parte de um caractere multibyte incompleto (mas, potencialmente válido).

**Observações:**

- Se o parâmetro `pEstado` for **NULL**, o estado de conversão será representado por uma variável local de duração fixa iniciada com o estado inicial de conversão. A função pode atualizar a variável apontada por este parâmetro, de modo que ela possa ser utilizada adiante na interpretação do próximo caractere de uma sequência de caracteres multibytes.
- Se `s` for **NULL** ou o caractere convertido for `L'\0'`, a função atribuirá o estado inicial de mudança à variável apontada por `pEstado`.
- A função armazena o valor **EILSEQ** na variável global **errno** quando encontra algum erro de codificação; neste caso, o estado de conversão torna-se indefinido.

**Exemplo:** A função `StrMBParaExtenso()` do programa a seguir copia um *string* multibyte num *string* extenso e, ao mesmo tempo, converte letras minúsculas em maiúsculas.

```
#include <stdio.h>
#include <wchar.h>
#include <stdlib.h>
#include <locale.h>
#include <string.h>
#include <wctype.h>

#define PORTUGUES_BRASIL "pt_BR.utf8"

/****
 *
 * Função StrMBParaExtenso(): retorna um string extenso
 *                             equivalente a um dado
 *                             string multibyte em
 *                             maiúsculas
 *
 * Argumentos: s (entrada) - o string multibyte a ser
 *                             convertido
 *
 * Retorno: ponteiro para o string convertido,
 *          se for possível a conversão; NULL,
 *          em caso contrário
 *
 ****/
```

```

wchar_t *StrMBParaExtenso(const char *s)
{
    size_t    nc; /* Número de caracteres resultantes */
    wchar_t   *strExtenso, *p, tmp;
    mbstate_t estado;
    size_t     nBytes;

    /* Na pior hipótese, o número de caracteres */
    /* extensos será igual ao número de bytes no */
    /* string multibyte (i.e., cada caractere */
    /* multibyte ocupa exatamente um byte). */
    nc = strlen(s);
    strExtenso = malloc ((nc + 1)*sizeof(wchar_t));

    /* Inicial o estado de conversão */
    memset(&estado, '\0', sizeof (estado));

    /* Guarda o início do string extenso */
    p = strExtenso;

    while ((nBytes = mbrtowc(&tmp, s, nc, &estado)) > 0){
        if (nBytes >= (size_t)-2) /* Caractere inválido */
            return NULL;

        *strExtenso++ = towupper(tmp);
        nc -= nBytes;
        s += nBytes;
    }

    *strExtenso++ = L'\0'; /* Termina o string extenso */

    /* p aponta para o início do string convertido */
    return p;
}

int main()
{
    wchar_t *strExtenso;
    char     *strMB = "A\u00E7\u00E3o"; /* "Ação" */

    if (!setlocale(LC_ALL, PORTUGUES_BRASIL)) {
        printf("Nao foi possivel alterar a localidade\n");
    }
}

```

```

        return 1;
    }

    wprintf(L"\nString multibyte: %s", strMB);

    strExtenso = StrMBParaExtenso(strMB);

    if (strExtenso)
        wprintf( L"\nString extenso convertido: %ls\n",
                strExtenso );
    else
        wprintf(L"\nNao foi possivel converter string\n");

    return 0;
}

```

Quando executado no sistema Linux, o último programa produz o seguinte resultado:

```

String multibyte: Ação
String extenso convertido: AÇÃO

```

### *mbstate\_t*

**Incluir:** <wchar.h>

**Descrição:** A função **mbstate\_t** testa se a variável apontada por seu argumento representa um estado inicial de mudança.

**Protótipo:**

```
int mbstate_t(const mbstate_t *pEstado)
```

**Parâmetro:** pEstado – ponteiro para uma variável do tipo **mbstate\_t** contendo um estado de conversão ou **NULL**.

**Retorno:** Um valor diferente de zero, se pEstado é **NULL** ou se \*pEstado representa um estado inicial de conversão; zero, caso contrário.

**Observação:** Esta função jamais precisará ser chamada por um programa que não use nenhuma codificação multibyte com estado.

**Exemplo:**

```

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

int main(void)
{
    char      *str = "ABC";
    mbstate_t  estado;
    wchar_t    cExtenso;
    int        n;

    /* Inicia estado de conversão */
    memset(&estado, 0, sizeof(estado));

    /* Converte o primeiro caractere */
    /* multibyte do string          */
    n = mbrtowc(&cExtenso, str, MB_CUR_MAX, &estado);

    /* Verifica se estado corrente de */
    /* conversão ainda é o inicial     */
    if (mbsinit(&estado))
        printf("Ainda em estado inicial de conversao\n");
    else
        printf("O estado de conversao foi alterado\n");

    return 0;
}

```

O programa apresentado como último exemplo tem propósito meramente ilustrativo. Ou seja, ele não tem nenhuma validade prática já que não usa nenhuma codificação com estado que justifique a chamada de **mbsinit()**.

*mbsrtowcs()*

**Incluir:** <wchar.h>

**Descrição:** A função **mbsrtowcs()** converte um *string* de caracteres multibytes num *string* de caracteres extensos, limitando o número de caracteres extensos armazenados.

**Protótipo:**

```
size_t mbsrtowcs( wchar_t *restrict strExtenso,
                  const char **restrict s, size_t nMax,
                  mbstate_t *restrict pEstado )
```

**Parâmetros:**

- **strExtenso** – array de elementos do tipo **wchar\_t** onde os caracteres extensos resultantes da conversão serão armazenados.
- **s** – ponteiro para um *string* de caracteres multibytes que serão convertidos.
- **nMax** – número máximo de caracteres extensos armazenados em **strExtenso**.
- **pEstado** – ponteiro para o estado de mudança corrente.

**Retorno:**

- O número de caracteres multibytes convertidos, se não ocorrer nenhum erro durante a conversão.
- **(size\_t)-1**, se ocorrer algum erro durante a conversão.
- **mbrtowc(0, "", 1, pEstado)**, se **s** for **NULL**.

**Observações:**

- A função armazena no máximo **nMax** caracteres extensos em **strExtenso** através de chamadas sucessivas da função **mbrtowc()**. Se, numa destas chamadas a função **mbrtowc()** retornar zero, o caractere **L'\0'** será armazenado em **strExtenso**.
- Ela função é afetada pelo valor da localidade correntemente atribuída à categoria **LC\_CTYPE**.
- Consulte também **mbrtowc()**.



**Exemplo:**

```

#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <locale.h>
#include <string.h>

#define TAMANHO      100
#define LOCALIDADE_BR  "pt_BR.utf8"

int main(void)
{
    char          strMB1[] = "Cora\u00E7\u00E3o"; /* "Coração" */
    char          strMB2[] = "A\u00E7\u00E3o"; /* "Ação" */
    const char    *pmb1 = strMB1;
    const char    *pmb2 = strMB2;
    mbstate_t     estado1, estado2;
    wchar_t       strExtens1[TAMANHO],
                  strExtens2[TAMANHO];

    /* Inicia estados de conversão */
    memset(&estado1, 0, sizeof(estado1));
    memset(&estado2, 0, sizeof(estado2));

    if (!setlocale(LC_ALL, LOCALIDADE_BR)) {
        printf("Nao foi possivel alterar localidade.\n");
        return EXIT_FAILURE;
    }

    mbsrtowcs(strExtens1, &pmb1, TAMANHO, &estado1);
    mbsrtowcs(strExtens2, &pmb2, TAMANHO, &estado2);

    printf( "\nPrimeiro string extenso: \"%ls\"",
            strExtens1 );
    printf( "\nSegundo string extenso: \"%ls\"\n",
            strExtens2);

    return EXIT_SUCCESS;
}

```

*wctomb()***Incluir:** <wchar.h>**Descrição:** A função **wctomb()** converte um caractere extenso num caractere multibyte e armazena o resultado num array.**Protótipo:**

```
size_t wctomb( char *restrict ar, wchar_t carExtenso,
               mbstate_t *restrict pEstado )
```

**Parâmetros:**

- *ar* – array onde os bytes resultantes da conversão serão armazenados. Se este parâmetro for **NULL**, a função retornará:

```
wctomb(arLocal, L'\0', pEstado)
```

onde *arLocal* é um array de duração fixa local à função.

- *carExtenso* – caractere extenso a ser convertido.
- *pEstado* – ponteiro para o estado de conversão ou **NULL**.

**Retorno:**

- O número de bytes armazenados em *ar*, se a conversão for bem sucedida.
- $(\text{size\_t}) - 1$ , se *carExtenso* representa um caractere extenso inválido.

**Observações:**

- Esta função é uma versão reiniciável da função **wctomb()** (v. **Seção 8.4.2**).
- O número de bytes armazenados no array *ar* não pode ser maior do que **MB\_CUR\_MAX**.
- Se *pEstado* for **NULL**, o estado de mudança usado na conversão será uma variável estática do tipo **mbstate\_t** local à função e iniciada com o estado inicial de conversão.
- Se um programa que chama esta função nunca usa codificações com estado, ele pode passar **NULL** como terceiro parâmetro.
- Se o segundo argumento for o caractere extenso nulo (*L'\0'*), a função ar-

mazenará no array, se necessário, uma sequência de bytes correspondente ao estado inicial de mudança.

- A variável apontada por `pEstado` pode ser alterada para refletir o estado corrente de conversão.
- Se `carExtenso` representa um caractere extenso inválido, a função armazena o valor **EILSEQ** na variável global **errno** e o estado de conversão resultante torna-se indefinido.

### Exemplo:

```
#include <stdio.h>
#include <wchar.h>
#include <locale.h>
#include <stdlib.h>

#define PORTUGUES_BRASIL "pt_BR.utf8" /* Linux */

int main(void)
{
    size_t      n, i;
    mbstate_t   estado;
    char        carMB[MB_CUR_MAX];
    wchar_t     carExtenso = L'\u00c7'; /* 'Ç' */

    if (!setlocale(LC_ALL, PORTUGUES_BRASIL)) {
        printf("Nao foi possivel alterar a localidade\n");
        return 1;
    }

    /* Inicia estado de conversão */
    memset(&estado, 0, sizeof(estado));

    /* O uso da variável 'estado' neste caso */
    /* não é necessário. Pode-se chamar a */
    /* wctomb() simplesmente assim: */
    /* n = wctomb(carMB, carExtenso, NULL); */

    n = wctomb(carMB, carExtenso, &estado);

    if (n != (size_t)-1) {
```

```

    printf("\nCaractere extenso: '%lc'", carExtenso);
    printf( "\nBytes no caractere multibyte "
            "convertido: " );
    for (i = 0; i < n; ++i)
        printf("%#x ", (unsigned char) *carMB++);
    putchar('\n');
} else
    printf("O caractere extenso e' invalido.\n");

return 0;
}

```

Resultado da execução o programa no Linux:

*Caractere extenso: 'Ç'*

*Bytes no caractere multibyte convertido: 0xc3 0x87*

*wcsrtombs()*

**Incluir:** <wchar.h>

**Descrição:** A função **wcsrtombs()** converte uma sequência de caracteres extensos numa sequência correspondente de caracteres multibytes.

**Protótipo:**

```

size_t wcsrtombs( char *restrict arrayMB,
                  const wchar_t **restrict carExtensos,
                  size_t nMax, mbstate_t *restrict pEstado )

```

**Parâmetros:**

- **arrayMB** – array onde os bytes resultantes da conversão serão escritos ou **NULL**.
- **carExtensos** – ponteiro para o array contendo os caracteres extensos que serão convertidos.
- **nMax** – número máximo de bytes que serão armazenados em **arrayMB**.
- **pEstado** – ponteiro para o estado corrente de mudança.

**Retorno:** O número de bytes resultantes da conversão, sem incluir o caractere nulo, ou `(size_t) - 1`, se um caractere extenso inválido for encontrado.

### Observações:

- Se o parâmetro `pEstado` for **NULL**, o estado de mudança usado na conversão será uma variável local de duração fixa do tipo **mbstate\_t** iniciada com o estado inicial de mudança.
- A conversão de cada caractere é igual àquela efetuada pela função **wcrtomb()**.
- Quando um caractere multibyte nulo é armazenado em `arrayMB`, a função armazena **NULL** em `*carExtensos`.
- O *string* armazenado no primeiro argumento é precedido por uma sequência de bytes que conduzem ao estado inicial de conversão.
- Quando a escrita iminente de outro caractere multibyte excede o valor `nMax`, esta função armazena em `*carExtensos` o endereço do próximo caractere extenso que seria convertido.
- Quando um caractere extenso não pode ser convertido num caractere multibyte válido, esta função armazena na variável **errno** o valor **EILSEQ** e o estado de mudança apontado pelo argumento `pEstado` torna-se indefinido.

### Exemplo:

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

#define MAX_BYTES      50
#define PORTUGUES_BRASIL "pt_BR.utf8"

int main(void)
{
    char        strMB[MAX_BYTES];
    wchar_t     *strExt = L"Eu \u2665 JP";
    wchar_t     *p;
    size_t      nBytes;
```

```

if (!setlocale(LC_ALL, PORTUGUES_BRASIL)) {
    printf("Nao foi possivel alterar a localidade\n");
    return 1;
}

p = strExt; /* Guarda início do string extenso */

nBytes = wcsrtombs( strMB, (const wchar_t **)&p,
                    MAX_BYTES, NULL );

printf("\nString extenso original: \"%ls\"", strExt);
printf("\n%d bytes foram convertidos", nBytes);
printf("\nO string convertido e': \"%s\"\n", strMB);

printf("\n-----\n");

    /* Reinicia o array strMB */
memset(strMB, '\0', sizeof(strMB));

    /* O valor de p foi alterado na */
    /* chamada da função wcsrtombs() */
p = strExt;

    /* Converte apenas 5 caracteres */
nBytes = wcsrtombs( strMB, (const wchar_t **)&p,
                    5, NULL);

printf("\nString extenso original: \"%ls\"", strExt);
printf("\n%d bytes foram convertidos", nBytes);
printf("\nO string convertido e': \"%s\"\n", strMB);

return 0;
}

```

Resultado de uma execução do último programa no Linux:

```

String extenso original: "Eu ♥ JP"
9 bytes foram convertidos
O string convertido e': "Eu ♥ JP"

```

-----

```
String extenso original: "Eu ♥ JP"
3 bytes foram convertidos
O string convertido e': "Eu "
```

Considerando a última chamada de **wcsrtombs()** no exemplo anterior, note que, quando esta função encontra o caractere '♥', ela já terá armazenado três caracteres no array `strMB`, cada um dos quais ocupa um byte<sup>93</sup>. Como o caractere '♥' ocupa três bytes no esquema de codificação UTF-8, armazená-lo no array causaria um excesso no número de caracteres armazenados no array, que foi especificado como 5 na referida chamada da função. Portanto, a função **wcsrtombs()** armazena no array `strMB` apenas os três primeiros caracteres resultantes da conversão do *string* extenso original. Logo, o array `strMB` não conteria um *string* se não fosse o fato de seus elementos terem sido iniciados com zero antes da chamada da função.

*wctob()*

**Incluir:** <wchar.h>

**Descrição:** A função **wctob()** converterá um caractere extenso em monobyte se esta conversão for possível.

**Protótipo:**

```
int wctob(wint_t cExtenso)
```

**Parâmetro:** `cExtenso` – um caractere extenso convertido em **wint\_t**.

**Retorno:** O caractere extenso convertido num caractere multibyte representado num único byte no estado de mudança inicial, se a conversão for possível; **EOF**, caso contrário.

**Observação:** Quando a conversão é possível, o resultado é expandido num valor do tipo **int** antes de ser retornado.

---

<sup>93</sup> O único caractere do string "Eu ♥ JP" que não faz parte do repertório ASCII é '♥' e, conforme foi visto na **Seção 7.6.1**, qualquer caractere ASCII é representado num único byte no esquema de codificação UTF-8.

**Exemplo:**

```

#include <stdio.h>
#include <wchar.h>
#include <locale.h>

#define PORTUGUES_BRASIL "pt_BR.utf8"

int main(void)
{
    wint_t cExtensol = L'U',
           cExtenso2 = L'\u00E3'; /* 'ã' */

    if (!setlocale(LC_ALL, PORTUGUES_BRASIL)) {
        printf("Nao foi possivel alterar a localidade\n");
        return 1;
    }

    printf( "'%lc' %spode ser representado por "
            "um caractere multibyte de um byte\n",
            cExtensol,
            wctob(cExtensol) != EOF ? "" : "NAO " );

    printf( "'%lc' %spode ser representado por "
            "um caractere multibyte de um byte\n",
            cExtenso2,
            wctob(cExtenso2) != EOF ? "" : "NAO " );

    return 0;
}

```

Resultado da execução do último programa no sistema Linux:

```

'U' pode ser representado por um caractere multibyte de um byte
'ã' NAO pode ser representado por um caractere multibyte de um byte

```



## 8.5.4 FUNÇÕES DE PROCESSAMENTO DE ARRAYS DE CARACTERES EXTENSOS

As funções apresentadas nesta seção atuam sobre arrays de caracteres extensos e não necessariamente sobre *strings* extensos. Isto é, as funções descritas nesta seção não contam com a ocorrência do caractere `␣'\0'` para encerrar o processamento de um array. Em vez disso, essas funções incluem um parâmetro do tipo **size\_t** que limita o número de bytes processados em cada bloco. Cada função apresentada nesta seção possui uma função análoga que atua sobre arrays de bytes (i.e., blocos) descrita na Seção 6.3.3.

*wmemchr()*

**Incluir:** `<wchar.h>`

**Descrição:** A função **wmemchr()** procura a primeira ocorrência de um caractere extenso num array de caracteres extensos limitando o número de caracteres extensos examinados.

**Protótipo:**

```
wchar_t *wmemchr( const wchar_t *arExt,
                  wchar_t cExtenso, size_t n )
```

**Parâmetros:**

- `arExt` – array de caracteres extensos onde será feita a busca.
- `cExtenso` – caractere extenso a ser procurado.
- `n` – número máximo de caracteres extensos que serão procurados.

**Retorno:** Ponteiro para o local onde se encontra a primeira ocorrência do caractere ou **NULL** se ele não for encontrado.

**Observação:** Consulte também **memchr()** (Seção 6.3.3).

**Exemplo:**

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

int main()
{
    wchar_t *s = L"Apenas um string extenso";

    printf( "O que resta de \"%ls\" a partir do "
           "caractere 'r' e' \"%ls\"", s,
           wmemchr(s, 'r', 16) );

    return 0;
}
```

**wmemcmp()****Incluir:** <wchar.h>

**Descrição:** A função **wmemcmp()** compara um número limitado de caracteres extensos de dois arrays .

**Protótipo:**

```
int wmemcmp( const wchar_t *bloco1,
             const wchar_t *bloco2, size_t n )
```

**Parâmetros:**

- `bloco1` – primeiro array de caracteres extensos.
- `bloco2` – segundo array de caracteres extensos.
- `n` – número de caracteres extensos que serão comparados.

**Retorno:**

- Zero, se os blocos são iguais.
- Um valor menor do que zero, se `bloco1` é menor do que `bloco2`
- Um valor maior do que zero, se `bloco1` é maior do que `bloco2`.

**Observação:** Consulte também **memcmp()** (Seção 6.3.3).

**Exemplo:**

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

int main()
{
    wchar_t s1[] = L"ABCDEFGF",
            s2[] = L"ABCDXYZ";

    printf(" s1 = %ls\n s2 = %ls\n wmemcmp(s1, s2, 4) = "
           "%d\n wmemcmp(s1, s2, 7) = %d\n", s1, s2,
           wmemcmp(s1, s2, 4), wmemcmp(s1, s2, 7));

    return 0;
}
```

Resultado do programa quando executado no Windows XP:

```
s1 = ABCDEFG
s2 = ABCDXYZ
wmemcmp(s1, s2, 4) = 0
wmemcmp(s1, s2, 7) = -19
```

*wmemcpy()*

**Incluir:** <wchar.h>

**Descrição:** A função **wmemcpy()** copia um número limitado de caracteres extensos de um array para outro.

**Protótipo:**

```
wchar_t *wmemcpy( wchar_t *restrict destino,
                  const wchar_t *restrict origem,
                  size_t n )
```

**Parâmetros:**

- `destino` – array que recebe a cópia.
- `origem` – array de onde os caracteres extensos são copiados.
- `n` – número de caracteres extensos que serão copiados.

**Retorno:** O primeiro argumento.

**Observações:**

- e os arrays de origem e de destino se sobrepõem, o resultado pode ser indefinido.
- Consulte também `memcpy()` (Seção 6.3.3).

**Exemplo:**

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

int main()
{
    wchar_t s1[80],
            s2[] = L"Apenas um string extenso";

    wmemcpy(s1, s2, 20);

    printf( "\ns2 = %ls \ns1 apos a chamada"
            " wmemcpy(s1, s2, 20) = %ls\n", s2, s1 );

    return 0;
}
```

*wmemmove()*

**Incluir:** <wchar.h>

**Descrição:** A função **wmemmove()** copia um número limitado de caracteres extensos de um array para outro, permitindo a sobreposição dos arrays.

**Protótipo:**

```
wchar_t *wmemmove( wchar_t *destino, const wchar_t *origem,
                  size_t n )
```

**Parâmetros:**

- destino – array que receberá os caracteres extensos copiados.
- origem – array contendo os caracteres extensos que serão copiados.
- n – número de caracteres extensos que serão copiados.

**Retorno:** O primeiro argumento.

**Observações:**

- A diferença entre **wmemmove()** e **wmemcpy()** é que **wmemmove()** permite sobreposição dos arrays.
- Consulte também **memmove()** (Seção 6.3.3) e **wmemcpy()**.

**Exemplo:**

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

int main()
{
    wchar_t s[] = L"Apenas um string extenso";

    printf( "String ANTES da chamada de wmemmove "
           "= %ls\n", s);
    printf( "\nString DEPOIS da chamada de wmemmove "
           "= %ls\n", wmemmove(s, s + 5, 10 ) );

    return 0;
}
```

*wmemset()*

**Incluir:** <wchar.h>

**Descrição:** A função **wmemset()** atribui um valor aos primeiros elementos de um array de caracteres extensos.

**Protótipo:**

```
wchar_t *wmemset(wchar_t *arExt, wchar_t ce, size_t n)
```

**Parâmetros:**

- `arExt` – array cujos elementos terão valores atribuídos.
- `ce` – valor que será atribuído aos `n` primeiros elementos do array `arExt`.
- `n` – número de elementos iniciais do array `arExt` que terão valores atribuídos.

**Retorno:** O primeiro argumento.

**Observação:** Consulte também **memset()** (Seção 6.3.3).

**Exemplo:**

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

int main(void)
{
    wchar_t array[] = L"Isto e' um teste.";

    printf( "Array antes de chamar wmemset(): %ls\n",
           array );

    wmemset(array, L'x', wcslen(array) - 1);

    printf( "Array depois de chamar wmemset(): %ls\n",
           array );

    return 0;
}
```

### 8.5.5 FUNÇÕES DE PROCESSAMENTO DE STRINGS EXTENSOS

Todas as funções apresentadas nesta seção possuem equivalentes para *strings* monobytes. Estas funções são apresentadas nas Seções 5.3.3 [função `strftime()`] e 6.3 [demais funções]. As principais diferenças entre as funções apresentadas aqui e suas correspondentes para *strings* monobytes dizem respeito aos tipos de caracteres com os quais elas lidam. Em outras palavras, a maioria das funções apresentadas na presente seção funciona da mesma maneira que as respectivas funções correspondentes. Por causa disso, as funções são descritas de modo mais sucinto na presente seção. Assim, em caso de dúvida no entendimento do funcionamento das funções apresentadas aqui, sugere-se que o leitor consulte a descrição mais completa apresentada nas referidas seções que descrevem funções equivalentes para *strings* monobytes. Encontrar a denominação da função equivalente a cada função apresentada aqui é simples: basta trocar o prefixo *wcs* que compõe o nome da função por *str*.

*wcscat()*

**Incluir:** <wchar.h>

**Descrição:** A função `wcscat()` concatena uma cópia de um *string* extenso ao final de outro *string* extenso.

**Protótipo:**

```
wchar_t *wcscat( wchar_t *restrict destino,
                  const wchar_t *restrict origem )
```

**Parâmetros:**

- *destino* – array contendo o *string* extenso que será acrescido de *origem*.
- *origem* – *string* extenso que será acrescentado ao *string* armazenado em *destino*.

**Retorno:** O primeiro argumento.

**Observações:**

- O comprimento do *string* extenso resultante será:

```
wcslen(destino) + wcslen(origem)
```

- O *string* extenso recebido como parâmetro em *destino* pode ter comprimento zero. Neste caso, **wscat()** funciona como **wscpy()**.
- Certifique-se de que o array que contém o *string* extenso *destino* tenha espaço suficiente para conter o resultado da operação.
- Consulte também **strcat()** (Seção 6.3.4).

**Exemplo:**

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

int main()
{
    wchar_t destino[80] = L"Isto e' ";
    wchar_t *origem = L"apenas um teste.";

    printf("String original: \n\t \"%ls\"\n", destino);
    printf( "\nString que contem os caracteres a ser"
           " concatenados: \n\t \"%ls\"\n", origem);

    wscat(destino, origem);

    printf( "\nString original apos a concatenacao "
           "usando wscat(destino, origem): \n\t "
           "\"%ls\"\n", destino );

    return 0;
}
```

**wcschr()****Incluir:** <wchar.h>

**Descrição:** A função **wcschr()** procura a primeira ocorrência de um caractere extenso num *string* extenso.

**Protótipo:**

```
wchar_t *wcschr(const wchar_t *strExt, wchar_t cExt)
```



**Parâmetros:**

- `strExt` – *string* extenso a ser pesquisado.
- `cExt` – caractere extenso a ser procurado.

**Retorno:** Um ponteiro para a primeira ocorrência de `cExt` em `strExt`, se o caractere extenso for encontrado, ou **NULL**, caso contrário.

**Observações:**

- A função considera o caractere extenso terminal de *string* extenso (`L'\0'`) como parte do *string* extenso.
- Consulte também `strchr()` (Seção 6.3.4).

**Exemplo:**

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

#define PORTUGUES_BRASIL "pt_BR.utf8"

int main()
{
    wchar_t *strExt = L"Cora\u00E7\u00E3o";
    wchar_t cExt = L'a';
    if (!setlocale(LC_ALL, PORTUGUES_BRASIL)) {
        printf("Nao foi possivel alterar a localidade\n");
        return 1;
    }

    printf( "O restante de \"%ls\" começando com a "
           "primeira ocorrencia de '%lc' é \"%ls\"\n",
           strExt, cExt, wcschr(strExt, cExt) );

    return 0;
}
```

Resultado do programa quando ele é executado no Linux:

O restante de "Coração" começando com a primeira ocorrência de 'a' é "ação"

*wscmp()*

**Incluir:** <wchar.h>

**Descrição:** A função **wscmp()** compara dois *strings* extensos e retorna um valor indicando se eles são iguais ou se um deles é menor do que o outro.

**Protótipo:**

```
int wscmp(const wchar_t *str1, const wchar_t *str2)
```

**Parâmetros:** *str1* e *str2* – *strings* extensos que serão comparados.

**Retorno:**

- Zero, se os *strings* extensos são iguais.
- Menor do que zero, se *str1* é menor do que *str2*.
- Maior do que zero, se *str1* é maior do que *str2*.

**Observações:**

- A comparação começa com o primeiro caractere extenso de cada *string* extenso e continua até que caracteres extensos correspondentes difiram, ou até que o final de um dos *strings* extensos seja atingido.
- Na maioria das situações, a função **wscmp()**, assim como **strcmp()**, deve ser usada apenas para testar se dois *strings* são iguais. Isto é, raramente, estas funções são usadas para ordenar *strings* [a não ser quando estas funções são usadas em conjunto com **wcsxfrm()** e **strxfrm()**, respectivamente].
- Consulte também **strcmp()** (Seção 6.3.4).

**Exemplo:**

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

```

int main(void)
{
    wchar_t strExt1[] = L"Isto e' um teste";
    wchar_t strExt2[] = L"Isto nao e' um teste";
    int      comparacao;

    comparacao = wcscmp(strExt1, strExt2);

    if (comparacao < 0)
        printf( "\"%ls\" e' menor do \"%ls\"\\n",
                strExt1, strExt2 );
    else if (comparacao > 0)
        printf( "\"%ls\" e' maior do \"%ls\"\\n",
                strExt1, strExt2 );
    else
        printf("Os strings sao iguais\\n");

    return 0;
}

```

## *wscoll()*

**Incluir:** <wchar.h>

**Descrição:** A função **wscoll()** compara dois *strings* extensos usando informações sobre colação da localidade corrente e retorna um valor indicando se eles são iguais ou se um deles é menor do que o outro.

**Protótipo:**

```
int wscoll(const wchar_t *str1, const wchar_t *str2)
```

**Parâmetros:** *str1* e *str2* – *strings* extensos que serão comparados.

**Retorno:**

- Zero, se os *strings* extensos são iguais.
- Menor do que zero, se *str1* é menor do que *str2*.

- Maior do que zero, se `str1` é maior do que `str2`.

### Observações:

- A comparação começa com o primeiro caractere extenso de cada *string* e continua até que caracteres extensos correspondentes difiram, ou até que o final de um dos *strings* extensos seja atingido.
- A diferença básica entre esta função e **wscmp()** é que a função apresentada aqui é capaz de aplicar regras de colação específicas de localidade à comparação de *strings*. Assim, **wscoll()** é usada em ordenação (ou, mais precisamente, colação) de *strings* (v. **Seção 6.4** e **Seção 7.7**).
- Esta função depende do valor corrente de **LC\_COLLATE**.
- Esta função é equivalente a **wscmp()** quando a localidade "C" é atribuída à categoria **LC\_COLLATE**.
- Consulte também **strcoll()** (**Seção 6.4.2**).

**Exemplo:** O programa a seguir ordena um array de *strings* com a função **qsort()** (v. **Seção 12.2.6**) usando **wscmp()** e **wscoll()**.

```
#include <wchar.h>
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>

#define PORTUGUES_BRASIL "pt_BR.utf8"
#define TAM_ARRAY(ar) sizeof(ar)/sizeof(ar[0])

/****
 *
 * Função Compara_wscmp(): função de colação baseada em
 *                          wscmp() que será usada com
 *                          qsort()
 *
 * Argumentos: a, b (entrada) - elementos que serão
 *                          comparados
 *
 * Retorno: semelhante ao retorno de wscmp()
 *
```

```

****/
int Compara_wcscmp(const void *a, const void *b)
{
    return wcscmp(*(wchar_t **)a, *(wchar_t **)b);
}

/****
*
* Função Compara_wcs coll(): função de colação baseada
*                               em wcs coll() que será usada
*                               com qsort()
*
* Argumentos: a, b (entrada) - elementos que serão
*                               comparados
*
* Retorno: semelhante ao retorno de wcs coll()
*
****/
int Compara_wcs coll(const void *a, const void *b)
{
    return wcs coll(*(wchar_t **)a, *(wchar_t **)b);
}

int main(void)
{
    char            *localidade;
    size_t          i, nElementos;
    const wchar_t *garotas[] = {
                                                L"M\u00E9rcia",
                                                L"Marta",
                                                L"M\u00E9lrcia",
                                                L"Mirtes",
                                                L"Marcela",
                                                L"\u00C9rica",
                                                L"\u00C1lvara",
                                                };

    localidade = setlocale(LC_ALL, PORTUGUES_BRASIL);
    if (!localidade) {
        printf("Nao foi possivel alterar a localidade\n");
        return 1;
    }
}

```

```

    printf("\nLocalidade corrente: %s\n", localidade);

    nElementos = TAM_ARRAY(garotas);

    printf("\nLista original: \n\t");
    for (i = 0; i < nElementos; ++i)
        printf("%ls ", garotas[i]);

    qsort( (void *)garotas, nElementos,
           sizeof(garotas[0]), Compara_wcscmp );

    printf("\n\nLista ordenada usando wcscmp():\n\t");
    for (i = 0; i < nElementos; ++i)
        printf("%ls ", garotas[i]);

    qsort( (void *)garotas, nElementos,
           sizeof(garotas[0]), Compara_wscoll );

    printf("\n\nLista ordenada usando wscoll():\n\t");
    for (i = 0; i < nElementos; ++i)
        printf("%ls ", garotas[i]);

    putchar('\n');

    return 0;
}

```

Resultado da execução do programa no sistema Linux:

*Localidade corrente: pt\_BR.utf8*

*Lista original:*

*Mércia Marta Márcia Mirtes Marcela Érica Álvora*

*Lista ordenada usando wcscmp():*

*Marcela Marta Mirtes Márcia Mércia Álvora Érica*

*Lista ordenada usando wscoll():*

*Álvora Érica Marcela Márcia Marta Mércia Mirtes*

*wscpy()*

**Incluir:** <wchar.h>

**Descrição:** A função **wscpy()** faz uma cópia de um *string* extenso.

**Protótipo:**

```
wchar_t *wscpy( wchar_t *restrict destino,
                const wchar_t *restrict origem )
```

**Parâmetros:**

- destino – array de caracteres extensos que receberá a cópia.
- origem – *string* extenso que é copiado para destino.

**Retorno:** O primeiro argumento.

**Observações:**

- Por ser capaz de limitar o número de caracteres copiados, é preferível usar a função **wscspn()** a usar **wscpy()**.
- Consulte também **strcpy()** (Seção 6.3.4).

**Exemplo:**

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

int main()
{
    wchar_t destino[80];

    wscpy(destino, L"Um string extenso");
    printf("Resultado da copia: \"%ls\\\"\\n", destino);

    return 0;
}
```

*wscspn()*

**Incluir:** <wchar.h>

**Descrição:** A função **wscspn()** examina um *string* extenso até encontrar algum caractere extenso que faz parte de outro *string* extenso.

**Protótipo:**

```
size_t wscspn( const wchar_t *string1,
               const wchar_t *string2 )
```

**Parâmetros:**

- *string1* – *string* extenso a ser examinado.
- *string2* – *string* extenso contendo os caracteres extensos que serão procurados.

**Retorno:** O índice em *string1* do primeiro caractere extenso de *string2* encontrado em *string1*.

**Observações:**

- O caractere extenso terminal de *string* extenso é levado em consideração.
- Consulte também **strcspn()** (Seção 6.3.4).

**Exemplo:**

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

int main()
{
    wchar_t  *strExt1 = L"Isto e' um teste";
    wchar_t  *strExt2 = L"AHbe";
    size_t    intersecao;

    intersecao = wscspn(strExt1, strExt2);
```



```

printf( "Os strings \"%ls\" e \"%ls\" se "
        "interceptam no caractere '%lc'\n",
        strExt1, strExt2, strExt1[intersecao]);

return 0;
}

```

### *wcsftime()*

**Incluir:** <wchar.h>

**Descrição:** A função **wcsftime()** cria um *string* extenso contendo data e hora especificadas utilizando um *string* extenso de formatação.

#### **Protótipo:**

```

size_t wcsftime( wchar_t *restrict arExt, size_t max,
                  const wchar_t *restrict formato,
                  const struct tm *restrict t )

```

#### **Parâmetros:**

- **arExt** – array de caracteres extensos que receberá o *string* contendo a data e a hora formatadas.
- **max** – número máximo de caracteres extensos armazenados em **arExt**.
- **formato** – *string* extenso de formatação (v. **Apêndice C**).
- **t** – ponteiro para uma estrutura **tm** contendo informações sobre data e hora (v. **Seção 5.3.1**).

#### **Retorno:**

- O número de caracteres extensos armazenados em **arExt**, quando a função obtém êxito e este número é menor do que **max**.
- Zero, se ocorrer algum erro ou o número de caracteres extensos requeridos for maior do que **max**.

**Observações:**

- O *string* extenso de formatação utilizado corresponde à versão extensa daquele que seria utilizado numa chamada equivalente da função **strftime()** (v. **Seção 5.3.3**).
- Todos os caracteres extensos comuns encontrados no *string* extenso de formatação são copiados sem modificação.
- Data e hora são formatadas de acordo com a localidade correntemente atribuída à categoria **LC\_TIME**.
- Consulte a descrição da função **strftime()** e os especificadores de formato de data e hora apresentados no **Apêndice C**.

**Exemplo:**

```
#include <stdio.h>
#include <wchar.h>
#include <time.h>
#include <locale.h>

#define MAX 200
#define PT_BR_WIN_XP "Portuguese_Brazil.850"
#define PT_BR_LINUX  "pt_BR.utf8"

int main(void)
{
    struct tm *agora;
    time_t      segundos;
    wchar_t     strExt[MAX];
    char        *local = PT_BR_LINUX;

    if (!setlocale(LC_ALL, local)) {
        printf("Nao foi possivel alterar a localidade\n");
        return 1;
    }

    time(&segundos);
    agora = localtime(&segundos);

    wcsftime( strExt, MAX,
              L"Passam %M minutos das %I horas (%z) "
```

```

        "%A, %B %d 20%y", agora );

    printf("%ls\n", strExt);

    return 0;
}

```

Resultado da execução do último programa no Windows XP:

Passam 22 minutos das 03 horas (Hora oficial do Brasil) sexta-feira, janeiro 16 2009

Resultado da execução do último programa no Linux:

Passam 24 minutos das 03 horas (-0300) sexta, janeiro 16 2009

*wcslen()*

**Incluir:** <wchar.h>

**Descrição:** A função **wcslen()** calcula o comprimento de um *string* extenso.

**Protótipo:**

```
size_t wcslen(const wchar_t *strExt)
```

**Parâmetro:** *strExt* – *string* extenso cujo comprimento será calculado.

**Retorno:** O número de caracteres extensos em *strExt*.

**Observações:**

- O comprimento não inclui o caractere extenso `L'\0'`.
- Consulte também **strlen()** (Seção 6.3.4).

**Exemplo:**

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

```

#include <wchar.h>

int main(void)
{
    FILE      *stream;
    wchar_t    strExtenso[] = L"Isto e' um teste.";
    wchar_t    arExtenso[80];
    wchar_t    *retorno;

    /*Abre arquivo para leitura e escrita */
    if(!(stream = fopen("Arq1.txt", "w+"))) {
        fprintf(stderr, "Arquivo nao pode ser aberto");
        return 1;
    }

    /* Escreve string extenso no arquivo */
    fputws(strExtenso, stream);

    rewind(stream); /* Volta ao início do arquivo */

    /* Tenta ler string extenso no arquivo */
    retorno = fgetws( arExtenso, wcslen(strExtenso) + 1,
                      stream );

    if (!retorno) {
        fprintf( stderr,
                "Ocorreu um erro durante a leitura\n" );
        return 1;
    }

    wprintf(L"\nString lido no arquivo: %ls", arExtenso);
    wprintf( L"\nTamanho do string lido: %d\n",
             wcslen(arExtenso) );

    fclose(stream);

    return 0;
}

```

*wcsncat()*

**Incluir:** <wchar.h>

**Descrição:** A função **wcsncat()** concatena um número limitado de caracteres extensos de um *string* extenso ao final de outro *string* extenso.

**Protótipo:**

```
wchar_t *wcsncat( wchar_t *restrict destino,
                  const wchar_t *restrict origem,
                  size_t n )
```

**Parâmetros:**

- *destino* – array contendo o *string* extenso que terá caracteres extensos anexados.
- *origem* – *string* extenso que cederá os caracteres extensos que serão acrescentados a *destino*.
- *n* – número máximo de caracteres extensos do *string* extenso *origem* que serão acrescentados a *destino*.

**Retorno:** O primeiro argumento.

**Observações:**

- O comprimento do *string* extenso resultante será (no máximo):  
`wcslen(destino) + n`
- O *string* extenso armazenado em *destino* pode ser um *string* extenso de comprimento zero. Neste caso, a função **wcsncat()** é equivalente a **wcsncpy()**.
- Consulte também **strncat()** (Seção 6.3.4).

**Exemplo:**

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

int main()
{
    wchar_t destino[80] = L"Isto é";
    wchar_t *origem = L" um teste apenas.";

    printf("String original: \n\t \"%ls\"\n", destino);
```

```

printf( "\nString que contem os caracteres "
        "a ser concatenados: "
        "\n\t \"%ls\"\n", origem );

wcsncat(destino, origem, 9);

printf( "\nString original apos a concatenacao "
        "usando wcsncat(destino, origem, 9): "
        "\n\t \"%ls\"\n", destino );

return 0;
}

```

### *wcsncmp()*

**Incluir:** <wchar.h>

**Descrição:** A função **wcsncmp()** compara um número limitado de caracteres extensos de dois *strings* extensos.

**Protótipo:**

```

int wcsncmp( const wchar_t *string1,
             const wchar_t *string2, size_t n )

```

**Parâmetros:**

- *string1* – primeiro *string* extenso a ser comparado.
- *string2* – segundo *string* extenso a ser comparado.
- *n* – número máximo de caracteres extensos que serão comparados.

**Retorno:**

- Zero, se os *strings* extensos são iguais.
- Menor do que zero, se *string1* é menor do que *string2*.
- Maior do que zero, se *string1* é maior do que *string2*.

**Observações:**

- A comparação começa com o primeiro caractere extenso de cada *string* extenso e continua até que caracteres extensos correspondentes difiram, até que *n* caracteres extensos tenham sido examinados ou até que o final de um dos *strings* extensos seja atingido.
- Esta função é similar a **wscmp()**, mas a função descrita aqui limita a comparação a *n* caracteres extensos.
- Consulte também **strncmp()** (Seção 6.3.4).

**Exemplo:**

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

int main(void)
{
    wchar_t string1[] = L"Isto e' um teste";
    wchar_t string2[] = L"Isto e' um teste tambem";
    int      comparacao;

    comparacao = wcsncmp(string1, string2, 5);

    printf("Usando wcsncmp(string1, string2, 5):\n");

    if (comparacao < 0)
        printf( "%ls\" e' menor do \"%ls\"\n",
                string1, string2 );
    else if (comparacao > 0)
        printf( "%ls\" e' maior do \"%ls\"\n",
                string1, string2 );
    else
        printf("Os strings sao iguais\n");

    comparacao = wcsncmp(string1, string2, 18);

    printf("\nUsando wcsncmp(string1, string2, 18):\n");

    if (comparacao < 0)
        printf( "%ls\" e' menor do \"%ls\"\n",
```

```

        string1, string2 );
    else if (comparacao > 0)
        printf( "\"%ls\" e' maior do \"%ls\"\\n",
            string1, string2 );
    else
        printf("Os strings sao iguais\\n");

    return 0;
}

```

### *wcsncpy()*

**Incluir:** <wchar.h>

**Descrição:** A função **wcsncpy()** copia um número limitado de caracteres extensos de um *string* extenso.

#### **Protótipo:**

```

wchar_t *wcsncpy( wchar_t *restrict destino,
                  const wchar_t *restrict origem,
                  size_t n )

```

#### **Parâmetros:**

- *destino* – array de caracteres extensos que receberá os caracteres extensos copiados.
- *origem* – *string* extenso cujos caracteres extensos serão copiados.
- *n* – número máximo de caracteres extensos que serão copiados.

**Retorno:** O primeiro argumento.

#### **Observações:**

- Certifique-se de que o array *destino* é suficientemente grande para conter os caracteres extensos copiados de *origem*.
- Consulte também **strncpy()** (Seção 6.3.4).



**Exemplo:**

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

int main()
{
    wchar_t destino[80];

    wcsncpy(destino, L"Pequeno", 10);
    printf( "String destino apos chamada de "
           "wcsncpy(destino, L\"Pequeno\", 10):"
           "\n\t \"%ls\"\n", destino );

    wcsncpy(destino, L"Um string grande", 10);

    printf("String destino apos chamada de "
           "wcsncpy(destino, L\"Um string grande\", 10):"
           "\n\t \"%ls\"\n", destino);

    return 0;
}
```

***wcspbrk()*****Incluir:** <wchar.h>

**Descrição:** A função **wcspbrk()** procura num *string* extenso a primeira ocorrência de qualquer caractere extenso contidos noutra *string*.

**Protótipo:**

<pre>wchar_t *wcspbrk( const wchar_t *string1,                   const wchar_t *string2 )</pre>
---

**Parâmetros:**

- *string1* – *string* extenso no qual será efetuada a busca.
- *string2* – *string* extenso contendo os caracteres extensos que serão procurados em *string1*.

**Retorno:** Ponteiro para o primeiro caractere extenso de `string2` encontrado em `string1` ou **NULL**, se nenhum caractere extenso de `string2` for encontrado em `string1`.

### Observações:

- Compare esta função com **wcscspn()**.
- Consulte também **strpbrk()** (Seção 6.3.4).

### Exemplo:

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

int main()
{
    wchar_t *strExt1 = L"Apenas um string";
    wchar_t *strExt2 = L"Teste";
    wchar_t *pOcorrencia;

    pOcorrencia = wcspbrk(strExt1, strExt2);

    if (pOcorrencia)
        printf( "Dentre os caracteres em \"%ls\", \n'%lc'"
               " e' o primeiro caractere a aparecer em "
               "\n\"%ls\"\n", strExt2, *pOcorrencia,
               strExt1 );
    else
        printf("Nao existe caractere em \"%ls\" que "
               "apareca em \"%ls\"\n", strExt2, strExt1);

    return 0;
}
```

*wcsrchr()*

**Incluir:** <wchar.h>

**Descrição:** A função **wcsrchr()** procura a última ocorrência de um caractere extenso num *string* extenso.

**Protótipo:**

```
wchar_t *wcsrchr(const wchar_t *strExt, wchar_t cExt)
```

**Parâmetros:**

- `strExt` – *string* extenso a ser examinado.
- `cExt` – caractere extenso a ser procurado no *string*.

**Retorno:** Ponteiro para a última ocorrência de `cExt` encontrado em `strExt` ou `NULL`, se o caractere extenso não for encontrado.

**Observações:**

- A função considera o caractere extenso terminal `L'\0'` como parte do *string* extenso.
- Consulte também **strrchr()** (Seção 6.3.4).

**Exemplo:**

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

int main()
{
    wchar_t *str = L"Apenas um string extenso";
    wchar_t c = L's';

    printf( "O restante de \"%ls\" começando com a "
           "ultima \nocorrenzia do caractere '%lc'\n"
           "e' \"%ls\"", str, c, wcsrchr(str, c) );

    return 0;
}
```

*wcsspn()*

**Incluir:** <wchar.h>

**Descrição:** A função **wcsspn()** procura o segmento inicial de um *string* extenso que contenha todos os caracteres extensos de outro *string* extenso.

**Protótipo:**

```
size_t wcsspn( const wchar_t *string1,
               const wchar_t *string2 )
```

**Parâmetros:**

- *string1* – *string* extenso a ser examinado.
- *string2* – *string* extenso contendo caracteres extensos que serão procurados.

**Retorno:** O comprimento do segmento inicial contido em *string1* que contém todos os caracteres extensos de *string2*.

**Observações:**

- O caractere terminal de *string* extenso (L'\0') é considerado parte apenas do primeiro *string* (*string1*).
- Compare esta função com **wcsstr()** (a seguir).
- Consulte também **strspn()** (Seção 6.3.4).

**Exemplo:**

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

int main(void)
{
    wchar_t  *str1 = L"ABCDEFGFGHIJH";
    wchar_t  *str2 = L"DCFAB";
    size_t   posicao;

    posicao = wcsspn(str1, str2);
```

```

printf( "\nPrimeiro caractere de '%ls' que "
        "nao esta em '%ls': %c\n",
        str1, str2, str1[posicao] );

return 0;
}

```

### *wcsstr()*

**Incluir:** <wchar.h>

**Descrição:** A função **wcsstr()** procura a primeira ocorrência de um *string* extenso em outro *string* extenso.

### **Protótipo:**

```

wchar_t *wcsstr( const wchar_t *string1,
                  const wchar_t *string2 )

```

### **Parâmetros:**

- *string1* – *string* extenso que será examinado.
- *string2* – *string* extenso que será procurado em *string1*.

**Retorno:** Ponteiro para a primeira ocorrência de *string2* em *string1* ou **NULL**, se *string2* não ocorre em *string1*.

**Observação:** Consulte também **strstr()** (Seção 6.3.4).

### **Exemplo:**

```

#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *strExt1 = L"Apenas um string extenso";
    wchar_t *strExt2 = L"str";
    wchar_t *pResultado;

    pResultado = wcsstr(strExt1, strExt2);
}

```

```

    if (pResultado)
        printf( "A primeira ocorrencia de \"%ls\" em "
                "\"%ls\" comeca em: \n\t \"%ls\"\n",
                strExt2, strExt1, pResultado );
    else
        printf("O string \"%ls\" nao ocorre em \"%ls\"\n",
                strExt2, strExt1);

    return 0;
}

```

### *wcstok()*

**Incluir:** <wchar.h>

**Descrição:** A função **wcstok()** divide um *string* extenso em partes (*tokens*).

**Protótipo:**

```

wchar_t *wcstok( wchar_t *restrict strExt,
                  const wchar_t *restrict separadores,
                  wchar_t **restrict ptr )

```

**Parâmetros:**

- *strExt* – *string* extenso a ser dividido em partes.
- *separadores* – *string* extenso contendo os separadores das partes.
- *ptr* – ponteiro para a última parte encontrada.

**Retorno:** Um ponteiro para uma parte de *strExt* se esta for encontrada ou **NULL**, se nenhuma parte for encontrada.

**Observações:**

- Esta função funciona de modo semelhante à função **strtok()** (v. **Seção 6.3.4**), mas, além de a função apresentada aqui utilizar caracteres extensos, ela ainda utiliza um argumento a mais do que a função **strtok()**. O terceiro argumento (*ptr*) que aparece no protótipo da função **wcstok()** é utilizado para armazenar a porção do *string* que segue o último *token* encontrado. Por exemplo, se o

*string* a ser dividido em *tokens* é `L"Um dois tres"` e espaço em branco é usado como separador de *tokens*, após a primeira chamada da função `wcstok()`, `ptr` estará apontando para a porção do *string* original que começa com *d*. Assim, quando esta função é chamada tendo `NULL` como primeiro argumento, a busca pelo próximo *token* começa na posição armazenada em `ptr`. A função `strtok()`, apresentada na **Seção 6.3.4**, também guarda a posição no *string* a partir de onde iniciará a busca pelo próximo *token*, mas faz isto usando uma variável local de duração fixa. Esta diferença faz com que a função `wcstok()` possa processar dois strings ao mesmo tempo, enquanto a função `strtok()` só pode processar um segundo string depois que termina de processar o primeiro.

- Esta função modifica o *string* extenso passado como argumento. Portanto, se necessário, faça uma cópia do mesmo antes de chamar a função.
- O *string* extenso separadores pode ser diferente em duas chamadas sucessivas.
- Em algumas versões da biblioteca GNU, utilizada pelo compilador gcc, a função `wcstok()` é implementada com os dois primeiros argumentos apenas, mas versões mais recentes desta biblioteca (e.g., libc6 2.8) implementam corretamente esta função.

### Exemplo:

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

int main()
{
    wchar_t string[] = L"Este e' um string com 7 tokens.";
    wchar_t separadores[] = L" .";
    wchar_t *token, *p;
    int i = 0;

    printf( "O string a ser separado em tokens e': "
           "\n\t\"%ls\"\n", string) ;

    printf("\nOs tokens sao:\n\n");

    token = wcstok(string, separadores, &p);
```

```

while (token) {
    printf("\tToken %d: \"%ls\"\n", ++i, token);
    token = wcstok(NULL, separadores, &p);
}

printf( "\n\nString original alterado por wcstok():"
        "\n\t\"%ls\"\n", string );

return 0;
}

```

Resultado de uma execução do último programa no Linux:

*O string a ser separado em tokens é:*

*"Este é um string com 7 tokens."*

*Os tokens são:*

```

Token 1: "Este"
Token 2: "é "
Token 3: "um"
Token 4: "string"
Token 5: "com"
Token 6: "7"
Token 7: "tokens"

```

*String original alterado por wcstok():*

*"Este"*

*wcsxfrm()*

**Incluir:** <wchar.h>

**Descrição:** A função **wcsxfrm()** cria uma chave de ordenação para um *string* extenso usando a ordem de colação estabelecida na categoria **LC\_COLLATE** e limitando o número de bytes na chave de ordenação.



**Protótipo:**

```
size_t wcsxfrm( wchar_t *restrict destino,
                const wchar_t *restrict origem,
                size_t n )
```

**Parâmetros:**

- *destino* – array de caracteres extensos que armazenará a chave de ordenação.
- *origem* – *string* extenso a ser transcrito.
- *n* – número máximo de caracteres extensos que serão armazenados em *destino*.

**Retorno:** O número de caracteres extensos armazenados no array *destino* (sem incluir o caractere extenso terminal `L'\0'`).

**Observações:**

- Quando esta função é bem sucedida, a chave de ordenação criada é um *string* de caracteres extensos que pode ser comparado com outra chave de ordenação usando-se `wscmp()`, obtendo-se o mesmo efeito que seria obtido se apenas a função `wscoll()` fosse utilizada com os strings originais.
- Usar `wcsxfrm()` e `wscmp()` pode ser mais eficiente do que usar `wscoll()` se for necessário comparar os mesmos *strings* mais de uma vez.
- Se o valor retornado for maior do que *n*, o conteúdo do array *destino* será indeterminado.
- O valor de *n* pode ser zero e, neste caso, *destino* pode ser **NULL**. O valor retornado, quando estes valores são usados como parâmetros, serve para dimensionar dinamicamente um array a ser usado como primeiro parâmetro numa chamada subsequente.
- Esta função é idêntica a `wcsncpy()` quando a localidade `"C"` é atribuída à categoria `LC_COLLATE`.
- Consulte também `strxfrm()` (Seção 6.3.4).

**Exemplo:**

```

#include <stdio.h>
#include <wchar.h>
#include <locale.h>

/* Windows XP */
#define PORTUGUES_BRASIL "Portuguese_Brazil.850"

int main()
{
    wchar_t    *origem1 = L"M\u00E9rcia";
    wchar_t    *origem2 = L"Mirtes";
    wchar_t    destino1[40];
    wchar_t    destino2[40];
    int        comparacao;

    /***** Usando localidade padrão "C" *****/

    if (!setlocale(LC_ALL, "C")) {
        printf("Nao foi possivel alterar a localidade\n");
        return 1;
    }

    printf("\nUsando a localidade padrao \"C\":\n\t");

    comparacao = wcscmp(origem1, origem2);

    if (!setlocale(LC_ALL, PORTUGUES_BRASIL)) {
        printf("Nao foi possivel alterar a localidade\n");
        return 1;
    }

    printf( "O string \"%ls\" %s o string \"%ls\"\n",
            origem1,
            (comparacao <= 0) ? "precede" : "sucede",
            origem2 );

    /***** Usando localidade português do Brasil *****/

    printf( "\nUsando a localidade \"%s\":\n\t",
            PORTUGUES_BRASIL );

```

```

wcsxfrm(destino1, origem1, wcslen(origem1));
wcsxfrm(destino2, origem2, wcslen(origem2));

comparacao = wcscmp(destino1, destino2);

printf( "O string \"%ls\" %s o string \"%ls\"\\n\\n",
        origem1,
        (comparacao <= 0) ? "precede" : "sucede",
        origem2 );

return 0;
}

```

Resultado do programa quando executado no Windows XP:

Usando a localidade padrao "C":

O string "Márcia" sucede o string "Mirtes"

Usando a localidade "Portuguese\_Brazil.850":

O string "Márcia" precede o string "Mirtes"

## 8.5.6 FUNÇÕES DE CONVERSÃO DE STRINGS EXTENSOS EM NÚMEROS

As funções de conversão de *strings* extensos em números são divididas em dois grupos: (1) funções que convertem *strings* extensos em números inteiros e (2) funções que convertem *strings* extensos em números de ponto flutuante. A única diferença entre as funções de um mesmo grupo é o tipo do número resultante da conversão. Portanto, estas funções são discutidas em grupo e não separadamente, como a maioria das demais funções da biblioteca padrão de C apresentadas no texto.

Todas as funções de conversão de *strings* extensos em números apresentadas nesta seção possuem equivalentes para *strings* monobytes declaradas em `<stdlib.h>` e apresentadas na **Seção 6.5**. As descrições apresentadas aqui são mais sucintas do que as mostradas na referida seção. Assim, caso surjam dúvidas no entendimento de alguma função apresentada aqui, sugere-se que o leitor consulte a discussão mais completa da função correspondente para *strings* monobytes apresentada na **Seção 6.5**.

*wcstol()*, *wcstoll()* (C99), *wcstoul()*, *wcstoull()* (C99)

**Incluir:** <wchar.h>

**Descrição:** As funções **wcstol()**, **wcstoll()**, **wcstoul()** e **wcstoull()** convertem *strings* extensos em inteiros dos tipos **long**, **long long**, **unsigned long** e **unsigned long long**, respectivamente.

**Protótipos:**

```
long wcstol( const wchar_t *restrict strExt,
             wchar_t **restrict final, int base )
```

```
long long wcstoll( const wchar_t *restrict strExt,
                   wchar_t **restrict final, int base )
```

```
unsigned long wcstoul( const wchar_t *restrict strExt,
                      wchar_t **restrict final, int base )
```

```
unsigned long long wcstoull( const wchar_t *restrict strExt,
                             wchar_t **restrict final,
                             int base )
```

**Parâmetros:**

- *strExt* – *string* extenso a ser convertido.
- *final* – se este parâmetro não for **NULL**, ao final da conversão, ele estará apontando para o primeiro caractere extenso do *string* que não foi convertido.
- *base* – base do número a ser convertido.

**Retorno:**

- O inteiro convertido, se não ocorrer erro de conversão e o valor convertido couber no tipo de retorno da função.
- Zero, se ocorrer erro de conversão.

- Se a magnitude do valor resultante da conversão for grande demais para ser representado no tipo de retorno da função<sup>94</sup>:
  - Mais ou menos **LONG\_MAX**, se a função chamada for **wcstol()**.
  - Mais ou menos **LLONG\_MAX**, se a função chamada for **wcstoll()**.
  - **ULONG\_MAX**, se a função chamada for **wcstoul()**.
  - **ULLONG\_MAX**, se a função chamada for **wcstoull()**.

#### Observações:

- A conversão prossegue até que o primeiro caractere inválido seja encontrado (incluindo `L'\0'`).
- O parâmetro `base` especifica qual base é usada e deve ser zero ou estar entre 2 e 36. Se ele for igual a zero, a base do número será determinada pelo seu formato. Isto é, se o número começar com `0x` ou `0X`, ele será considerado hexadecimal; se ele começar com zero, a base octal será assumida; em outros casos, a base será considerada decimal.
- Qualquer valor inválido usado como base faz com que o retorno seja zero e o parâmetro `final` aponte para o início de `strExt`, o que permite checar se uma conversão foi bem sucedida.
- Se a magnitude do valor resultante da conversão for grande demais para ser representada no tipo de retorno da função, ela atribuirá **ERANGE** à variável **errno** (v. **Seção 11.5**).
- Consulte também **strtol()**, **strtoll()**, **strtoul()** e **strtoull()** (**Seção 6.5.1**).

**Exemplo:** O programa a seguir demonstra o uso da função **wcstol()**.

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

int main()
{
    long    li;
    wchar_t *string = L"-1234567abc",
```

---

<sup>94</sup> Essas macros são definidas no cabeçalho `<limits.h>` (v. **Seção 2.3**).

```

        *resto;

    li = wcstol(string, &resto, 0);

    printf("String original: \"%ls\"\n", string);
    printf( "\nValor convertido para long int: %ld\n",
        li );
    printf( "\nResto do string original que nao foi "
        "convertido: \"%ls\"\n", resto );

    return 0;
}

```

**Exemplo:** O programa a seguir demonstra o uso da função `wcstoul()`.

```

#include <stdio.h>
#include <wchar.h>

int main()
{
    unsigned long    ul;
    wchar_t          *string = L"1234567abc",
                    *resto;

    ul = wcstoul(string, &resto, 0);

    printf("String original: \"%ls\"\n", string);
    printf( "\nValor convertido para long int: %ld\n",
        ul );
    printf( "\nResto do string original que nao foi "
        "convertido: \"%ls\"\n", resto );

    return 0;
}

```

*wcstod()*, *wcstof()* (C99), *wcstold()*

**Incluir:** `<wchar.h>`

**Descrição:** As funções `wcstod()`, `wcstof()` e `wcstold()` convertem *strings* extensos em números de ponto flutuante dos tipos **double**, **float** e **long double**, respectivamente.

**Protótipos:**

```
double wcstod( const wchar_t *restrict strExt,
               wchar_t **restrict final )
```

```
float wcstof( const wchar_t *restrict strExt,
              wchar_t **restrict final )
```

```
long double wcstold( const wchar_t *restrict strExt,
                     wchar_t **restrict final )
```

**Parâmetros:**

- `strExt` – *string* extenso a ser convertido.
- `final` – se este parâmetro não for **NULL**, ao final da conversão ele estará apontando para o primeiro caractere do *string* que não foi convertido.

**Retorno:**

- O número de ponto flutuante convertido, se a conversão for possível e representável no tipo de retorno da função.
- Zero, se nenhuma conversão for possível.
- Se o número resultante da conversão for excessivamente grande para ser representado no tipo de retorno da função<sup>95</sup>:
  - **HUGE\_VAL**, se a função chamada for **wcstod()**.
  - **HUGE\_VALF**, se a função chamada for **wcstof()**
  - **HUGE\_VALL**, se a função chamada for **wcstold()**.
- Se o número resultante da conversão for excessivamente pequeno para ser representado no tipo de retorno da função, ela retornará um valor cuja magnitude é o menor valor maior do que zero que possa ser representado no tipo de retorno da função.

---

<sup>95</sup> Essas macros são definidas no cabeçalho `<math.h>` (v. **Seção 3.6.2**).

**Observações:**

- Se houver espaços em branco no início do *string*, eles serão saltados. A verificação de espaços em branco é feita de acordo com a função **isspace()** (v. **Seção 8.6.3**).
- O número contido no *string* pode incluir:
  - Sinal
  - Ponto
  - e ou E (notação científica)
  - +INF ou -INF, representando, respectivamente,  $+\infty$  ou  $-\infty$
  - +NAN ou -NAN (representando um valor não numérico)
  - 0x ou 0X mais uma sequência de dígitos hexadecimais (C99)
- Quando ocorre *overflow*, essas funções atribuem **ERANGE** à variável **errno** (v. **Seção 11.5**).
- Consulte também **strtod()**, **strtof()** e **strtold()** (**Seção 6.5.2**).

**Exemplo:** O programa a seguir demonstra o uso da função **wcstod()**.

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

int main()
{
    double    d;
    wchar_t   *str = L"51.2 + alguma coisa",
              *final;

    d = wcstod(str, &final );

    printf( "O string \"%ls\" foi convertido"
           " em %f\n", str, d);
    printf( "O string \"%ls\" nao fez parte "
           "da conversao\n", final );
```



```
    return 0;
}
```

## 8.6 CLASSIFICAÇÃO E TRANSFORMAÇÃO DE CARACTERES EXTENSOS: <wctype.h>

No cabeçalho <wctype.h> são declaradas funções utilizadas para classificar e transformar caracteres extensos. Excetuando-se as funções **wctype()**, **towctrans()** e **wctrans()**, para cada função apresentada aqui, existe uma função semelhante que atua sobre caracteres monobytes declarada no cabeçalho <ctype.h> (v. **Seção 6.2.1**). As funções que possuem correspondentes para caracteres monobytes declaradas em <ctype.h> serão apresentadas aqui com brevidade e, em caso de dúvidas sobre a descrição de uma dada função apresentada aqui, consulte a discussão mais detalhada da função correspondente apresentada na **Seção 6.2.1**. O nome da função correspondente a cada função apresentada aqui é obtida removendo-se *w* do nome da função utilizada com caracteres extensos. Por exemplo, a função **iswalnum()** declarada em <wctype.h> corresponde à função **isalnum()** declarada em <ctype.h>.

As funções declaradas no cabeçalho <wctype.h> usam informações de localidade que constam na categoria **LC\_CTYPE** e o funcionamento destas funções depende do código de caracteres utilizado.

### 8.6.1 TIPOS

*wctrans\_t*

**Incluir:** <wctype.h>

**Descrição:** O tipo **wctrans\_t** representa regras de conversões entre caracteres extensos específicas de uma localidade. Um valor deste tipo é obtido por meio de uma chamada da função **wctrans()**.

*wctype\_t*

**Incluir:** <wchar.h> ou <wctype.h>

**Descrição:** O tipo **wctype\_t** representa categorias de caracteres extensos específicas de localidade. Um valor deste tipo é obtido por meio de uma chamada da função **wctype()**. Este tipo também é definido em `<wchar.h>` (v. **Seção 8.5.1**).

*wint\_t*

**Incluir:** `<wchar.h>` ou `<wctype.h>`

**Descrição:** O tipo **wint\_t** é um inteiro usado como tipo de retorno e de parâmetros para representar valores do tipo **wchar\_t**. Este tipo também é definido em `<wchar.h>` (v. **Seção 8.5.1**).

## 8.6.2 MACRO WEOF

**Incluir:** `<wchar.h>` ou `<wctype.h>`

**Descrição:** A macro **WEOF**, utilizada com caracteres extensos, é equivalente a **EOF**, utilizada com caracteres simples. Esta macro é usada para indicar o final de um *stream* de caracteres extensos ou uma condição de erro. Esta macro também é definida em `<wchar.h>` (v. **Seção 8.5.2**).

## 8.6.3 FUNÇÕES DE CLASSIFICAÇÃO DE CARACTERES EXTENSOS

Assim como as funções correspondentes declaradas em `<ctype.h>` (v. **Seção 6.2**), a maioria das funções de classificação de caracteres extensos tem o mesmo formato<sup>96</sup>:

```
int nome(wint_t carExtenso)
```

Onde:

- *nome* é o nome da função e sempre começa com *isw* e termina com um pretenso mnemônico que denota o nome de uma propriedade de classificação de caracteres [e.g., **iswalph()**].

---

<sup>96</sup> As únicas exceções são as funções **iswctype()** e **wctype()** que serão apresentadas mais adiante.

- `carExtenso` é o parâmetro único da função e representa um caractere extenso.

A tarefa de cada uma dessas funções é verificar se o caractere extenso recebido como parâmetro satisfaz uma dada propriedade (i.e., faz parte de uma determinada categoria). Por exemplo, **`iswalpha()`** verifica se o caractere extenso recebido como argumento é uma letra.

Todas essas funções possuem especificações de retorno semelhantes. Isto é, todas elas retornam um valor diferente de zero se a propriedade sendo testada é satisfeita ou zero, caso contrário. Por exemplo, a função **`iswalpha()`** retorna um valor diferente de zero se o caractere extenso testado for uma letra ou zero se este não for o caso.

A **Tabela 8-4** apresentada a seguir completa a descrição dessas funções.

FUNÇÃO	TESTA SE O CARACTERE EXTENSO RECEBIDO COMO ARGUMENTO É...
<b><code>iswalnum()</code></b>	Alfanumérico (i.e., letra ou dígito)
<b><code>iswalpha()</code></b>	Letra
<b><code>iswblank()</code></b> (C99)	Espaços em branco que separam palavras numa mesma linha
<b><code>iswcntrl()</code></b>	Caractere de controle
<b><code>iswdigit()</code></b>	Dígito (0–9)
<b><code>iswgraph()</code></b>	Imprimível (exceto espaço em branco)
<b><code>iswlower()</code></b>	Letra minúscula
<b><code>iswprint()</code></b>	Imprimível (incluindo espaços em branco)
<b><code>iswpunct()</code></b>	Símbolo de pontuação
<b><code>iswspace()</code></b>	Espaço em branco (qualquer)
<b><code>iswupper()</code></b>	Letra maiúscula
<b><code>iswxdigit()</code></b>	Dígito hexadecimal (a–f, A–F, 0–9)

Tabela 8-4: Funções de classificação de caracteres extensos.

Qualquer função apresentada na **Tabela 8-4** pode ser implementada como macro e a maioria delas é afetada pelo valor corrente de **`LC_CTYPE`**.

Deve-se notar que a diferença entre as funções **`iswblank()`** e **`iswspace()`** é que a primeira função considera apenas espaços em branco utilizados para separar palavras numa mesma linha de texto, enquanto a segunda função considera qualquer espaço em branco, incluindo caracteres de quebra de linha. Assim, qualquer caractere que satisfaça **`iswblank()`** também satisfaz **`iswspace()`**, mas a recíproca não é verdadeira.

**Exemplo:** O programa a seguir demonstra o uso da função **iswblank()**.

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

int main(void)
{
    /* Uso de iswblank() */
    printf("O caractere L'\\n' %s espaco em branco\\n",
           iswblank(L'\\n') ? "e'" : "nao e'");
    printf("O caractere L'$' %s espaco em branco\\n",
           iswblank(L'$') ? "e'" : "nao e'");
    printf("O caractere L'\\t' %s espaco em branco\\n",
           iswblank(L'\\t') ? "e'" : "nao e'");
    printf("O caractere L' ' %s espaco em branco\\n",
           iswblank(L' ') ? "e'" : "nao e'");

    return 0;
}
```

Resultado de uma execução do último programa no Linux:

```
O caractere L'\\n' nao e' espaco em branco
O caractere L'$' nao e' espaco em branco
O caractere L'\\t' e' espaco em branco
O caractere L' ' e' espaco em branco
```

**Exemplo:** Compare o resultado do último programa que demonstra o uso da função **iswblank()** com o programa a seguir que demonstra o uso de **iswspace()**.

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

int main(void)
{
    /* Uso de iswspace() */
    printf("O caractere L'\\n' %s espaco em branco\\n",
           iswspace(L'\\n') ? "e'" : "nao e'");
    printf("O caractere L'$' %s espaco em branco\\n",
           iswspace(L'$') ? "e'" : "nao e'");
```

```

printf("O caractere L'\\t' %s espaco em branco\\n",
       iswspace(L'\\t') ? "e'" : "nao e'");
printf("O caractere L' ' %s espaco em branco\\n",
       iswspace(L' ') ? "e'" : "nao e'");

return 0;
}

```

Resultado de uma execução do último programa no sistema Linux:

```

O caractere L'\\n' e' espaco em branco
O caractere L'$' nao e' espaco em branco
O caractere L'\\t' e' espaco em branco
O caractere L' ' e' espaco em branco

```

**Exemplo:** O programa a seguir demonstra o uso das funções **iswcntrl()**, **iswpunct()**, **iswprint()** e **iswgraph()**.

```

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

int main(void)
{
    /* Uso de iswcntrl() */
    printf("\\nO caractere L'\\n' %s caractere de controle\\n",
           iswcntrl(L'\\n') ? "e'" : "nao e'");
    printf("O caractere L'$' %s caractere de controle\\n",
           iswcntrl(L'$') ? "e'" : "nao e'");
    printf("O caractere L'#' %s caractere de controle\\n",
           iswcntrl(L'#') ? "e'" : "nao e'");

    /* Uso de iswpunct() */
    printf("\\nO caractere L'? ' %s caractere de pontuacao\\n",
           iswpunct(L'?') ? "e'" : "nao e'");
    printf("O caractere L'B' %s caractere de pontuacao\\n",
           iswpunct(L'B') ? "e'" : "nao e'");
    printf("O caractere L': ' %s caractere de pontuacao\\n",
           iswpunct(L':') ? "e'" : "nao e'");

    /* Uso de iswprint() */
}

```

```

printf("\nO caractere L'\n' %s caractere imprimivel\n",
       iswprint(L'\n') ? "e" : "nao e");
printf("O caractere L'$' %s caractere imprimivel\n",
       iswprint(L'$') ? "e" : "nao e");
printf("O caractere L'\a' %s caractere imprimivel\n",
       iswprint(L'\a') ? "e" : "nao e");

/* Uso de iswgraph() */
printf("\nO caractere L'\n' %s caractere grafico\n",
       iswgraph(L'\n') ? "e" : "nao e");
printf("O caractere L'$' %s caractere grafico\n",
       iswgraph(L'$') ? "e" : "nao e");
printf("O caractere L'\a' %s caractere grafico\n",
       iswgraph(L'\a') ? "e" : "nao e");

return 0;
}

```

### Resultado de execução do programa no Windows XP:

```

O caractere L'\n' e' caractere de controle
O caractere L'$' nao e' caractere de controle
O caractere L'#' nao e' caractere de controle

O caractere L'?' e' caractere de pontuacao
O caractere L'B' nao e' caractere de pontuacao
O caractere L':' e' caractere de pontuacao

O caractere L'\n' nao e' caractere imprimivel
O caractere L'$' e' caractere imprimivel
O caractere L'\a' nao e' caractere imprimivel

O caractere L'\n' nao e' caractere grafico
O caractere L'$' e' caractere grafico
O caractere L'\a' nao e' caractere grafico

```

**Exemplo:** O programa a seguir demonstra o uso das funções **iswdigit()**, **iswalnum()** e **iswxdigit()**.

```

#include <stdio.h>
#include <wctype.h>

```

```

int main(void)
{
    /* Uso de iswdigit() */
    printf("O caractere L'5' %s digito\n",
           iswdigit(L'5') ? "e'" : "nao e'");
    printf("O caractere L'$' %s digito\n",
           iswdigit(L'$') ? "e'" : "nao e'");

    /* Uso de iswalpha() */
    printf("\nO caractere L'z' %s letra\n",
           iswalpha(L'z') ? "e'" : "nao e'");
    printf("O caractere L'3' %s letra\n",
           iswalpha(L'3') ? "e'" : "nao e'");

    /* Uso de iswalnum() */
    printf("\nO caractere L'z' %s letra ou digito\n",
           iswalnum(L'z') ? "e'" : "nao e'");
    printf("O caractere L'3' %s letra ou digito\n",
           iswalnum(L'3') ? "e'" : "nao e'");
    printf("O caractere L'@' %s letra ou digito\n",
           iswalnum(L'@') ? "e'" : "nao e'");

    /* Uso de iswxdigit() */
    printf("\nO caractere L'z' %s digito hexadecimal\n",
           iswxdigit(L'z') ? "e'" : "nao e'");
    printf("O caractere L'3' %s digito hexadecimal\n",
           iswxdigit(L'3') ? "e'" : "nao e'");
    printf("O caractere L'C' %s digito hexadecimal\n",
           iswxdigit(L'C') ? "e'" : "nao e'");

    return 0;
}

```

**Resultado de execução do último programa no Windows XP:**

```

O caractere L'5' e' digito
O caractere L'$' nao e' digito

O caractere L'z' e' letra
O caractere L'3' nao e' letra

```

```

O caractere L'z' e' letra ou digito
O caractere L'3' e' letra ou digito
O caractere L'@' nao e' letra ou digito

O caractere L'z' nao e' digito hexadecimal
O caractere L'3' e' digito hexadecimal
O caractere L'C' e' digito hexadecimal

```

**Exemplo:** O programa a seguir mostra a influência da localidade atribuída à categoria **LC\_CTYPE** em funções declaradas no cabeçalho `<wctype.h>`.

```

#include <stdio.h>
#include <wctype.h>
#include <wchar.h>
#include <locale.h>

#define PORTUGUES_BRASIL "pt_BR.utf8" /* Linux */

/****
 *
 * Função EhLetraLocal(): verifica se um caractere
 *                       extenso é uma letra na
 *                       localidade especificada e
 *                       imprime o resultado do teste
 *
 * Argumentos: cExt (entrada) - o caractere que será
 *                       testado
 *             localidade (entrada) - a localidade
 *
 * Retorno: Nada
 *
 ****/

void EhLetraLocal(wchar_t cExt, const char *localidade)
{
    char *local;

    local = setlocale(LC_CTYPE, localidade);
    if (!local) {
        printf("\nNao foi possivel alterar a localidade");
        return;
    }
}

```



```

printf("\nA localidade corrente e': \"%s\"", local);
printf( "\nNesta localidade, o caractere "
        "%s letra\n",
        iswalph(cExt) ? "e'" : "nao e'" );
}

int main(void)
{
    wchar_t c = L'\u00E7'; /* O caractere é 'ç' */

    /* Usa a localidade do sistema */
    EhLetraLocal(c, "");

    /* Usa a localidade português do Brasil */
    EhLetraLocal(c, PORTUGUES_BRASIL);

    /* Usa a localidade padrão */
    EhLetraLocal(c, "C");

    return 0;
}

```

Resultado de uma execução do último programa no Linux:

*A localidade corrente e': "en\_US.UTF-8"*  
*Nesta localidade, o caractere e' letra*

*A localidade corrente e': "pt\_BR.utf8"*  
*Nesta localidade, o caractere e' letra*

*A localidade corrente e': "C"*  
*Nesta localidade, o caractere nao e' letra*

*iswctype()*

**Incluir:** <wctype.h>

**Descrição:** A função **iswctype()** testa se um dado caractere extenso pertence a uma dada categoria de caracteres extensos.

**Protótipo:**

```
int iswctype(wint_t ce, wctype_t categoria)
```

**Parâmetros:**

- `ce` – o caractere extenso que se deseja testar.
- `categoria` – uma categoria de caracteres extensos, cujo valor deve ser obtido por meio de uma chamada de **wctype()**.

**Retorno:** Um valor diferente de zero se o caractere extenso fizer parte da categoria especificada; zero, caso contrário.

**Observações:**

- Se o argumento `ce` não corresponder a **WEOF** ou a um caractere extenso válido, o comportamento da função será indefinido.
- Se o valor do argumento `categoria` não for um valor retornado por **wctype()** ou este valor tiver sido invalidado por uma mudança de localidade, o comportamento da função será indefinido.
- Esta função é necessária em localidades que definem outras categorias de caracteres além daquelas definidas na localidade "C". Caso contrário, ela não é necessária, pois suas chamadas podem ser substituídas por chamadas de outras funções declaradas em `<wctype.h>`, como mostra a **Tabela 8-5** a seguir. De acordo com o padrão ISO C99, as chamadas de funções apresentadas em cada linha da **Tabela 8-5** são equivalentes em qualquer localidade.

PROPRIEDADE	CHAMADA DE iswctype()	CHAMADA EQUIVALENTE
"alnum"	iswctype(ce, wctype("alnum"))	isalnum(ce)
"alpha"	iswctype(ce, wctype("alpha"))	isalpha(ce)
"blank"	iswctype(ce, wctype("blank"))	isblank(ce)
"cntrl"	iswctype(ce, wctype("cntrl"))	iscntrl(ce)
"digit"	iswctype(ce, wctype("digit"))	isdigit(ce)
"graph"	iswctype(ce, wctype("graph"))	isgraph(ce)
"lower"	iswctype(ce, wctype("lower"))	islower(ce)
"print"	iswctype(ce, wctype("print"))	isprint(ce)
"punct"	iswctype(ce, wctype("punct"))	ispunct(ce)

"space"	<code>iswctype(ce, wctype("space"))</code>	<code>isspace(ce)</code>
"upper"	<code>iswctype(ce, wctype("upper"))</code>	<code>isupper(ce)</code>
"xdigit"	<code>iswctype(ce, wctype("xdigit"))</code>	

Tabela 8-5: Chamadas de funções equivalentes a chamadas de `iswctype()`.

**Exemplo:** Veja exemplo de `wctype()`.

`wctype()`

**Incluir:** `<wctype.h>`

**Descrição:** A função `wctype()` retorna um valor que representa uma regra de classificação de caracteres extensos que pode ser utilizada com a função `iswctype()`.

**Protótipo:**

`wctrans_t wctype(const char *propriedade)`

**Parâmetro:** `propriedade` – *string* que descreve uma propriedade de caracteres extensos de acordo com a localidade atribuída à categoria **LC\_CTYPE** e de acordo com a qual caracteres extensos podem ser categorizados.

**Retorno:** Um valor diferente de zero, se a localidade correntemente atribuída a **LC\_CTYPE** definir uma regra de classificação cujo nome coincida com o parâmetro `propriedade`; zero, caso contrário.

**Observações:**

- Quando esta função retorna um valor diferente de zero, ele pode ser usado como segundo argumento da função `iswctype()`.
- Esta função constrói um valor do tipo **wctrans\_t** que representa uma categoria de caracteres extensos identificada pelo único argumento.
- Os valores retornados por esta função são válidos até que a categoria de localidade **LC\_CTYPE** seja alterada [e.g., por meio de uma chamada da função `setlocale()`].

- Os *strings* apresentados na **Tabela 8-6** representam categorias de caracteres extensos requeridas pelo padrão ISO C99, mas uma dada implementação de C pode incluir outras categorias. Esses *strings* são válidos em qualquer localidade como argumentos para a função **wctype()**.

STRING	CATEGORIA QUE REPRESENTA
"alnum"	Caracteres alfanuméricos
"alpha"	Letras
"cntrl"	Caracteres de controle
"digit"	Dígitos
"graph"	Caracteres gráficos
"lower"	Letras minúsculas
"print"	Caracteres imprimíveis
"punct"	Caracteres de pontuação
"space"	Espaços em branco
"upper"	Letras maiúsculas
"xdigit"	Dígitos hexadecimais

Tabela 8-6: Strings representando categorias de caracteres extensos.

**Exemplo:** O programa a seguir demonstra o uso das funções **wctype()** e **iswctype()**.

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

int main()
{
    printf( "O caractere L'z' %s letra\n",
            iswctype(L'z', wctype("alpha")) ?
            "e'" : "nao e'" );

    printf( "O caractere L'$' %s digito\n",
            iswctype(L'$', wctype("digit")) ?
            "e'" : "nao e'");

    printf( "O caractere L'z' %s letra ou digito\n",
            iswctype(L'z', wctype("alnum")) ?
```

```

        "e'" : "nao e'" );

    return 0;
}

```

## 8.6.4 FUNÇÕES DE TRANSFORMAÇÃO DE CARACTERES EXTENSOS

*towctrans()*

**Incluir:** <wctype.h>

**Descrição:** A função **towctrans()** retorna o caractere extenso recebido como parâmetro convertido de acordo com uma transformação especificada.

**Protótipo:**

```
wint_t towctrans(wint_t ce, wctrans_t trans)
```

**Parâmetros:**

- *ce* – o caractere extenso a ser transformado.
- *trans* – a transformação pela qual o caractere *ce* passará; este valor deverá ter sido retornado pela função **wctrans()** (v. adiante).

**Retorno:** O caractere extenso *ce* transformado e alargado em **wint\_t**.

**Observação:** Quando esta função for chamada, a localidade correntemente atribuída à categoria **LC\_CTYPE** deverá ser a mesma usada durante a chamada da função **wctrans()** que retornou o valor do argumento *trans*.

**Exemplo:** Veja o exemplo da função **wctrans()**.

*towlower()*

**Incluir:** <wctype.h>

**Descrição:** A função **towlower()** converte uma letra maiúscula em minúscula.

**Protótipo:**

```
wint_t tolower(wint_t ce)
```

**Parâmetro:** *ce* – caractere extenso a ser convertido.

**Retorno:** O caractere extenso convertido em letra minúscula, se ele for uma letra maiúscula; o próprio caractere extenso sem modificação, caso contrário.

**Observação:** Consulte também **tolower()** (Seção 6.2.2).

**Exemplo:** Veja o exemplo da função **towupper()**

*towupper()*

**Incluir:** <wctype.h>

**Descrição:** A função **towupper()** converte uma letra minúscula em maiúscula.

**Protótipo:**

```
wint_t towupper(wint_t ce)
```

**Parâmetro:** *ce* – caractere extenso a ser convertido.

**Retorno:** O caractere extenso convertido em letra maiúscula, se ele for uma letra minúscula; o próprio caractere extenso sem modificação, caso contrário.

**Observação:** Consulte também **toupper()** (Seção 6.2.2).

**Exemplo:** O programa a seguir demonstra o uso das funções **iswlower()**, **iswupper()**, **tolower()** e **towupper()**.

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

int main(void)
{
    /* Uso de iswlower() */
```

```

printf("\nO caractere L'u' %s minuscula",
       islower(L'u') ? "e'" : "nao e'");
printf("\nO caractere L'L' %s minuscula",
       islower(L'L') ? "e'" : "nao e'");
printf("\nO caractere L'$' %s minuscula",
       islower(L'$') ? "e'" : "nao e'");

/* Uso de iswupper() */
printf("\n\nO caractere L'u' %s maiuscula",
       iswupper(L'u') ? "e'" : "nao e'");
printf("\nO caractere L'L' %s maiuscula",
       iswupper(L'L') ? "e'" : "nao e'");
printf("\nO caractere L'$' %s maiuscula",
       iswupper(L'$') ? "e'" : "nao e'");

/* Uso de towlower() */
printf("\n\nO caractere L'u' convertido em minuscula"
       " e' %c", towlower(L'u'));
printf( "\nO caractere L'L' convertido em minuscula"
       " e' %c", towlower(L'L') );
printf( "\nO caractere L'$' convertido em minuscula"
       " e' %c\n", towlower(L'$') );

/* Uso de towupper() */
printf( "\n\nO caractere L'u' convertido em maiuscula"
       " e' %c", towupper(L'u'));
printf( "\nO caractere L'L' convertido em maiuscula"
       " e' %c", towupper(L'L'));
printf( "\nO caractere L'$' convertido em maiuscula"
       " e' %c\n", towupper(L'$'));

return 0;
}

```

*wctrans()*

**Incluir:** <wctype.h>

**Descrição:** A função **wctrans()** retorna um valor que representa uma transformação de caracteres extensos.

**Protótipo:**

```
wctrans_t wctrans(const char *nome)
```

**Parâmetro:** nome – *string* que descreve a transformação a ser retornada pela função.

**Retorno:** Um valor diferente de zero, se a localidade correntemente atribuída a **LC\_CTYPE** define uma transformação entre caracteres extensos cujo nome coincida com o argumento nome; zero, caso contrário.

**Observações:**

- Quando o valor retornado por esta função é diferente de zero, ele pode ser usado como segundo argumento numa chamada da função **towctrans()**.
- Os nomes de transformações "tolower" e "toupper" são válidos em qualquer localidade. Assim, uma chamada:

```
towctrans(c, wctrans("tolower"));
```

é sempre equivalente a:

```
towlower(c);
```

e uma chamada:

```
towctrans(c, wctrans("toupper"));
```

é sempre equivalente a:

```
toupper(c);
```

- Na localidade "C", os únicos nomes de transformações válidos são "tolower" e "toupper".
- O valor retornado por esta função permanece válido até que a categoria **LC\_CTYPE** seja modificada [e.g., por meio de uma chamada de **setlocale()**].

**Exemplo:** O programa a seguir demonstra o uso das funções **wctrans()** e **towctrans()**.

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



```

int main(void)
{
    char      *alfabeto = "abcdefghijklmnopqrstuvxz";
    wchar_t   *strExtenso;
    int       i;
    size_t    tamAlfabeto;

    tamAlfabeto = strlen(alfabeto);

    strExtenso = malloc(sizeof(wchar_t)*(tamAlfabeto+1));

    mbstowcs(strExtenso, alfabeto, 2*(tamAlfabeto + 1));

    printf("Alfabeto minusculo: %ls\n", strExtenso);

    for (i = 0; i < tamAlfabeto; ++i)
        strExtenso[i] = (wchar_t)towctrans(strExtenso[i],
                                           wctrans("toupper"));

    printf("Alfabeto maiusculo: %ls\n", strExtenso);

    return 0;
}

```

## 8.7 EXERCÍCIOS DE REVISÃO

1. Defina os seguintes conceitos:
  - (a) Caractere extenso
  - (b) Caractere multibyte
  - (c) *String* multibyte
  - (d) Ponto de código
  - (e) Esquema de codificação de caracteres
  - (f) Codificação de caracteres
  - (g) Estado de mudança
  - (h) Variável de estado

- (i) Mudança de estado
  - (j) Estado inicial
  - (k) Estado de conversão
2. Qual é o significado dos seguintes componentes no nome de uma função declarada em `<wchar.h>`?
    - (a) *mb*
    - (b) *mbs*
    - (c) *wc*
    - (d) *wcs*
    - (e) *wmem*
  3. (a) Como caracteres multibytes são implementados em C? (b) Existe algum tipo especial para armazenamento de caracteres multibytes?
  4. Como a macro predefinida **\_\_STDC\_ISO\_10646\_\_** é interpretada?
  5. (a) Em que cabeçalho é definida a macro **MB\_LEN\_MAX**? (b) Idem para a macro **MB\_CUR\_MAX**. (c) Como cada uma destas macros é expandida?
  6. O que deve ser armazenado numa variável do tipo **mbstate\_t**?
  7. Que funções declaradas no cabeçalho `<wchar.h>` são afetadas pela categoria de localidade **LC\_COLLATE**?
  8. Que funções de processamento de *strings* declaradas em `<string.h>` podem ser usadas com *strings* multibytes?
  9. Suponha que uma operação comum precise ser executada sobre *strings* multibytes. Suponha ainda que esta operação seja implementada em funções declaradas nos cabeçalhos `<string.h>` e `<wchar.h>`, mas elas não sejam convenientes para uso com *strings* multibytes. Quais são as possíveis soluções?
  10. Como uma variável do tipo **mbstate\_t** pode ser iniciada?
  11. Por que se deve evitar o processamento de dois ou mais *strings* multibytes simultaneamente por uma mesma função de conversão de *strings* multibytes declarada em `<stdlib.h>`? Este problema aflige funções correspondentes

declaradas em `<wchar.h>`?

12. Descreva o funcionamento das seguintes funções declaradas em `<stdlib.h>`:

- (a) **mblen()**
- (b) **mbstowcs()**
- (c) **mbtowc()**
- (d) **wcstombs()**
- (e) **wctomb()**

13. (a) Quais são as funções declaradas em `<wchar.h>` equivalentes às seguintes funções declaradas em `<stdlib.h>`? (b) O que as funções declaradas em `<wchar.h>` têm a mais que suas respectivas equivalentes declaradas em `<stdlib.h>`?

- (a) **mblen()**
- (b) **mbstowcs()**
- (c) **mbtowc()**
- (d) **wcstombs()**
- (e) **wctomb()**

14. Para que serve a função **mbstowcs()**?

15. Descreva o funcionamento das seguintes funções:

- (a) **btowc()**
- (b) **wctob()**

16. Quais são as diferenças entre as funções **wcstok()**, declarada em `<wchar.h>`, e **strtok()**, declarada em `<string.h>`?

17. Que funções declaradas em `<wchar.h>` são usadas em colação de *strings* extensos?

18. Para que servem os tipos **wctrans\_t** e **wctype\_t**?

19. Em que situações práticas devem ser usadas as funções **towctrans()** e **wctrans()**?

# *Capítulo 9*

---

*Funções com listas de argumentos va-  
riáveis*

## 9.1 INTRODUÇÃO

Existem funções cujos argumentos não são completamente especificados a priori. Ou seja, o número e os tipos de alguns argumentos de tal função podem variar entre uma chamada e outra da mesma. Tais argumentos são denominados **argumentos variáveis**<sup>97</sup>. O cabeçalho `<stdio.h>` (v. **Capítulo 10**) é repleto de declarações de funções desta natureza, como, por exemplo, `scanf()` e `printf()`.

Em definições e alusões de funções com argumentos variáveis, estes argumentos são representados por três pontos seguidos. Por exemplo, o protótipo da função `printf()` apresenta-se como:

```
int printf(const char *formato, ...);
```

Existem ainda outras funções que aparentam ter argumentos fixos, mas, na realidade, utilizam, indiretamente, argumentos variáveis. Tais funções possuem, pelo menos, um argumento do tipo `va_list`. Novamente, várias destas funções são declaradas no cabeçalho `<stdio.h>`. Um exemplo desta última categoria de funções é a função `vprintf()` (v. **Capítulo 10**), cujo protótipo é:

```
int vprintf(const char *formato, va_list outrosArgs);
```

Como pode ser observado, o protótipo da função `vprintf()` contém exatamente dois argumentos e os tipos destes argumentos são bem definidos. Mas, acontece que o tipo `va_list`, utilizado no segundo argumento, serve exatamente para definir listas de argumentos variáveis. Isto significa que, essencialmente, os protótipos das funções `printf()` e `vprintf()` descrevem funções semelhantes (mas, não equivalentes). De fato, a diferença entre estas funções refere-se à maneira como cada uma é usada (detalhes sobre esta diferença serão apresentados no **Capítulo 10**).

O tipo `va_list` e as quatro macros necessárias para o processamento de argumentos variáveis são definidos no cabeçalho `<stdarg.h>`. Para definir as funções da primeira categoria descrita algumas linhas atrás e para poder chamar as funções da segunda categoria, é necessário usar este tipo e, pelo menos, três dessas macros.

O objetivo deste capítulo é descrever os componentes do cabeçalho `<stdarg.h>` e apresentar exemplos de definições de funções que usam, direta ou indiretamente, argumentos variáveis. Os componentes deste cabeçalho são apresentados em ordem de provável utilização, em vez de em ordem alfabética, como ocorre com a maioria dos componentes apresentados no presente texto.

---

97 Aqui, o termo *variável* denota o fato de o número e os tipos dos argumentos que recebem esta denominação não serem especificados na definição ou em alusões de uma função.

## 9.2 SUPORTE PARA LISTAS DE ARGUMENTOS VARIÁVEIS: <stdarg.h>

O cabeçalho `<stdarg.h>` serve para dar suporte para definições de funções de listas variáveis de argumentos. Este cabeçalho apresenta um tipo e quatro macros definidos resumidamente na **Tabela 9-1**.

COMPONENTE	O QUE É	O QUE FAZ
<b>va_list</b>	Tipo	Define listas de argumentos variáveis.
<b>va_arg</b>	Macro	Resulta no próximo argumento de uma lista de argumentos variáveis.
<b>va_copy</b>	Macro	Copia o conteúdo de uma variável do tipo <b>va_list</b> para outra.
<b>va_end</b>	Macro	Encerra o processamento de uma lista de argumentos variáveis.
<b>va_start()</b>	Macro	Inicia o processamento de uma lista de argumentos variáveis.

Tabela 9-1: Componentes definidos em `<stdarg.h>`.

### 9.2.1 TIPO `va_list`

**Incluir:** `<stdarg.h>`

**Descrição:** O tipo `va_list` serve para definir listas de argumentos variáveis e é tipicamente usado em duas situações:

- **Como tipo de uma variável local.** Neste caso, a função [e.g., `printf()`] que define tal variável recebe *diretamente* como parâmetro uma lista de argumentos variáveis. A variável local é utilizada no processamento desses argumentos.
- **Como tipo de um argumento de função.** Neste caso, a função [e.g., `vprintf()`] usa *indiretamente* argumentos variáveis.

**Exemplo:** Existem muitos exemplos de uso deste tipo ao longo deste livro, notadamente na **Seção 9.4** e no **Capítulo 10**.

## 9.2.2 MACROS

*va\_start()*

**Incluir:** <stdarg.h>

**Descrição:** A macro **va\_start()** inicia o processamento de uma lista de argumentos variáveis.

**Protótipo:**

```
void va_start(va_list argumentos, ultimoArgumentoFixo)
```

**Parâmetros:**

- `argumentos` – uma lista de argumentos variáveis.
- `ultimoArgumentoFixo` – nome do último argumento fixo passado para a função.

**Observações:**

- Esta macro faz o parâmetro `argumentos` apontar para o primeiro argumento variável da lista de argumentos variáveis e deve sempre ser usada para iniciar este parâmetro antes de o processamento dos argumentos ser iniciado.
- Se o parâmetro `ultimoArgumentoFixo` for declarado com **register** ou tiver ponteiro para função ou array como tipo, ou com um tipo mais estreito (i.e., **short** ou **char**) do que o tipo **int**, o comportamento da macro **va\_start()** será indefinido.

*va\_arg()*

**Incluir:** <stdarg.h>

**Descrição:** A macro **va\_arg()** fornece o próximo argumento de uma lista de argumentos variáveis.

**Protótipo:**

```
tipo va_arg(va_list argumentos, tipo)
```

**Parâmetros:**

- `argumentos` – lista de argumentos variáveis.
- `tipo` – tipo do próximo argumento da lista a ser acessado.

**Retorno:** O valor do próximo argumento da lista de argumentos variáveis recebida como parâmetro.

**Observações:**

- A lista de argumentos recebida como parâmetro deve ter sido iniciada com a macro `va_start()`.
- É importante que se tenha um meio de determinar quando o processamento da lista de argumentos variáveis deve ser encerrado.
- Para que os valores dos argumentos variáveis sejam retornados corretamente, é importante que se informe precisamente, no segundo argumento da macro `va_arg()`, o tipo do próximo argumento a ser retornado.
- Cada chamada da macro `va_arg()` modifica a lista de argumentos de modo que argumentos sucessivos são retornados na ordem em que se encontram. Se não houver nenhum próximo argumento ou se o tipo especificado não for compatível com o tipo do próximo argumento, o comportamento desta macro será indefinido, exceto nos seguintes casos:
  - Um tipo é inteiro com sinal, o outro tipo é o tipo inteiro correspondente sem sinal e o valor é representável em ambos os tipos.
  - Um tipo é `void *` e o outro é `char *`.
- Como os tipos `char` (`signed` ou `unsigned`), `short` (`signed` ou `unsigned`) e `_Bool` são implicitamente promovidos para `int` (`signed` ou `unsigned`), os tipos `char`, `short` e `_Bool` não devem ser utilizados como segundo parâmetro da macro `va_arg()` (v. Volume I).

`va_end()`

**Incluir:** `<stdarg.h>`

**Descrição:** A macro `va_end()` serve para encerrar o processamento de uma lista de argumentos variáveis.



**Protótipo:**

```
void va_end(va_list argumentos)
```

**Parâmetro:** `argumentos` – lista de argumentos variáveis.

**Observações:**

- Esta macro é usada para liberar eventuais recursos usados por uma variável do tipo **va\_list**. Esta macro invalida o parâmetro recebido; i.e., ele não deve ser mais usado, a não ser que a macro **va\_start()** ou **va\_copy()** seja invocada novamente com este mesmo parâmetro.
- Cada invocação das macros **va\_start()** e **va\_copy()** deve casar com uma invocação correspondente da macro **va\_end()**. Se a macro **va\_end()** não for chamada para cada invocação de **va\_start()** ou **va\_copy()**, o comportamento do programa será indefinido<sup>98</sup>.

*va\_copy()* (C99)

**Incluir:** `<stdarg.h>`

**Descrição:** A macro **va\_copy()** copia o conteúdo de uma variável do tipo **va\_list** para outra variável deste mesmo tipo.

**Protótipo:**

```
va_copy(va_list destino, va_list origem)
```

**Parâmetros:**

- `destino` – variável do tipo **va\_list** que receberá a cópia.
- `origem` – variável do tipo **va\_list** que será copiada.

**Observações:**

- Esta macro faz uma cópia do parâmetro `origem` no estado em que ele se encontra. Em outras palavras, esta macro não reinicia a lista de argumentos

<sup>98</sup> Em algumas implementações (e.g., gcc), a macro **va\_end()** simplesmente não tem nenhum efeito; i.e., tanto faz usá-la ou não. Mas, em nome da portabilidade, é sempre aconselhável utilizar esta macro.

copiada em `destino`, de modo que este parâmetro só estará apontando para o primeiro elemento da lista de argumentos variáveis se a macro `va_arg()` ainda não tiver sido usada no momento da chamada da macro `va_copy()`.

- Se, no instante da cópia, o parâmetro `origem` não apontar para o primeiro argumento da lista de argumentos variáveis devido ao fato de já ter sido usado pela macro `va_arg()` e deseja-se reiniciar o parâmetro `destino`, deve-se invocar as macros `va_end()` e `va_start()`, nesta ordem, passando-lhes `destino` como parâmetro.

## 9.3 COMO CRIAR FUNÇÕES COM LISTAS DE ARGUMENTOS VARIÁVEIS

Conforme antecipado no início deste capítulo, existem duas categorias de funções que usam argumentos variáveis:

- **Funções que usam argumentos variáveis *diretamente*.** Em cabeçalhos e protótipos destas funções, argumentos variáveis são representados por três pontos. Chamar uma função desta categoria não requer nenhum conhecimento especial, mas é preciso usar o material exposto neste capítulo na definição de uma função dessa natureza.
- **Funções que usam argumentos variáveis *indiretamente*.** Estas funções aparentam ter um número fixo de argumentos, mas têm, pelo menos, um argumento do tipo `va_list`, o que, indiretamente, as torna funções com argumentos variáveis. Tanto a criação quanto as chamadas de funções desta categoria requerem o conhecimento apresentado neste capítulo.

### 9.3.1 USO DIRETO DE LISTAS DE ARGUMENTOS VARIÁVEIS

Uma função com argumentos variáveis deve possuir pelo menos um argumento fixo (i.e., com identificador e tipo definidos). Além disso, os argumentos fixos da função devem preceder os argumentos variáveis, representados por três pontos, em seu cabeçalho. A implementação de uma função contendo argumentos variáveis deve seguir quatro passos básicos que serão descritos a seguir.

**Passo 1.** O primeiro passo na definição de uma função com argumentos variáveis consiste em definir no corpo da função uma variável do tipo `va_list` para representar a lista de argumentos. Esta variável é utilizada para acessar os argumentos variáveis da função.

**Passo 2.** O segundo passo no processamento dos argumentos variáveis resume-se em fazer com que a variável do tipo **va\_list** definida na função aponte para o primeiro argumento variável da lista de argumentos. Isto é obtido com o uso da macro **va\_start()**. Conforme apresentado na **Seção 9.2.2**, esta macro recebe dois argumentos: o primeiro argumento será a variável do tipo **va\_list** definida no início da função e o segundo será o nome do último parâmetro fixo da mesma função.

**Passo 3.** O terceiro passo da implementação de uma função com argumentos variáveis consiste em cumprir os objetivos da função; ou seja, acessar e processar todos os argumentos necessários para satisfazer os requisitos da função. O acesso sequencial aos argumentos variáveis de uma função é obtido por meio da macro **va\_arg()**, normalmente invocada no interior de um laço de repetição que se encerra quando todos os argumentos tiverem sido acessados e processados.

**Passo 4.** O último passo de criação da função é finalizar o processamento da lista de argumentos variáveis, após processar todos os argumentos variáveis, invocando a macro **va\_end()** (**Seção 9.2.2**). Esta macro deve *sempre* ser utilizada para encerrar o processamento da lista de argumentos variáveis; caso contrário, o programa poderá apresentar um comportamento indefinido quando a função for executada.

O seguinte esboço de função em pseudocódigo tem como objetivo clarificar o que foi exposto:

```
void F(int umInt, double umDouble, ...)
{
    va_list args; /* Passo 1 */

    /* Passo 2: Inicia a lista de argumentos variáveis. */
    /* 'umDouble' é o último argumento fixo. */
    va_start(args, umDouble);

    /* Passo 3: Acessa e processa os argumentos */
    while(o último elemento da lista não tiver sido acessado) {
        proximoArgumento = va_arg(args, umTipo);

        /* Processa o argumento 'proximoArgumento' que */
        /* deve ser de um tipo compatível com 'umTipo' */
    }

    va_end(args); /* Passo 4: Encerra o processamento da lista */
}
```

Na **Seção 9.4** serão apresentados exemplos reais de implementação de funções com listas de argumentos variáveis.

### 9.3.2 USO INDIRETO DE LISTAS DE ARGUMENTOS VARIÁVEIS

Existem duas abordagens recomendáveis para o uso indireto de argumentos variáveis numa função. Estas abordagens serão descritas a seguir.

#### *Abordagem 1: A função inicia e encerra o processamento dos argumentos*

Neste caso, a função precisa receber um argumento fixo que indique onde deve iniciar a lista de argumentos variáveis. Este argumento é utilizado na invocação da macro **va\_start()** para iniciar a lista de argumentos variáveis. A implementação desta abordagem segue os três últimos passos descritos na **Seção 9.3.1**. Isto é, o único passo que não precisa ser seguido é o primeiro, pois a lista de argumentos variáveis já é recebida como parâmetro pela função. Por outro lado, a função que chama uma função que usa esta abordagem precisa seguir apenas o primeiro passo descrito na **Seção 9.3.1** antes de executar a chamada.

Essa abordagem é esquematizada em pseudocódigo a seguir:

```
void FuncaoChamada(int umInt, va_list args)
{
    /* Inicia a lista de argumentos variáveis. */
    /* 'umInt' deve ser o último argumento fixo. */
    va_start(args, umInt);

    /* Acessa e processa os argumentos */
    while(o último elemento da lista não tiver sido acessado)
    {
        proximoArgumento = va_arg(args, umTipo);

        /* Processa o argumento 'proximoArgumento' que */
        /* deve ser de um tipo compatível com 'umTipo' */
    }

    va_end(args); /* Encerra o processamento da lista */
}

void FuncaoQueChama(int umInt,...)
```

```

{
    int        umInteiro;
    va_list argumentos; /* Passo 1 */

    /* Chama a função sem iniciar a */
    /* lista de argumentos variáveis */
    FuncaoChamada(umInteiro, argumentos);

    /* ... */
}

```

*Abordagem 2: A função nem inicia nem encerra o processamento dos argumentos*

Nesta abordagem, a responsabilidade pela iniciação e pelo encerramento da lista de argumentos variáveis fica a cargo da função chamadora. Isto é, a função que implementa esta abordagem segue apenas o **Passo 3** descrito na **Seção 9.3.1**; os demais passos descritos naquela seção são implementados pela função chamadora. Neste caso, a função não precisa ser informada, por meio de um parâmetro fixo, onde começam os argumentos variáveis.

Esta abordagem, esquematizada a seguir, é a mais utilizada por funções com argumentos variáveis indiretos da biblioteca padrão de C (v. **Capítulo 10**).

```

void FuncaoChamada(va_list args)
{
    /* Acessa e processa os argumentos que */
    /* supostamente já foram iniciados      */
    while(o último elemento da lista não for acessado) {
        proximoArgumento = va_arg(args, umTipo);

        /* ... */
    }
}

void FuncaoQueChama(int umInt,...)
{
    va_list argumentos; /* Passo 1 */

    /* Passo 2: Inicia a lista de argumentos variáveis */
    va_start(argumentos, umInt);

    /* Passo 3: Acesso e processamento dos argumentos */

```

```

        /* são delegados para a função chamada.                */
        FuncaoChamada(argumentos);

        va_end(argumentos); /* Encerra o processamento */
    }

```

É importante ressaltar que, em qualquer abordagem, quando se passa uma variável do tipo **va\_list** como argumento para uma função que invoca a macro **va\_arg()**, essa variável deve ser considerada indeterminada após o retorno da função. Assim, antes de ser reutilizada, essa variável deve ser passada como parâmetro para as macros **va\_end()** e **va\_start()**, que devem ser invocadas nesta ordem.

## 9.4 EXEMPLOS DE FUNÇÕES COM LISTAS DE ARGUMENTOS VARIÁVEIS

**Exemplo 1:** O programa a seguir utiliza uma função com lista de argumentos variáveis para calcular a média de um número indefinido de valores do tipo **double**.

```

#include <stdio.h>
#include <stdarg.h>

extern double Media(int, ...); /* Alusão */

int main()
{
    double arReal[4] = {22.5, 3.5, 7.1, -9.12};

    printf( "\nMedia dos valores %f, %f, %f e %f: %f\n",
            arReal[1], arReal[2], arReal[3], arReal[4],
            Media(4, arReal[1], arReal[2], arReal[3],
                  arReal[4]) );

    return 0;
}

/****
*
* Função Media(): calcula a média de um número
*                 indefinido de valores double
*
* Argumentos: n (entrada) - número de valores

```

```

*          ... (entrada) - valores cuja média
*                          será calculada
*
* Retorno: a média dos valores recebido como argumentos
*
****/

double Media(int n, ...)
{
    double total = 0;
    va_list argumentos;
    int i;

    va_start( argumentos, n );

    for ( i = 1; i <= n; i++ )
        total += va_arg( argumentos, double );

    va_end( argumentos );

    return total/n;
}

```

**Exemplo 2:** O seguinte programa utiliza uma função com lista de argumentos variáveis para concatenar um número arbitrário de *strings*.

```

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

/**
*
* Função ConcatenaStrings(): concatena um número
*                          arbitrário de strings
*
* Parâmetros: strings (no mínimo um), sendo
*               que o último string deve ser NULL
*
* Retorno: um string que representa o resultado
*           da concatenação ou NULL se não for

```

```

*           possível alocar o espaço necessário
*
*** /
char* ConcatenaStrings(const char *str, ...)
{
    va_list      lista1, lista2;
    size_t       tamanho = 1;
    const char   *s;
    char         *concatenacao;

    va_start(lista1, str);

    va_copy(lista2, lista1);

    /* Determina a quantidade de espaço necessário */
    /* para conter a concatenação, que é igual à */
    /* soma dos tamanhos de todos os strings de */
    /* entrada mais 1. */
    for ( s = str; s != NULL;
          s = va_arg(lista1, const char *) )
        tamanho += strlen(s);

    va_end(lista1);

    concatenacao = malloc(tamanho);

    if (!concatenacao) /* Não foi possível alocar */
        return NULL; /*o espaço necessário */

    /* Transforma num string vazio */
    *concatenacao = '\0';

    /* Concatena os strings */
    for ( s = str; s != NULL;
          s = va_arg(lista2, const char *) )
        strcat(concatenacao, s);

    va_end(lista2);

    return concatenacao;
}

int main()

```



```

{
    const char *str1 = "Primeiro string",
               *str2 = "Segundo string",
               *str3 = "Terceiro string",
               *str4 = "Quarto string",
               *str5 = " ",
               *resultado;

    printf( "Strings originais:\n"
           "\t\"%s\" \n\t\"%s\" \n\t\"%s\" "
           "\n\t\"%s\" \n\t\"%s\"",
           str1, str2, str3, str4, str5 );

    resultado = ConcatenaStrings( str1, str5, str2,
                                  str5, str3, str5,
                                  str4, NULL );

    printf( "\n\nResultado da concatenacao: \n\t\"%s\"",
           resultado );

    return 0;
}

```

**Exemplo 3:** O programa apresentado a seguir contém uma função de argumentos variáveis que calcula a soma de um número indefinido de valores do tipo **double**. O aspecto mais interessante desta função é que ela pode encerrar o processamento da lista de argumentos variáveis antes que todos os argumentos tenham sido processados (i.e., quando o resultado parcial da soma atinge um valor tão grande que é considerado infinito).

```

#include <stdio.h>
#include <stdarg.h>
#include <float.h>
#include <math.h>

/****
 *
 * Função SomaDouble(): calcula a soma de um número
 *                      indefinido de parcelas do
 *                      tipo double
 *
 * Argumentos: nTermos (entrada) - número de parcelas

```

```

*                               que serão somados
*                               ... (entrada) - as parcelas que serão
*                               somadas
*
* Retorno: a soma das parcelas ou infinito se o
*          resultado parcial da soma atingir um
*          valor muito grande
*
****/
double SomaDouble(int nTermos, ...)
{
    va_list args;
    double  umArgumento, soma = 0.0;
    int      i;

    va_start (args, nTermos);

    for (i = 0; i < nTermos; i++){
        umArgumento = va_arg(args, double);
        soma += umArgumento;
        /* Encerra, se o resultado */
        /* atingir um valor infinito */
        if (!isfinite(soma))
            break;
    }

    va_end (args);

    return soma;
}

int main(void)
{
    double d1 = 1.5, d2 = DBL_MAX, d3 = DBL_MAX,
           d4 = 4.5;
    double resultado;

    printf( "Parcelas: %e, %e, %e e %e\n",
           d1, d2, d3, d4 );

    resultado = SomaDouble(4, d1, d2, d3, d4);

    if (isfinite(resultado))

```

```

    printf("Soma: %e\n", resultado);
else
    printf("O resultado da soma e' infinito\n");

return 0;
}

```

**Exemplo 4:** O programa apresentado a seguir contém uma função com argumentos variáveis que simplesmente calcula o produto de um número indefinido de valores do tipo **int**. Essencialmente, esse exemplo não apresenta novidades com relação aos exemplos anteriores, mas é exposto aqui com o intuito de que ele seja comparado com o **Exemplo 5** que o segue.

```

#include <stdio.h>
#include <stdarg.h>

/****
 *
 * Função MultiInt(): calcula o produto de um número
 *                    indefinido de termos do tipo int
 *
 * Argumentos: nTermos (entrada) - número de termos que
 *                               serão multiplicados
 *              ... (entrada) - os termos que serão
 *                               multiplicados
 *
 * Retorno: o produto dos termos
 *
 ****/
int MultiInt(int nTermos, ...)
{
    va_list  listaArgs;
    int      i, produto = 1;

    va_start(listaArgs, nTermos);

    for (i = 0; i < nTermos; ++i)
        produto *= va_arg(listaArgs, int);

    va_end(listaArgs);

    return produto;
}

```

```

}

int main(void)
{
    int d1 = 1;
    int d2 = 2;
    int d3 = 3;
    int d4 = 4;

    printf( "\nArgumentos: %d, %d, %d e %d\n",
            d1, d2, d3, d4 );
    printf("Produto: %d\n", MultiInt(4, d1, d2, d3, d4));

    return 0;
}

```

**Exemplo 5:** O programa a seguir realiza essencialmente o mesmo que o programa apresentado no **Exemplo 4**, mas utiliza uma abordagem diferente. Isto é, enquanto a função `MultiInt()` daquele programa utiliza diretamente uma lista de argumentos variáveis, a função `MultiInt2()` do programa a seguir faz isto indiretamente.

```

#include <stdio.h>
#include <stdarg.h>

/****
 *
 * Função MultiInt2(): calcula o produto de um número
 *                    indefinido de termos do tipo int
 *
 * Argumentos: nTermos (entrada) - número de termos que
 *                    serão multiplicados
 *              termos (entrada) - os termos que serão
 *                    multiplicados
 *
 * Retorno: o produto dos termos
 *
 ****/
int MultiInt2(int nTermos, va_list termos)
{
    int i, produto = 1;

    for (i = 0; i < nTermos; i++)

```

```

        produto *= va_arg(termos, int);

    return produto;
}

/****
 *
 * Função ApresentaProduto(): calcula e apresenta o
 *                          produto de um número
 *                          indefinido de termos
 *                          do tipo int
 *
 * Argumentos: nTermos (entrada) - número de termos que
 *                          serão multiplicados
 *              ... (entrada) - os termos que serão
 *                          multiplicados
 *
 * Retorno: Nada
 *
 ****/
void ApresentaProduto(int nTermos, ...)
{
    va_list listaArgs;
    int      i, produto;

    va_start(listaArgs, nTermos);

    printf("\nNumeros: ");

    /* Imprime os termos */
    for (i = 0; i < nTermos; i++)
        printf("%d ", va_arg(listaArgs, int));

    /* Para processar a lista de argumentos      */
    /* novamente, é necessário reiniciá-la.      */
    /* Mas, antes é necessário chamar a macro   */
    /* va_end para encerrar o uso iniciado com */
    /* a última chamada de va_start:            */
    va_end(listaArgs);

    /* Evidentemente, esta própria função poderia */
    /* calcular e apresentar o produto dos números. */
    /* Mas, a título de ilustração, esta tarefa */

```

```

        /* será delegada para a função MultiInt2().      */

        /* Reinicia a lista de argumentos */
        va_start(listaArgs, nTermos);

        produto = MultiInt2(nTermos, listaArgs);

        printf("\nProduto dos numeros: %d\n", produto);

        va_end(listaArgs);
    }

    int main(void)
    {
        int i1 = 1, i2 = 2, i3 = 3, i4 = 4;

        ApresentaProduto(4, i1, i2, i3, i4);

        return 0;
    }

```

Note, neste último exemplo, que, diferentemente do que ocorre com o **Exemplo 4**, é necessária uma função intermediária entre **main()** e **MultiInt2()** para que esta última possa ser chamada. Esta função intermediária [i.e., **ApresentaProduto()**] é responsável pela iniciação e finalização da lista de argumentos variáveis passada para a função **MultiInt2()**. Ou seja, a segunda abordagem descrita na **Seção 9.3.2** é usada aqui.

## 9.5 EXERCÍCIOS DE REVISÃO

1. O que é uma lista de argumentos variáveis?
2. (a) O que caracteriza uma função que usa argumentos variáveis diretamente?  
(b) O que caracteriza uma função que usa argumentos variáveis indiretamente?
3. Qual é o tipo da variável utilizada no processamento de listas de argumentos variáveis?
4. (a) Em que situações o uso da macro **va\_copy()** se faz necessário? (b) O uso desta macro pode ser evitado?

5. Que passos devem ser seguidos para a implementação de uma função que usa listas de argumentos variáveis diretamente?
6. Que passos devem ser seguidos para a implementação de uma função que usa listas de argumentos variáveis indiretamente?
7. Descreva as duas abordagens apresentadas no texto para a implementação de funções que usam listas de argumentos variáveis indiretamente.
8. (a) Descreva o uso da macro **va\_end()**. (b) Qual é a importância desta macro no processamento de listas de argumentos variáveis?
9. O que pode ocorrer quando se invoca a macro **va\_end()** duas vezes seguidas utilizando como parâmetro uma mesma variável que representa uma lista de argumentos variáveis?
10. É possível criar uma função com argumentos variáveis com o seguinte protótipo? Explique.  

```
int F(...)
```
11. Como funciona a macro **va\_arg()**?
12. Quais são os requisitos que devem ser seguidos para uma invocação bem-sucedida da macro **va\_arg()**?
13. Por que os tipos **char**, **short** e **\_Bool** não devem ser utilizados como segundo parâmetro da macro **va\_arg()**?
14. Que cuidados devem ser tomados quando se usa a macro **va\_copy()**?

# *Capítulo 10*

---

*Entrada e saída*



## 10.1 INTRODUÇÃO

O cabeçalho `<stdio.h>` da biblioteca padrão de C provê suporte para operações básicas de entrada e saída em C. Além disso, existem funções para entrada e saída de caracteres e *strings* extensas declaradas no cabeçalho `<wchar.h>`. A maior parte dos componentes deste cabeçalho já foi suficientemente descrita no **Capítulo 8**, de modo que, no presente capítulo, apenas suas funções de entrada e saída serão apresentadas.

## 10.2 CONCEITOS FUNDAMENTAIS DE ENTRADA E SAÍDA

Esta seção apresenta sucintamente conceitos básicos necessários para entender e usar as facilidades de entrada e saída providas pela biblioteca padrão. Estes conceitos são completamente explorados no **Volume I**, de tal forma que o objetivo da presente seção é apenas fornecer uma referência rápida para tais conceitos.

### 10.2.1 PROCESSAMENTO DE ENTRADA E SAÍDA

**Processamento de entrada e saída** consiste em copiar dados entre a memória principal e os dispositivos externos. Mais precisamente, uma operação de entrada copia dados de um dispositivo de entrada para a memória principal, enquanto uma operação de saída copia dados da memória principal para um dispositivo de saída.

### 10.2.2 STREAMS

Um *stream* consiste em um array unidimensional de bytes e é um conceito importante em processamento de arquivos em C porque permite tratar quaisquer dispositivos de entrada ou saída da mesma maneira, de modo que o programador não precise preocupar-se com as diversas diferenças entre eles.

### 10.2.3 PROCESSAMENTO DE ARQUIVOS

Em C, **arquivo** refere-se a qualquer dispositivo que possa ser utilizado como repositório de dados para um programa.

Existem dois tipos de processamento de arquivos em C: processamento de arquivos **baseado em *streams*** e processamento de arquivos **baseado em sistemas**. As principais características destes dois tipos de processamento são apresentadas na **Tabela 10-1**.

BASEADO EM...	CARACTERIZA-SE POR..	PRÓS	CONTRAS
<i>Streams</i>	<ul style="list-style-type: none"> <li>• Utilizar a biblioteca padrão</li> </ul>	<ul style="list-style-type: none"> <li>• Portabilidade</li> <li>• Facilidade de uso</li> </ul>	<ul style="list-style-type: none"> <li>• Relativamente ineficiente</li> </ul>
Sistemas	<ul style="list-style-type: none"> <li>• Utilizar diretamente o sistema operacional</li> </ul>	<ul style="list-style-type: none"> <li>• Eficiência</li> </ul>	<ul style="list-style-type: none"> <li>• Não tem portabilidade</li> </ul>

Tabela 10-1: Tipos de processamento de arquivos em C.

Qualquer arquivo pode ser associado a um *stream* e as funções da biblioteca padrão de C permitem acessar arquivos usando o conceito de *stream*.

## 10.2.4 FORMATOS DE ARQUIVOS

O **formato** de um arquivo refere-se ao fato de as sequências de bytes que compõem o arquivo serem ou não interpretadas. Existem dois tipos de formatos de arquivos: **arquivos de texto** (ou **formatados**) e **arquivos binários** (ou **não formatados**).

Os bytes que constituem um arquivo de texto são interpretados como caracteres organizados em linhas, que terminam com um caractere de quebra de linha ('`\n`'). Quando um arquivo de texto é escrito, o conteúdo armazenado em memória é mapeado em caracteres que farão parte do arquivo. Quando um arquivo de texto é lido, ocorre o mapeamento inverso: sequências de caracteres são interpretadas antes de serem armazenadas em memória.

Em arquivos binários, não existe interpretação de conteúdo. Isto é, num arquivo binário cada bit é lido ou escrito exatamente como ele aparece no arquivo.

## 10.2.5 ACESSO A ARQUIVOS

Arquivos podem ser processados por meios de dois tipos de acesso:

- **Acesso sequencial** – os bytes do arquivo são acessados um a um na ordem em que se encontram no arquivo.
- **Acesso direto** – um dado conjunto de bytes pode ser acessado num local arbitrário do arquivo sem que os bytes que o precedam sejam necessariamente acessados. Nem todo arquivo permite este tipo de acesso<sup>99</sup>.

Processamento de arquivo com acesso sequencial envolve o uso de um laço de repetição que começa a processar um arquivo a partir de seu início e encerra quando o final do arquivo é atingido ou alguma outra condição é satisfeita. Por outro lado, processamento com acesso direto consiste em mover o apontador de posição do arquivo para o local desejado e executar a operação de leitura ou escrita requerida.

### 10.2.6 STREAMS PADRONIZADOS: `stdin`, `stdout` e `stderr`

Existem três *streams* de texto associados aos meios de entrada e saída considerados padrões (e.g., teclado e monitor de vídeo), que são automaticamente abertos no início da execução de qualquer programa em C. Estes *streams* são representados pelas variáveis globais **`stdin`**, **`stdout`** e **`stderr`** aludidas no cabeçalho `<stdio.h>` e discutidas na **Seção 10.6**.

### 10.2.7 ENTRADA E SAÍDA FORMATADAS

Uma operação de **entrada** (ou **leitura**) **formatada** consiste em ler caracteres num *stream* de texto, convertê-los de acordo com um *string* de formatação e armazenar os valores convertidos em memória. Existe uma coleção de funções na biblioteca padrão de C, denominada **família `scanf`**, dedicada à entrada formatada (v. **Seção 10.7.6**).

**Saída** (ou **escrita**) **formatada** consiste em escrever, num *stream* de texto, caracteres que representam valores de tipos de dados conhecidos de acordo com uma especificação de formato. A **família `printf`** é um conjunto de funções da biblioteca padrão de C que se destina à saída formatada (v. **Seção 10.7.7**).

Qualquer função de entrada ou saída formatada faz uso de um parâmetro especial, denominado **string de formatação**, que determina, por meio de **especificadores de formato**, como a conversão dos dados lidos ou escritos deve ser efetuada. Em sua

---

<sup>99</sup> Por exemplo, *streams* associados a um terminal de computador não permitem acesso direto, enquanto aqueles associados a arquivos armazenados em disco o permitem.

forma mais simples, um especificador de formato é constituído pelo caractere `%` seguido de um ou mais caracteres que informam como o dado correspondente a ser lido ou escrito será interpretado. Normalmente, *strings* de formatação usados por funções da família `scanf` possuem apenas especificadores de formato e espaços em branco, enquanto aqueles usados por funções da família `printf` podem conter caracteres comuns. Um **string extenso de formatação** é um *string* de formatação constituído por caracteres extensos. Os mesmos especificadores de formato utilizados em *strings* de formatação comuns podem ser utilizados em *strings* extensos de formatação. O **Apêndice B** apresenta todos os especificadores de formatos usados por funções de entrada ou saída formatada.

## 10.2.8 BUFFERING

Um **buffer** é uma área de memória onde dados lidos ou que serão escritos num arquivo são armazenados temporariamente. O uso de buffers permite que o acesso a dispositivos de entrada ou saída, que é relativamente lento quando comparado ao acesso à memória principal, seja minimizado.

**Buffering** refere-se ao uso de buffers em operações de entrada ou saída. Em C, existem dois tipos de *buffering*:

- **Buffering de linha** – o sistema armazena caracteres até que um caractere de quebra de linha (`'\n'`) seja encontrado ou até que o buffer esteja cheio.
- **Buffering de bloco** – o sistema armazena bytes até que o buffer esteja cheio (independentemente de o caractere `'\n'` ser encontrado ou não).

*Streams* associados a arquivos armazenados externamente têm *buffering* de bloco, os *streams* **stdin** e **stdout** têm *buffering* de linha e o *stream* **stderr** não usa *buffering*.

## 10.2.9 ORIENTAÇÃO DE STREAMS

Existem dois conjuntos de funções para leitura e escrita de caracteres e *strings* de acordo com o tipo de caractere utilizado na operação: funções para caracteres mono-bytes [e.g., **printf()**], declaradas em `<stdio.h>`, e funções para caracteres extensos [e.g., **wprintf()**], declaradas em `<wchar.h>`. Quando um *stream* é aberto, ele permite o uso de funções pertencentes a apenas um destes conjuntos de funções. O que determina qual dos conjuntos é utilizado por um *stream* é a **orientação do stream**.

Existem duas categorias de *streams* de acordo com a orientação:

- **Streams com orientação monobyte.** Um *stream* tem orientação monobyte quando uma operação de leitura ou escrita no *stream* processa um caractere monobyte.
- **Streams com orientação extensa.** Um *stream* tem orientação extensa quando uma operação de leitura ou escrita no *stream* lê ou escreve um caractere extenso, conforme a definição do tipo **wchar\_t** na implementação corrente. Os caracteres escritos num *stream* com orientação extensa são armazenados no arquivo associado ao *stream* como caracteres extensos (i.e., do tipo **wchar\_t**). Funções que executam operações de leitura ou escrita implicitamente fazem conversão entre caracteres extensos e caracteres multibytes na codificação em vigor. Esta conversão pode ser com estado (v. **Seção 7.3.1**) e, por esta razão, cada *stream* com orientação extensa armazena o estado corrente de conversão.

Após um *stream* ser aberto e antes de qualquer leitura ou escrita no mesmo, ele não possui orientação. Após a execução de uma função de entrada ou saída de caracteres extensos, o *stream* passa a ter orientação extensa. De modo semelhante, se for executada uma função de entrada ou saída de caracteres monobytes, o *stream* terá orientação monobyte. Quando um *stream* tem orientação extensa, não devem ser usadas funções de entrada ou saída de caracteres monobytes, e funções de entrada ou saída de caracteres extensos não devem ser aplicadas a *stream* com orientação monobyte. Essas regras aplicam-se a quaisquer *streams*, incluindo **stdin**, **stdout** e **stderr** (v. **Seção 10.6**).

A função **fwide()** (v. **Seção 10.8**) pode ser chamada para consultar a orientação de um *stream*. Antes da primeira operação de entrada ou saída num *stream*, esta função pode também ser usada para estabelecer sua orientação. Após o estabelecimento da orientação de um *stream*, ela só poderá ser alterada reabrindo-se o *stream* usando-se, por exemplo, a função **freopen()** (v. **Seção 10.7.1**).

Na prática, provavelmente, só faz sentido usar mais de um tipo de orientação com os *streams* **stdin**, **stdout** e **stderr**, como mostra o exemplo a seguir.

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

int main()
{
    /* Após a chamada de printf() a seguir a      */
    /*
```

```

    /* orientação de stdout passa a ser monobyte */
    printf("\nEste string sera impresso.\n");

    /* A função wprintf() requer que a      */
    /* orientação de stdout seja extensa. */
    /* Usar a função fwide() não resolve! */
    wprintf(L"\nEste string NAO sera impresso.\n");

    /* Apesar de tipicamente usarem o mesmo dispositivo, */
    /* stdout e stderr são streams diferentes. Como      */
    /* stderr está sendo usado pela primeira vez por     */
    /* fwprintf(), este stream terá orientação extensa  */
    fwprintf(stderr, L"\nImpresso sem problemas.\n");

    return 0;
}

```

## 10.3 PROCESSAMENTO DE ARQUIVOS EM C NA PRÁTICA

O processamento de um arquivo em C segue o seguinte algoritmo geral:

1. Defina um ponteiro para *stream* (v. **Seção 10.3**).
2. Associe o arquivo que deseja processar a um *stream* usando a função **fopen()** (v. **Seção 10.7.1**) e atribua o endereço retornado por esta função ao ponteiro definido no **Passo 1**.
3. Se o endereço atribuído ao ponteiro no **Passo 2** for **NULL**, o arquivo não foi aberto e não poderá ser processado. Portanto, o processamento deve ser imediatamente encerrado.
4. Se o acesso ao arquivo for:
  - 4.1 Sequencial: crie um laço de repetição, contendo as operações de leitura ou escrita desejadas, até que uma dada condição encerre o laço. Condições que, necessariamente, devem encerrar o laço são ocorrência de erro e chegada ao final de arquivo (v. **Seção 10.7.12**).
  - 4.2 Direto: mova o apontador de posição do arquivo até o ponto desejado usando **fseek()** ou **fsetpos()** (v. **Seção 10.7.9**) e, então, execute a operação de leitura ou escrita desejada. Este passo pode ser executado quantas vezes forem necessárias.

5. Quando o processamento do arquivo estiver concluído, feche-o usando **fclose()** (v. **Seção 10.6.1**).

As operações de leitura e escrita, às quais o **Passo 4** do algoritmo faz referência, são encontradas nas seguintes seções deste capítulo:

- *Processamento de caracteres (Seção 10.7.3)* – as funções descritas nesta seção são as funções de entrada e saída mais fundamentais da biblioteca padrão de C, pois elas leem ou escrevem bytes. Portanto, estas funções podem ser usadas com qualquer tipo de arquivo.
- *Processamento de linhas (Seção 10.7.4)* – as funções apresentadas nesta seção são adequadas apenas para processamento de arquivos de texto contendo caracteres monobytes ou multibytes (v. **Capítulo 8**). Antes de usar alguma função descrita nesta seção, certifique-se de que o *stream* é orientado por byte (v. **Seção 10.2.9**).
- *Processamento de blocos (Seção 10.7.5)* – as funções descritas aqui são mais apropriadas para arquivos binários, mas também podem ser usadas com arquivos de texto.
- *Entrada e saída formatadas de caracteres monobytes (Seções 10.7.6 e 10.7.7)* – as funções apresentadas nestas seções devem ser usadas com *streams* de texto orientados por byte.

Provavelmente, as funções de leitura e escritas mencionadas anteriormente são aquelas usadas com mais frequência num programa. Além delas, talvez o programador precise usar em circunstâncias mais específicas as funções de leitura e escrita descritas nas seguintes seções:

- *Entrada e saída formatadas para caracteres extensos (Seção 10.8)* – as funções descritas nesta seção devem ser usadas com *streams* de texto com orientação extensa.
- *Formatação em memória* – as funções discutidas na **Seção 10.7.8** são usadas para escrita ou leitura de dados formatados em arrays de caracteres monobytes. A **Seção 10.8** apresenta funções semelhantes para caracteres extensos.

Outras funções de entrada e saída da biblioteca padrão de C que não foram referidas nesta seção têm papéis coadjuvantes com relação àquelas mencionadas aqui [e.g., **ftell()**] ou têm uso mais restrito [e.g., **perror()**].

## 10.4 TIPOS

Todos os tipos descritos nesta seção são declarados no arquivo `<stdio.h>`. O tipo **wint\_t** utilizado por algumas funções de entrada ou saída que lidam com caracteres extensos é definido em `<wchar.h>` e descrito no **Capítulo 8**.

*fpos\_t*

**Incluir:** `<stdio.h>`

**Descrição:** **fpos\_t** é o tipo de um valor que representa uma posição num arquivo. Ele é o tipo do valor retornado pela função **fgetpos()** e recebido como argumento por **fsetpos()**.

**Exemplo:** Veja o exemplo apresentado para a função **fsetpos()** (**Seção 10.7.9**).

*FILE*

**Incluir:** `<stdio.h>`

**Descrição:** Os campos de uma estrutura do tipo **FILE** armazenam informações sobre um arquivo.

**Observações:**

- O conceito de *stream* é implementado em C por meio de ponteiros para estruturas do tipo **FILE**. Antes de processar um arquivo utilizando o conceito de *stream*, deve-se declarar um ponteiro para uma estrutura deste tipo, como, por exemplo:

```
FILE *umStream;
```

Frequentemente, este ponteiro é denominado **ponteiro para stream** ou simplesmente **stream**, e o identificador utilizado para rotulá-lo envolve o uso da palavra *stream*.



- A implementação dos campos de uma estrutura do tipo **FILE** é dependente do sistema operacional em uso e, portanto, não deve ser acessada diretamente. Isto é, o programador deve manipular arquivos utilizando apenas ponteiros para *streams* em conjunto com as funções apresentadas no capítulo corrente.

**Exemplo:** Existem inúmeros exemplos de uso do tipo **FILE** no presente capítulo.

## 10.5 MACROS

As macros definidas no cabeçalho `<stdio.h>` e suas respectivas expansões são apresentadas na **Tabela 10-2**. Na última coluna desta tabela aparece uma referência à seção onde o uso da respectiva macro é explorado.

MACRO	EXPANSÃO	REFERÊNCIA
<b>_IOFBF</b>	Um valor inteiro indicando <i>buffering</i> de bloco.	<b>Seção 10.7.2</b>
<b>_IOLBF</b>	Um valor inteiro indicando <i>buffering</i> de linha.	<b>Seção 10.7.2</b>
<b>_IONBF</b>	Um valor inteiro indicando ausência de <i>buffering</i> .	<b>Seção 10.7.2</b>
<b>BUFSIZ</b>	Um valor inteiro que representa o tamanho do buffer utilizado pela função <b>setbuf()</b> .	<b>Seção 10.7.2</b>
<b>EOF</b>	Um valor negativo indicando final de arquivo ou ocorrência de erro numa operação de entrada ou saída.	Diversas seções deste capítulo
<b>SEEK_CUR</b>	Um valor inteiro que indica a posição corrente do apontador de posição do arquivo.	<b>Seção 10.7.9</b>
<b>SEEK_END</b>	Um valor inteiro que indica a posição inicial de um arquivo.	<b>Seção 10.7.9</b>
<b>SEEK_SET</b>	Um valor inteiro que indica a posição final de um arquivo.	<b>Seção 10.7.9</b>

MACRO	EXPANSÃO	REFERÊNCIA
<b>FILENAME_MAX</b>	Tamanho máximo de um <i>string</i> utilizado para representar um nome de arquivo.	<b>Seção 10.7.1</b>
<b>FOPEN_MAX</b>	Número máximo de arquivos que podem estar simultaneamente abertos.	<b>Seção 10.7.1</b>
<b>L_tmpnam</b>	Número de caracteres que uma implementação requer para armazenar nomes de arquivos temporários criados pela função <b>tmpnam()</b> .	<b>Seção 10.7.11</b>
<b>TMP_MAX</b>	Número mínimo de nomes de arquivos distintos criados pela função <b>tmpnam()</b> .	<b>Seção 10.7.11</b>

Tabela 10-2: Macros definidas em &lt;stdio.h&gt;.

A macro **WEOF**, que é do tipo **wint\_t** e utilizada com caracteres extensos, é equivalente a **EOF**, utilizada com caracteres simples. Isto é, a macro **WEOF** é usada para indicar final de arquivo ou ocorrência de erro numa operação de entrada ou saída envolvendo caracteres extensos. Esta macro é definida nos cabeçalhos <wchar.h> e <wctype.h> (v. **Capítulo 8**).

## 10.6 VARIÁVEIS GLOBAIS

As variáveis globais **stderr**, **stdin** e **stdout** representam os três *streams* de entrada e saída automaticamente abertos no início da execução de um programa. Eles são todos *streams* de texto e uma descrição sumária destes *streams* é apresentada na **Tabela 10-3**.

STREAM PADRÃO	ASSOCIADO A...
<b>stdin</b>	Entrada padrão de dados (tipicamente, o teclado)
<b>stdout</b>	Saída padrão de dados (tipicamente, o monitor de vídeo)
<b>stderr</b>	Saída padrão de mensagens de erro (tipicamente, o monitor de vídeo)

Tabela 10-3: Variáveis globais representando *streams* padronizados.

## 10.7 FUNÇÕES

### 10.7.1 ABERTURA E FECHAMENTO DE ARQUIVOS

*fopen()*

**Incluir:** <stdio.h>

**Descrição:** A função **fopen()** aloca dinamicamente uma estrutura do tipo **FILE** e preenche seus campos com informações sobre um arquivo.

**Protótipo:**

```
FILE *fopen( const char *restrict nome,
             const char *restrict modo )
```

**Parâmetros:**

- *nome* – *string* que representa um nome de arquivo construído de acordo com as regras do sistema operacional vigente.
- *modo* – *string* que representa um **modo de acesso**. Existe um conjunto de modos de acesso para arquivos de texto e outro para arquivos binários. O conjunto de modos de acesso para arquivos de texto é apresentado na **Tabela 10-4**. Os especificadores de modo de acesso para arquivos binários são obtidos acrescentando-se a letra *b*<sup>100</sup> ao final de cada nome na **Tabela 10-4**.

MODO DE ACESSO	DESCRIÇÃO
"r"	Abre um arquivo de texto existente apenas para leitura.
"w"	Cria um novo arquivo de texto apenas para escrita. Se o arquivo já existir, seu conteúdo será destruído.
"a"	Abre um arquivo de texto existente para acréscimo; i.e., com escrita ao final do arquivo. Se o arquivo com o nome especificado não existir, um novo arquivo com este nome será criado.
"r+"	Abre um arquivo de texto existente para leitura e escrita.

MODO DE ACESSO	DESCRIÇÃO
"w+"	Cria um novo arquivo de texto para leitura e escrita. Se o arquivo já existir, seu conteúdo será destruído.
"a+"	Abre um arquivo de texto existente ou cria um novo arquivo para leitura e acréscimo. Podem-se ler dados em qualquer parte do arquivo, mas eles podem ser escritos apenas ao final do arquivo.

Tabela 10-4: Modos de acesso a arquivos de texto.

**Retorno:** Um ponteiro para a estrutura **FILE** alocada e associada ao arquivo cujo nome é recebido como argumento, se a operação for bem sucedida; **NULL**, se não for possível abrir o arquivo especificado.

#### Observações:

- O que determina se um arquivo será considerado um arquivo de texto ou um arquivo binário é seu modo de abertura, e as funções declaradas em `<stdio.h>` funcionam de modo diferente nos dois modos de abertura (v. **Volume I**).
- Antes de tentar processar um arquivo, deve-se testar o valor retornado pela função **fopen()** para verificar se o arquivo foi aberto sem problemas. Por exemplo, o trecho de programa a seguir tenta abrir um arquivo chamado `Arq.txt` para leitura apenas:

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

...
FILE *stream;

/* Tenta abrir o arquivo */
stream = fopen("Arq2.txt", "r");

if (!stream) { /* O arquivo não pode ser aberto */
    /* Trecho de programa a ser executado */
    /* quando o arquivo não pode ser aberto */
    ...
} else { /* O arquivo foi aberto sem problemas */
    /* Trecho de programa que processa o arquivo */
    ...
}
```

- O número máximo de arquivos que podem estar simultaneamente abertos no sistema operacional corrente é determinado pela macro **FOPEN\_MAX**, definida em `<stdio.h>`.
- A macro **FILENAME\_MAX**, definida em `<stdio.h>`, determina o tamanho máximo do *string* que pode ser utilizado para representar um nome de arquivo no sistema operacional corrente.

**Exemplo:**

```
#include <stdio.h>

int main(void)
{
    FILE *stream;

    /* Abre o arquivo para escrita */
    stream = fopen("Arq1.txt", "r");

    /* Verifica se a abertura do */
    /* arquivo ocorreu sem erros */
    if (stream == NULL) {
        printf("O arquivo Arq1.txt nao pode ser aberto.");
        return 1;
    }

    fclose(stream); /* Fecha o arquivo */

    return 0;
}
```

*fclose()***Incluir:** `<stdio.h>`

**Descrição:** A função **fclose()** é utilizada para fechar um arquivo quando o programa não precisa mais processá-lo.

**Protótipo:**

```
int fclose(FILE *stream)
```

**Parâmetro:** *stream* – *stream* associado a um arquivo aberto.

**Retorno:** Zero, se o arquivo for fechado com sucesso; **EOF**, caso ocorra algum erro durante a operação.

**Observações:**

- Ao fechar-se um arquivo, libera-se o espaço ocupado pela estrutura **FILE** associada ao arquivo e alocada dinamicamente pela função **fopen()** quando ele foi aberto.
- Um erro frequente entre os iniciantes em C é usar um nome do arquivo como argumento, em vez de um ponteiro de *stream*, numa chamada de **fclose()** [e.g., `fclose("teste.dat")`].
- Buffers alocados pelo sistema para um arquivo são automaticamente liberados quando ele é fechado. Se um buffer for associado a um arquivo de saída, o conteúdo dele é descarregado para o respectivo arquivo. No caso de um arquivo com *buffering* aberto apenas para leitura, o conteúdo do buffer é simplesmente descartado.
- Buffers criados com **setbuf()** ou com **setvbuf()** (v. **Seção 10.7.2**) não são automaticamente liberados.
- Após o fechamento de um arquivo, o *stream* associado a ele não deve mais ser utilizado no programa, a não ser que o mesmo *stream* seja novamente associado a um arquivo por meio de uma nova chamada de **fopen()**.

**Exemplo:**

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    char str[] = "Este e' um arquivo de texto";

    stream = fopen("Arq1.txt", "w");
```

```

    /* Testa se arquivo foi aberto */
    if (!stream) {
        fprintf(stderr, "Arquivo nao pode ser aberto");
        return 1;
    }

    /* Escreve conteúdo do string no arquivo */
    fprintf(stream, "%s", str);

    /* Se fclose() retornar um valor diferente */
    /* de zero ocorreu algum erro na tentativa */
    /* de fechamento do arquivo. */
    if (fclose(stream)) {
        printf("O arquivo nao pode ser corretamente fechado.");
        return 1;
    }

    return 0;
}

```

### *freopen()*

**Incluir:** <stdio.h>

**Descrição:** A função **freopen()** fecha o arquivo associado a um *stream* especificado e depois associa este *stream* ao arquivo cujo nome é recebido como argumento.

### **Protótipo:**

```

FILE *freopen( const char *restrict nomeDoArquivo,
               const char *restrict modo,
               FILE *stream )

```

### **Parâmetros:**

- *nomeDoArquivo* – *string* contendo o nome do arquivo a ser aberto.
- *modo* – *string* contendo o modo de abertura (v. **Tabela 10-4**).
- *stream* – ponteiro para a estrutura **FILE** associada a um arquivo aberto.

**Retorno:** O parâmetro `stream`, quando a função obtém êxito; **NULL**, caso contrário.

**Observações:**

- Inicialmente, a função tenta fechar o *stream* recebido como terceiro argumento, mas, se não conseguir, este fato será ignorado. Portanto, o *stream* passado como argumento poderá ser fechado mesmo que não possa ser reaberto.
- O nome do arquivo (primeiro argumento) pode ser **NULL**. Neste caso, a função tentará alterar o modo de abertura conforme especificado pelo segundo argumento como se o nome do arquivo correntemente associado ao *stream* tivesse sido usado. O novo modo de abertura permitido depende de implementação, assim como se o *stream* será inicialmente fechado ou não.
- Se a função for bem sucedida, a orientação do *stream* (v. **Seção 10.2.9**) e os sinalizadores de erro e final de arquivo serão zerados. Além disso, o estado de mudança associado ao *stream* tornar-se-á o estado inicial de mudança (v. **Seção 7.3.1**).
- A função **freopen()** é frequentemente utilizada para associar um *stream* padrão (**stdin**, **stdout** ou **stderr**) a um determinado arquivo-texto, como mostra o seguinte fragmento de programa:

```
FILE *stream;
...
stream = freopen("Entrada.txt", "r", stdin);
```

**Exemplo:**

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    char c;

    /* Associa 'stream' com Arq1.txt */
    stream = fopen("Arq1.txt", "w");

    /* fprintf() escreverá em Arq1.txt */
    fprintf(stream, "Texto escrito em Arq1.txt\n");
```



```

        /* Associa 'stream' com Arq2.txt */
        stream = freopen("Arq2.txt", "w", stream);

        /* fprintf() escreverá em Arq2.txt */
        fprintf(stream, "Texto escrito em Arq2.txt\n");

        /* Associa Arq3.txt com stdout */
        freopen("Arq3.txt", "w", stdout);

        /* printf() escreverá em Arq3.txt */
        printf("Texto escrito em Arq3.txt");

        /* Associa Arq1.txt com stdin */
        freopen("Arq1.txt", "r", stdin);

        /* scanf() lerá um caractere em Arq1.txt */
        scanf("%c", &c);

        /* Imprime caractere lido em stderr */
        fprintf(stderr, "Carctere lido: %c\n", c);

        fclose(stream);

        return 0;
    }

```

## 10.7.2 GERENCIAMENTO DE BUFFERS

### *setbuf()*

**Incluir:** <stdio.h>

**Descrição:** A função **setbuf()** associa um buffer a um *stream*.

**Protótipo:**

```
void setbuf(FILE *restrict stream, char *restrict buffer)
```

**Parâmetros:**

- *stream* – ponteiro para a estrutura **FILE** associada a um arquivo aberto.

- `buffer` – array de caracteres com número de bytes igual a, pelo menos, **BUFSIZ**. Este parâmetro servirá como buffer após a execução de uma chamada desta função. Se ele for **NULL**, não haverá *buffering*.

**Exemplo:**

```
#include <stdio.h>

int main()
{
    char    buffer[BUFSIZ];
    FILE    *stream1, *stream2;

    stream1 = fopen("Arq1.txt", "w");
    stream2 = fopen("Arq2.txt", "w");

    /* stream1 terá buffer */
    setbuf(stream1, buffer);
    fputs( "Alguma coisa escrita em \"Arq1.txt\"",
           stream1 );
    fflush(stream1);

    /* stream2 não terá buffer */
    setbuf(stream2, NULL);
    fputs( "Alguma coisa escrita em \"Arq2.txt\"",
           stream2 );

    fclose(stream1);
    fclose(stream2);

    return 0;
}
```

*setvbuf()***Incluir:** <stdio.h>

**Descrição:** A função `setvbuf()` associa um buffer a um *stream*, permitindo especificar o tamanho do buffer e o tipo de *buffering*.

**Protótipo:**

```
int setvbuf( FILE *restrict stream,
             char *restrict bloco,
             int tipo, size_t tamanho )
```

**Parâmetros:**

- `stream` – ponteiro para a estrutura **FILE** associada a um arquivo aberto.
- `bloco` – array de caracteres (bloco) que servirá como buffer.
- `tamanho` – tamanho do bloco em bytes.
- `tipo` – Tipo de buffer a ser utilizado, que deve ser especificado usando uma das macros (definidas em `<stdio.h>`) apresentadas na **Tabela 10-5**.

MACRO	BUFFERING
<code>_IOFBF</code>	De bloco
<code>_IOLBF</code>	De linha
<code>_IONBF</code>	Nenhum (i.e., entrada ou saída sem <i>buffering</i> )

**Tabela 10-5:** Macros de especificação de *buffering* usados pela função `setvbuf()`.

**Retorno:** Zero, se a função obtém êxito; um valor diferente de zero, caso contrário.

**Observações:**

- Esta função deve ser usada apenas antes da execução de qualquer operação de entrada ou saída no *stream* especificado como parâmetro.
- Quando o segundo argumento numa chamada desta função é **NULL**, um buffer é alocado usando `malloc()`. Este buffer será automaticamente liberado quando o arquivo for fechado.
- Quando o segundo argumento numa chamada desta função não é **NULL**, como na chamada:

```
setbuf(meuStream, array);
```

ela é equivalente a uma chamada da função **setvbuf()**:

```
(void) setvbuf(meuStream, array, _IOFBF, BUFSIZ);
```

onde **BUFSIZ** é a macro, definida em `<stdio.h>`, que especifica o tamanho do buffer utilizado pela função **setbuf()**.

- Uma causa frequente de erro é alocar um buffer como uma variável automática local e não fechar o arquivo antes de retornar da função na qual o buffer foi declarado (v. **Volume I**).

### Exemplo:

```
#include <stdio.h>

#define TAM_BUFFER 20 /* Um buffer bem pequeno */
                      /* para testar o programa */

int main ()
{
    char    buffer[TAM_BUFFER];
    FILE    *stream;
    char    *msg = "\nExamine o conteudo do arquivo "
                  "\"Arq1.txt\", feche-o e digite "
                  "[ENTER] para continuar";

    stream = fopen("Arq1.txt", "w");

    /* O stream terá buffering de linha */
    if(setvbuf(stream, buffer, _IOLBF, TAM_BUFFER)) {
        printf("Nao foi possivel estabelecer buffer\n");
        return 1;
    }

    /* O string a seguir será escrito */
    /* no buffer e não no arquivo      */
    fputs("Teste", stream);

    printf("%s", msg);
    getchar();

    /* O conteúdo do buffer será enviado para o */
    /* arquivo juntamente com o caractere '\n' */
    fputc('\n', stream);
}
```

```

printf("%s", msg);
getchar();

fputs( "Parte destes caracteres serao enviados ao "
      "arquivo apos excederem o tamanho do buffer.",
      stream);

printf("%s", msg);
getchar();

fclose (stream);

printf("%s", msg);
printf("\nO programa sera' encerrado.");
getchar();

return 0;
}

```

## *fflush()*

**Incluir:** <stdio.h>

**Descrição:** A função **fflush()** descarrega a área de buffer associada a um *stream* de saída, enviando o conteúdo do buffer para o arquivo associado ao respectivo *stream*.

**Protótipo:**

<pre>int fflush(FILE *stream)</pre>
-------------------------------------

**Parâmetro:** *stream* – *stream* associado a um arquivo aberto ou **NULL**.

**Retorno:** Zero, se a função for bem sucedida; **EOF**, se ocorrer algum erro durante sua execução.

**Observações:**

- Esta função serve para descarregar apenas buffers associados a *streams* de saída. Não existe nenhuma função na biblioteca padrão de C que descarregue *streams* de entrada.

- Se o argumento desta função for **NULL**, todos os buffers do programa que a chama serão descarregados.
- Esta função não tem nenhum efeito sobre um arquivo sem buffer associado.

**Exemplo:** Sem o uso de **fflush()**, o programa a seguir poderia não imprimir a mensagem "Antes da divisao".

```
#include <stdio.h>

int main()
{
    int i = 1, j = 0;

    printf("Antes da divisao");
    fflush(stdout);

    /* Divisão proposital por zero */
    i = i / j;

    printf("\nDepois da divisao\n");

    return 0;
}
```

Quando esse programa é executado no Linux, obtém-se o seguinte no meio de saída:

```
Antes da divisao
Floating point exception
```

Removendo-se a chamada de **fflush()** do programa, o resultado obtido é apenas:

```
Floating point exception
```

### 10.7.3 PROCESSAMENTO DE CARACTERES MONOBYTES

*getc() e fgetc()*

**Incluir:** <stdio.h>

**Descrição:** As funções **getc()** e **fgetc()** leem um caractere num *stream* especificado.

**Protótipos:**

```
int getc(FILE *stream)
```

```
int fgetc(FILE *stream)
```

**Parâmetro:** *stream* – um *stream* aberto que permite leitura.

**Retorno:** O caractere lido convertido em **int** ou **EOF**, se ocorrer algum erro de leitura ou o final de arquivo tiver sido atingido.

**Observação:** **getc()** é normalmente implementada como macro e pode ser mais eficiente do que **fgetc()**, que é sempre implementada como função.

**Exemplo:** O programa a seguir imprime no meio de saída padrão o conteúdo do arquivo-texto cujo nome é introduzido na linha de comando. Neste programa, o arquivo é lido caractere a caractere usando-se a função (ou macro) **getc()**, mas o mesmo resultado seria obtido se esta função fosse substituída por **fgetc()**.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE *stream;
    char c;

    if (2 != argc) {
        printf( "\nEste programa deve ser usado assim:\n"
               " %s <nome do arquivo texto>\n", argv[0]);
        return 1;
    }

    /* Tenta abrir para leitura o arquivo cujo */
    /* nome foi introduzido na linha de comando */
    stream = fopen(argv[1], "r");
```

```

if (!stream) {
    printf("Nao foi possivel abrir o arquivo\n");
    return 1;
}

/* Imprime o conteúdo do arquivo até */
/* que seu final seja atingido      */
while((c = getc(stream)) != EOF)
    printf("%c", c);

fclose(stream);

return 0;
}

```

### *putc() e fputc()*

**Incluir:** <stdio.h>

**Descrição:** As funções **putc()** e **fputc()** escrevem um caractere num *stream*.

**Protótipos:**

```
int putc(int caractere, FILE *stream)
```

```
int fputc(int caractere, FILE *stream)
```

Parâmetros:

- *caractere* – caractere a ser impresso.
- *stream* – *stream* aberto para escrita.

**Retorno:** O caractere impresso quando há êxito; **EOF**, caso contrário.

**Observação:** **putc()** é normalmente implementada como macro e pode ser mais eficiente do que **fputc()**, que é sempre implementada como função.



**Exemplo:** A chamada de **fputc()** no programa a seguir pode ser substituída por uma chamada de **putc()** sem alterar o resultado.

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    char *str = "Isto e' um teste.";

    /* Abre Arq1.txt para escrita */
    if ( !(stream = fopen("Arq1.txt", "w")) ) {
        fprintf(stderr, "Arquivo nao foi aberto");
        return 1;
    }

    /* Escreve cada caractere do */
    /* string 'str' em Arq1.txt */
    while (*str)
        fputc(*str++, stream);

    fclose(stream);

    return 0;
}
```

### *getchar()*

**Incluir:** <stdio.h>

**Descrição:** A função **getchar()** lê um caractere no meio de entrada padrão.

**Protótipo:**

```
int getchar(void)
```

**Retorno:** Quando esta função obtém êxito, o caractere lido em **stdin** convertido em **int**; **EOF**, caso contrário.

**Observação:** Uma chamada de **getchar()** é equivalente à chamada: **fgetc(stdin)**.

**Exemplo:**

```
#include <stdio.h>

int main(void)
{
    int c;

    printf("Digite uma frase: ");

    /* Imprime a frase caractere por caractere */
    /* caractere até encontrar '\n' ou EOF */
    while (c != '\n' && c != EOF) {
        c = getchar();
        putchar(c);
    }

    putchar('^');
    putchar('\n');
    putchar('|');
    printf("\n\nEsta foi a frase que voce digitou\n");

    return 0;
}
```

*putchar()***Incluir:** <stdio.h>**Descrição:** A função **putchar()** escreve um caractere no meio de saída padrão.**Protótipo:**

int putchar(int caractere)

**Parâmetro:** caractere – o caractere a ser impresso.**Retorno:** Um inteiro sem sinal correspondente ao caractere impresso se a escrita for efetuada com êxito; caso contrário, **EOF**.**Observação:** A chamada de **putchar()**:

```
putchar(caractere);
```

é equivalente à chamada de **fputc()**:

```
fputc(caractere, stdout);
```

### Exemplo:

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

int main(void)
{
    int c;

    /* Imprime todas as letras do      */
    /* conjunto de caracteres corrente */
    for(c = 0; c <= CHAR_MAX; ++c)
        if (isalpha(c))
            putchar(c);

    return 0;
}
```

## 10.7.4 PROCESSAMENTO DE LINHAS

O processamento de arquivos linha por linha é conveniente apenas para arquivos de texto. Existem duas funções declaradas em `<stdio.h>` que leem e escrevem uma linha num dado *stream* de texto, respectivamente: **fgets()** e **fputs()**.

*fgets()*

**Incluir:** `<stdio.h>`

**Descrição:** A função **fgets()** lê caracteres num *stream* e armazena-os num array até encontrar um caractere de quebra de linha, chegar ao final do arquivo ou atingir o número máximo de caracteres especificado.

**Protótipo:**

```
char *fgets(char *restrict ar, int n, FILE *restrict stream)
```

**Parâmetros:**

- `ar` – array onde os caracteres lidos serão armazenados.
- `n` – número máximo de caracteres que serão armazenados no array `ar`.
- `stream` – *stream* onde será feita a leitura.

**Retorno:** `NULL`, quando o final do arquivo é atingido antes de a função armazenar qualquer caractere no array; caso contrário, o argumento `ar`.

**Observação:** Esta função escreve automaticamente um caractere nulo (`'\0'`) após o último caractere armazenado no array. Assim, o número máximo de caracteres que ela lê é  $n - 1$ .

**Exemplo:** O programa a seguir imprime o conteúdo do próprio arquivo-fonte.

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

#define TAM_ARRAY 256

int main(int argc, char *argv[])
{
    FILE *stream;
    char linha[TAM_ARRAY];
    int contador = 0;
    char *nomeArqFonte, *nomeExec, *p;

    /* O primeiro argumento na linha de          */
    /* comando é o nome do arquivo executável */
    nomeExec = argv[0];

    /* Verifica se o arquivo tem extensão (DOS/Windows) */
    if (p = strchr(nomeExec, '.')) /* Arquivo tem extensão */
        *p = '\0';                /* Remove a extensão */

    /* Determina o provável nome do arquivo-fonte */
    nomeArqFonte = malloc(strlen(nomeExec) + 3);
    nomeArqFonte = strcat(nomeExec, ".c");
```

```

if (!(stream = fopen(nomeArqFonte, "r"))) {
    fprintf( stderr,
        "\nNao pude encontrar arquivo-fonte" );
    return 1;
}

printf("\nConteudo do arquivo fonte:\n\n");

while (fgets(linha, TAM_ARRAY, stream)) {
    linha[ strlen(linha) - 1 ] = '\0'; /* Remove '\n' */
    printf("Linha %3d: %s\n", ++contador, linha);
}

/* Se fgets() retorna NULL, ocorreu um erro */
/* ou o final do arquivo foi encontrado */
if (!feof(stream)) { /* Ocorreu um erro */
    fprintf( stderr,
        "\nOcorreu um erro lendo o arquivo-fonte" );
    return 1;
}

printf("\nTotal de linhas no arquivo-fonte: %d\n",
    contador);

return 0;
}

```

## *fputs()*

**Incluir:** <stdio.h>

**Descrição:** A função **fputs()** escreve os caracteres de um *string* num *stream*.

**Protótipo:**

```
int fputs(const char *restrict s, FILE *restrict stream)
```

**Parâmetros:**

- *s* – *string* que será escrito.
- *stream* – *stream* onde será feita a escrita.

**Retorno:** Um valor positivo quando a escrita é bem sucedida; caso contrário, **EOF**.

**Observação:** Esta função não insere um caractere de quebra de linha após a escrita do último caractere do *string*, como faz a função semelhante **puts()** (v. adiante).

### Exemplo:

```
#include <stdio.h>

int main(void)
{
    FILE    *stream;

    /* Abre Arq1.txt para escrita */
    stream = fopen("Arq1.txt", "w");

    if (stream) { /* Abertura foi OK */
        /* A frase será escrita em Arq1.txt */
        fputs("\nO mar e' azul.\n", stream);
    } else { /* Arquivo não foi aberto */
        /* A frase será escrita na saída */
        /* padrão de mensagens de erro */
        fputs("O arquivo nao pode ser aberto.", stderr);
        return 1;
    }

    fclose(stream);

    return 0;
}
```

*gets()*

**Incluir:** <stdio.h>

**Descrição:** A função **gets()** lê caracteres no *stream* padrão de entrada e armazena-os num array de caracteres até atingir um caractere de quebra de linha ou o final do *stream*.

**Protótipo:**

```
char *gets(char *array)
```

**Parâmetro:** `array` – array com capacidade para conter os caracteres lidos mais o caractere terminal de *string*.

**Retorno:** Seu único parâmetro, se a função obtém êxito; **NULL**, caso contrário.

**Observações:**

- Se o *string* passado como argumento não tiver espaço suficiente para conter os caracteres lidos, poderá haver corrupção de memória. Como é impossível prever o espaço necessário para armazenar os caracteres que um usuário possa digitar no meio de entrada padrão, o conselho mais seguro com relação a esta função é: *nunca a utilize*.
- Dê preferência ao uso da função **fgets()**, que permite limitar o número de caracteres lidos.

**Exemplo:**

```
#include <stdio.h>

int main(void)
{
    char string[80]; /* Nunca é suficiente para gets() */

    printf("\nIntroduza um string: ");

    /* E se o usuário digitar */
    /* mais de 80 caracteres??? */
    gets(string);

    printf("\nO string introduzido foi: \"%s\"\n", string);

    return 0;
}
```

No último exemplo, em vez da chamada de **gets()**, é preferível usar a seguinte chamada de **fgets()**:

```
fgets(string, 80, stdin);
```

*puts()*

**Incluir:** <stdio.h>

**Descrição:** A função **puts()** escreve um *string* no meio de saída padrão.

**Protótipo:**

```
int puts(const char *string)
```

**Parâmetros:** *string* – o *string* a ser impresso em **stdout**.

**Retorno:** Um valor positivo, se a função obtém êxito; caso contrário, **EOF**.

**Observação:** A chamada de **puts()**:

```
puts(str);
```

tem o mesmo efeito que a chamada de **printf()**:

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

**Exemplo:**

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

```
int main(void)
```

```
{
```

```
    char str[] = "Isto e' um teste.";
```

```
    /* Imprime o string no meio de saída padrão */
    puts(str);
```

```
    return 0;
```

```
}
```



## 10.7.5 PROCESSAMENTO DE BLOCOS

Um bloco de memória é um array unidimensional de bytes que podem ser agrupados para constituir elementos de um array. As funções declaradas em `<stdio.h>` usadas para leitura e escrita de blocos são **fread()** e **fwrite()**, respectivamente.

*fread()*

**Incluir:** `<stdio.h>`

**Descrição:** A função **fread()** lê num *stream* dados com tamanho e número especificados e armazena-os num bloco.

**Protótipo:**

```
size_t  fread( void    *restrict ptr, size_t tamanho,
               size_t  nItens, FILE *restrict stream )
```

**Parâmetros:**

- `ptr` – ponteiro para um bloco onde os dados lidos serão armazenados.
- `tamanho` – tamanho de cada item lido.
- `nItens` – número de itens que serão lidos e armazenados no bloco.
- `stream` – *stream* onde será feita a leitura.

**Retorno:** O número de itens que foram lidos e armazenados; zero, se o final do arquivo for atingido ou ocorrer algum erro antes da leitura de qualquer elemento.

**Observações:**

- O valor retornado por esta função deverá ser igual ao valor do seu terceiro argumento, a não ser que ocorra um erro ou o final do *stream* seja atingido.
- Esta função pode ser utilizada para escrever valores de quaisquer tipos e não apenas arrays como parece sugerir seu protótipo. Por exemplo:

```
double umDouble;
FILE *stream = fopen("arq.dat", "rb");
fread(&umDouble, sizeof(double), 1, stream);
```

- O valor zero retornado por esta função é ambíguo porque pode indicar que o final do arquivo foi atingido ou que ocorreu algum erro antes da leitura de qualquer elemento. Esta ambiguidade pode ser resolvida utilizando-se **ferror()** ou **feof()** (v. **Seção 10.7.12**).

### Exemplo:

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

#define MAX 20

int main(void)
{
    FILE *stream;
    char str[] = "Isto e' um teste.";
    char ar[MAX];
    size_t caracteresLidos;
    int i;

    /* Abre arquivo para leitura e escrita */
    if ( !(stream = fopen("Arq1.txt", "w+")) ) {
        fprintf(stderr, "Arquivo nao pode ser aberto.\n");
        return 1;
    }

    /* Escreve string no arquivo, limitando */
    /* o número de caracteres escritos a MAX */
    fprintf(stream, "%.*s", MAX, str);

    /* Volta ao início do arquivo */
    rewind(stream);

    caracteresLidos = fread(ar, 1, MAX, stream);

    printf( "\nNumero de caracteres lidos: %d\n",
           caracteresLidos );

    printf( "\nCaracteres lidos: \"" );

    /* Imprime os caracteres lidos no arquivo */
}
```

```

    for (i = 0; i < caracteresLidos; ++i)
        putchar(ar[i]);

    printf("\n\n");

    fclose(stream);

    return 0;
}

```

### *fwrite()*

**Incluir:** <stdio.h>

**Descrição:** A função **fwrite()** lê dados num bloco e escreve-os num *stream*, permitindo especificar o tamanho de cada item e o número de itens escritos.

#### **Protótipo:**

```

size_t  fwrite( const void  *restrict ptr, size_t tamanho,
                size_t nItens, FILE *restrict stream )

```

#### **Parâmetros:**

- *ptr* – ponteiro para um bloco onde os dados serão lidos.
- *tamanho* – tamanho de cada item a ser escrito.
- *nItens* – número de itens que serão escritos.
- *stream* – *stream* onde será feita a escrita.

**Retorno:** O número de itens que foram escritos no *stream* especificado.

#### **Observações:**

- Quando ocorre algum erro de escrita, o valor retornado por esta função é menor do que o terceiro argumento.
- A função **fwrite()** pode ser utilizada para escrever valores de quaisquer tipos e não apenas arrays como parece sugerir seu protótipo. Por exemplo, para escrever num arquivo um único valor do tipo **double** armazenado em memória, pode-se usar o seguinte trecho de programa:

```
FILE    *stream = open("MeuArquivo.dat", "wb");
double umDouble;
...
fwrite(&umDouble, sizeof(double), 1, stream);
```

**Exemplo:**

```
#include <stdio.h>

#define TAM_ARRAY(ar) sizeof(ar)/sizeof(ar[0])

int main(void)
{
    FILE *stream;
    int arrayInt[] = {1, 2, 3, 4, 5, 6, 7, 8, 9},
        arrayIntLido[9],
        i;

    /* Abre Arq1.bin para escrita */
    if ( !(stream = fopen("Arq1.bin", "wb")) ) {
        fprintf(stderr, "Arquivo nao pode ser aberto");
        return 1;
    }

    /* Escreve o array 'arrayInt' em Arq1.bin */
    fwrite( arrayInt, sizeof(arrayInt[0]),
            TAM_ARRAY(arrayInt), stream );

    /* Reabre Arq1.bin; agora, para leitura */
    if ( !freopen("Arq1.bin", "rb", stream) ) {
        fprintf(stderr, "Arquivo nao pode ser reaberto");
        return 1;
    }

    /* Lê o array no arquivo Arq1.bin */
    fread( arrayIntLido, sizeof(arrayInt[0]),
            TAM_ARRAY(arrayInt), stream );

    printf("\nArray lido no arquivo: { ");

    /* Exibe os elementos do array em stdout */
    for(i = 0; i < 8; ++i)
```

```

    printf("%d, ", arrayIntLido[i]);

    printf("%d }\n", arrayIntLido[i]); /* Último elemento */

    fclose(stream);

    return 0;
}

```

## 10.7.6 ENTRADA FORMATADA: A FAMÍLIA DE FUNÇÕES SCANF

A família de funções `scanf` consiste em um conjunto de funções com as seguintes características comuns:

- Todas as funções funcionam de modo análogo à função **`scanf()`**: leem caracteres, converte-os de acordo com um *string* de formatação (v. **Apêndice B**) e armazenam os valores convertidos em variáveis.
- Todas as funções retornam o número de valores lidos, convertidos e armazenados em variáveis, quando bem sucedidas; quando ocorre algum erro, elas retornam **EOF**.
- Todas as funções utilizam *scanf* na composição de seus nomes.

A família de funções `scanf` é apresentada de forma resumida na **Tabela 10-6**.

FUNÇÃO	DESCRIÇÃO RESUMIDA
<b><code>fscanf()</code></b>	Lê dados formatados num <i>stream</i> de texto especificado.
<b><code>fwscanf()</code></b>	Lê dados formatados no <i>stream</i> especificado, usando um <i>string</i> extenso de formatação.
<b><code>scanf()</code></b>	Lê dados formatados em <b><code>stdin</code></b> .
<b><code>sscanf()</code></b>	Lê dados formatados num <i>string</i> .
<b><code>swscanf()</code></b>	Lê dados num <i>string</i> extenso usando um <i>string</i> extenso de formatação.
<b><code>vfscanf()</code> (C99)</b>	Semelhante à função <b><code>fscanf()</code></b> , mas usa argumentos variáveis indiretamente por meio de um parâmetro do tipo <b><code>va_list</code></b> .
<b><code>vswscanf()</code> (C99)</b>	Semelhante à função <b><code>scanf()</code></b> , mas usa argumentos variáveis indiretamente por meio de um parâmetro do tipo <b><code>va_list</code></b> .

FUNÇÃO	DESCRIÇÃO RESUMIDA
<b>vsscanf()</b> (C99)	Semelhante à função <b>sscanf()</b> , mas usa argumentos variáveis indiretamente por meio de um parâmetro do tipo <b>va_list</b> .
<b>vfwscanf()</b> (C99)	Semelhante à função <b>fwscanf()</b> , mas usa argumentos variáveis indiretamente por meio de um parâmetro do tipo <b>va_list</b> .
<b>vswscanf()</b> (C99)	Lê dados formatados num <i>string</i> extenso usando um <i>string</i> extenso de formatação e armazena os dados lidos em variáveis cujos endereços são passados numa lista de argumentos variáveis do tipo <b>va_list</b> .
<b>wscanf()</b> (C99)	Semelhante à função <b>wscanf()</b> , mas usa argumentos variáveis indiretamente por meio de um parâmetro do tipo <b>va_list</b> .
<b>wscanf()</b>	Lê dados formatados usando um <i>string</i> extenso de formatação em <b>stdin</b> .

Tabela 10-6: Família de funções scanf.

As funções da família `scanf` que começam com a mesma letra têm outras afinidades além das já enumeradas aqui. Para facilitar a identificação de funções desta família, o significado de cada prefixo usado com seus membros é apresentado na **Tabela 10-7**.

PREFIXO	SIGNIFICADO
<b>f</b>	A leitura de dados é feita num <i>stream</i> especificado como argumento.
<b>s</b>	A leitura de dados é feita num <i>string</i> <sup>101</sup> .
<b>v</b>	Um dos argumentos da função é do tipo <b>va_list</b> .
<b>w</b>	A leitura é efetuada usando um <i>string</i> extenso de formatação; se a leitura for feita num <i>string</i> , ele será constituído de caracteres extensos.

Tabela 10-7: Prefixos utilizados em nomes de membros da família scanf.

Alguns prefixos apresentados na **Tabela 10-7** podem ser usados de modo combinado (e.g., *vf* e *vs*).

As funções **sscanf()** e **vsscanf()** da família `scanf`, utilizadas em formatação em memória serão apresentadas na **Seção 10.7.8**, enquanto as funções desta família que

---

<sup>101</sup> Note que a letra *s* no início de *scanf* não é um prefixo.

lidam com caracteres e *strings* extensos serão apresentadas na **Seção 10.8**. As demais funções que compõem a família `scanf` serão apresentadas a seguir.

*fscanf()*

**Incluir:** `<stdio.h>`

**Descrição:** A função **fscanf()** lê dados formatados num *stream* especificado.

**Protótipo:**

```
int fscanf( FILE *restrict stream,
            const char *restrict formato, ... )
```

**Parâmetros:**

- *stream* – *stream* onde será feita a leitura.
- *formato* – *string* de formatação (v. **Apêndice B**).
- ... – representa endereços de variáveis onde os valores lidos serão armazenados.

**Retorno:** O número de itens lidos e armazenados em variáveis; **EOF**, se o final do *stream* for encontrado durante a leitura ou ocorrer algum erro de leitura.

**Observações:**

- A única diferença entre a função **fscanf()** e a função **scanf()** é que **fscanf()** permite a especificação de um *stream*, enquanto **scanf()** lê apenas no *stream* padrão **stdin**.
- Consulte a descrição da função **scanf()** para maiores detalhes.

**Exemplo:**

```
#include <stdio.h>

int main(void)
{
    char    string[20];
    int     umInt, nValoresLidos;
    FILE    *stream;
```

```

    /* Tenta abrir o arquivo */
    if (!(stream = fopen("Arq1.txt", "w+"))) {
        printf("Nao foi possivel abrir o arquivo");
        return 1;
    }

    /* Imprime um string e um inteiro no arquivo */
    fprintf(stream, "String 125");

    rewind(stream); /* Volta ao início do stream */

    nValoresLidos = fscanf( stream, "%20s %d",
                           string, &umInt );

    if (nValoresLidos == 2)
        printf("O valor lido foi: %d", umInt);
    else
        printf("Nao foi lido nenhum valor inteiro");

    fclose(stream);

    return 0;
}

```

### *scanf()*

**Incluir:** <stdio.h>

**Descrição:** A função **scanf()** permite a leitura de valores de vários tipos simultaneamente no meio da entrada padrão (**stdin**).

**Protótipo:**

```
int scanf(const char *restrict formato, ...)
```

**Parâmetros:**

- *formato* – *string* de formatação (v. **Apêndice B**).
- ... – representa endereços de variáveis onde os valores lidos serão armazenados.



**Retorno:** O número de itens lidos e armazenados em variáveis; **EOF**, se o final do *stream stdin* for encontrado durante a leitura.

### Observações:

- O uso correto desta e de outras funções da família `scanf` requer que haja um endereço de variável para cada especificador de formato no *string* de formatação e que o tipo de cada variável seja compatível com a especificação de formato correspondente.
- Quando caracteres que não são parte de especificadores de formato são incluídos no *string* de formatação, esta função espera que o usuário digite cada caractere exatamente como ele aparece no *string* de formatação.
- A função **`scanf()`** não permite a leitura de *strings* contendo espaços em seu interior. O mesmo aplica-se a outros membros da família `scanf`.

### Exemplo:

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

```
int main()
```

```
{
```

```
    int    inteiro;
```

```
    float  real;
```

```
    int    nValoresLidos, c;
```

```
    /* Apresenta um prompt inicial para o usuário */
```

```
    printf("\nDigite um numero inteiro "
```

```
           "seguido de um numero real: " );
```

```
    /* Garante que o usuário recebe a mensagem */
```

```
    fflush(stdout);
```

```
    /* Permanece no laço até que o usuário */
```

```
    /* digite os dados corretamente.      */
```

```
    while(1) {
```

```
        nValoresLidos = scanf("%d %f", &inteiro, &real);
```

```
        if (2 == nValoresLidos)
```

```
            break; /* Usuário digitou corretamente */
```

```

    if (1 == nValoresLidos) /* Um prompt mais elaborado */
        printf( "\nO segundo valor digitado esta' "
                "incorreto. Digite os dois valores "
                "novamente: " );
    else if (0 == nValoresLidos) /* Idem */
        printf( "\nOs dois valores digitados estao "
                "incorretos. Digite-os novamente: " );
    else { /* Só existem as três possibilidades acima */
        fprintf( stderr, "\nEste programa contem um bug "
                "nao identificado." );
        return 1;
    }

    /* Limpa o buffer de entrada. Se */
    /* isto não for feito, o programa */
    /* entra num laço sem fim. */
    while ( (c = getchar()) != '\n' && c != EOF )
        ; /* Intencionalmente vazio */

    printf( "\nOs valores introduzidos foram: %d e %f\n",
            inteiro, real );

    return 0;
}

```

### *vfscanf()*

**Incluir:** <stdio.h>

**Descrição:** A função **vfscanf()** lê entrada formatada num *stream* utilizando indiretamente uma lista de argumentos variáveis.

**Protótipo:**

```

int vfscanf( FILE *restrict stream,
             const char *restrict formato,
             va_list args )

```

**Parâmetros:**

- `stream` – *stream* onde será feita a leitura.
- `formato` – *string* de formatação (v. **Apêndice B**).
- `args` – lista de argumentos variáveis contendo endereços de variáveis onde os valores lidos serão armazenados (v. **Capítulo 9**).

**Retorno:** O número de itens lidos e armazenados em variáveis; **EOF**, se o final do *stream* for encontrado durante a leitura ou ocorrer algum erro de leitura.

**Observações:**

- A única aparente diferença entre esta função e **fscanf()** é que **fscanf()** possui uma lista de argumentos variáveis, enquanto **vfscanf()** possui exatamente três parâmetros. Mas, o terceiro parâmetro da função **vfscanf()** é do tipo **va\_list** que representa uma lista de argumentos variáveis (v. **Capítulo 9**). Portanto, a real diferença entre as funções **fscanf()** e **vfscanf()** diz respeito apenas às situações nas quais cada uma delas pode ser usada. Ou seja, a função **vfscanf()** permite ser chamada por uma função com uma lista de parâmetros variáveis utilizando estes mesmos parâmetros, enquanto a função **fscanf()** não permite isto.
- Consulte as descrições das funções **fscanf()** e **vsprintf()** e o **Capítulo 9**.

**Exemplo:**

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

/*****
 *
 * Função LeDadosLinha(): lê dados formatados de acordo com
 *                        um string de formatação na n-ésima
 *                        linha de um arquivo-texto
 *
 * Argumentos: arquivo (entrada) - nome do arquivo
 *            n (entrada) - índice da linha (começa em 1)
 *            formato (entrada) - string de formatação
 *            ... (saída) - endereços de variáveis que
 *                        terão valores atribuídos
 */
```

```

*
* Retorno: número de variáveis que tiveram valores
*         atribuídos
*
****/

int LeDadosLinha( const char *arquivo, unsigned n,
                  char *formato, ... )
{
    int      nValoresAtribuidos;
    va_list  args;
    FILE     *stream;
    int      caractere;
    unsigned i;

    if (!(stream = fopen(arquivo, "r")))
        return 0;

    if (!n) { /* Indexação de linhas começa em 1 */
        fclose(stream);
        return 1;
    }

    /* Lê e descarta linhas até atingir */
    /* a n-ésima linha do arquivo      */
    for (i = 1; i < n; ++i) {
        do {
            caractere = getc(stream);
        } while ((caractere != '\n') && (caractere != EOF));

        if (caractere == EOF) { /* Final do arquivo atingido */
            fclose(stream); /* antes da linha desejada */
            return 0;
        }
    }

    /* Neste ponto, o apontador de posição do arquivo */
    /* aponta para o início da linha desejada. A função */
    /* vfscanf() é chamada para completar o serviço.   */

    va_start(args, formato);
    nValoresAtribuidos = vfscanf(stream, formato, args);
    va_end(args);
}

```

```

    fclose(stream);

    return nValoresAtribuidos;
}

int main(void)
{
    int    umInt, nValoresLidos;
    float  umFloat;
    char   umChar;

    nValoresLidos = LeDadosLinha( "Arq2.txt", 5, "%f %d %c",
                                   &umFloat, &umInt, &umChar );

    printf( "\nNumero de valores lidos e atribuidos: %d",
            nValoresLidos );

    if (nValoresLidos == 3)
        printf( "\nValores lidos: %f %d %c\n",
                umFloat, umInt, umChar );

    return 0;
}

```

No exemplo anterior, a função `LeDadosLinha()` armazena os argumentos variáveis recebidos numa variável local do tipo **va\_list** e chama a função **vfscanf()** passando-lhe como terceiro parâmetro esta variável. Uma chamada equivalente não seria possível com o uso da função **fscanf()**.

*vfscanf()*

**Incluir:** <stdio.h>

**Descrição:** A função **vfscanf()** lê dados formatados no meio de entrada padrão (**stdin**) usando indiretamente uma lista de argumentos variáveis.

**Protótipo:**

```
int vscanf(const char *restrict formato, va_list args)
```

**Parâmetros:**

- *formato* – *string* de formatação (v. **Apêndice B**).
- *args* – lista de argumentos variáveis contendo endereços de variáveis nas quais os valores lidos serão armazenados (v. **Capítulo 9**).

**Retorno:** O número de itens lidos e armazenados em variáveis; **EOF**, se o final de **stdin** for encontrado durante a leitura ou ocorrer algum erro de leitura.

**Observações:**

- As funções **vscanf()** e **scanf()** são semelhantes, pois ambas leem dados formatados em **stdin**.
- As funções **vscanf()** e **vfscanf()** são semelhantes, pois ambas possuem um argumento do tipo **va\_list**.
- Consulte também as descrições das funções **scanf()** e **vfscanf()** e o **Capítulo 9**.

**Exemplo:**

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

/****
*
* Função LeituraComPrompt(): incorpora apresentação de
*                               prompt com entrada de dados
*                               em stdin
*
* Argumentos: prompt (entrada) - a mensagem a ser
*                               apresentada ao usuário
*                               em stdout
*             formato (entrada) - string de formatação
*             ... (saída) - endereços de variáveis
*
* Retorno: o mesmo que scanf()
```

```

*
****/

int LeituraComPrompt( const char *prompt,
                      const char *formato, ... )
{
    int      nValoresAtribuidos;
    va_list args;

    /* Apresenta o prompt para o usuário */
    printf(prompt);
    fflush(stdout);

    /* Delega a tarefa restante para vscanf() */
    va_start(args, formato);
    nValoresAtribuidos = vscanf(formato, args);
    va_end(args);

    return nValoresAtribuidos;
}

int main(void)
{
    int      umInt, nValores;
    float umFloat;

    nValores = LeituraComPrompt( "\nDigite um inteiro e "
                                "um real: ", "%d %f",
                                &umInt, &umFloat );

    printf( "Numero de valores lidos e atribuidos: %d\n",
            nValores );

    return 0;
}

```

### 10.7.7 SAÍDA FORMATADA: A FAMÍLIA DE FUNÇÕES PRINTF

A família de funções printf consiste em um conjunto de funções com as seguintes propriedades comuns:

- Todas as funções funcionam de modo análogo à função **printf()**, escrevendo caracteres que representam valores de tipos de dados conhecidos de acordo com uma especificação de formato.
- Todas as funções usam um *string* de formatação como argumento e os especificadores de formato que podem ser utilizados no *string* de formatação de cada função são os mesmos (v. **Apêndice B**).
- Todas as funções retornam o número de caracteres escritos, quando não ocorre nenhum erro, ou um valor negativo, caso contrário.
- Todas as funções contêm *printf* na composição de seus nomes.

A família de funções printf é apresentada de forma resumida na **Tabela 10-8**.

FUNÇÃO	DESCRIÇÃO RESUMIDA
<b>fprintf()</b>	Escreve num <i>stream</i> de texto especificado.
<b>fwprintf()</b>	Escreve num <i>stream</i> especificado usando um <i>string</i> extenso de formatação.
<b>printf()</b>	Escreve no <i>stream</i> de saída padrão <b>stdout</b> .
<b>snprintf()</b> (C99)	Escreve num array de caracteres, limitando o número de caracteres escritos.
<b>sprintf()</b>	Semelhante à função <b>snprintf()</b> , mas não permite limitar o número de caracteres escritos.
<b>swprintf()</b>	Escreve, num array de caracteres extensos, dados formatados de acordo com um <i>string</i> extenso de formatação.
<b>vfprintf()</b>	Semelhante à função <b>fprintf()</b> , mas usa argumentos variáveis indiretamente por meio de um parâmetro do tipo <b>va_list</b> .
<b>vfwprintf()</b>	Escreve num <i>stream</i> usando um <i>string</i> extenso de formatação e uma lista de argumentos variáveis do tipo <b>va_list</b> .
<b>vprintf()</b>	Semelhante à função <b>printf()</b> , mas usa argumentos variáveis indiretamente por meio de um parâmetro do tipo <b>va_list</b> .
<b>vsnprintf()</b> (C99)	Semelhante à função <b>snprintf()</b> , mas usa argumentos variáveis indiretamente por meio de um parâmetro do tipo <b>va_list</b> .



FUNÇÃO	DESCRIÇÃO RESUMIDA
<b>vsprintf()</b>	Semelhante à função <b>sprintf()</b> , mas usa argumentos variáveis indiretamente por meio de um parâmetro do tipo <b>va_list</b> .
<b>vswprintf()</b>	Escreve num array de caracteres extensos usando um <i>string</i> extenso de formatação e uma lista de argumentos variáveis do tipo <b>va_list</b> .
<b>vwprintf()</b>	Escreve em <b>stdout</b> usando um <i>string</i> extenso de formatação e uma lista de argumentos variáveis do tipo <b>va_list</b> .
<b>wprintf()</b>	Escreve no meio de saída padrão <b>stdout</b> usando um <i>string</i> extenso de formatação.

Tabela 10-8: Família de funções printf.

Os membros da família printf que possuem um mesmo prefixo têm alguma afinidade entre si. O significado de cada prefixo (caractere) usado com funções da família printf é apresentado na **Tabela 10-9**. Estes prefixos podem ser usados de modo combinado (e.g., *vf* e *vsn*).

PREFIXO	SIGNIFICADO
<b>f</b>	A escrita é feita num <i>stream</i> especificado como argumento.
<b>s</b>	A escrita é feita num array de caracteres.
<b>v</b>	Um dos argumentos da função é do tipo <b>va_list</b> .
<b>n</b>	A função é capaz de limitar o número de caracteres escritos.
<b>w</b>	A função usa um <i>string</i> extenso de formatação; se a escrita for feita num array, seus elementos serão caracteres extensos.

Tabela 10-9: Prefixos utilizados em nomes de membros da família printf.

As funções **snprintf()**, **sprintf()**, **vsnprintf()** e **vsprintf()** são utilizadas em formatação em memória e serão apresentadas na **Seção 10.7.8**. As funções da família printf que lidam com caracteres e *strings* extensos serão apresentadas na **Seção 10.8**. As demais funções que compõem a família printf serão apresentadas a seguir.

*fprintf()***Incluir:** <stdio.h>**Descrição:** A função **fprintf()** imprime dados formatados num *stream* especificado.**Protótipo:**

```
int fprintf( FILE *restrict stream,
             const char *restrict formato, ... )
```

**Parâmetros:**

- *stream* – *stream* onde será feita a escrita.
- *formato* – *string* de formatação (v. **Apêndice B**).
- ... – representa os valores que serão escritos.

**Retorno:** O número de caracteres escritos no *stream* se não ocorrer nenhum erro; caso contrário, um valor negativo.

**Observações:**

- Os dados que serão impressos, representados por três pontos no protótipo da função, podem ser constantes, variáveis ou expressões.
- A única diferença entre as funções **fprintf()** e **printf()** é que a primeira permite a especificação do *stream* onde será feita a escrita, enquanto a segunda escreve sempre em **stdout**.

**Exemplo:**

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

int main()
{
    FILE *stream;
    time_t sec;
```

```

stream = fopen("Arq1.txt", "a");

if (stream) {
    time(&sec);
    fprintf( stream, "\nUltima atualizacao deste arquivo: %.24s",
             ctime(&sec) );
}

return 0;
}

```

### *printf()*

**Incluir:** <stdio.h>

**Descrição:** A função **printf()** escreve dados formatados no *stream* padrão **stdout**.

**Protótipo:**

```
int printf(const char *restrict formato, ...)
```

**Parâmetros:**

- *formato* – *string* de formatação (v. **Apêndice B**).
- ... – representa os dados que serão escritos.

**Retorno:** O número de caracteres escritos em **stdout** se não ocorrer nenhum erro; caso contrário, um valor negativo.

**Observações:**

- Os dados que serão impressos, representados por três pontos no protótipo da função, podem ser constantes, variáveis ou expressões.
- A única diferença entre as funções **fprintf()** e **printf()** é que a primeira permite a especificação do *stream* onde será feita a escrita, enquanto a segunda escreve sempre em **stdout**.

**Exemplo:** Existem inúmeros exemplos de uso de **printf()** ao longo do texto.

*vfprintf()*

**Incluir:** <stdio.h>

**Descrição:** A função **vfprintf()** escreve dados formatados num *stream* especificado usando indiretamente uma lista de argumentos variáveis.

**Protótipo:**

```
int vfprintf( FILE *restrict stream,
              const char *restrict formato,
              va_list args )
```

**Parâmetros:**

- *stream* – *stream* onde será feita a escrita.
- *formato* – *string* de formatação (v. **Apêndice B**).
- *args* – lista de argumentos variáveis contendo os valores que serão escritos.

**Retorno:** O número de caracteres escritos no *stream* se não ocorrer nenhum erro; caso contrário, um valor negativo.

**Observações:**

- A diferença entre as funções **vfprintf()** e **fprintf()** é que **fprintf()** usa diretamente argumentos variáveis, representados por três pontos em seu protótipo, enquanto **vfprintf()** usa argumentos variáveis indiretamente por meio de um argumento do tipo **va\_list**.
- Consulte o **Capítulo 9** para entender a diferença entre os usos direto e indireto de listas de argumentos variáveis.

**Exemplo:**

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

```
#include <stdlib.h>

/****
 *
 * Função AbortaComMensagem(): apresenta uma mensagem de erro
 *                             e aborta o programa
 *
 * Argumentos: formato (entrada) - string de formatação da
 *                             mensagem a ser impressa
 *             ... (entrada) - componentes da mensagem de erro
 *
 * Retorno: Nada
 *
 ****/

void AbortaComMensagem(char *formato, ...)
{
    va_list argumentos;

    va_start(argumentos, formato) ;
    vfprintf(stderr, formato, argumentos);
    fputc('\n', stderr);

    exit(1);
}

main()
{
    int x = 1, y = -1;

    if (x != y)
        AbortaComMensagem( "\nOs valores de x (%d) e y (%d) "
                           "deveriam ser iguais", x, y );

    return 0;
}

vprintf()
```

**Incluir:** <stdio.h>

**Descrição:** A função **vprintf()** escreve dados formatados na saída padrão **stdout** usando indiretamente uma lista de argumentos variáveis.

**Protótipo:**

```
int vprintf(const char *restrict formato, va_list args)
```

**Parâmetros:**

- `formato` – *string* de formatação (v. **Apêndice B**).
- `args` – lista de argumentos variáveis contendo os dados que serão escritos.

**Retorno:** O número de caracteres escritos em **stdout** se não ocorrer nenhum erro; caso contrário, um valor negativo.

**Observações:**

- A diferença entre as funções **vprintf()** e **printf()** é que **printf()** usa diretamente argumentos variáveis, representados por três pontos em seu protótipo, enquanto **vprintf()** usa argumentos variáveis indiretamente por meio de um argumento do tipo **va\_list**.
- Consulte o **Capítulo 9** para entender a diferença entre os usos direto e indireto de listas de argumentos variáveis.

**Exemplo:**

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

int SaidaFormatada(char *formato, ...)
{
    va_list      argumentos;
    int          retorno;
    const char *const msg = "\nResultado: ";

    printf(msg);

    va_start(argumentos, formato);
    retorno = vprintf(formato, argumentos);
    va_end(argumentos);

    return retorno;
}
```

```

int main()
{
    int    umInteiro = 32;
    float  umFloat = -4.9;

    SaidaFormatada("%d %f\n", umInteiro, umFloat);

    return 0;
}

```

## 10.7.8 FORMATAÇÃO EM MEMÓRIA

Nos cabeçalhos `<stdio.h>` e `<wchar.h>` são declaradas funções que permitem entrada e saída entre um programa e arrays de caracteres. Este processo é denominado **formatação em memória** e permite que o programa formate texto num *string* antes de enviá-lo para algum meio de saída.

As funções utilizadas em formatação em memória incluem quatro membros da família `scanf` e seis membros da família `printf`, que são os seguintes:

- `sscanf()`
- `swscanf()`
- `vsscanf()` (C99)
- `vswscanf()` (C99)
- `snprintf()` (C99)
- `sprintf()`
- `swprintf()`
- `vsprintf()` (C99)
- `vsnprintf()` (C99)
- `vswprintf()`

Membros da família `scanf` utilizados em formatação em memória (i.e., as quatro primeiras funções apresentadas ) retornam o número de variáveis que tiveram valores alterados, ou **EOF** se o final do *string* onde é feita a leitura for atingido.

Os membros da família `printf` utilizados em formatação em memória (i.e., as seis últimas funções apresentadas ) não levam em consideração o caractere '`\0`' quando retornam o número de caracteres escritos. Além disso, o valor retornado por uma função que limita o número de caracteres escritos [i.e., uma função que contém *n* no nome – e.g., **`snprintf()`**] pode ser utilizado para calcular o tamanho do array necessário para conter o resultado de uma operação de escrita.

Nesta seção, serão apresentadas as funções utilizadas em formatação em memória declaradas em `<stdio.h>`; enquanto aquelas utilizadas com a mesma finalidade declaradas em `<wchar.h>` serão apresentadas na **Seção 10.8**.

*sscanf()*

**Incluir:** `<stdio.h>`

**Descrição:** A função **`sscanf()`** lê dados num *string*.

**Protótipo:**

```
int sscanf( const char *restrict string,
            const char *restrict formato, ... )
```

**Parâmetros:**

- *string* – *string* onde será feita a leitura.
- *formato* – *string* de formatação (v. **Apêndice B**).
- ... – representa endereços de variáveis onde os valores lidos serão armazenados.

**Retorno:** O número de itens lidos quando a operação é bem sucedida; **EOF**, se o final do *string* for atingido prematuramente.

**Exemplo:**

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



```
#include <string.h>

/****
*
* Função ConverteDataParaISO(): converte um string de data
*                               no formato usado pela
*                               macro __DATE__ para o
*                               formato padrão ISO
*
* Exemplo: "Sep 16 1992" é convertida em "1992-09-16"
*
* Argumentos: strConversao (saída) - a data no formato ISO
*             data (entrada) - a data no formato da macro
*                               __DATE__
*
* Retorno: ponteiro para um string contendo a data no
*          formato ISO
*
****/

char *ConverteDataParaISO( char *strConversao,
                           const char *data )
{
    char *meses__DATE[12] = { "Jan", "Feb", "Mar", // Meses no
                              "Apr", "May", "Jun", // formato
                              "Jul", "Aug", "Sep", // __DATE__
                              "Oct", "Nov", "Dec" };

    char  strMes[4], /* Strings que contêm o mês, ... */
          strDia[3], /* o dia e ... */
          strAno[5]; /* o ano no formato __DATE__ */
    int   i;

    /* Separa a data em strings */
    /* contendo dia, mês e ano */
    sscanf(data, "%s %s %s", strMes, strDia, strAno);

    /* Encontra mês e insere a ordem (1-12) no string */
    for ( i = 0 ; i < 12; ++i) {
        if (!strcmp(strMes, meses__DATE[i])) {
            sprintf(strMes, "%0.2i", i + 1 );
            break;
        }
    }
}
```

```

    }

    /* A função deveria verificar se a data é */
    /* válida antes de prosseguir, mas, neste */
    /* exemplo, isto não é importante.      */

    /* Combina data, mes e ano no          */
    /* string que contém a conversão */
    sprintf( strConversao, "%04.4s/%02.2s/%02.2s",
            strAno, strMes, strDia );

    return strConversao;
}

/****
 *
 * Função main(): imprime a data de compilação
 *               deste arquivo no formato ISO
 *
 ****/

int main(void)
{
    char  dataISO[11];

    printf( "\nA data de compilacao no formato ISO e': %s\n",
            ConverteDataParaISO(dataISO, __DATE__) );

    return 0;
}

```

### *vsscanf()* (C99)

**Incluir:** <stdio.h>

**Descrição:** A função **vsscanf()** lê dados formatados num *string* usando um parâmetro que representa uma lista de argumentos variáveis.

**Protótipo:**

```
int vsscanf( const char *restrict string,
            const char *restrict formato,
            va_list args )
```

**Parâmetros:**

- *string* – *string* onde será feita a leitura.
- *formato* – *string* de formatação (v. **Apêndice B**).
- *args* – lista de argumentos variáveis representando endereços de variáveis onde os valores lidos serão armazenados

**Retorno:** Número de itens lidos, quando a operação é bem sucedida; **EOF**, se o final do *string* for atingido prematuramente.

**Observações:**

- A diferença entre as funções **vsscanf()** e **sscanf()** é que **sscanf()** utiliza uma lista de argumentos variáveis (representada por três pontos no protótipo) diretamente, enquanto **vsscanf()** usa uma lista de argumentos variáveis indiretamente por meio de um parâmetro do tipo **va\_list**. Esta diferença reflete-se no modo como estas funções são utilizadas.
- Consulte também a descrição da função **sscanf()** e o **Capítulo 9**.

**Exemplo:**

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

int LeituraEmString(const char *s, const char *formato, ...)
{
    int      nValoresAtribuidos;
    va_list args;

    printf("\nString onde sera' feita a leitura: %s\n", s);

    va_start(args, formato);
```

```

    nValoresAtribuidos = vsscanf(s, formato, args);
    va_end(args);

    return nValoresAtribuidos;
}

int main(void)
{
    int    umInt, nValoresLidos;
    float  umFloat;
    char   entrada[] = "12 3.1415 X algo mais";

    nValoresLidos = LeituraEmString( entrada, "%d %f",
                                     &umInt, &umFloat );

    printf( "\nNumero de valores lidos e atribuidos: %d",
            nValoresLidos );

    if (2 == nValoresLidos)
        printf( "\n\nValores lidos: %d e %f\n",
                umInt, umFloat );

    return 0;
}

```

### *snprintf()* (C99)

**Incluir:** <stdio.h>

**Descrição:** A função **snprintf()** escreve dados formatados num array limitando o número de caracteres escritos.

**Protótipo:**

```

int snprintf( char *restrict array, size_t n,
              const char *restrict formato, ... )

```

**Parâmetros:**

- **array** – array onde será feita a escrita ou **NULL** (v. a seguir).
- **n** – número máximo de caracteres que serão escritos no array, incluindo o caractere terminal de *string* `'\0'`.

- `formato` – *string* de formatação (v. **Apêndice B**).
- `...` – representa os valores que serão escritos no array de acordo com o *string* de formatação.

**Retorno:** O número de caracteres, sem incluir o caractere terminal `'\0'`, que são escritos no array se `n` for suficientemente grande.

#### Observações:

- A função **`snprintf()`** deve, sempre que disponível<sup>102</sup>, ser usada preferencialmente com relação à função **`sprintf()`** como proteção contra corrupção de memória.
- Um valor de retorno maior do que ou igual a `n` indica que a função não armazenou todos os caracteres que deveriam ter sido armazenados no array.
- Se `n` for igual a zero, o primeiro argumento pode ser **`NULL`**. Usando-se zero como segundo argumento, pode-se determinar o tamanho do array necessário para conter o resultado e, então, alocar dinamicamente este array no tamanho exato, como mostra o trecho de programa a seguir:

```
char                *p,
const char *const   formato = "\n%4.0d\t%4.2f\t%8.3f\n";
int                x, tamanho;
float              y;
long double        z;
...
tamanho = sprintf(NULL, 0, formato, x, y, z);
p = malloc(tamanho + 1);
sprintf(p, tamanho + 1, formato, x, y, z);
```

**Exemplo 1:** O programa a seguir demonstra o uso da função **`snprintf()`**.

```
#include <stdio.h>

#define MAX 50

int main()
{
    char    ar[MAX];
```

---

102

A função **`snprintf()`** foi introduzida pelo padrão C99.

```

double  x = -123.45,
        y = 67.89,
        z = 123.01,
        resultado;
int      carEscritos = 0;

resultado = x * y + z;

carEscritos = snprintf( ar, MAX,
                        "Usando os valores %f, %f"
                        " e %f, o resultado foi %f.\n",
                        x, y, z, resultado );

#ifdef TESTE
    printf("\nValor retornado = %d\n\n", carEscritos);
#endif

    printf("Conteudo do array:\n\t\"%s\"", ar);

    fflush(stdout);

    if ( carEscritos >= MAX )
        fprintf( stderr, "\n\nO string resultante foi truncado:\n\t"
                 "%d caracteres nao foram escritos no array\n",
                 carEscritos - MAX + 1 );

    return 0;
}

```

### Resultado da execução do programa no Linux:

Valor retornado = 85

Conteudo do array:

"Usando os valores -123.450000, 67.890000 e 123.0"

O string resultante foi truncado:

36 caracteres nao foram escritos no array

**Exemplo 2:** O programa a seguir demonstra novamente o uso da função **snprintf()**.

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

int main()
{
    char    *ar;
    double  x = -123.45,
            y = 67.89,
            z = 123.01,
            resultado;

    int     carEscritos = 0;
    char    *formato = "Usando os valores %f, %f"
                      " e %f,\no resultado foi %f.";

    resultado = x * y + z;

    /* Calcula o espaço necessário */
    /* para conter o string resultante */
    carEscritos = snprintf( NULL, 0, formato,
                           x, y, z, resultado );

    ar = malloc(carEscritos + 1);

    carEscritos = snprintf( ar, carEscritos + 1, formato,
                           x, y, z, resultado );

#ifdef TESTE
    printf("\nValor retornado = %d\n\n", carEscritos);
#endif

    printf("Conteudo do array:\n\"%s\"\n", ar);

#ifdef TESTE
    printf( "\nComprimento do string no array = %d\n\n",
            strlen(ar) );
#endif

    return 0;
}
```

Resultado da execução do programa no Linux:

```
Valor retornado = 84
```

```
Conteúdo do array:
```

```
"Usando os valores -123.450000, 67.890000 e 123.010000,  
o resultado foi -8258.010500."
```

```
Comprimento do string no array = 84
```

Observe que o segundo exemplo de **snprintf()** é semelhante ao primeiro, mas, enquanto no primeiro exemplo o *string* é truncado, no segundo exemplo, o *string* é integralmente armazenado no array.

**NB:** Os dois últimos exemplos foram compilados usando o compilador gcc 4.3.2 e a biblioteca glib 2.8 (libc6) no sistema Ubuntu 8.10. O uso de uma versão anterior da biblioteca glib pode produzir um resultado bem diferente porque, em algumas destas versões, a função **snprintf()** não foi corretamente implementada de acordo com o padrão ISO C99.

*sprintf()*

**Incluir:** <stdio.h>

**Descrição:** A função **sprintf()** escreve num array de caracteres dados formatados de acordo com um *string* de formatação, acrescentando sempre o caractere terminal '\0' ao final do processo de escrita.

**Protótipo:**

```
int sprintf( char *restrict array,  
             const char *restrict formato, ...)
```

**Parâmetros:**

- *array* – array onde será feita a escrita.
- *formato* – *string* de formatação (v. **Apêndice B**).
- ... – representa valores que serão escritos no array de acordo com o *string* de formatação.



**Retorno:** O número de caracteres escritos no array, sem incluir o caractere terminal `'\0'`.

### Observações:

- O uso desta função não é recomendado, uma vez que ela pode escrever além do limite do array. Em vez de usá-la, é preferível usar **snprintf()**, pois esta função permite limitar o número máximo de caracteres que serão escritos no array.
- Um erro comum no uso de **sprintf()** é achar que o uso de um especificador de largura (v. **Apêndice B**) pode resolver o problema de corrupção de memória, mas isto não ocorre, porque a largura especifica um valor mínimo, e não um valor máximo como se poderia supor. Por exemplo, na chamada da função **sprintf()** a seguir, o valor do especificador de largura (i.e, 4) informa à função que, pelo menos, quatro caracteres devem ser escritos na variável apontada por `ar`.

```
sprintf(ar, "%4d", n);
```

### Exemplo:

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

/****
 *
 * Função HoraPadrao(): converte uma hora no formato da
 *                      macro __TIME__ no formato HHMMSS
 *
 * Exemplo: "10:49:31" é transformada em "104931"
 *
 * Parâmetros: hora (entrada) - string no formato da macro
 *                      __TIME__ (HH:MM:SS)
 *                      horaPadrao (saída) - string no formato HHMMSS
 *
 * Retorno: hora no formato HHMMSS ou NULL, se ocorrer erro
 *
 ****/

char *HoraPadrao(char *horaPadrao, const char *hora)
{
    char strAux[12]; /* String auxiliar de conversão */
    char h[3], /* Hora */
          m[3], /* Minutos */
```

```

        s[3]; /* Segundos */
    int  teste;

    teste = sscanf(hora, "%2s %*c %2s %*c %2s", h, m, s);

    if (teste != 3)
        return NULL;

    /* Divide o string em horas, minutos e segundos */
    sprintf(horaPadrao, "%s%s%s", h,m,s);

    return horaPadrao;
}

int main(void)
{
    char str[7];

    printf( "\nHora de compilacao no formato HH:MM:SS: %s",
        __TIME__ );
    printf( "\nHora de compilacao no formato HHMMSS: %s\n",
        HoraPadrao(str, __TIME__) );

    return 0;
}

```

### *vsprintf()*

**Incluir:** <stdio.h>

**Descrição:** A função **vsprintf()** escreve num array de caracteres dados formatados de acordo com um *string* de formatação utilizando uma lista de argumentos variáveis como parâmetro.

**Protótipo:**

```

int vsprintf( char *restrict array,
              const char *restrict formato,
              va_list args )

```

**Parâmetros:**

- `array` – array onde será feita a escrita.
- `formato` – *string* de formatação (v. **Apêndice B**).
- `args` – lista de argumentos variáveis contendo os valores que serão escritos no array.

**Retorno:** O número de caracteres escritos no array, sem incluir o caractere `'\0'`.

**Observações:**

- A diferença entre as funções **`vsprintf()`** e **`sprintf()`** é que esta última usa uma lista de argumentos variáveis diretamente, enquanto **`vsprintf()`** usa uma lista de argumentos variáveis indiretamente por meio de um parâmetro do tipo **`va_list`**.
- A função **`vsprintf()`** não permite limitar o número de caracteres escritos no array recebido como argumento. Por isso, o programador deve dar preferência ao uso da função **`vsnprintf()`**, apresentada adiante.
- Consulte também as descrições de **`sprintf()`** e **`vsnprintf()`**, e o **Capítulo 9**.

**Exemplo:**

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

#define TAMANHO 80

/*****
 *
 * Função Acrescenta(): acrescenta um string formatado
 *                       ao final de outro string usando
 *                       a função vsprintf()
 *
 * Argumentos: str (entrada/saída) - string que terá outro
 *                       acrescentado
 *                       formato (entrada) - string de formatação
 *                       ... (entrada) - dados que serão formatados
 */
```

```

*                                     pelo string de formatação
*
* Retorno: ponteiro para str
*
****/

char *Acrescenta(char *str, const char *formato, ...)
{
    va_list args;

    if (str && formato) {
        va_start(args, formato);
        vsprintf(str + strlen(str), formato, args);
        va_end(args);
    }

    return str;
}

int main(void)
{
    char str[TAMANHO + 1] = "\nValor digitado: ";
    int numero, teste;

    printf("\nDigite um numero inteiro: ");
    teste = scanf("%d", &numero);

    if (!teste) {
        printf("\nVoce nao digitou um numero inteiro.\n");
        return 1;
    }

    Acrescenta(str, "%d.\n", numero);

    puts(str);

    return 0;
}

```

*vsprintf()* (C99)

**Incluir:** <stdio.h>

**Descrição:** A função **vsnprintf()** escreve num array de caracteres dados formatados de acordo com um *string* de formatação utilizando uma lista de argumentos variáveis como parâmetro e permitindo limitar o número de caracteres escritos.

**Protótipo:**

```
int vsnprintf( char *restrict array, size_t n,
               const char *restrict formato,
               va_list args )
```

**Parâmetros:**

- **array** – array onde será feita a escrita.
- **n** – número máximo de caracteres que serão escritos, incluindo o caractere terminal de *string* ('\\0').
- **formato** – *string* de formatação (v. **Apêndice B**).
- **args** – lista de argumentos variáveis contendo os valores que serão escritos no array.

**Retorno:** O número de caracteres, sem incluir o caractere '\\0', escritos no array se **n** for suficientemente grande.

**Observações:**

- A diferença entre as funções **vsnprintf()** e **snprintf()** é que esta última usa uma lista de argumentos variáveis diretamente, enquanto **vsnprintf()** usa uma lista de argumentos variáveis indiretamente por meio de um parâmetro do tipo **va\_list**.
- Consulte também a descrição da função **vsprintf()** e o **Capítulo 9**.

**Exemplo:**

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

#define TAM_ARRAY 20
```

```

void ArmazenaSaida(const char *str, const char *formato,...)
{
    va_list args;
    int      n;

    va_start(args, formato);

    n = vsnprintf(str, TAM_ARRAY, formato, args);

    if (n >= TAM_ARRAY)
        fprintf(stderr, "\nO tamanho do array deveria ser:"
                    " %d\n", n );

    va_end(args);
}

int main(void)
{
    char string[TAM_ARRAY];
    char formato[] = "%s %s %s\n";

    ArmazenaSaida( string, formato, "Janeiro", "Fevereiro",
                    "Marco" );

    printf("\nMeses de verao: %s\n\n", string);

    return 0;
}

```

### 10.7.9 FUNÇÕES DE POSICIONAMENTO (ACESSO DIRETO)

Existem cinco funções declaradas em `<stdio.h>` que podem ser utilizadas na movimentação do apontador de posição de um arquivo. Estas funções são essenciais em processamento de arquivos com acesso direto e serão descritas a seguir.

*fseek()*

**Incluir:** `<stdio.h>`

**Descrição:** A função **fseek()** move o apontador de posição de um arquivo para um local especificado por seus argumentos.

**Protótipo:**

```
int fseek(FILE *stream, long int distancia, int deOnde)
```

**Parâmetros:**

- *stream* – *stream* associado a um arquivo que suporta acesso direto.
- *distancia* – deslocamento, medido a partir do terceiro argumento, aplicado ao apontador de posição do arquivo.
- *deOnde* – posição no arquivo a partir de onde a distância (segundo argumento) será medida. Este argumento pode assumir o valor de uma das macros apresentados na **Tabela 10-10**.

MACRO	REPRESENTA...
<b>SEEK_SET</b>	o início do arquivo
<b>SEEK_CUR</b>	a posição corrente do apontador de posição
<b>SEEK_END</b>	o final do arquivo

Tabela 10-10: Macros de posicionamento em arquivos.

**Retorno:** Zero, quando a função é bem sucedida; um valor diferente de zero, quando ela não consegue deslocar o apontador de posição do arquivo para a posição desejada.

**Observações:**

- Os bytes num arquivo são indexados a partir de zero.
- Esta função zera o indicador de final de arquivo e desfaz qualquer efeito da função **ungetc()** (v. **Seção 10.7.13**) no *stream* especificado.
- Se a macro **SEEK\_END** for utilizada como valor do terceiro argumento de **fseek()** e o arquivo tiver sido aberto apenas para leitura, a distância (segundo argumento) deve ser negativa. Por outro lado, se a macro **SEEK\_SET** for utilizada, a distância deve ser sempre positiva, independentemente do modo de abertura do arquivo.

- Em *streams* binários, a distância utilizada com **fseek()** pode ser qualquer valor inteiro que não faça o apontador de posição do arquivo ultrapassar os limites do arquivo (de acordo com o modo de abertura). Para *streams* de texto, o segundo argumento de **fseek()** deve ser zero ou um valor retornado por **ftell()** (v. adiante) para o mesmo *stream*.
- Pode-se testar se um arquivo permite acesso direto usando-se o seguinte trecho de programa:

```
int teste = fseek(stream, 0L, SEEK_CUR);

if (!teste) { /* O arquivo permite acesso direto */
    ...
} else { /* O arquivo NÃO permite acesso direto */
    clearerr(stream); /* Elimina a condição de erro */
                    /* provocada pela chamada de fseek() */
    ... /* Acesso sequencial apenas */
}
```

**Observação:** Consulte também a descrição da função **ftell()**.

### Exemplo:

```
#include <stdio.h>

typedef struct {
    long    ID;
    double  valor;
} tRegistro;

void CriaArquivoDeRegistros()
{
    tRegistro registros[] = { {32L, -2.5},
                              {56L, 4.23},
                              {22L, -0.34},
                              {123L, 5.44},
                              {13L, 7.22} };

    FILE *stream;

    if ( !(stream = fopen("Registros.dat", "wb")) ) {
        perror("Nao foi possivel criar arquivo de registros");
    }
}
```



```

        return;
    }

    fwrite( registros, sizeof(tRegistro),
           sizeof(registros)/sizeof(registros[0]), stream );

    fclose(stream);
}

int main()
{
    FILE      *stream;
    tRegistro  umRegistro;
    long       chave = 123L; /* Valor de ID procurado */

    CriaArquivoDeRegistros();

    if ( !(stream = fopen("Registros.dat", "rb")) )
        perror("Nao foi possivel abrir arquivo de registros");
    else
        do { /* Procura registro com ID = chave */
            if (fread(&umRegistro.ID, sizeof(long), 1, stream)
                < 1) {
                fprintf( stderr, "Registro com ID %ld nao foi"
                        " encontrado\n", chave );

                break;
            } else /* Salta o resto do registro */
                if (fseek( stream,
                           sizeof(tRegistro) - sizeof(long),
                           SEEK_CUR )) {
                    perror("fseek() falhou");
                    break;
                }
        } while (umRegistro.ID != chave);

    if (umRegistro.ID == chave)
        printf("\nA chave %ld foi encontrada\n", chave);

    return 0;
}

```

*ftell()***Incluir:** <stdio.h>

**Descrição:** A função **ftell()** informa onde se encontra correntemente o apontador de posição do arquivo associado ao *stream* especificado como argumento.

**Protótipo:**

```
long ftell(FILE *stream)
```

**Parâmetro:** *stream* – *stream* cuja posição se deseja consultar.

**Retorno:** Se a operação for bem sucedida, a posição corrente do apontador de posição do arquivo associado ao *stream* recebido como argumento; caso contrário, -1L.

**Observações:**

- Esta função é frequentemente usada para guardar o valor corrente do apontador de posição de modo que se possa, posteriormente, retornar àquela posição após a execução de alguma operação de entrada ou saída.
- A posição retornada por **ftell()** é sempre medida a partir do início do arquivo.
- Em *streams* binários, o valor retornado por **ftell()** representa o número de bytes a partir do início do arquivo.
- Em *streams* de texto, o valor retornado por **ftell()** representa um valor dependente de implementação que faz sentido apenas quando utilizado como distância numa chamada subsequente da função **fseek()**.

**Exemplo:** O programa a seguir tenta encontrar uma palavra num arquivo de texto. Se ela for encontrada, o arquivo será impresso a partir da linha que a contém; caso contrário, o arquivo será impresso a partir de seu início. Além de **ftell()**, este programa usa também a função **rewind()** (v. adiante).

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

#define MAX_LINHA 256
#define NOME_ARQ "Teste.txt"
```

```

void ConstroiArqTeste()
{
    FILE *stream;
    char *conteudo[] = { "bola", "bala", "bela",
                        "balde", "bula", "bilhar" };

    int i,
        nElementos = sizeof(conteudo)/sizeof(conteudo[0]);

    if ( !(stream = fopen(NOME_ARQ, "w")) )
        return;

    for (i = 0; i < nElementos; ++i)
        fprintf(stream, "%s\n", conteudo[i]);

    fclose(stream);
}

int main(int argc, char **argv)
{
    FILE *stream;
    long posicao = 0L;
    char linha[MAX_LINHA];
    char *p, palavra[MAX_LINHA];
    int c, numLinha = 0;

    ConstroiArqTeste();

    printf("\nDigite a palavra a ser procurada: ");
    p = fgets(palavra, MAX_LINHA, stdin);

    if (!p && *p != '\n') {
        fprintf(stderr, "\nEntrada incorreta. Bye.\n");
        return 1;
    }

    /* Se o caractere de quebra de linha */
    /* foi incluído no string, remove-o */
    if (palavra[strlen(palavra) - 1] == '\n')
        palavra[strlen(palavra) - 1] = '\0';

    if ( !(stream = fopen(NOME_ARQ, "r")) ) {
        fprintf(stderr, "\nNao pode abrir o arquivo\n");
    }
}

```

```

        return 1;
    }

do {
    /* Marca o inicio da linha corrente */
    if ( (posicao = ftell(stream)) == -1L ) {
        fprintf( stderr, "\nNao foi possivel obter "
                " a posicao no arquivo\n" );
        return 1;
    }

    numLinha++;

    /* Lê a proxima linha do arquivo */
    if (!fgets(linha, MAX_LINHA, stream))
        break;
} while (!strstr(linha, palavra));

/* Ou a palavra foi encontrada ou fim de arquivo */
if ( feof( stream ) ) {
    fprintf( stderr, "\nNao foi possivel encontrar "
            "\"%s\" em %s\n", palavra, NOME_ARQ );
    rewind(stream); /* Volta ao início do arquivo */
} else {
    printf( "\nA palavra \"%s\" foi encontrada "
            "na linha %d do arquivo \"%s\"\n",
            palavra, numLinha, NOME_ARQ );

    /* Move o apontador de arquivo para */
    /* o inicio da linha contendo a chave */
    fseek( stream, posicao, 0 );
}

/* Se a palavra foi encontrada, imprime o */
/* arquivo a partir da linha que contém a */
/* palavra. Caso contrário, imprime todo */
/* conteúdo do arquivo em stdout. */

printf( "\nConteúdo do arquivo a partir de seu inicio "
        "ou da palavra encontrada:\n\n" );

while((c = fgetc(stream)) != EOF)
    putchar(c);

```

```

    fclose(stream);

    return 0;
}

```

## *fgetpos()*

**Incluir:** <stdio.h>

**Descrição:** A função **fgetpos()** informa onde se encontra correntemente o apontador de posição do arquivo associado ao *stream* especificado como argumento.

**Protótipo:**

```
int fgetpos(FILE *restrict stream, fpos_t *restrict posicao)
```

**Parâmetros:**

- *stream* – *stream*, que permite acesso direto, cuja posição se deseja consultar.
- *posicao* – endereço da variável onde será armazenado o valor da posição corrente do apontador de posição do arquivo.

**Retorno:** Um valor diferente de zero, se ocorrer algum erro; zero, caso contrário.

**Observações:**

- O valor armazenado na variável cujo endereço é passado como segundo parâmetro é útil apenas como parâmetro numa chamada subsequente de **fsetpos()**.
- A função **fgetpos()** é semelhante à função **ftell()**, mas há importantes diferenças que serão apresentadas na descrição da função **fsetpos()**.
- Consulte a descrição da função **fsetpos()**.

**Exemplo:** Veja o exemplo apresentado para a função **fsetpos()**.

*fsetpos()*

**Incluir:** <stdio.h>

**Descrição:** A função **fsetpos()** move o apontador de posição de um arquivo para um novo local obtido por meio de uma chamada prévia de **fgetpos()**.

**Protótipo:**

```
int fsetpos(FILE *stream, const fpos_t *posicao)
```

**Parâmetros:**

- *stream* – *stream* aberto que permite acesso direto.
- *posicao* – endereço de uma variável do tipo **fpos\_t** cujo valor foi obtido por meio de uma chamada de **fgetpos()**.

**Retorno:** Zero, se a função obtém êxito; um valor diferente de zero, caso contrário.

**Observações:**

- Esta função zera o indicador de final de arquivo e desfaz qualquer efeito da função **ungetc()** (v. **Seção 10.7.13**) no *stream* especificado.
- Quando falha, esta função atribui um valor diferente de zero à variável global **errno**.
- As funções **fgetpos()** e **fsetpos()** são parecidas, respectivamente, com as funções **ftell()** e **fseek()** apresentadas antes. As principais diferenças entre estes pares de funções são:
  - **ftell()** e **fseek()** usam o tipo **long int** para representar deslocamentos num arquivo, enquanto **fgetpos()** e **fsetpos()** usam o tipo **fpos\_t** para representá-los. Este tipo é escolhido de modo a permitir deslocamentos arbitrariamente grandes. Por outro lado, os posicionamentos permitidos pela funções **ftell()** e **fseek()** são limitados pelo tamanho do tipo **long int**. O padrão C99 não garante que o tipo **fpos\_t** seja inteiro.
  - A função **fseek()** permite que se especifique a origem do deslocamento, enquanto os deslocamentos promovidos pela função **fsetpos()** são sempre medidos a partir do início do arquivo.

- o **fgetpos()** e **fsetpos()** levam em consideração o estado de mudança de um *stream* de caracteres multibytes com estado (v. **Seção 7.3.1**), enquanto **ftell()** e **fseek()** não têm obrigação de fazer isso. Isto é, quando se move o apontador de posição de um arquivo usando **fsetpos()**, esta função atualiza o indicador de estado de mudança do *stream*, de modo que a próxima operação de leitura seja informada sobre qual é o estado de mudança prevalente para os próximos caracteres.
- Consulte também **fseek()** e **ftell()**.

**Exemplo:**

```
#include <stdio.h>

#define MAX_LINHA 256

int main()
{
    FILE    *stream;
    fpos_t   marca;
    char    *p, linha[MAX_LINHA];
    int     i, c;

    if ( !(stream = fopen("Teste.txt", "r+")) ) {
        fprintf(stderr, "\nNao foi possivel abrir arquivo\n");
        return 1;
    }

    /* Salta três linhas no arquivo */
    for (i = 0; i < 3; ++i)
        fgets(linha, MAX_LINHA, stream);

    /* Guarda a posição corrente */
    if (fgetpos(stream, &marca)) {
        perror("\nGuardando posicao no arquivo");
        return 1;
    }

    /* Imprime o conteúdo do arquivo na */
    /* tela a partir da quarta linha    */
    printf( "\nConteudo do arquivo a partir "
```

```

        "da quarta linha:\n\n" );

while((c = fgetc(stream)) != EOF)
    putchar(c);

/* Retorna à posição guardada */
if (fsetpos(stream, &marca)) {
    perror("\nRestaurando posicao no arquivo");
    return 1;
}

/* Redireciona a entrada */
/* padrão para um arquivo */
freopen("TesteCopia.txt", "w", stdout);

/* Imprime o conteúdo do arquivo em */
/* arquivo a partir da quarta linha */
printf( "Conteudo do arquivo a partir "
        "da quarta linha:\n\n" );

while((c = fgetc(stream)) != EOF)
    putchar(c);

fclose(stream);

return 0;
}

```

*rewind()*

**Incluir:** <stdio.h>

**Descrição:** A função **rewind()** retorna o apontador de posição de um arquivo para o início do arquivo.

**Protótipo:**

void rewind(FILE *stream)
---------------------------

**Parâmetro:** *stream* – *stream* aberto que permite acesso direto.



**Observações:**

- Esta função zera os indicadores de erro e de final de arquivo do *stream* recebido como argumento.
- Consulte também **fseek()**.

**Exemplo:** O programa a seguir imprime em **stdout** o conteúdo de um arquivo-texto duas vezes. Na primeira vez, as letras são impressas como minúsculas e, na segunda vez, elas são impressas como maiúsculas.

```
#include <stdio.h>

int main(int argc, char** argv)
{
    FILE *stream;
    int c;

    if ( !(stream = fopen("Teste.txt", "r")) ) {
        fprintf(stderr, "\nNao foi possivel abrir arquivo\n");
        return 1;
    }

    printf("\nConteudo do arquivo em letras minusculas:\n" );

    while ((c = fgetc(stream)) != EOF)
        putchar(tolower(c));

    rewind( stream );

    printf("\nConteudo do arquivo em letras maiusculas:\n" );

    while ((c = fgetc(stream)) != EOF)
        putchar(toupper(c));

    fclose( stream );

    return 0;
}
```

## 10.7.10 GERENCIAMENTO DE ARQUIVOS

*remove()*

**Incluir:** <stdio.h>

**Descrição:** A função **remove()** exclui o arquivo cujo nome é passado como argumento.

**Protótipo:**

```
int remove(const char *nomeDoArquivo)
```

**Parâmetro:** nomeDoArquivo – nome do arquivo a ser removido.

**Retorno:** Zero, quando a função obtém êxito; um valor diferente de zero, caso contrário.

**Observação:** O resultado da operação será dependente de implementação se o arquivo a ser removido estiver aberto.

**Exemplo:** O programa a seguir permite que se especifique, em linha de comando, o nome de um arquivo a ser excluído utilizando a função **remove()**.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    if (argc != 2) {
        printf( "\nErro: este programa deve ser "
                "usado assim:\n\n\t%s "
                "<arquivo-a-ser-apagado>\n", argv[0]);
        return 1;
    }

    /* Exclui o arquivo */
    if(remove(argv[1]))
        printf( "\nOcorreu um erro ao tentar remover "
                "o arquivo \"%s\"\n", argv[1] );
    else
        printf("\nArquivo \"%s\" foi removido\n", argv[1]);
}
```

```

    return 0;
}

```

## *rename()*

**Incluir:** <stdio.h>

**Descrição:** A função **rename()** altera o nome de um arquivo.

### **Protótipo:**

```

int rename( const char *nomeAtual,
            const char *nomeNovo )

```

### **Parâmetros:**

- `nomeAtual` – nome atual do arquivo a ser renomeado.
- `nomeNovo` – nome que o arquivo terá após ser renomeado.

**Retorno:** Zero, quando a função obtém êxito; um valor diferente de zero, caso contrário.

**Observação:** Se o arquivo a ser renomeado estiver aberto ou se já existir um arquivo com o novo nome especificado como argumento, o resultado será dependente de implementação.

**Exemplo:** O programa a seguir utiliza a função **rename()** para rebatizar um arquivo. O nome atual do arquivo e novo nome desejado para ele são passados como argumentos para o programa.

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    if (argc != 3) {
        printf( "\nErro: este programa deve "
                "ser usado assim:\n\n\t%s"
                " <nome-atual> <novo-nome>\n", argv[0] );
        return 1;
    }
}

```

```

    }

    /* Renomea o arquivo */
    if(rename(argv[1], argv[2]))
        printf( "\nOcorreu um erro ao tentar "
                "renomear o arquivo \"%s\"\n", argv[2] );
    else
        printf( "\nO arquivo \"%s\" foi renomeado "
                "para \"%s\"\n", argv[1], argv[2] );

    return 0;
}

```

## 10.7.11 ARQUIVOS TEMPORÁRIOS

*tmpfile()*

**Incluir:** <stdio.h>

**Descrição:** A função **tmpfile()** cria um arquivo temporário e abre-o para atualização no modo "w+b".

**Protótipo:**

FILE *tmpfile(void)
---------------------

**Retorno:** Um ponteiro para o *stream* associado ao arquivo recém-criado, se não ocorrer nenhum erro; ou **NULL**, se o arquivo não puder ser criado.

**Observação:** Se o caminho que leva ao arquivo temporário não for modificado após a criação do arquivo, este arquivo será removido pelo sistema quando for fechado ou quando o programa terminar.

**Exemplo:**

```

#include <stdio.h>

int main()
{
    FILE *stream,

```

```

        *streamTmp;
int    c;

if (!(stream = fopen("Arq1.txt", "r"))) {
    fprintf(stderr, "\nNao foi possivel abrir arquivo");
    return 1;
}

if (!(streamTmp = tmpfile())) {
    fprintf( stderr,
             "\nNao foi possivel abrir arquivo temporario");
    return 1;
}

while ((c = fgetc(stream)) != EOF)
    if (fputc(c, streamTmp) == EOF)
        break;

fclose(stream);

/* O conteúdo armazenado no arquivo temporário */
/* seria processado aqui. Neste exemplo, ocorre */
/* apenas a impressão deste conteúdo na tela. */

rewind(streamTmp);

while((c = fgetc(streamTmp)) != EOF)
    putchar(c);

fclose(streamTmp);

return 0;
}
tmpnam()

```

**Incluir:** <stdio.h>

**Descrição:** A função **tmpnam()** cria um *string* que pode ser utilizado como nome de um arquivo temporário.

**Protótipo:**

```
char *tmpnam(char *nomeDoArquivo)
```

**Parâmetro:** `nomeDoArquivo` – **NULL** ou um array, capaz de conter pelo menos **L\_tmpnam** caracteres, que receberá o nome do arquivo.

**Retorno:** Quando o argumento não é **NULL**, o endereço do array recebido como argumento. Caso contrário, o endereço de um array de duração fixa local à função contendo o nome do arquivo gerado.

**Observações:**

- A macro **L\_tmpnam** especifica o tamanho mínimo que um array deve ter para ser capaz de conter um nome de arquivo criado por esta função, enquanto a macro **TMP\_MAX** especifica o número mínimo de nomes de arquivos distintos criados pela função. Ambas as macros são definidas em `<stdio.h>`.
- A função **tmpnam()** gera apenas nomes de arquivo; i.e., ela não cria, abre ou remove arquivos.
- Quando o argumento é **NULL**, uma chamada subsequente da função pode modificar o conteúdo do array cujo endereço é retornado.
- No intervalo entre a geração de um nome de arquivo temporário e a criação do arquivo com este nome, pode ser que algum outro processo, executado paralelamente com o programa, crie um arquivo temporário com o mesmo nome.
- Arquivos temporários criados com o auxílio da função **tmpnam()** não são removidos automaticamente quando o programa termina, de modo que é responsabilidade do programador excluir arquivos assim criados [e.g., usando **remove()**].
- Em termos práticos, é mais recomendável usar a função **tmpfile()** para a criação de arquivos temporários.

**Exemplo:**

```
#include <stdio.h>

int main()
{
    char nome[L_tmpnam];
```

```

FILE *stream,
      *streamTmp;
int   c;

    /* Abre arquivo principal */
if (!(stream = fopen("Arq1.txt", "r"))) {
    fprintf(stderr, "\nNao foi possivel abrir arquivo\n");
    return 1;
}

    /* Determina nome de arquivo temporário */
tmpnam(nome);

#ifdef TESTE
    printf("\nNome do arquivo temporario: %s\n", nome);
#endif

    /* Abre arquivo temporário */
if (!(streamTmp = fopen(nome, "w+"))) {
    fprintf( stderr, "\nNao foi possivel abrir"
              " arquivo temporario\n" );
    return 1;
}

while ((c = fgetc(stream)) != EOF)
    if (fputc(c, streamTmp) == EOF)
        break;

fclose(stream);

    /* O conteúdo armazenado no arquivo temporário */
    /* seria processado aqui. Neste exemplo, ocorre */
    /* apenas a impressão deste conteúdo na tela. */

rewind(streamTmp);

printf("\nConteudo do arquivo temporario:\n\n");

while((c = fgetc(streamTmp)) != EOF)
    putchar(c);

fclose(streamTmp);

```

```

        /* Arquivos temporários como o criado aqui */
        /* não são removidos automaticamente. É      */
        /* preciso removê-lo explicitamente.          */
    if (remove(nome))
        printf("\n\nNao foi possivel remover arquivo temporario\n");
    else
        printf("\n\nArquivo temporario foi removido\n");

    return 0;
}

```

### 10.7.12 DETECÇÃO DE ERROS EM *STREAMS*

*feof()*

**Incluir:** <stdio.h>

**Descrição:** A função **feof()** informa quando o final do *stream* recebido como argumento foi atingido.

**Protótipo:**

```
int feof(FILE *stream)
```

**Parâmetro:** *stream* – *stream* que se deseja testar.

**Retorno:** Um valor diferente de zero se foi tentada uma operação de leitura além do final do arquivo; zero, caso contrário.

**Observações:**

- A função **feof()** sinaliza o final do arquivo apenas quando há uma tentativa de leitura além do final do arquivo.
- Num *stream* de texto, pode-se usar tanto a macro **EOF** quanto a função **feof()** para testar se o final de um arquivo foi atingido, mas, num *stream* binário, deve-se utilizar apenas a função **feof()** com este objetivo.
- O uso de **EOF** para testar quando o final de um *stream* é atingido (mesmo quando o *stream* em questão é de texto) requer muito cuidado por parte do programador (v. **Apêndice D**). Portanto, dê preferência ao uso da função **feof()**.



**Exemplo:**

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    int    c;

    if (!(stream = fopen("Arq1.txt", "r"))) {
        printf("\nArquivo nao pode ser aberto\n");
        return 1;
    }

    /* Lê e imprime na tela os caracteres do arquivo */
    while ( (c = fgetc(stream)) != EOF )
        putchar(c);

    /* Se o final do arquivo não foi atingido */
    /* depois de fgetc() retornar EOF, é porque */
    /* ocorreu um erro de leitura. */
    if (!feof(stream))
        printf("\nOcorreu um erro de leitura no arquivo\n");

    fclose(stream);

    return 0;
}
```

***clearerr()*****Incluir:** <stdio.h>

**Descrição:** A função **clearerr()** remove qualquer sinalização de erro ou de final de arquivo no *stream* recebido como argumento.

**Protótipo:**

void clearerr(FILE \*stream)

**Parâmetro:** *stream* – *stream* aberto cujos sinalizadores de erro serão removidos.

**Observações:**

- Quando o indicador de erro de um *stream* assume um valor diferente de zero, operações executadas nele continuarão retornando um status de erro até que a função **clearerr()** seja chamada.
- As funções de posicionamento **fseek()**, **fsetpos()** e **rewind()** também zeram o indicador de final de arquivo de um *stream*, mas não fazem o mesmo com o indicador de erro.

**Exemplo:** Veja exemplo da função **ferror()**.

*ferror()*

**Incluir:** <stdio.h>

**Descrição:** A função **ferror()** verifica se existe alguma sinalização de erro num *stream*.

**Protótipo:**

```
int ferror(FILE *stream)
```

**Parâmetro:** *stream* – *stream* aberto a ser verificado.

**Retorno:** Um valor diferente de zero se existir algum erro associado ao *stream*; zero, caso contrário.

**Observações:**

- Tipicamente esta função é chamada em conjunto com a função **clearerr()**, como mostra o exemplo a seguir.
- Consulte também **clearerr()**.

**Exemplo:**

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

```

int main(void)
{
    FILE *stream;

    /* Abre o arquivo somente para leitura */
    if (!(stream = fopen("Arq1.txt", "r"))) {
        fprintf(stderr, "\nNao foi possivel abrir arquivo\n");
        return 1;
    }

    /* Gera uma condição de erro */
    /* tentando escrever no arquivo */
    fputc('c', stream);

    /* Verifica se há alguma sinalização */
    /* de erro para o arquivo */
    if (ferror(stream)) {
        printf("\nOcorreu um erro de escrita\n");
        /* Zera indicadores de erro e de EOF */
        clearerr(stream);
    }

    fclose(stream);

    return 0;
}

```

### 10.7.13 FUNÇÕES `perror()` E `ungetc()`

*perror()*

**Incluir:** <stdio.h>

**Descrição:** A função `perror()` escreve no *stream* `stderr` uma mensagem de erro correspondente ao valor armazenado na variável global `errno`.

**Protótipo:**

```
void perror(const char *string)
```

**Parâmetro:** `string` – `NULL` ou um *string* que será concatenado ao início da mensagem de erro.

**Observação:** Quando o parâmetro é **NULL**, o conteúdo da mensagem de erro impressa pela função **perror()** é o mesmo do *string* retornado pela função **strerror()** (v. **Seção 6.3.4**).

**Exemplo:**

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

#define MAX_MSG 255 /* Tamanho máximo da mensagem */

int main(int argc, char** argv)
{
    char    msg[MAX_MSG] = "";
    FILE    *stream;

    if (!( stream = fopen( "ArqInexistente", "r" ))) {
        snprintf( msg, MAX_MSG,
                  "\nFuncao %s, arquivo %s, \nlinha %d",
                  __func__, __FILE__, __LINE__ );

        perror(msg);

        return 1;
    }

    return 0;
}
```

*ungetc()*

**Incluir:** <stdio.h>

**Descrição:** A função **ungetc()** insere um caractere num *stream* aberto para leitura.

**Protótipo:**

<pre>int ungetc(int c, FILE *stream)</pre>
--

**Parâmetros:**

- `c` – caractere (expandido em **int**) a ser inserido no *stream*.
- `stream` – *stream* que permite leitura.

**Retorno:** O caractere inserido no *stream*, se a função obtém êxito; **EOF**, caso ocorra algum erro durante a operação.

**Observações:**

- O caractere inserido no *stream* será lido na próxima operação de leitura neste *stream*.
- Se o primeiro argumento recebido pela função for **EOF**, a operação falhará e o *stream* não será alterado.
- Uma chamada bem sucedida desta função elimina uma eventual sinalização de final de arquivo no *stream*.
- Quando uma chamada de **fseek()**, **fsetpos()** ou **rewind()** é executada com êxito, ela descarta qualquer caractere inserido no *stream* por meio de **ungetc()**.
- O padrão ISO não garante a inserção de dois ou mais caracteres num *stream* com o uso repetido da função **ungetc()** sem chamadas intercaladas de operações de leitura no *stream*, como mostram os trechos de programa a seguir:

```
ungetc('A', stdin); /* Inserção garantida */
ungetc('B', stdin); /* Pode não haver inserção */
...
ungetc('A', stdin); /* Inserção garantida */
getchar();
ungetc('B', stdin); /* Inserção garantida */
```

**Exemplo:**

```
#include <stdio.h>

#define MAX_DIGITOS 20

int main()
{
    FILE *stream;
    int    c, i = 0;
    char  digitos[MAX_DIGITOS + 1];
```

```

    /* Abre arquivo para escrita e leitura */
    if ( !(stream = fopen("Arq1.txt", "w+")) ) {
        fprintf(stderr, "\nImpossível abrir o arquivo\n");
        return 1;
    }

    /* Escreve um string alfanumérico no arquivo */
    fprintf(stream, "123245abcd");

    rewind(stream); /* Volta ao início do stream */

    /* Coleta os dígitos que se encontram no início do */
    /* arquivo. Se algum caractere que não é dígito for */
    /* lido, ele será devolvido ao stream. O número de */
    /* dígitos lidos é limitado a MAX_DIGITOS.          */

    for (i = 0; i < MAX_DIGITOS; ++i) {
        c = getc(stream);
        if (isdigit(c))
            digitos[i] = c;
        else
            break;
    }

    digitos[i] = '\0'; /* Termina o string numérico */

    /* O número máximo de dígitos foi lido ou */
    /* um caractere que não é dígito foi lido */
    /* ou o final do arquivo foi atingido ou */
    /* ocorreu um erro de leitura.          */

    if (ferror(stream)) {
        fprintf(stderr, "\nOcorreu um erro de leitura\n");
        return 1;
    }

    /* Se um caractere que não é dígito */
    /* foi lido, devolve-o ao stream.    */
    if (!isdigit(c) && c != EOF)
        if (ungetc(c, stream) == EOF) {
            fprintf(stderr, "\nA funcao ungetc() falhou.\n");
            return 1;
        }
    }

```

```

#ifdef TESTE
    printf("\nUltimo caractere lido: %c", c);
    printf("\nCaractere lido no arquivo: %c", getc(stream));
#endif

    printf("\nDigitos lidos no inicio do arquivo: %s\n",
           digitos);

    fclose(stream);

    return 0;
}

```

## 10.8 FUNÇÕES DE ENTRADA E SAÍDA DE CARACTERES E STRINGS EXTENSOS

Todas as funções apresentadas nesta seção são declaradas no cabeçalho `<wchar.h>`. É importante lembrar que os *streams* usados com funções de leitura ou escrita apresentados nesta seção devem ter orientação extensa ou não ter nenhuma orientação (v. Seção 10.2.9).

*fgetwc()* e *getwc()*

**Incluir:** `<wchar.h>`

**Descrição:** As funções **fgetwc()** e **getwc()** leem o próximo caractere extenso num *stream*.

**Protótipos:**

```
wint_t fgetwc(FILE *stream)
```

```
wint_t getwc(FILE *stream)
```

**Parâmetro:** *stream* – *stream* aberto para leitura.

**Retorno:** Se a operação for bem sucedida, o próximo caractere extenso encontrado no *stream* convertido em **wint\_t**; se ocorrer algum erro ou o final do *stream* for atingido prematuramente, **WEOF**.

**Observação:** A única diferença entre estas duas funções é que **getwc()** pode ser implementada como macro, enquanto **fgetwc()** é sempre implementada como função.

**Exemplo:** O exemplo a seguir usa **fgetwc()**, mas o mesmo resultado seria obtido se, em vez desta função, tivesse sido usada a função (ou macro) **getwc()**.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE      *stream;
    wint_t     cExtenso;

    if (!(stream = fopen("Dados.txt", "r"))) {
        printf( "Nao foi possivel abrir o arquivo:"
               " \"Dados.dat\"\\n" );
        return 1;
    }

    errno = 0;

    while ((cExtenso = fgetwc(stream)) != WEOF)
        printf("Caractere extenso lido = %lc\\n", cExtenso);

    if (errno == EILSEQ) {
        printf("Encontrado um caractere extenso invalido.\\n");
        return 1;
    }

    fclose(stream);

    return 0;
}
```

*fgetws()*

**Incluir:** <wchar.h>



**Descrição:** A função `fgetws()` lê caracteres extensos num *stream* e armazena-os num array até encontrar um caractere extenso de quebra de linha, chegar ao final do arquivo ou atingir o número máximo de caracteres extensos especificado.

**Protótipo:**

```
wchar_t *fgetws( wchar_t *restrict arExt, int n,
                 FILE *restrict stream )
```

**Parâmetros:**

- `arExt` – array de caracteres extensos onde os caracteres extensos lidos mais o caractere `L'\0'` serão armazenados.
- `n` – número máximo de caracteres que serão armazenados no array (incluindo o caractere `L'\0'`).
- `stream` – *stream* aberto para leitura.

**Retorno:** `NULL`, se ocorrer erro durante a leitura; caso contrário, o primeiro argumento.

**Observações:**

- Um caractere extenso nulo (`L'\0'`) é sempre o último caractere acrescentado ao final do array, de modo que o número máximo de caracteres que esta função lê é  $n - 1$ .
- Se um caractere extenso de quebra de linha for lido, ele será incluído no *string* extenso.

**Exemplo:**

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

#define TAM_ARRAY 80

int main()
{
    FILE      *stream;
    wchar_t   ar[TAM_ARRAY];
```

```

wchar_t *p;
stream = fopen("TesteExtenso.txt", "r");

if (!stream)
    perror("Abrindo arquivo de entrada");

p = fgetws(ar, sizeof(ar), stream);

if (!p)
    perror("Lendo arquivo de entrada");
else
    wprintf(L"\nString lido: \"%s\"", p);

return 0;
}

```

### *fputwc()* e *putwc()*

**Incluir:** <wchar.h>

**Descrição:** As funções **fputwc()** e **putwc()** escrevem um caractere extenso num *stream*.

#### **Protótipos:**

wint\_t fputwc(wchar\_t ce, FILE \*stream)

wint\_t putwc(wchar\_t ce, FILE \*stream)

#### **Parâmetros:**

- *ce* – caractere extenso a ser escrito.
- *stream* – *stream* aberto para escrita.

**Retorno:** O caractere extenso escrito no *stream* ou **WEOF** quando ocorre um erro.

**Observação:** A única diferença entre estas duas funções é que **putwc()** pode ser implementada como macro, enquanto **fputwc()** é sempre implementada como função.

**Exemplo:** O exemplo a seguir usa **fputwc()**, mas o mesmo resultado seria obtido se, em vez desta função, tivesse sido usada a função (ou macro) **putwc()**.

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

int main(void)
{
    FILE      *arquivo;
    wchar_t    c = L'C';

    arquivo = fopen("Arq2.txt", "w");

    if ( fputwc(c, arquivo) == WEOF ) {
        printf("Erro ao tentar escrever no arquivo");
        return 1;
    } else
        printf("O caractere %lc foi escrito no arquivo", c);

    fclose(arquivo);

    return 0;
}
```

### *fputws()*

**Incluir:** <wchar.h>

**Descrição:** A função **fputws()** escreve um *string* extenso num *stream*.

**Protótipo:**

```
int fputws( const wchar_t *restrict strExt,
            FILE *restrict stream )
```

**Parâmetros:**

- `strExt` – *string* extenso a ser escrito.
- `stream` – *stream* aberto para escrita.

**Retorno:** -1, se ocorre algum erro; caso contrário, um valor não negativo.

**Observação:** Quando encontra um caractere extenso inválido, esta função atribui **EILSEQ** à variável global **errno** (v. **Seção 11.5**).

**Exemplo:**

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

int main(void)
{
    FILE      *stream;
    wchar_t *strExtenso = L"Isto e' um teste.";

    if (!(stream = fopen("Arq.dat", "w"))) {
        fprintf(stderr, "Nao foi possivel abrir arquivo.\n");
        return 1;
    }

    errno = 0;

    if ( fputws(strExtenso, stream) == -1 ) {
        fprintf( stderr, "Nao foi possivel completar"
                  " a operacao.\n");

        if (errno == EILSEQ)
            fprintf( stderr, "Foi encontrado um caractere"
                    " extenso invalido.\n" );

        return 1;
    }

    fclose(stream);

    return 0;
}
```

*fwide()*

**Incluir:** <wchar.h>

**Descrição:** A função **fwide()** consulta ou configura a orientação de um *stream* (v. **Seção 10.2.9**).

**Protótipo:**

```
int fwide(FILE *stream, int orientacao)
```

**Parâmetros:**

- *stream* – *stream* aberto para leitura ou escrita.
- *orientacao* – valor inteiro que representa o modo de orientação do *stream*:
  - Se este valor for maior do que zero, a função tentará tornar extensa a orientação do *stream*.
  - Se este valor for menor do que zero, a função tentará orientar o *stream* por byte.
  - Se este valor for igual a zero, não haverá tentativa de alteração da orientação do *stream*.

**Retorno:**

- Um valor negativo, se o *stream* tiver orientação monobyte.
- Um valor positivo, se o *stream* tiver orientação extensa.
- Zero, se o *stream* não for orientado.

**Observação:** Esta função não serve para alterar a orientação de um *stream*. Ou seja, uma vez que a orientação do *stream* tenha sido estabelecida, deve-se reabrir o *stream* usando **freopen()** (v. **Seção 10.7.1**) antes de reorientá-lo.

**Exemplo:**

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

void Orientacao(const char *intro, FILE *stream)
{
    int orientacao;
```

```

orientacao = fwide(stream, 0);

if (orientacao < 0)
    printf( "\n%s\n\tStream tem orientacao monobyte.\n",
            intro );
else if (orientacao > 0)
    printf( "%s\n\tStream tem orientacao extensa.\n",
            intro );
else
    printf( "%s\n\tStream nao tem orientacao.\n",
            intro );
}

int main(void)
{
    FILE *stream;

    stream = fopen("Arq.dat", "w");

    Orientacao("Apos a abertura do arquivo:", stream);

    /* Orienta o stream por byte */
    fwide(stream, -1);
    Orientacao("Apos chamada fwide(stream, -1):", stream);

    /* Tenta orientação extensa */
    fwide(stream, 1);
    Orientacao("Apos chamada fwide(stream, 1):", stream);

    fclose(stream);

    stream = fopen("Arq.dat", "w");

    Orientacao("Apos reabertura do arquivo:", stream);

    fwprintf(stream, L"um string");

    Orientacao("Apos chamar fwprintf()", stream);

    fclose(stream);

    return 0;
}

```

*fwprintf()***Incluir:** <wchar.h>**Descrição:** A função **fwprintf()** escreve dados num *stream* usando um *string* extenso de formatação.**Protótipo:**

```
int fwprintf( FILE *restrict stream,
              wchar_t *restrict formato, ... )
```

**Parâmetros:**

- *stream* – *stream* de texto aberto para escrita.
- *formato* – *string* extenso de formatação (v. **Apêndice B**).

**Retorno:** O número de caracteres extensos escritos, se não ocorrer erro na operação; caso contrário, um valor negativo.**Observação:** Esta função faz parte da família printf e é semelhante à função **fprintf()** apresentada na **Seção 10.7.7**.**Exemplo:**

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

#define LOCALE_BR "pt_BR.utf8"

int main()
{
    wchar_t palavraLocal[] = L"Cora\u00E7\u00E3o";
    char    palavraSemAcento[] = "Coracao";
    char    *localidade;

    localidade = setlocale(LC_ALL, LOCALE_BR);

    if (!localidade)
        fprintf( stderr, "Nao foi possivel alterar a "
```

```

        "localidade para %s.\n"
        "A localidade corrente e' %s.\n",
        localidade, setlocale(LC_ALL, NULL));

    fprintf( stdout, L"Palavra: %ls\n"
              "Palavra sem acento: %s\n",
              palavraLocal, palavraSemAcento );

    return 0;
}

```

### *fwscanf()*

**Incluir:** <wchar.h>

**Descrição:** A função **fwscanf()** lê dados formatados num *stream* especificado usando um *string* extenso de formatação.

**Protótipo:**

```

int fwscanf( FILE *restrict stream,
             const wchar_t *restrict formato, ... )

```

**Parâmetros:**

- *stream* – *stream* de texto aberto para leitura.
- *formato* – *string* extenso de formatação (v. **Apêndice B**).

**Retorno:** O número de variáveis que tiverem valores atribuídos ou **EOF** se o final do *stream* for atingido antes que qualquer variável tenha um valor atribuído.

**Observação:** Esta função faz parte da família *scanf* e é semelhante à função **fs-*canf()*** apresentada na **Seção 10.7.6**.

**Exemplo:**

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



```
#include <wchar.h>

int main(void)
{
    int    umInt, nValoresLidos = 0;
    FILE   *stream;

    stream = fopen("Arql.dat", "rt");

    if (!stream) {
        printf("Nao foi possivel abrir o arquivo");
        return 1;
    }

    nValoresLidos = fwscanf(stream, L"%d", &umInt);

    if (nValoresLidos != EOF)
        printf("O valor lido foi: %d", umInt);
    else
        printf("Nao foi lido nenhum valor");

    fclose(stream);

    return 0;
}
```

## *getwchar()*

**Incluir:** <wchar.h>

**Descrição:** A função **getwchar()** lê o próximo caractere extenso no *stream* **stdin**.

**Protótipo:**

```
wint_t getwchar(void)
```

**Retorno:** O caractere extenso lido em **stdin** convertido em **wint\_t**, se a função obtém êxito; caso contrário, **WEOF**.

**Observações:**

- Chamar esta função é o mesmo que efetuar a chamada: `fgetc(stdin)`.
- Esta operação pode ser implementada como macro.

**Exemplo:** Veja exemplo de `ungetc()`.

*putwchar()*

**Incluir:** `<wchar.h>`

**Descrição:** A função `putwchar()` escreve um caractere extenso em **stdout**.

**Protótipo:**

<code>wint_t putwchar(wchar_t ce)</code>
--

**Parâmetro:** `ce` – caractere extenso a ser escrito.

**Retorno:** O caractere extenso escrito, se a função obtém êxito; caso contrário, **WEOF**.

**Observação:** A chamada:

```
putwchar(ce);
```

tem o mesmo efeito que:

```
fputc(ce, stdout);
```

**Exemplo:** Veja exemplo de `ungetc()`.

*swprintf()*

**Incluir:** `<wchar.h>`

**Descrição:** A função `swprintf()` escreve, num array de caracteres extensos, dados formatados conforme especificado por um *string* extenso de formatação e acrescenta o caractere extenso terminal `L'\0'` ao final do processo de escrita.



```

    nCarEscritos = swprintf( str, MAX, formato, umInt,
                           umDouble, umString, umChar );

    printf( "String:\n\t%ls \n\nNumero de caracteres "
           "escritos: %d\n", str, nCarEscritos );

    return 0;
}

```

### *swscanf()*

**Incluir:** <wchar.h>

**Descrição:** A função **swscanf()** lê dados formatados num *string* extenso usando um *string* extenso de formatação.

### **Protótipo:**

```

int swscanf( const wchar_t *restrict entrada,
             const wchar_t *restrict formato, ... )

```

### **Parâmetros:**

- *entrada* – *string* extenso onde será feita a leitura.
- *formato* – *string* extenso de formatação (v. **Apêndice B**).
- ... – representa endereços de variáveis onde valores lidos serão armazenados.

**Retorno:** O número de itens lidos, convertidos e armazenados; **EOF**, se a função tentar ler além do final do *string* extenso sem ter atribuído valor a nenhuma variável.

### **Observações:**

- O número de especificadores de formato no segundo argumento deve ser igual ao número de argumentos variáveis.
- Esta função faz parte da família *scanf* e é utilizada para formatação em memória (v. **Seção 10.7.8**).
- Esta função é semelhante à função **sscanf()** apresentada na **Seção 10.7.8**.

**Exemplo:**

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

int main(void)
{
    int      umInt, nValoresLidos;
    float    umFloat;
    char      umChar;
    wchar_t  entrada[] = L"12 3.1415 X algo mais";

    nValoresLidos = swscanf( entrada, L"%d %f %c",
                             &umInt, &umFloat, &umChar );

    printf( "\nString onde foi feita a leitura: \n\t%s",
            entrada );
    printf( "\n\nNumero de valores lidos e atribuidos: %d",
            nValoresLidos );

    if (3 == nValoresLidos) {
        printf( "\n\nValores lidos:\n \t%d, %f e %c",
                umInt, umFloat, umChar );
    }

    return 0;
}
```

***ungetwc()*****Incluir:** <wchar.h>

**Descrição:** A função **ungetwc()** insere um caractere extenso num *stream* aberto para leitura. Este caractere será lido na próxima chamada de alguma função que leia caracteres extensos no *stream*.

**Protótipo:**

<pre>wint_t ungetwc(wint_t ce, FILE *stream)</pre>
--

**Parâmetros:**

- `ce` – caractere extenso a ser inserido no *stream*.
- `stream` – *stream* aberto para leitura.

**Retorno:** O caractere extenso inserido no *stream*, se a função obtém êxito; **WEOF**, se ocorrer algum erro na operação ou se `ce` for igual a **WEOF**.

**Observações:**

- Uma chamada subsequente de **ungetwc()** sem uma chamada intercalada de alguma função de leitura de caracteres extensos no mesmo *stream* recebido como argumento por esta função pode fazer com que **ungetwc()** não funcione adequadamente.
- Consulte descrição da função **ungetc()** (Seção 10.7.13).

**Exemplo:** O programa a seguir demonstra o uso das funções **ungetwc()**, **putwchar()** e **getwchar()**.

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

int main(void)
{
    int c;

    ungetwc(L'A', stdin);

    wprintf(L"Caractere inserido em stdin: ");
    putwchar(getwchar());

    return 0;
}
```

**vfwprintf()**

**Incluir:** `<wchar.h>`

**Descrição:** A função **vfwprintf()** escreve num *stream* usando um *string* extenso de formatação e uma lista de argumentos variáveis do tipo **va\_list**.

**Protótipo:**

```
int vfwprintf( FILE *restrict stream,
               const wchar_t *restrict formato,
               va_list args )
```

**Parâmetros:**

- *stream* – *stream* aberto para escrita.
- *formato* – *string* extenso de formatação (v. **Apêndice B**).
- *args* – lista de argumentos variáveis contendo os valores que serão escritos (v. **Capítulo 9**).

**Retorno:** O número de caracteres extensos escritos se não ocorrer nenhum erro; um número negativo, caso contrário.

**Observações:**

- A diferença entre as funções **vfwprintf()** e **fwprintf()** é que **fwprintf()** usa diretamente argumentos variáveis, representados por três pontos em seu protótipo, enquanto **vfwprintf()** usa argumentos variáveis indiretamente por meio de um parâmetro do tipo **va\_list** (v. **Capítulo 9**).
- Esta função faz parte da família printf e é semelhante à função **vfprintf()** apresentada na **Seção 10.7.7**.

**Exemplo:**

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

/****
 * Função EscreveEmStream(): escreve dados formatados no
 *                               stream especificado usando
 *                               vfwprintf()
 ****/

int EscreveEmStream(FILE *stream, const wchar_t *formato,
                    ...)
```

```

{
    va_list  argumentos;
    int      retorno;

    va_start(argumentos, formato);

    retorno = vfwprintf(stream, formato, argumentos);

    va_end(argumentos);

    return retorno;
}

int main(void)
{
    FILE *stream;
    int   umInteiro = 30;
    float umFloat = 7.45;
    char  umString[80] = "Teste";

    stream = tmpfile(); /* Cria um arquivo temporário */

    if (!stream) {
        perror("Arquivo temporario nao pode ser criado.");
        return 1;
    }

    EscreveEmStream( stream, L"%d %f %s",
                    umInteiro, umFloat, umString );

    rewind(stream);

    fscanf( stream,"%d %f %s",
            &umInteiro, &umFloat, umString );

    printf("%d %f %s\n", umInteiro, umFloat, umString);

    fclose(stream);

    return 0;
}

```



***vfwscanf()* (C99)****Incluir:** <wchar.h>

**Descrição:** A função **vfwscanf()** lê dados formatados num *stream* usando um *string* extenso de formatação e uma lista de argumentos variáveis.

**Protótipo:**

```
int vfwscanf( FILE *restrict stream,
              const wchar_t *restrict formato,
              va_list args )
```

**Parâmetros:**

- *stream* – *stream* onde será feita a leitura.
- *formato* – *string* extenso de formatação (v. **Apêndice B**).
- *args* – lista de argumentos variáveis contendo endereços de variáveis onde os valores lidos serão armazenados (v. **Capítulo 9**).

**Retorno:** Número de valores lidos, convertidos e armazenados em variáveis; EOF, se ocorrer algum erro, sem que nenhum valor tenha sido lido e atribuído.

**Observações:**

- A diferença entre as funções **vfwscanf()** e **fwscanf()** é que **fwscanf()** usa diretamente argumentos variáveis, representados por três pontos em seu protótipo, enquanto **vfwscanf()** usa argumentos variáveis indiretamente por meio de um parâmetro do tipo **va\_list** (v. **Capítulo 9**).
- A função **vfwscanf()** faz parte da família **scanf** e é semelhante à função **vfs-canf()** apresentada na **Seção 10.7.6**.

**Exemplo:**

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

```

/****
 * Função LeEmStream(): Lê dados formatados no stream
 *                      especificado usando vfwscanf()
 ****/

int LeEmStream(FILE *stream, const wchar_t *formato, ...)
{
    int      nValoresAtribuidos;
    va_list args;

    va_start(args, formato);

    nValoresAtribuidos = vfwscanf(stream, formato, args);

    va_end(args);

    return nValoresAtribuidos;
}

int main(void)
{
    int    umInt, nValoresLidos;
    float umFloat;
    char  umChar;

    printf("Digite um inteiro, um real e um caractere:\n");

    nValoresLidos = LeEmStream( stdin, L"%d %f %c",
                               &umInt, &umFloat, &umChar );

    if (nValoresLidos != EOF)
        printf( "Numero de valores lidos e atribuidos: %d",
                nValoresLidos );
    else
        printf("Ocorreu um erro de leitura");

    return 0;
}

```

*vswprintf()*

**Incluir:** <wchar.h>

**Descrição:** A função **vswprintf()** escreve um número limitado de caracteres extensos num array usando um *string* de formatação extenso e argumentos supridos por uma lista de argumentos variáveis.

**Protótipo:**

```
int vswprintf( wchar_t *restrict arExt, size_t n,
              const wchar_t *restrict formato,
              va_list args )
```

**Parâmetros:**

- **arExt** – array de caracteres extensos onde será feita a escrita.
- **n** – número máximo de caracteres extensos escritos.
- **formato** – *string* extenso de formatação (v. **Apêndice B**).
- **args** – lista de argumentos variáveis contendo os dados que serão escritos (v. **Capítulo 9**).

**Retorno:** Se não ocorrer nenhum erro durante a operação, o número de caracteres extensos escritos; **EOF**, se ocorrer algum erro.

**Observações:**

- A diferença entre as funções **vswprintf()** e **swprintf()** é que **swprintf()** usa diretamente argumentos variáveis, representados por três pontos em seu protótipo, enquanto **vswprintf()** usa argumentos variáveis indiretamente por meio de um parâmetro do tipo **va\_list** (v. **Capítulo 9**).
- A função **vswprintf()** faz parte da família **printf** e é utilizada para formatação em memória (v. **Seção 10.7.8**).
- Esta função é semelhante à função **vsnprintf()** [e não à função **vsprintf()**] apresentada na **Seção 10.7.8**.

**Exemplo:**

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

```

#define MAX 80

/****
 * Função EscreveEmArray(): escreve dados formatados no
 *                          array de caracteres extensos
 *                          especificado usando vswprintf()
 ****/

int EscreveEmArray(wchar_t ar[], const wchar_t *formato,
                  ...)
{
    va_list argumentos;
    int      retorno;

    va_start(argumentos, formato);

    retorno = vswprintf(ar, MAX, formato, argumentos);

    va_end(argumentos);

    return retorno;
}

int main(void)
{
    wchar_t array[MAX];
    int      umInteiro = 33;
    float    umFloat = 44.5;
    wchar_t umString[MAX] = L"Isto e' um teste";

    EscreveEmArray( array, L"%d %f %ls",
                   umInteiro, umFloat, umString );

    printf("Conteudo do array: %ls\n", array);

    return 0;
}

```

*vswscanf()* (C99)

**Incluir:** <wchar.h>

**Descrição:** A função **vswscanf()** (C99) lê dados formatados num *string* extenso e armazena-os em endereços de variáveis passados como parâmetros numa lista de argumentos variáveis do tipo **va\_list**.

**Protótipo:**

```
int vswscanf( const wchar_t *restrict strExt,
              const wchar_t *restrict formato,
              va_list args )
```

**Parâmetros:**

- **strExt** – *string* extenso onde será feita a leitura.
- **formato** – *string* extenso de formatação (v. **Apêndice B**).
- **args** – lista de argumentos variáveis contendo endereços de variáveis onde os valores serão armazenados (v. **Capítulo 9**).

**Retorno:** O número de itens lidos, convertidos e armazenados; **EOF**, se ela tentar ler além do final do *string* sem ter atribuído valor a nenhuma variável.

**Observações:**

- A função **vswscanf()** faz parte da família **scanf** e é utilizada para formatação em memória (v. **Seção 10.7.8**).
- Esta função é semelhante à função **vsscanf()** apresentada na **Seção 10.7.8**.

**Exemplo:**

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

/****
 * Função LeEmStream(): Lê dados formatados no string
 *                       especificado usando vswscanf()
 ****/

int LeEmString(const wchar_t *s, const wchar_t *formato,
              ...)
```

```

{
    int      nValoresAtribuidos;
    va_list args;

    va_start(args, formato);

    nValoresAtribuidos = vswscanf(s, formato, args);

    va_end(args);

    return nValoresAtribuidos;
}

int main(void)
{
    int      umInt, nValoresLidos;
    float    umFloat;
    char      umChar;
    wchar_t  entrada[] = L"12 3.1415 X algo mais";

    nValoresLidos = LeEmString( entrada, L"%d %f %c",
                                &umInt, &umFloat, &umChar );

    printf( "\nString onde foi feita a leitura: \n\t%s",
            entrada );

    if (nValoresLidos == EOF) {
        printf("Ocorreu um erro de leitura");
        return 1;
    }

    printf( "\n\nNumero de valores lidos e atribuidos: %d",
            nValoresLidos );

    if (3 == nValoresLidos) {
        printf( "\n\nValores lidos:\n \t%d, %f e %c",
                umInt, umFloat, umChar );
    }

    return 0;
}

```

## *vwprintf()*

**Incluir:** <wchar.h>

**Descrição:** A função **vwprintf()** escreve em **stdout** usando um *string* extenso de formatação e uma lista de argumentos variáveis.

**Protótipo:**

```
int vwprintf( const wchar_t *restrict formato,
              va_list argumentos )
```

**Parâmetros:**

- *formato* – *string* extenso de formatação (v. **Apêndice B**).
- *argumentos* – lista de argumentos variáveis contendo os valores que serão escritos (v. **Capítulo 9**).

**Retorno:** O número de caracteres extensos escritos, se não ocorrer nenhum erro; **EOF**, caso contrário.

**Observações:**

- A diferença entre as funções **vwprintf()** e **wprintf()** é que **wprintf()** usa diretamente argumentos variáveis, representados por três pontos em seu protótipo, enquanto **vwprintf()** usa argumentos variáveis indiretamente por meio de um parâmetro do tipo **va\_list** (v. **Capítulo 9**).
- Consulte também a descrição da função **wprintf()**.

**Exemplo:**

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

int Imprime(const wchar_t *formato, ...)
{
    va_list argumentos;
    int      retorno;
```

```

    va_start(argumentos, formato);

    retorno = vwprintf(formato, argumentos);

    va_end(argumentos);

    return retorno;
}

int main()
{
    int      umInteiro = 32;
    float     umFloat = 44.9;
    wchar_t *umString = L"Isto e' um teste";

    Imprime(L"%d %f %ls\n", umInteiro, umFloat, umString);

    return 0;
}

```

### *vwscanf()* (C99)

**Incluir:** <wchar.h>

**Descrição:** A função **vwscanf()** lê dados formatados no *stream* padrão **stdin** usando um *string* extenso de formatação e uma lista de argumentos variáveis.

**Protótipo:**

```
int vwscanf(const wchar_t *restrict formato, va_list args)
```

**Parâmetros:**

- **formato** – *string* extenso de formatação (v. **Apêndice B**).
- **args** – lista de argumentos variáveis contendo endereços de variáveis onde os valores lidos serão armazenados (v. **Capítulo 9**).

**Retorno:** O número de valores lidos, convertidos e armazenados em variáveis. Se ocorrer algum erro sem que nenhum valor tenha sido atribuído a alguma variável, **EOF**.



**Observações:**

- A função **vwscanf()** faz parte da família **scanf** e é semelhante à função **vscanf()** apresentada na **Seção 10.7.6**.
- Consulte também **wscanf()**.

**Exemplo:**

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

int Leitura(const wchar_t *formato, ...)
{
    int      nValoresAtribuidos;
    va_list args;

    va_start(args, formato);
    nValoresAtribuidos = vwscanf(formato, args);
    va_end(args);

    return nValoresAtribuidos;
}

int main(void)
{
    int      umInt, nValoresLidos;
    float umFloat;
    char      umChar;

    printf("Digite um inteiro, um real e um caractere:\n");

    nValoresLidos = Leitura( L"%d %f %c",
                           &umInt, &umFloat, &umChar );

    if (nValoresLidos != EOF)
        printf( "Numero de valores lidos e atribuidos: %d",
               nValoresLidos );
    else
        printf("Ocorreu algum erro de leitura");
}
```

```
    return 0;
}
```

*wprintf()*

**Incluir:** <wchar.h>

**Descrição:** A função **wprintf()** imprime dados formatados no *stream* padrão **stdout** usando um *string* extenso de formatação.

**Protótipo:**

```
int wprintf(const wchar_t *restrict formato, ...)
```

**Parâmetro:** *formato* – *string* extenso de formatação (v. **Apêndice B**).

**Retorno:** O número de caracteres extensos escritos, se não ocorrer nenhum erro; um valor negativo, caso contrário.

**Exemplo:** Veja exemplo de **wscanf()**.

*wscanf()*

**Incluir:** <wchar.h>

**Descrição:** A função **wscanf()** lê entrada formatada em **stdin** usando um *string* extenso de formatação.

**Protótipo:**

```
int wscanf(const wchar_t *restrict formato, ...)
```

**Parâmetro:** *formato* – *string* extenso de formatação (v. **Apêndice B**).

**Retorno:** Número de variáveis que tiveram valores atribuídos; **EOF**, se o final de **stdin** for encontrado sem que nenhum valor tenha sido atribuído a alguma variável ou se ocorrer erro durante a operação.

**Observações:**

- A diferença entre as funções **vwscanf()** e **wscanf()** é que **wscanf()** usa diretamente argumentos variáveis, representados por três pontos em seu protótipo, enquanto **vwscanf()** usa argumentos variáveis indiretamente por meio de um parâmetro do tipo **va\_list** (v. **Capítulo 9**).
- A função **wscanf()** faz parte da família **scanf** e é semelhante à função **scanf()** apresentada na **Seção 10.7.6**.

**Exemplo:** O programa a seguir demonstra o uso das funções **wprintf()** e **wscanf()**.

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

int main( void )
{
    int      umInt, resultado;
    float    umFloat;
    char      umChar, str[80];
    wchar_t  umLChar, strL[80];

    printf( "Digite um inteiro, um real e um "
           "caractere (nesta ordem): " );

    resultado = wscanf( L"%d %f %c", &umInt,
                       &umFloat, &umChar );

    printf( "Numero de valores lidos e atribuidos: %d\n",
           resultado );

    /* A chamada de wprintf() a seguir poderá não */
    /* imprimir nada porque o stream stdout tem */
    /* orientação monobyte devido às chamadas */
    /* anteriores de printf(). */
    if (resultado == 3)
        wprintf( L"\nValores lidos e atribuidos:\n\t%d "
                L"\n\t%f \n\t%c \n",
                umInt, umFloat, umChar );

    return 0;
}
```

## 10.9 EXERCÍCIOS DE REVISÃO

1. Defina: (a) processamento de arquivos baseado em *streams* e (b) processamento de arquivos baseado em sistemas. (c) Que vantagens cada um destes tipos de processamento oferece com relação ao outro?
2. Defina os seguintes conceitos:
  - (a) Arquivo de texto
  - (b) Arquivo binário
  - (c) Acesso sequencial
  - (d) Acesso direto
  - (e) Entrada formatada
  - (f) Saída formatada
  - (g) Buffering
  - (h) Orientação de *stream*
3. Que funções da biblioteca padrão de C podem ser usadas com um *stream* com orientação: (a) monobyte e (b) extensa.
4. O que representam as variáveis globais **stderr**, **stdin** e **stdout**?
5. (a) O que significa abrir um arquivo? (b) Como esta operação é realizada em C?
6. Descreva os seguintes modos de abertura de arquivos:
  - (a) "r"
  - (b) "w"
  - (c) "a"
  - (d) "r+"
  - (e) "w+"
  - (f) "a+"

7. (a) O que significa fechar um arquivo? (b) Qual é a importância de se fechar um arquivo num programa em C?
8. Como se pode determinar o número máximo de arquivos que podem estar abertos simultaneamente durante a execução de um programa escrito em C?
9. O que representa a macro **FILENAME\_MAX**?
10. Cite duas situações práticas nas quais a função **freopen()** é útil.
11. O que há de errado com a seguinte chamada da função **fflush()**?  

```
fflush(stdin);
```
12. (a) Apresente duas maneiras de testar se o final de um arquivo foi atingido durante uma operação de leitura num *stream* de texto. (b) A resposta ao item (a) também se aplica a *streams* binários? Explique.
13. Em que diferem as funções **getc()** e **fgetc()**?
14. Suponha que *streamA* e *streamB* são dois *streams* abertos em modo binário, sendo que o primeiro *stream* é aberto para leitura e o segundo é aberto para escrita. Escreva um trecho de programa que demonstre como copiar o conteúdo do primeiro *stream* para o segundo.
15. Qual é a diferença entre as funções **getchar()** e **fgetc()**?
16. (a) Descreva os parâmetros da função **fgets()**. (b) O que esta função retorna?
17. Descreva as diferenças entre as funções **gets()** e **fgets()**.
18. (a) Por que se aconselha que a função **gets()** nunca seja utilizada? (b) Se é sempre aconselhável não usar esta função, o que justifica sua presença na biblioteca padrão de C?
19. (a) Quais são as diferenças entre as macros **EOF** e **WEOF**? (b) Em que cabeçalhos estas macros são definidas?
20. O que têm em comum as funções das famílias **printf** e **scanf** que começam com *v*?
21. Para que servem as funções **fseek()** e **ftell()**?
22. Que vantagens oferecem as funções **fsetpos()** e **fgetpos()** com relação às funções **fseek()** e **ftell()**?

23. Apresente o significado de cada macro a seguir:

(a) **SEEK\_SET**

(b) **SEEK\_CUR**

(c) **SEEK\_END**

24. O que representam as macros a seguir:

(a) **L\_tmpnam**

(b) **TMP\_MAX**

25. Que vantagens a função **tmpfile()** oferece com relação à função **tmpnam()**?

26. Como são tipicamente usadas as funções **clearerr()** e **ferror()**?

27. (a) Em que situações o uso da função **rename()** não é portátil? (b) Em que situações o uso da função **remove()** não é portátil?

28. Qual é relação existente entre as funções **perror()**, declarada em `<stdio.h>`, e **strerror()**, declarada em `<string.h>`?

29. Em que situação o padrão ISO de C não garante que a função **ungetc()** inserirá um caractere num *stream* aberto para leitura?

30. (a) Uma função de escrita de caracteres extensos pode realizar uma escrita num *stream* com orientação monobyte? (b) Uma função de leitura de caracteres monobytes pode realizar uma leitura num *stream* com orientação extensa? Explique.

31. (a) Como se consulta a orientação de um *stream*? (b) Como se altera a orientação de um *stream*?

32. Certo livro de programação em C ensina a ler um valor do tipo **int** no meio de entrada padrão da seguinte maneira:

```
char linha[100];    /* Linha lida */
...
fgets(linha, sizeof(linha), stdin);
sscanf(linha, "%d", &valorInteiro);
```

Por que esta não é uma boa abordagem? [**Dica:** O que acontece se o usuário decide *testar* o programa e digita 200 caracteres?]

## 33. Por que o seguinte programa imprime um resultado inesperado?

```

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

char *NomeTmp(void)
{
    char        nome[30];          /* Contém o nome gerado */
    static int numeroSeq = 0; /* Número sequencial acrescentado */
                                /* ao nome do arquivo          */

    ++numeroSeq;

    strcpy(nome, "tmp");

    nome[3] = numeroSeq + '0';

    nome[4] = '\0';

    return nome;
}

int main()
{
    printf("Nome: %s\n", NomeTmp());

    return(0);
}

```

*As resoluções das questões a seguir são relacionadas a especificadores de formato e requerem que o leitor consulte o **Apêndice B**.*

34. (a) Os especificadores de formato %d e %i da família printf têm o mesmo efeito? (b) E os especificadores de formato %d e %i da família scanf têm o mesmo efeito?

35. Como são interpretados os seguintes especificadores de formato da família printf?

(a) %a

(b) %A

(c) %c

(d) %e

(e) %E

(f) %f

(g) %F

(h) %g

(i) %G

(j) %n

(k) %o

(l) %p

(m) %s

(n) %u

(o) %x

(p) %X

36. Descreva o efeito do uso dos seguintes componentes na formação de especificadores de formato da família printf:

(a) Sinalizador

(b) Largura

(c) Precisão

(d) Prefixo

37. Como deve ser usado o asterisco na composição de um especificador de formato da família scanf?

38. Qual é o efeito na composição de um especificador de formato da família scanf do uso do componente: (a) largura e (b) prefixo?

39. Os especificadores de formato %5c e %5s da família scanf são equivalentes?

40. Como são interpretados os seguintes especificadores de formato da família scanf?



- (a) `%5[abcd]`
- (b) `%5[^abcd]`
- (c) `%5[a-z]`
- (d) `%5[^a-z]`

41. Que tipos de especificadores de formato da família `scanf` são equivalentes?
42. Qual é o efeito do uso do prefixo `l` com os tipos de especificadores de formato `c` e `s` da família `scanf`?

# *Capítulo 11*

---

*Suporte para legibilidade e robustez*

## 11.1 INTRODUÇÃO

Um programa é **robusto** quando é capaz de lidar não apenas com situações normais mas também com aquelas consideradas anormais; i.e., que não são desejadas num programa. Por exemplo, dados introduzidos incorretamente pelo usuário ou ausência de um arquivo importante para o funcionamento do programa constituem situações indesejáveis (mas, bastante prováveis) e um programa considerado robusto deve antecipar e saber tratar tais situações.

Este capítulo explora três cabeçalhos da biblioteca padrão cujos componentes podem contribuir para a robustez de um programa:

- `<assert.h>` – que define a macro **assert()** usada em teste e depuração de programas (v. **Seção 11.4**).
- `<errno.h>` – que contém uma alusão à variável global **errno** e definições de macros (constantes) associadas a condições de erros (v. **Seção 11.5**).
- `<signal.h>` – que dá suporte para tratamento de sinais em C (v. **Seção 11.6**).
- `<setjmp.h>` – no qual são declaradas duas funções que dão suporte para desvios entre funções cujo uso prático mais comum é a implementação de mecanismos de tratamento de exceções (v. **Seção 11.7**).

Este capítulo também discute dois cabeçalho que, de algum modo, contribuem para a melhoria de legibilidade e portabilidade de programas:

- `<stdbool.h>` – que define macros que servem como identificadores alternativos para o tipo **\_Bool** (v. **Seção 11.2**).
- `<iso646.h>` – no qual são definidas macros que representam identificadores alternativos para alguns operadores de C (v. **Seção 11.3**).

# 11.2 MACROS PARA O TIPO BOOLEANO: <stdbool.h> (C99)

O cabeçalho <stdbool.h> provê macros que representam identificadores alternativos para o tipo **\_Bool**. O objetivo destas macros é melhorar a legibilidade de programas escritos em C, assim como facilitar o transporte de programas escritos nesta linguagem para C++ ou Java.

A **Tabela 11-1** a seguir enumera as macros definidas no cabeçalho <stdbool.h> e suas respectivas expansões.

MACRO	EXPANSÃO
<b>bool</b>	<b>_Bool</b>
<b>false</b>	0
<b>true</b>	1
<b>__bool_true_false_are_defined</b>	1

Tabela 11-1: Macros definidas em <stdbool.h> e expansões correspondentes.

**Exemplo:** O programa a seguir exemplifica o uso de macros definidas em <stdbool.h>.

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

/* Se seu compilador não suporta esta          */
/* característica de C99 ou se o comentário    */
/* da linha a seguir for removido, o trecho    */
/* de programa correspondente a #else será     */
/* compilado. Caso contrário, o trecho de     */
/* programa correspondente a #if será compilada */

/* #undef __bool_true_false_are_defined */
int main()
{
    int  x = -1, y = 7;

    #if __bool_true_false_are_defined

        bool teste = false;
```

```

printf("Usando macros definidas em <stdbool.h>.\n");

if (x < y)
    teste = true;

#else

    _Bool teste = 0;

printf( "Macros definidas em <stdbool.h>"
        " NAO sao usadas.\n" );

if (x < y)
    teste = 1;

#endif

return 0;
}

```

## 11.3 NOMES LEGÍVEIS PARA OPERADORES: <iso646.h>

No cabeçalho <iso646.h>, encontram-se macros que proveem identificadores alternativos para alguns operadores de C. O objetivo primário destas macros é prover substituições para operadores que usam caracteres ausentes em alguns conjuntos de caracteres definidos pelo padrão ISO 646.

O antigo padrão ISO 646, de modo semelhante ao padrão ISO 8859 (v. **Seção 6.1**), define vários conjuntos de caracteres<sup>102</sup>. Entretanto, diferentemente do que ocorre com o padrão ISO 8859, que preserva todos os caracteres do repertório ASCII, o padrão ISO 646 substitui alguns caracteres *pouco usados* do código ASCII por outros caracteres usados em alguns scripts. Infelizmente, alguns destes caracteres *pouco usados* em linguagens naturais (e.g., # e ^) são *bastante usados* na escrita de programas em C. Daí a necessidade de definir dígrafos, trígrafos (v. **Volume I**) e macros como alternativas para escrita desses caracteres ausentes.

---

102 Não confunda o código de caracteres ISO 646 com aquele definido pelo padrão ISO 10646.

A **Tabela 11-2** apresenta as macros definidas no cabeçalho `<iso646.h>` e suas respectivas expansões.

MACRO	EXPANDE-SE NO OPERADOR...
<b>and</b>	<code>&amp;&amp;</code>
<b>and_eq</b>	<code>&amp;=</code>
<b>bitand</b>	<code>&amp;</code>
<b>bitor</b>	<code> </code>
<b>compl</b>	<code>~</code>
<b>not</b>	<code>!</code>
<b>not_eq</b>	<code>!=</code>
<b>or</b>	<code>  </code>
<b>or_eq</b>	<code> =</code>
<b>xor</b>	<code>^</code>
<b>xor_eq</b>	<code>^=</code>

Tabela 11-2: Macros dfinidas em `<iso646.h>` e expansões correspondentes.

Utilizando as macros apresentadas na **Tabela 11-2**, a instrução:

```
if (x < 10 && !y || z) {  
    ...  
}
```

seria considerada equivalente a:

```
if (x < 10 and not y or z) {  
    ...  
}
```

Muito raramente, é necessário usar, nos dias atuais, as macros definidas em `<iso646.h>` para compensar a ausência de caracteres – o que é o verdadeiro objetivo destas macros – mas elas também podem melhorar a legibilidade de programas escritos em C.

**Exemplo:** O programa a seguir ilustra o uso das macros definidas em `<iso646.h>`.

```
#include <stdio.h>  
#include <iso646.h>  
  
int main()
```

```

{
    int x = -1, y = 7, z = -3;

    if (x < 10 && !y || z)
        printf("A condicao foi satisfeita\n");
    else
        printf("A condicao NAO foi satisfeita\n");

    /* A condição do if a seguir tem a mesma      */
    /* interpretação da condição do if anterior */
    if (x < 10 and not y or z)
        printf("A condicao foi satisfeita\n");
    else
        printf("A condicao NAO foi satisfeita\n");

    return 0;
}

```

## 11.4 MACRO `assert()`: `<assert.h>`

O propósito do cabeçalho `<assert.h>` é definir a macro **`assert()`** utilizada em teste e depuração de programas.

**Incluir:** `<assert.h>`

**Descrição:** A macro **`assert()`** imprime uma mensagem de erro e aborta o programa [chamando **`abort()`**] quando a expressão condicional recebida como parâmetro resulta em zero.

**Protótipo:**

```
void assert(int exp)
```

**Parâmetro:** `exp` – expressão condicional.

**Observações:**

- Se a expressão condicional resultar em zero, uma mensagem incluindo a expressão condicional, o nome do arquivo e o número da linha que contém a chamada da macro serão impressos na saída padrão de erros **`stderr`** e o programa será encerrado. A mensagem impressa em **`stderr`** pode ser algo como:

*Assertion failed: <exp>, file <arquivo>, line <linha>*

- Esta macro deve ser utilizada apenas nas fases de teste e depuração do programa (v. **Volume I**).
- Se a diretiva<sup>103</sup>:  

```
#define NDEBUG
```

for colocada antes da diretiva:  

```
#include <assert.h>
```

os usos de **assert()** não terão efeito no arquivo que contém estas diretivas.
- O problema com a macro **assert()** é que ela não permite adaptar uma mensagem mais amigável na língua nativa do usuário ou programador. O **Volume I** ensina como definir uma macro semelhante a **assert()** capaz de apresentar mensagens ao agrado do programador.

**Exemplo:** Se o comentário a seguir for removido, a macro **assert()** não terá nenhum efeito.

```
/* #define NDEBUG */
#include <assert.h>

int main()
{
    int x = -10;

    assert(x > 0);

    return 0;
}
```

## 11.5 CLASSIFICAÇÃO DE ERROS: <errno.h>

O objetivo do cabeçalho <errno.h> é definir macros (constantes inteiras) associadas a algumas situações de exceções com as quais podem se deparar muitas funções da biblioteca padrão de C. Este cabeçalho também declara uma variável global que estas funções de biblioteca utilizam para armazenar valores indicativos de erros.

---

<sup>103</sup> O nome da macro **NDEBUG** é derivado de *no debugging*, em inglês; i.e., *sem depuração*, em português.



## 11.5.1 MACROS

A **Tabela 11-3** apresenta as macros que devem ser definidas em `<errno.h>` de acordo com o padrão ISO de C.

MACRO	INTERPRETAÇÃO
<b>EDOM</b>	Argumento inválido passado para uma função matemática.
<b>EILSEQ</b>	Sequência de caracteres multibyte inválida.
<b>ERANGE</b>	Valor resultante de uma operação grande demais.

Tabela 11-3: Macros representantes de condições de erro.

A **Tabela 11-4** mostra quais funções da biblioteca padrão de C usam cada macro indicadora de erro definida em `<errno.h>`.

MACRO	FUNÇÕES QUE USAM A MACRO
<b>EDOM</b>	Todas as funções declaradas no cabeçalho <code>&lt;math.h&gt;</code> que possuem restrições de domínio [e.g., <b>asin()</b> , <b>log2()</b> ].
<b>EILSEQ</b>	<b>fgetwc()</b> , <b>fputwc()</b> , <b>mbrlen()</b> , <b>mbrtowc()</b> , <b>mbsrtowcs()</b> , <b>wcrtomb()</b> , <b>wcsrtombs()</b> .
<b>ERANGE</b>	Todas as funções declaradas no cabeçalho <code>&lt;math.h&gt;</code> que podem produzir <i>overflow</i> ou <i>underflow</i> mais as funções: <b>strtod()</b> , <b>strtof()</b> , <b>strtoumax()</b> , <b>strtol()</b> , <b>strtold()</b> , <b>strtoll()</b> , <b>strtoul()</b> , <b>strtoull()</b> , <b>strtoumax()</b> , <b>wcstod()</b> , <b>wcstof()</b> , <b>wcstoumax()</b> , <b>wcstol()</b> , <b>wcstold()</b> , <b>wcstoll()</b> , <b>wcstoul()</b> , <b>wcstoull()</b> , <b>wcstoumax()</b> .

Tabela 11-4: Funções da biblioteca padrão que usam macros de erros.

Tipicamente, implementações de C proveem várias outras macros que representam condições de erro além daquelas apresentadas na **Tabela 11-3**.

## 11.5.2 VARIÁVEL GLOBAL `errno`

**Incluir:** `<errno.h>`

**Descrição:** A variável **errno** é utilizada por diversas funções da biblioteca padrão de C para armazenar valores que indicam a ocorrência de algum erro durante a execução destas funções.

**Observações:**

- Quando um programa inicia sua execução, a variável **errno** é iniciada com zero.
- Para testar se uma dada função da biblioteca padrão armazenou algum valor em **errno**, o programa deve atribuir zero a esta variável imediatamente antes da chamada dessa função<sup>104</sup>.
- Na realidade, **errno** não precisa necessariamente ser uma variável global, mas isto é um detalhe que só interessa a quem implementa a biblioteca. Do ponto de vista do programador, para todos os efeitos práticos, não há nenhum problema em imaginar que **errno** é sempre uma variável global.

**Exemplo:** O programa a seguir demonstra o uso da variável global **errno** e da macro **EDOM**.

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

int main()
{
    double x = 1.5, y = 0.5, z;

    /* Aqui, não é necessário zerar */
    /* a variável errno, mas é bom */
    /* adquirir o hábito de fazer */
    /* isto para não esquecer quando */
    /* for realmente necessário */
    errno = 0;

    /* A função acos() atribui o valor */
    /* EDOM à variável global errno se */
    /* o valor do argumento estiver fora */
    /* do intervalo [-1, 1] */
    z = acos(x);

    if (errno == EDOM)
        printf( "O valor %f esta' fora do intervalo"
```

---

<sup>104</sup> Mesmo que o programa ainda não tenha chamado nenhuma função de biblioteca que porventura possa alterar o valor de **errno**, esta é uma prática de programação sempre recomendável para evitar algum esquecimento.

```

        " valido de acos()\n", x );
else
    printf("Valor de acos(%f): %f\n", x, z);

    /* Aqui, é estritamente necessário */
    /* zerar a variável errno. Caso */
    /* contrário, ela poderá continuar */
    /* com o valor que lhe foi atribuído */
    /* na chamada anterior de acos(). */
    errno = 0;

    z = acos(y);

    if (errno == EDOM)
        printf( "O valor %f esta' fora do intervalo"
                " valido de acos()\n", y );
    else
        printf("Valor de acos(%f): %f\n", y, z);

    return 0;
}

```

Quando executado, o programa do último exemplo produz como resultado a seguinte impressão no meio de saída padrão:

```

O valor 1.500000 esta' fora do intervalo valido de acos()
Valor de acos(0.500000): 1.047198

```

Observe no programa anterior que a variável **errno** é zerada duas vezes. A primeira atribuição de zero a esta variável feita pelo programa é dispensável<sup>105</sup>, mas a segunda é essencial. Isto é, se esta segunda atribuição for removida, o programa imprimirá incorretamente como resultado:

```

O valor 1.500000 esta' fora do intervalo valido de acos()
O valor 0.500000 esta' fora do intervalo valido de acos()

```

A segunda linha impressa pelo programa está incorreta porque o valor 0.5 é perfeitamente válido como parâmetro de **acos()**. Neste caso, o valor obtido pela variável **errno** foi decorrente da primeira chamada de **acos()**, e não da segunda chamada desta função. Daí a necessidade de zerar a variável **errno** antes da segunda chamada de **acos()**.

---

<sup>105</sup> Lembre-se de que a variável **errno** começa com zero ao início de qualquer programa.

## 11.6 TRATAMENTO DE SINAIS: <signal.h>

O propósito do cabeçalho <signal.h> é oferecer suporte para tratamento de sinais em C. No corrente contexto, **sinais** constituem uma forma de comunicação entre programas, incluindo a comunicação entre um sistema operacional e os programas executados sob sua supervisão. Mais precisamente, um sinal é usado por um programa para notificar outro programa da ocorrência de algum evento. Por exemplo, quando o usuário de um programa digita [CTRL-C], o sistema operacional DOS (ou Unix) envia um sinal para o programa informando-lhe a ocorrência deste evento.

Um sistema operacional pode enviar vários sinais para um programa a fim de notificá-lo sobre a ocorrência de eventos incomuns. Tais eventos incluem erros (e.g., acesso ilegal de memória) e interrupções de hardware (e.g., exceção de ponto flutuante). Um programa pode escolher uma das seguintes alternativas para lidar com um dado sinal:

- Permitir que o sinal receba o tratamento padrão.
- Instalar um **tratador de sinal**, que é uma função responsável pela resposta do programa ao sinal e é chamada automaticamente quando o programa recebe o sinal que ela está apta a tratar.
- Ignorar o sinal.

A primeira alternativa não requer nenhum preparativo por parte do programa, mas o tratamento padrão para a maioria dos sinais significa simplesmente abortar o programa.

A instalação de um tratador de sinal (segunda alternativa) é efetuada em dois passos:

1. **Registro** de uma função que será responsável pelo tratamento do sinal. Em C, esta operação é realizada utilizando-se a função **signal()** declarada no cabeçalho <signal.h>.
2. **Implementação** da função de tratamento de sinal de modo a responder ao sinal de forma conveniente ao programa. Esta função deve ter um protótipo específico e sua implementação deve seguir algumas regras que serão discutidas na **Seção 11.6.3**.

O cabeçalho <signal.h> provê duas funções predefinidas de tratamento de sinais representadas por dois ponteiros de funções: **SIG\_DFL** – que implementa um

tratamento padrão – e **SIG\_IGN** – que ignora o respectivo sinal e pode ser usado para implementar a última alternativa mencionada (v. **Seções 11.6.2 e 11.6.3**).

Quando um programa recebe um sinal, o sistema operacional interrompe a execução normal do programa<sup>106</sup>. Se o programa tiver registrado uma função tratadora de sinal para o referido sinal, ela será invocada; caso contrário, um tratador padrão de sinal será invocado. A execução do programa permanece suspensa até que o tratador de sinal retorne.

Tipicamente, sinais são enviados pelo sistema operacional para um programa em decorrência da digitação de uma sequência de teclas (e.g., CTRL-C) ou devido à ocorrência de uma condição de exceção (e.g., tentativa de execução de uma divisão inteira por zero).

Sinais são divididos em duas categorias:

- **Sinais síncronos** são aqueles enviados (artificialmente) pelo próprio programa que os recebe. As funções **raise()** (v. **Seção 11.6.3**) e **abort()** (v. **Seção 12.2.3**) são capazes de enviar sinais para o próprio programa no qual são chamadas. Uma característica evidente de um sinal síncrono é que o programa pode prever quando ele vai ser emitido.
- **Sinais assíncronos** são sinais enviados por outros programas, inclusive pelo sistema operacional. Um programa nunca sabe quando vai receber, ou mesmo se vai receber, um sinal assíncrono e isto pode acontecer durante a execução de qualquer instrução não atômica.

Por razões de portabilidade, uma função tratadora de sinal assíncrono deve executar apenas as seguintes operações:

- Chamar a função **signal()** (v. **Seção 11.6.3**).
- Atribuir valores a variáveis do tipo **sig\_atomic\_t** (v. **Seção 11.6.1**).
- Simplesmente, retornar.

---

106 Esta interrupção pode ocorrer durante a execução de qualquer instrução que não seja atômica.

Tratadores de sinais síncronos podem ainda, de modo portátil, abortar o programa chamando **abort()** ou **exit()** (v. **Seção 12.2.3**), ou chamar **longjmp()** (v. **Seção 11.7.2**).

Sinais podem ser emitidos por um programa para indicar a existência de condições de exceção dentro do próprio programa. Estes sinais podem ser emitidos pela função **raise()**, declarada em `<signal.h>`, e são considerados **artificiais**, porque a intenção primária do uso de sinais é, conforme mencionado anteriormente, comunicação entre programas (e não comunicação dentro de um mesmo programa).

O restante desta seção descreve em detalhes os componentes do cabeçalho `<signal.h>`.

## 11.6.1 Tipo `sig_atomic_t`

**Incluir:** `<signal.h>`

**Descrição:** O tipo inteiro **`sig_atomic_t`** é recomendado para definir variáveis não locais utilizadas por tratadores de sinais. É garantido que o acesso a uma variável deste tipo é **atômico**. Em outras palavras, a leitura ou escrita de uma variável deste tipo é sempre efetuada numa única instrução em linguagem de máquina, de modo que seu valor nunca é parcialmente lido ou escrito<sup>107</sup>.

### Observações:

- A existência deste tipo faz-se necessária porque, como um sinal assíncrono pode ocorrer durante a execução de qualquer instrução não atômica (em linguagem de máquina), uma variável que necessite de mais de uma instrução (em linguagem de máquina) para leitura ou escrita pode ter sido atualizada apenas parcialmente no instante em que o sinal é recebido. Assim, se esta variável for compartilhada entre um tratador de sinal e outras partes do programa, quando o tratador for invocado, ele estará utilizando esta variável com um valor inválido.
- O padrão de C recomenda que variáveis do tipo **`sig_atomic_t`** também sejam definidas com o qualificador **`volatile`** (v. **Volume I**).
- Variáveis locais de um tratador de sinal não precisam utilizar nem este tipo, nem o qualificador **`volatile`**.

---

<sup>107</sup> Os tipos **`signed char`** e **`unsigned char`** também possuem esta propriedade.

**Exemplo:** Veja exemplos da função `signal()` (Seção 11.6.3).

## 11.6.2 MACROS

A **Tabela 11-5** apresenta as macros (constantes), definidas em `<signal.h>`, que representam os sinais preconizados pelo padrão de C.

SINAL	O QUE MOTIVA O SINAL É...
<b>SIGFPE</b>	Tentativa de execução de alguma operação aritmética ilegal. <i>Este sinal é irrecuperável.</i>
<b>SIGILL</b>	Tentativa de execução de alguma instrução ilegal. <i>Este sinal é irrecuperável.</i>
<b>SIGSEGV</b>	Tentativa de acesso ilegal de memória. <i>Este sinal é irrecuperável.</i>
<b>SIGABRT</b>	Encerramento anormal (i.e., aborto) do programa.
<b>SIGTERM</b>	Solicitação do sistema operacional para encerrar o programa.
<b>SIGINT</b>	Interrupção causada pela digitação de <code>[CTRL-C]</code> .

Tabela 11-5: Sinais predefinidos e respectivas causas.

Observe na **Tabela 11-5** que os três primeiros sinais – **SIGFPE**, **SIGILL** e **SIGSEGV** – são irrecuperáveis. Isto significa que um tratador de qualquer um desses sinais não deve retornar. Isto é, ele deve abortar o programa, pois, caso contrário, o programa poderá ter um comportamento indefinido. Para os demais sinais da **Tabela 11-5**, se a função tratadora de sinal retornar, o programa será retomado a partir do ponto em que se encontrava quando o sinal foi recebido pelo programa.

Os sinais definidos pelo padrão de C são tão poucos e tão genéricos que um programa que precisa comunicar-se com outros programas, incluindo o sistema operacional, de modo mais sofisticado, precisa utilizar sinais específicos de cada plataforma, o que compromete a portabilidade do programa.

Dependendo do sistema operacional utilizado, outros sinais podem ser definidos em `<signal.h>`. Macros que representam sinais são constantes inteiras não negativas e são facilmente identificadas, pois todas começam com `SIG`. Muitos sistemas operacionais não permitem que um programa instale tratadores para alguns destes sinais ou os ignorem. Por exemplo, o sistema Linux não permite que um programa instale tratador para `SIGSTOP` e `SIGKILL` ou ignore estes sinais.

O cabeçalho `<signal.h>` também define macros que representam ponteiros para funções predefinidas. Estas macros, apresentadas na **Tabela 11-6**, podem ser usadas como segundo argumento da função **signal()** apresentada adiante.

PONTEIRO	AÇÃO EXECUTADA
<b>SIG_DFL</b>	Ação padrão
<b>SIG_IGN</b>	Nenhuma (i.e., o sinal será ignorado)

Tabela 11-6: Ponteiros predefinidos de funções de tratamento de sinal.

As macros apresentadas na **Tabela 11-5** e na **Tabela 11-6** serão comentadas mais especificamente a seguir.

### ***SIGFPE***

**Incluir:** `<signal.h>`

**Descrição:** Apesar de o sinal **SIGFPE** ter seu nome derivado de *ponto flutuante*, este sinal pode ser emitido por uma variedade de razões em virtude da execução de uma operação aritmética que não precisa necessariamente envolver números de ponto flutuante. Como exemplos de operações que causam o envio do sinal **SIGFPE** para o programa que tenta executá-las têm-se:

```
int x = 5/0; /* Divisão por zero*/
int y = INT_MIN/-1; /* Valor não pode ser representado */
```

**Observação:** Apesar de ser qualificado como irrecuperável, em alguns casos, um programa que recebe o sinal **SIGFPE** pode ser retomado. Mas, em nome da segurança, é melhor não correr o risco.

### ***SIGILL***

**Incluir:** `<signal.h>`

**Descrição:** Um programa recebe o sinal **SIGILL** quando tenta executar uma operação ilegal, o que pode ocorrer por diversas razões, incluindo:

- Tentativa de execução de uma função por meio de um ponteiro para função que não foi iniciado apropriadamente.



- Sobrescrita do apontador de instrução armazenado na pilha de execução, fazendo-o apontar para dados, em vez de uma instrução.
- Compilação de um programa para um conjunto específico de instruções e tentativa de executá-lo numa máquina que usa outro conjunto de instruções.
- Tentativa de execução de uma instrução que requer privilégios especiais.
- Geração de uma instrução incorreta por um compilador defeituoso.

## *SIGSEGV*

**Incluir:** <signal.h>

**Descrição:** O sinal **SIGSEGV** é recebido por um programa quando ele tenta acessar um endereço em memória cujo acesso não lhe é permitido ou quando ele tenta acessar um endereço com acesso permitido para algumas operações, mas executando uma operação não permitida. Por exemplo, em sistemas que armazenam *strings* constantes em posições de acesso apenas para leitura (v. **Volume I**), a instrução de atribuição a seguir causa o envio do sinal **SIGSEGV** para o programa que a executa:

```
char *str = "Bola";
str[1] = 'C'; /* Pode causar o aborto do programa */
```

## *SIGABRT*

**Incluir:** <signal.h>

**Descrição:** O sinal **SIGABRT** é enviado por um programa para si mesmo por meio de uma chamada da função **abort()**. Claro que este sinal poderá ser tratado, mas a execução do programa não poderá ser retomada, pois, tão logo a função de tratamento do sinal retorne, o programa será abortado devido à chamada precedente da função **abort()** que deu origem ao sinal. Tipicamente, uma função de tratamento para este tipo de sinal executa algumas operações pendentes (e.g., fechamento de arquivos) antes da finalização do programa.

## *SIGTERM*

**Incluir:** <signal.h>

**Descrição:** O sinal **SIGTERM** é um sinal enviado a um programa solicitando *gentilmente* que ele encerre sua execução. Uma função que faz o tratamento deste tipo de sinal deve ultimar os preparativos para realmente encerrar o programa.

**Observação:** Em sistemas operacionais da família Unix, existe um sinal, denominado **SIGKILL**, que não é tão *gentil* quanto **SIGTERM**. Ou seja, o sinal **SIGKILL** não pode ser nem tratado, nem ignorado, de modo que a emissão deste sinal força inexoravelmente o final de qualquer programa. Quando estes sistemas precisam ser desligados, eles inicialmente emitem o sinal **SIGTERM** para cada processo. Então, se um dado processo não se encerra após alguns segundos, ele será encerrado por meio da emissão de **SIGKILL**.

## *SIGINT*

**Incluir:** `<signal.h>`

**Descrição:** A macro **SIGINT** corresponde a sinais enviados ao programa quando o usuário digita `[CTRL-C]`. Uma função de tratamento para este tipo de sinal pode retornar sem problemas [v. exemplos da função **signal()** mais adiante].

## *SIG\_DFL*

**Incluir:** `<signal.h>`

**Descrição:** A macro **SIG\_DFL** é um ponteiro para função associado a uma função definida pela implementação que realiza a ação padrão para um dado tipo de sinal.

**Observação:** Para a maioria dos sinais, a ação padrão consiste em simplesmente abortar o programa.

## *SIG\_IGN*

**Incluir:** `<signal.h>`

**Descrição:** A macro **SIG\_IGN** é um ponteiro para função associado a uma função definida pela implementação que meramente ignora um dado tipo de sinal.

**Observações:**

- Conforme foi citado antes, nem todo sinal pode ser ignorado.
- Mesmo quando é permitido ignorar um sinal, proceder assim não significa que o programa que ignore o sinal continuará sendo executado sem problemas.

*SIG\_ERR*

**Incluir:** <signal.h>

**Descrição:** A macro **SIG\_ERR** é retornada pela função **signal()** (v. adiante) quando não consegue registrar uma função de tratamento de sinal e serve apenas para testar se uma chamada de **signal()** foi bem sucedida ou não.

**11.6.3 FUNÇÕES***raise()*

**Incluir:** <signal.h>

**Descrição:** A função **raise()** emite um sinal para o próprio programa que a chama. Se o programa tiver uma função para tratamento do sinal enviado, ela será chamada.

**Protótipo:**

```
int raise(int sinal)
```

**Parâmetro:** *sinal* – o sinal a ser emitido; os possíveis valores são representados pelas macros apresentadas na **Tabela 11-5** ou alguma outra macro específica de implementação.

**Retorno:** Zero, se a função obtém êxito; um valor diferente de zero, caso contrário.

**Observações:**

- Esta função é útil para sinalizar a ocorrência de alguma condição de exceção (v. **Seção 11.7.4**).
- Consulte também **signal()**.

**Exemplo:** Veja exemplos da função `signal()`.

*signal()*

**Incluir:** `<signal.h>`

**Descrição:** A função `signal()` instala uma função de tratamento de sinal que é automaticamente chamada quando o programa recebe um sinal especificado.

**Protótipo**<sup>108</sup>:

```
void (*signal(int sinal, void (* Funcao)(int) ))(int)
```

**Parâmetros:**

- `sinal` – valor inteiro correspondente ao sinal que será tratado (v. **Seção 11.6.2**).
- `Funcao` – ponteiro para a função a ser chamada quando surge a condição de exceção especificada por `sinal`. A função atribuída a este ponteiro deve ter o seguinte protótipo:

```
void nome-da-função(int sinal)
```

**Retorno:**

- Um ponteiro para a função de tratamento de sinal instalada anteriormente para o sinal especificado, se a operação for bem sucedida.
- **SIG\_ERR**, se ocorrer erro na tentativa de instalar a função de tratamento de sinal.

**Observações:**

- O programador pode definir sua própria função de tratamento de sinal ou utilizar um dos ponteiros predefinidos **SIG\_DFL** e **SIG\_IGN**. Se a função for representada por um destes ponteiros predefinidos, a ação correspondente, apresentada na **Tabela 11-6**, será executada.

---

<sup>108</sup> Se você não estudou o **Capítulo 10** do **Volume I**, pode ter alguma dificuldade para entender este protótipo. O que está sendo declarado aqui é uma função que recebe dois argumentos e retorna um ponteiro para função. O segundo argumento da função declarada também é um ponteiro para função.

- Logo antes de chamar a função de tratamento de sinal instalada, o sistema executa a chamada:

```
signal(sinal, SIG_DFL);
```

Isto significa que o tratamento padrão (**SIG\_DFL**) para aquele tipo de sinal será subsequentemente aplicado, a não ser que a função **signal()** seja novamente chamada, dentro da função de tratamento ora em execução, para reinstalar uma função de tratamento de sinal específica. Tipicamente, a função de tratamento reinstala a si mesma (v. exemplos abaixo).

- O ponteiro retornado pela função **signal()**, quando ele é diferente de **SIG\_ERR**, é útil para reinstalar a função de tratamento de sinal vigente antes da instalação da nova função.
- Como não há garantia que a maioria das funções da biblioteca seja reentrante, estas funções não devem ser chamadas por um tratador de sinais. A justificativa é que um programa pode receber um sinal a qualquer instante, mesmo quando está executando uma função da biblioteca padrão. Quando isto acontece e a tratadora de sinal chama esta função, ela deve ser reentrante.
- Quando o sinal tratado é assíncrono, uma função de tratamento de sinal deve evitar chamar qualquer função da biblioteca padrão, a não ser **abort()**, **\_Exit()**, ou **signal()**. A não observância desta recomendação pode levar a um comportamento indefinido do programa.
- Se o sinal tratado é decorrente de uma chamada de **raise()** ou **abort()**, a única recomendação é que a função de tratamento de sinal não chame **raise()**.

**Exemplo:** O programa a seguir mostra como o tratamento de sinais depende do sistema operacional no qual o programa é executado.

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

void TrataSIGFPE(int sig)
{
    fprintf(stderr, "\nO programa tentou executar "
                  "uma operacao aritmetica ilegal"
                  " e sera' abortado. Tchau.\n");
```

```

    exit(1); /* É o melhor a fazer nesses casos */
}

void TrataSIGSEGV(int sig)
{
    fprintf(stderr, "\nO programa tentou um acesso "
                  " ilegal de memoria e sera' "
                  "abortado. Tchau.\n");

    exit(1); /* É o melhor a fazer nesses casos */
}

void TrataSIGILL(int sig)
{
    fprintf(stderr, "\nO programa tentou executar uma "
                  "operacao ilegal e sera' abortado."
                  " Tchau.\n");

    exit(1); /* É o melhor a fazer nesses casos */
}

int main()
{
    const char *const msg = "Nao foi possivel instalar "
                           "um tratador de sinal";

    int    c;
    float x;
    char *p = NULL;
    unsigned char ar[4] = {0xff, 0xff, 0xff, 0xff};

    /* O ponteiro de função F definido a seguir */
    /* aponta para uma instrução ilegal          */
    /* (0xFFFFFFFF) processadores Pentium        */
    void (*F)() = (void (*)())ar;

    if (signal(SIGFPE, TrataSIGFPE) == SIG_ERR) {
        perror(msg);
        return 1;
    }

    if (signal(SIGSEGV, TrataSIGSEGV) == SIG_ERR) {
        perror(msg);
    }
}

```

```

        return 1;
    }

    if (signal(SIGILL, TrataSIGILL) == SIG_ERR) {
        perror(msg);
        return 1;
    }

    printf( "\nEscolha o tipo de sinal que deseja que"
           " o programa receba:\n" );
    printf( "\t1. Acesso ilegal de memoria.\n"
           "\t2. Operacao aritmetica ilegal.\n"
           "\t3. Operacao ilegal (inexistente).\n" );
    printf("\nSua opcao: ");

    while((c = getchar()) != '1' && c != '2' && c != '3')
        getchar(); /* Melhor: LimpaBuffer() [v. Vol I] */

    if ('1' == c)
        *p = 5; /* O ponteiro p é NULL. Portanto */
               /* esta instrução causará uma */
               /* tentativa de acesso ilegal */
               /* de memória. */
    else if ('2' == c)
        c = 10 / 0; /* Evidentemente, esta é */
                  /* uma operação de ponto- */
                  /* flutuante ilegal. */
    else
        F(); /* Esta função aponta para uma */
            /* instrução ilegal (inexistente). */

    return 0;
}

```

Esse programa funciona conforme esperado no sistema Linux Ubuntu 8.10 (e provavelmente em outros sistemas operacionais da família Unix), mas, quando o usuário escolhe a terceira opção do programa no sistema Windows XP, não se obtém o resultado desejado. Isto ocorre porque Windows XP não permite tratamento do sinal **SIGILL**.

**Exemplo:** O programa a seguir demonstra o uso de **signal()** e da implementação de uma função para tratar o sinal **SIGINT**.

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

/****
 *
 * Função TrataSIGINT(): trata o sinais do tipo SIGINT
 *
 * Argumentos: sinal (entrada) - o valor SIGINT
 *
 * Retorno: Nada
 *
 ****/

void TrataSIGINT(int sinal)
{
    printf( "Voce acaba de digitar CTRL-C."
           " Aguarde um pouco mais.\n" );

    /* É preciso reinstalar o tratador de sinal */
    signal(SIGINT, TrataSIGINT);
}

int main(void)
{
    int i, j;

    printf( "Realizando uma longa e inutil operacao."
           " Experimente digitar CTRL-C.\n\n" );

    signal(SIGINT, TrataSIGINT);

    /* Este é apenas um laço relativamente */
    /* demorado para manter o programa em */
    /* execução enquanto os sinais são */
    /* testados. Dependendo do computador */
    /* utilizado deve-se ajustar os valores */
    /* de modo que o tempo gasto não seja */
    /* longo ou curto demais. */
    for (i = 0; i < 5000; i++)
        for (j = 0; j < 1000000; j++);
}
```



```

    return 0;
}

```

O último programa funciona bem tanto no sistema Linux Ubuntu 8.1 quanto no sistema Windows XP. Se você desejar encerrar este programa enquanto ele estiver em execução, digite [CTRL-Break] no Windows ou [CTRL-\] em sistemas operacionais da família Unix. No Windows, [CTRL-Break] causa a emissão do sinal SIGBREAK, enquanto, em sistemas da família Unix, [CTRL-\] faz com que o programa receba o sinal SIGQUIT. Estes sinais não fazem parte do padrão de C.

**Exemplo:** O programa a seguir demonstra o uso de uma única função para tratamento de vários tipos de sinais. Ele também demonstra o uso da função **raise()** para enviar sinais síncronos para o próprio programa.

```

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

void LimpaBuffer(void);
char LeOpcao(int min, int max);
void TrataSinais(int sinal);

/* A variável global a seguir indica */
/* se o usuário digitou CTRL-C ou não */
volatile sig_atomic_t digitouCTRLC = 0;

int main(void)
{
    int x, opcao, i = ULONG_MAX;

    /* Pode-se, obviamente, instalar uma função */
    /* para cada sinal, mas aqui apenas uma      */
    /* função é usada para vários sinais         */
    signal( SIGABRT, TrataSinais );
    signal( SIGFPE, TrataSinais );
    signal( SIGINT, TrataSinais );
    signal( SIGSEGV, TrataSinais );
    signal( SIGTERM, TrataSinais );

```

```

while(1) {
    printf( "\nOpcoes:\n\n"
           "\t0:Sai do programa\n"
           "\t1:Testa sinal SIGABRT\n"
           "\t2:Testa sinal SIGFPE\n"
           "\t3:Testa sinal SIGINT\n"
           "\t4:Testa sinal SIGSEGV\n"
           "\t5:Testa sinal SIGTERM\n" );

    opcao = LeOpcao('0', '5');

    switch( opcao ) {
        case '0':
            return 0;
        case '1':
            abort();
            break;
        case '2':
            x = 10/0;
            break;
        case '3':
            digitouCTRLC = 0;
            printf( "\nDigite CTRL-C para causar"
                   " uma interrupcao\n" );
            /* Dá um tempo para o usuário */
            /*digitar CTRL-C */
            while (i--)
                if (digitouCTRLC)
                    break;

            if (!digitouCTRLC) {
                printf( "\nVoce nao digitou CTRL-C. "
                       "O programa gerara' o sinal "
                       "correspondente.\n" );
                raise(SIGINT);
            }
            break;
        case '4':
            printf("\nSimulacao de tentativa de acesso "
                  "ilegal a memoria usando raise().");
            raise(SIGSEGV);
            break;
        case '5':

```

```

        printf("\nSimulacao de solicitacao do SO "
               "para terminar o programa usando "
               "raise().");
        raise(SIGTERM);
        break;
    } /* switch */
} /* while */

return 0;
}

/****
 * Função LimpaBuffer(): Lê e descarta caracteres
 *                      encontrados na entrada padrão
 * Argumentos: Nenhum
 * Retorno: Nada
 ****/
void LimpaBuffer(void)
{
    int c;

    while( (c = fgetc(stdin)) != EOF && c != '\n' )
        ; /* Corpo Vazio */
}

/****
 * Função LeOpcao(): Lê e valida a opção digitada
 *                  pelo usuário
 * Argumentos: min (entrada) - o menor valor que o
 *                  usuário deve digitar
 *                  max (entrada) - o maior valor que o
 *                  usuário deve digitar
 * Retorno: A opção escolhida pelo usuário
 ****/
char LeOpcao(int min, int max)
{
    int op;

    while (1) {
        printf("\nDigite sua opcao: ");
        op = getchar();

        if (op >= min && op <= max) {

```

```

        LimpaBuffer();
        break;
    } else {
        printf("\nOpcao invalida. Tente novamente.\n");
        LimpaBuffer();
    }
}

return op;
}

/****
* Função TrataSinais(): trata diversos tipos de
*                       sinais recebidos pelo programa
* Argumentos: sinal (entrada) - o sinal recebido
*                       pelo programa
* Retorno: Nada
****/
void TrataSinais(int sinal)
{
    int opcao = 0;

    printf("\n\nFuncao de tratamento de sinais:");

    switch( sinal ) {
        case SIGABRT:
            printf( "\n\tSinal de termino anormal recebido."
                    "\n\tO programa sera encerrado quando "
                    "esta funcao retornar.\n\tDigite uma "
                    "tecla para encerrar o programa." );
            getchar();
            break;
        case SIGFPE:
            printf( "\n\tSinal de operacao aritmetica "
                    "invalida recebido.\n\tSe esta funcao "
                    "retornar o programa entrara em loop "
                    "infinito.\n\tEste tipo de erro e' "
                    "irrecuperavel. Portanto, o programa "
                    "\n\tsera' abortado quando voce "
                    "digitar ENTER" );
            getchar();
            exit(1); /* Sem esta instrucao o */
            break; /* programa entra em loop */
    }
}

```

```

case SIGINT:
    printf( "\n\tSinal de interrupcao "
           "[CTRL-C] recebido.\n" );
    digitouCTRLC = 1;
    break;

case SIGSEGV:
    printf( "\n\tSinal de tentativa de acesso "
           "ilegal a memoria recebido.\n" );
    break;
case SIGTERM:
    printf( "\n\tSinal de solicitacao do SO para "
           "encerrar o programa recebido.\n" );
    break;
}

/* É necessário reinstalar a função de */
/* tratamento de sinal; caso contrário, */
/* da próxima vez que surgir este sinal, */
/* SIG_DFL será chamada. */
signal(sinal, TrataSinais);
}

```

**Exercício:** Execute o último programa nos sistemas Linux e Windows e observe as diferenças.

**Exemplo:** O programa a seguir é, ao mesmo tempo, interessante e espirituoso, pois mostra como um estudante poderia enganar seu instrutor mascarando a ocorrência de uma violação de memória<sup>109</sup>. Tente entender este programa, mas não o utilize com a intenção original do estudante...

```

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

/*****

```

---

<sup>109</sup> Este programa é baseado num programa sugerido em nota de aula do professor Jian Huang do departamento de Engenharia Elétrica e Ciência da Computação da Universidade do Tennessee. **Cuidado:** Em sistemas operacionais da família Windows você pode ter dificuldade para encerrar este programa. Em sistemas da família Unix, basta digitar [CTRL-] para encerrá-lo.

```

*
* Função TrataSIGINT(): impede que o programa seja
*                       encerrado com CTRL-C
*
* Argumentos: sinal (entrada) - o sinal recebido
*                       (i.e., SIGINT)
*
* Retorno: Nada
*
****/
void TrataSIGINT(int sinal)
{
    signal(SIGINT, TrataSIGINT);

    while(1)
        ; /* Vazio */
}

/****
*
* Função TrataSIGSEGV(): "faz de conta" que existe um
*                       problema com o servidor de rede
*
* Argumentos: sinal (entrada) - o sinal recebido
*                       (i.e., SIGSEGV)
*
* Retorno: Nada
*
****/
void TrataSIGSEGV(int sinal)
{
    signal(SIGINT, TrataSIGINT);

    fprintf( stderr,
              "O servidor nao esta' respondendo."
              " Tentando novamente...\n" );

    while(1)
        ; /* Vazio */
}

int main(void)
{

```

```

char *s = NULL;

signal(SIGSEGV, TrataSIGSEGV);

    /* A instrução seguinte causa o */
    /* recebimento do sinal SIGSEGV */
strcpy(s, "Bola");

return 0;
}

```

## 11.7 DESVIOS GENERALIZADOS: <setjmp.h>

Uma instrução de desvio, como **goto** ou qualquer outra (v. **Capítulo 1 do Volume I**), não permite que se executem desvios entre funções. As funções descritas nesta seção ignoram os protocolos normais de chamada e retorno de funções e permitem desvios entre funções.

Provavelmente, a aplicação prática mais comum destes desvios é a implementação de tratamento de exceções em C simulando um mecanismo semelhante àquele encontrado em linguagens orientadas a objeto, como C++ e Java.

### 11.7.1 Tipo jmp\_buf

**Incluir:** <setjmp.h>

**Descrição:** **jmp\_buf** é o tipo de uma variável usada para armazenar as informação salvas por **setjmp()** e restauradas por **longjmp()**. Mais precisamente, uma variável deste tipo é capaz de armazenar o conteúdo dos registradores e parte da pilha de execução, que representam o estado de execução de um programa.

**Observação:** Consulte também a **Seção 11.7.3**.

### 11.7.2 FUNÇÕES longjmp() E setjmp()

As funções **setjmp()** e **longjmp()** devem ser utilizadas em conjunto e numa ordem específica. Ou seja, um programa deve chamar **setjmp()** antes de executar qualquer chamada correspondente de **longjmp()**. A apresentação destas funções na presente

seção seguirá esta ordem de chamada.

**Observação importante:** Ao estudar esta seção, o leitor deve considerar que **setjmp()** e **longjmp()** não são funções convencionais. Isto é, a função **longjmp()** ignora o protocolo normal de chamada e retorno de funções. Por exemplo, a função **longjmp()** nunca retorna ao ponto em que foi chamada. Portanto, não tente entender o funcionamento dessas funções fazendo analogia com outras funções. Um completo entendimento destas funções requer algum conhecimento básico de Assembly e de como um programa é executado.

*setjmp()*

**Incluir:** <setjmp.h>

**Descrição:** A função **setjmp()** prepara o programa para a execução de um desvio entre funções efetuado por uma chamada posterior de **longjmp()**. Ou seja, esta função guarda informações relevantes sobre o estado corrente de execução do programa que poderão posteriormente ser usadas por **longjmp()** para retornar ao ponto do programa onde **setjmp()** foi chamada.

**Protótipo:**

```
int setjmp(jmp_buf buffer)
```

**Parâmetro:** estado – estrutura do tipo **jmp\_buf** que armazena o conteúdo dos registradores e parte da pilha de execução no momento da chamada da função.

**Retorno:**

- Zero quando a função é chamada diretamente; i.e., quando o argumento `estado` está sendo iniciado
- Um valor diferente de zero, quando o retorno é decorrente de uma chamada de **longjmp()**. Neste último caso, o valor retornado é o valor do segundo argumento de **longjmp()** e este valor nunca é igual a zero<sup>110</sup>.

---

<sup>110</sup> Conforme foi antecipado, as funções **setjmp()** e **longjmp()** não são funções *normais*. Quando a função **longjmp()** retorna, ela o faz como se tivesse retornando de uma chamada correspondente de **setjmp()**. Se você não entendeu o funcionamento desta função, considere isto normal, pois é virtualmente impossível entender o funcionamento de **setjmp()** isoladamente, sem levar em consideração o comportamento de **longjmp()**, que será explorado mais adiante.



**Observações:**

- A função **setjmp()** deve ser sempre chamada antes de uma chamada correspondente de **longjmp()**.
- A função que chama **setjmp()** não deve retornar antes de **longjmp()** ser chamada. Caso contrário, o resultado será indefinido.
- Quando um parâmetro de função ou variável local está sujeita a uma modificação entre uma chamada de **setjmp()** e uma chamada correspondente de **longjmp()**, deve-se qualificar o parâmetro ou a variável com **volatile** para prevenir um resultado inesperado. Por exemplo:

```
void F(volatile tipo *ptr)
{
    volatile int vi;
    jmp_buf      buffer;

    if (setjmp(buffer)) {
        /* Retorno via longjmp(). Aqui, ptr e vi */
        /* terão valores definidos pois eles foram */
        /* qualificados com volatile.                */
    } else {
        /* Retorno de setjmp() */
    }

    /* Instruções que alteram ptr e vi */
}
```

- Consulte também a descrição de **longjmp()** e a **Seção 11.7.3**.

**Exemplo:** Veja exemplo da função **longjmp()**.

*longjmp()*

**Incluir:** <setjmp.h>

**Descrição:** A função **longjmp()** permite desviar o fluxo de execução de um programa para um ponto previamente estabelecido por meio de uma chamada da função **setjmp()**.

**Protótipo:**

```
void longjmp(jmp_buf estado, int retorno_setjmp)
```

**Parâmetros:**

- `estado` – estado de execução do programa iniciado por uma chamada prévia de `setjmp()`.
- `retorno_setjmp` – valor retornado por `setjmp()` quando a chamada de `longjmp()` é encerrada.

**Retorno:** A função `longjmp()` nunca retorna no sentido convencional; i.e., ao ponto do programa onde foi chamada. Em vez disso, ela retorna como se `setjmp()` tivesse retornado com o valor de retorno determinado por `retorno_setjmp`.

**Observações:**

- A função `setjmp()` deve ser chamada para iniciar o parâmetro `estado` antes de `longjmp()` ser chamada.
- A função que chama `setjmp()` e define a variável do tipo `jmp_buf` passada para `setjmp()` deve ainda estar em execução e não pode retornar antes de uma chamada correspondente de `longjmp()`. Caso contrário, o resultado é indefinido.
- Se o valor do segundo parâmetro for zero, ele será substituído por 1.
- O padrão de C recomenda que o valor retornado por `setjmp()` indiretamente via `longjmp()` seja utilizado apenas em comparações (v. exemplos a seguir).
- O uso de `longjmp()` e `setjmp()` pode tornar um programa ainda mais difícil de entender do que o uso de desvios incondicionais (v. **Capítulo 1 do Volume I**).
- Consulte também a descrição de `setjmp()` e a **Seção 11.7.3**.

**Exemplo:** O exemplo a seguir não tem nenhuma utilidade prática, mas ilustra bem o funcionamento conjunto de `longjmp()` e `setjmp()`.

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

```

/****
 *
 * Esta função serve apenas para
 * demonstrar o uso de longjmp()
 *
 ****/

void UmaFuncao(jmp_buf status)
{
    /* Se o valor do segundo argumento for          */
    /* trocado por zero, o resultado será o mesmo */
    longjmp(status, 1);

    /* Se fosse uma função convencional, */
    /* longjmp() retornaria neste ponto. */
    printf("\nEsta mensagem nunca sera' impressa\n");
}

int main(void)
{
    int      retorno;
    jmp_buf estado;
    double   umDouble = 10.5;

    /* Quando setjmp() é chamada diretamente, */
    /* ela sempre retorna zero.                */

    /* Tanto setjmp() quanto longjmp()      */
    /* retornam neste ponto do programa. */
    retorno = setjmp(estado);

    printf("Esta mensagem sera' impressa duas vezes\n");

    if (retorno) { /* setjmp() retornou indiretamente */
                    /* por meio de longjmp().          */
        printf( "longjmp() retornou com valor %d\n",
                retorno );

        /* Aqui, o valor da variável umDouble */
        /* é indefinido. Ela deveria ter sido */
        /* definida com volatile.              */
        printf("Valor de umDouble: %f", umDouble);
    }
}

```

```

        return retorno;
    } else /* setjmp() foi diretamente chamada */
        printf( "Valor retornado por setjmp(): %d\n",
                retorno );

    umDouble *= 2.0;
    printf("Valor de umDouble: %f", umDouble); /* OK */

    UmaFuncao(estado);

    return 0;
}

```

Quando compilado e executado no Windows XP, o último programa apresenta o seguinte resultado:

```

Esta mensagem sera' impressa duas vezes
Valor retornado por setjmp(): 0
Valor de umDouble: 21.000000
Esta mensagem sera' impressa duas vezes
longjmp() retornou com valor 1
Valor de umDouble: 21.000000

```

Na execução do programa do último exemplo, a variável `umDouble` apresenta o mesmo valor nas duas chamadas de **printf()** que a imprimem, mas o padrão de C não garante que isto ocorra. Isto é, esta garantia só haveria se a variável `umDouble` fosse definida com **volatile**.

A **Seção 11.7.4** apresenta exemplos práticos de uso das funções **longjmp()** e **setjmp()**.

### 11.7.3 COMO ENTENDER **longjmp()** E **setjmp()**

Quando a função **setjmp()** é chamada, ela guarda os valores armazenados nos registradores, incluindo o apontador de pilha (`sp`) e o apontador de frame (`fp`), na estrutura do tipo **jmp\_buf** recebida como argumento. Antes de explicar em maiores detalhes o funcionamento de **setjmp()** e **longjmp()**, é necessário fazer uma observação importante com relação ao tipo **jmp\_buf**.

Na realidade, o tipo **jmp\_buf** não é exatamente uma estrutura, apesar de a literatura de programação em C com frequência se referir a este tipo como tal. Se isto fosse verdade, como uma função poderia armazenar algum valor num parâmetro real deste

tipo<sup>111</sup>? De fato, este tipo é definido como um array de um único elemento que é a própria estrutura a qual a literatura sobre o assunto costuma fazer referência. Tipicamente, este tipo é definido conforme esboçado a seguir:

```
typedef struct {
    unsigned int  fp;    /* Apontador de frame    */
    unsigned int  sp;    /* Apontador de pilha    */
    unsigned char sreg;  /* Registrador de status */
    unsigned int  pc;    /* Apontador de instrução */
    ...              /* Outros registradores  */
} jmp_buf[1];
```

O estado de execução de um programa depende do conteúdo da memória alocada para sua execução e de seus registradores. Assim, quando a função **longjmp()** é chamada recebendo como primeiro argumento uma variável contendo os registradores armazenados numa chamada de **setjmp()**, ela restaura os conteúdos dos registradores e a execução do programa continua como se tivesse havido um retorno da função **setjmp()**. Isto ocorre porque o apontador de instrução (**pc**) é restaurado junto com os demais registradores.

Como **setjmp()** armazena o conteúdo de todos os registradores, incluindo **sp** e **fp**, se a função **longjmp()** for chamada após a função que fez a chamada correspondente de **setjmp()** ter retornado, os dados armazenados na respectiva estrutura **jmp\_buf** tornar-se-ão inválidos. Isto ocorre porque, quando **longjmp()** restaurar os registradores **sp** e **fp**, a pilha terá uma configuração diferente. Para ilustrar essa situação, considere o programa a seguir.

**Exemplo:** O programa a seguir será provavelmente abortado porque a função **F1()** que chama **setjmp()** retorna antes da chamada correspondente de **longjmp()**.

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

int F1(char *s, jmp_buf estado)
{
    int i;
    i = setjmp(estado);
```

---

<sup>111</sup> Lembre-se de que, em C, só existe passagem de parâmetros por valor (v. **Capítulo 3** do **Volume I**). Assim, se uma estrutura é recebida como parâmetro, qualquer alteração feita pela função na estrutura incide apenas na cópia do parâmetro real recebido. Portanto, após o retorno da função, a estrutura passada como parâmetro teria o mesmo valor que tinha antes da chamada desta função.

```

    printf("F1(): i = %d, s = %s\n", i, s);

    return i;
}

void F2(int j, jmp_buf estado)
{
    printf("F2(): j = %d\n", j);

    longjmp(estado, j);
}

int main()
{
    jmp_buf estado;

    F1("Asa", estado);

    F2(5, estado);

    return 0;
}

```

Quando o programa inicia, a configuração da pilha de execução pode ser visualizada como mostra a **Figura 11-1** a seguir.

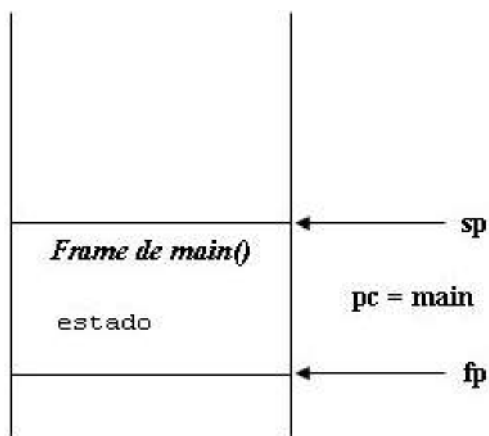


Figura 11-1: Configuração da pilha ao início do programa.

Na **Figura 11-1**, estado é a variável definida no início da função **main()** e  $pc = main$  significa que o apontador de instrução aponta para a próxima instrução a ser executada na função **main()**. Quando esta função chama **F1()**, ela, inicialmente, armazena, na pilha, os parâmetros utilizados na chamada, e a configuração da pilha torna-se aquela mostrada na **Figura 11-2**<sup>112</sup>.

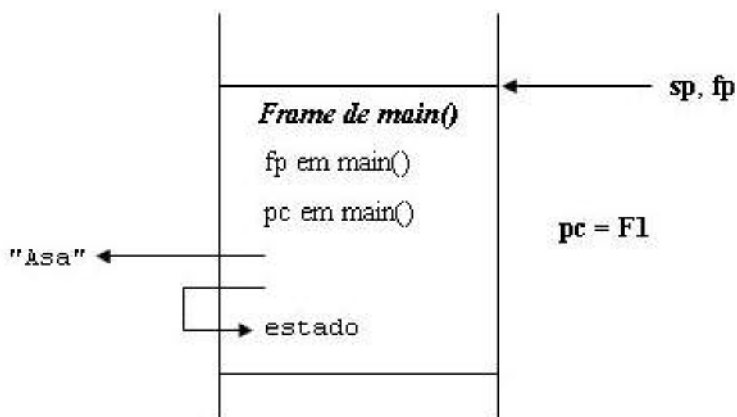


Figura 11-2: Configuração da pilha quando **F1()** é chamada.

Ambos os parâmetros armazenados na pilha na chamada de **F1()** são ponteiros: o primeiro aponta para a variável **estado** definida em **main()** e o segundo aponta para o local em memória onde a *string* "Àsa" se encontra armazenado. Ao iniciar sua execução, a função **F1()** armazena a variável local **i** e a configuração da pilha passa a ser a mostrada na **Figura 11-3** a seguir.

<sup>112</sup>  $pc = F1$  significa que o apontador de instrução aponta para a próxima instrução (em linguagem de máquina) a ser executada na função **F1()**.

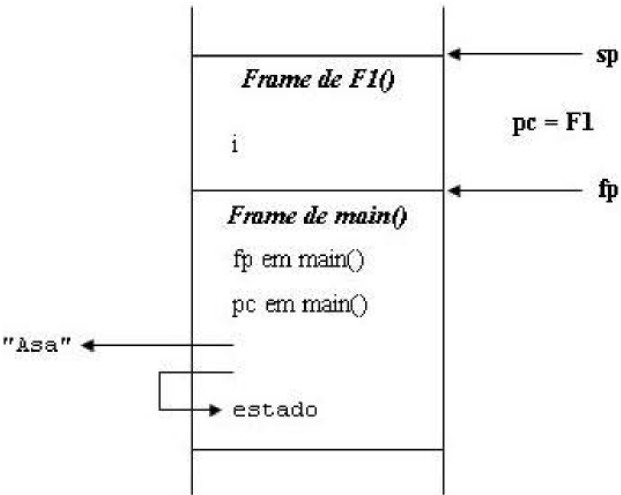


Figura 11-3: Configuração da pilha quando começa a execução de F1().

Em seguida, a função F1 ( ) chama as funções **setjmp()** e **printf()**. Utilizando o segundo parâmetro recebido por F1 ( ) , a função **setjmp()** atualiza o conteúdo da variável **estado** definida em **main()**. Ao retornar, a função F1 ( ) deixa a pilha no estado em que se encontrava antes de sua chamada, exceto pelo fato de a variável **estado** ter sido alterada, como mostra a **Figura 11-4** a seguir<sup>113</sup>.

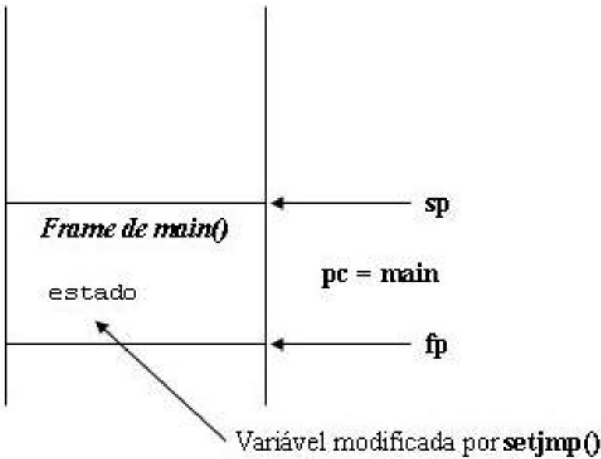


Figura 11-4: Configuração da pilha quando F1() retorna.

113 *pc = main* significa que o apontador de instrução aponta para a próxima instrução (em linguagem de máquina) a ser executada na função **main()**.



Prosseguindo, a função **main()** chama a função **F2()**. Esta última função requer dois parâmetros que são, então, armazenados na pilha, como mostra a **Figura 11-5**:

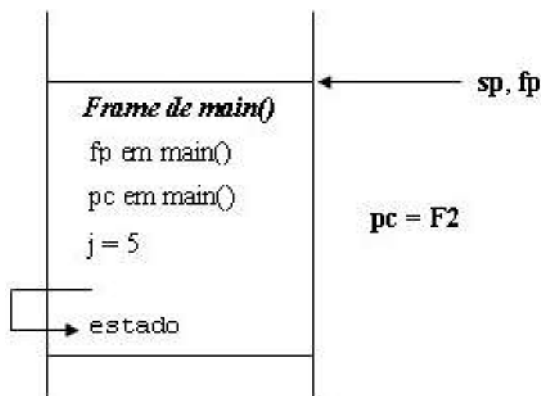


Figura 11-5: Configuração da pilha quando F2() é chamada.

Então, a função **F2()** chama **printf()** e, em seguida, **longjmp()**. Esta última função restaura os registradores que foram guardados na chamada de **setjmp()**. Acontece que, quando foram guardados, estes registradores referiam-se ao estado do programa quando a função **F1()** foi chamada (v. **Figura 11-2**) e esta função já encerrou sua execução. Assim, a pilha de execução assume a configuração mostrada na **Figura 11-6**.

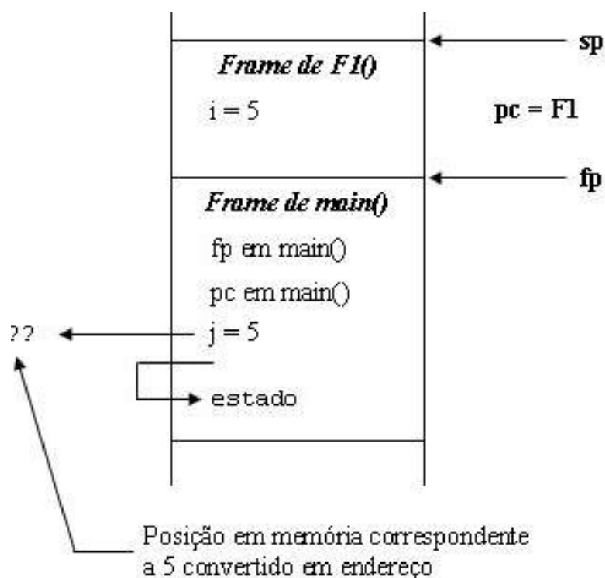


Figura 11-6: Configuração da pilha quando F2() retorna.

Conforme pode ser observado na **Figura 11-6**, quando os registradores são restaurados, a função `F1()` passa a ser executada a partir do ponto onde `setjmp()` foi chamada, conforme esperado. Acontece que, agora, a pilha não apresenta mais a configuração esperada por `F1()`. Em particular, no local da pilha onde esta função esperava encontrar um *string* (v. **Figura 11-3**), ela encontra o valor inteiro 5 correspondente ao parâmetro com o qual a função `main()` chamou a função `F2()` (v. **Figura 11-5**). Assim, quando a função `printf()` é chamada por `F1()`, ela tenta interpretar a posição em memória correspondente ao valor inteiro 5 encontrado na pilha como um *string*. Por conseguinte, provavelmente, o programa será abortado<sup>114</sup>.

### 11.7.4 TRATAMENTO DE EXCEÇÕES USANDO `longjmp()` E `setjmp()`

O uso prático mais comum das funções `setjmp()` e `longjmp()` é o tratamento de exceções. **Exceções** são situações que impedem o funcionamento normal de um programa ou, em outras palavras, alteram seu fluxo normal de execução. Como exemplos de exceções pode-se citar: tentativa de abertura para leitura de um arquivo inexistente, tentativa de execução de uma operação ilegal (e.g., divisão por zero) e tentativa frustrada de alocação dinâmica de memória.

**Tratamento de exceção** é uma resposta apresentada por um programa a uma circunstância excepcional que surge enquanto o programa está sendo executado. O tratamento de exceções oferecido pela linguagem C é bastante rudimentar e consiste basicamente em uma combinação das seguintes abordagens:

- Uso de um valor de retorno que indica quando uma função não pode ser executada normalmente [e.g., a função `malloc()` retorna `NULL` quando não consegue alocar memória]. Esta abordagem apresenta dois problemas básicos: (1) a função que faz a chamada precisa sempre testar o valor retornado prejudicando assim o fluxo natural do programa e (2) nem sempre resta um valor considerado inválido que possa ser retornado pela função (e.g., se uma função pode retornar como válido qualquer valor inteiro possível, não lhe restará nenhum valor para indicar a ocorrência de exceção).
- Uso de sinais. Esta abordagem utiliza as funções `raise()` e `abort()` para gerar sinais e é bastante limitada (v. **Seção 11.6** e **Seção 11.8**).

---

<sup>114</sup> De fato, este programa foi abortado em testes realizados nos sistemas Windows XP e Linux Ubuntu 8.10.

- Uso de uma variável global, como **errno** (v. **Seção 11.5**) para indicar ocorrência de erro. Existem três problemas com esta abordagem: (1) o programador precisa iniciar esta variável antes de chamar a função, (2) o programa precisa testá-la após chamar a função e (3) se o programa usar múltiplas linhas de execução, talvez não seja possível determinar qual foi a função que alterou o valor da variável quando for o caso.

Linguagens modernas, como C++, Java e C#, utilizam um mecanismo elegante de tratamento de exceções que supera os problemas aqui apontados. Todas estas linguagens utilizam pequenas variações de um mecanismo básico. A seguir será descrito brevemente o tratamento de exceções usado em C++<sup>115</sup>.

O mecanismo de tratamento de exceções em C++ consiste basicamente numa forma de transferir o controle de uma parte do programa que gera uma exceção para outra parte dele que faz o tratamento adequado da exceção. A abordagem possui três componentes<sup>116</sup>:

- **Lançador de exceção.** Este componente é uma instrução, representada pela palavra-chave (de C++) *throw*, semelhante a uma instrução de retorno. Mas o valor retornado por esta instrução não corresponde ao tipo de valor retornado normalmente pela função. Quando ocorre uma condição de exceção, esta instrução é executada e isto é denominado **lançamento de exceção**.
- **Bloco *try*.** Sintaticamente, este componente é como um bloco comum de C, mas é precedido pela palavra-chave (de C++) *try*. Neste bloco são colocadas chamadas de funções que podem lançar exceções, mas outras instruções e chamadas de funções que não lançam exceções também podem ser inseridas nele.
- **Blocos *catch*.** Um bloco *catch* é semelhante em C a uma função de um único argumento, mas sem tipo de retorno. Cada bloco *catch* é um **tratador de exceção** e o tipo do argumento único que ele recebe corresponde exatamente ao tipo de exceção que ele trata.

Cada bloco *try* está associado a tantos blocos *catch* quanto forem os tipos de exceções que podem ser lançadas em seu interior. O bloco *catch* que casa com o tipo de alguma exceção lançada no bloco *try* associado tem suas instruções executadas. Se nenhuma exceção é lançada, todos os blocos *catch* são saltados<sup>117</sup>.

---

115 Uma completa discussão sobre este mecanismo está além do escopo deste livro. A descrição apresentada aqui é, sem dúvida, rudimentar e deve ser suficiente apenas para facilitar o entendimento de como este mecanismo pode ser razoavelmente implementado em C usando as funções **longjmp()** e **setjmp()**.

116 Existe ainda um componente denominado *finally*, mas este não receberá consideração aqui porque não contribui para a discussão que segue.

117 Pode-se especificar um bloco *catch* capaz de capturar qualquer tipo de exceção utilizando-se *elipse* (três pontos).

Por exemplo, suponha que uma função `F()` lance exceções dos tipos `tExcecao1` e `tExcecao2`. Então, a chamada desta função deve ser colocada num bloco *try* com dois blocos *catch* associados. Cada bloco *catch* é responsável pelo tratamento de uma das exceções lançadas pela função `F()`, conforme mostrado esquematicamente a seguir (lembre-se que este código é em C++ e não em C):

```
int main()
{
    ...
    try {      /* Bloco try */
        ...
        F(); /* Esta função pode lançar exceção */
        ...
    }
    catch(tExcecao1 e)    /* Bloco catch */
    {
        /* Tratamento de exceção do tipo tExcecao1 */
    }
    catch(tExcecao2 e)    /* Outro bloco catch */
    {
        /* Tratamento de exceção do tipo tExcecao2 */
    }
    ...
}
```

Note que o esquema apresentado funciona mesmo que a função `F()` não lance diretamente nenhuma exceção mas chame funções que lancem as exceções `tExcecao1` e `tExcecao1`.

Em resumo, a sequência de eventos quando ocorre uma exceção é a seguinte:

1. O programa está sendo executado normalmente fora de qualquer bloco *try*.
2. O fluxo de execução entra num bloco *try*.
3. Ocorre uma exceção em alguma função chamada dentro do bloco *try*.
4. A função lança uma exceção.
5. O controle é transferido para o bloco *catch* que trata a respectiva exceção<sup>118</sup>.

---

<sup>118</sup> Quando uma exceção é lançada e não é capturada por nenhum bloco *catch*, o mecanismo de exceção chama uma função que, como padrão, encerra o programa.

O mecanismo de tratamento de exceção examinado aqui e aqueles suportados por outras linguagens modernas dependem de suporte *runtime*. Portanto, tal mecanismo não pode ser implementado completamente em C, cujo padrão não propõe suporte semelhante. Entretanto, pode-se razoavelmente simular um mecanismo semelhante ao descrito usando-se **setjmp()** e **longjmp()**. Os exemplos apresentados a seguir ilustram a abordagem de uso de **setjmp()** e **longjmp()** em tratamento de exceções.

**Exemplo:** O próximo programa simula o mecanismo de tratamento de exceções de C++ usando as funções **setjmp()** e **longjmp()**.

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

/* Tipos de exceções que este programa pode lançar */
typedef enum {EX1 = 1, EX2, EX3} tExcecao;

jmp_buf estado; /* Variável global */

/* Alusões */
extern void F1(int);
extern void F2(int);
extern void F3(int);

void F1(int x)
{
    printf("\nExecutando F1()");

    if (!x)
        longjmp(estado, EX1);
    else if (x < 0)
        longjmp(estado, EX2);

    F2(x);
}

void F2(int x)
{
    printf("\nExecutando F2()");

    F3(x);
}

void F3(int x)
```

```

{
    printf("\nExecutando F3()");

    if (x < 10)
        longjmp(estado, EX3);
}

int main(int argc, char** argv)
{
    switch(setjmp(estado)) {
        case 0:
            printf("\nDentro do bloco try");
            F1(5);
            break;
        case EX1:
            printf("\nExcecao EX1 capturada\n");
        case EX2:
            printf("\nExcecao EX2 capturada\n");
        case EX3:
            printf("\nExcecao EX3 capturada\n");
    }

    return 0;
}

```

Quando executado esse programa imprime o seguinte no meio de saída padrão:

```

Dentro do bloco try
Executando F1()
Executando F2()
Executando F3()
Excecao EX3 capturada

```

No último programa, a simulação do mecanismo *try-catch* de C++ é realizada por meio da instrução **switch**, e o lançamento de exceções é implementado usando-se chamadas de **longjmp()**. A variável `estado` é definida como global para evitar que ela tenha de ser passada para cada função que lance exceção. Finalmente, os tipos de exceções que este programa pode lançar são as constantes do tipo enumeração `tExcecao`<sup>119</sup>.

---

<sup>119</sup> É realmente importante que nenhuma das constante tenha valor zero, pois elas serão utilizadas como segundo argumento em chamadas de **longjmp()** e, conforme foi visto, se este valor for 0, esta função o substituirá por 1. Além disso, como estas constantes farão parte de casos de uma instrução **switch**, ter-se-iam dois casos com valor 0. Para garantir que nenhuma constante de uma enumeração assuma o valor zero, basta atribuir um valor positivo qualquer à primeira constante da enumeração (v. **Volume I**).

A lógica do programa anterior em si é simples de entender. A expressão usada com a instrução **switch** consiste numa chamada de **setjmp()** e esta chamada retorna zero, conforme visto antes. Portanto, esta porção inicial da instrução **switch** em conjunto com o bloco:

```
case 0:
    printf("\nDentro do bloco try");
    F1(5);
    break;
```

corresponde a um bloco *try* de C++, pois sabe-se que, nesse local, **longjmp()** ainda não foi chamada (i.e., nenhuma exceção foi lançada ainda). Os demais casos da instrução **switch** correspondem a blocos *catch* de C++, pois eles serão acionados quando **setjmp()** retornar (virtualmente) um valor diferente de 0. Mas, isto só ocorre quando **longjmp()** é chamada (i.e., quando uma exceção é lançada).

**Exemplo:** A simulação do mecanismo de tratamento de exceções de C++ implementada no último programa torna-se ainda mais evidente com o uso de macros, como mostra o programa a seguir que é equivalente ao programa anterior.

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

#define TRY      do{ switch(setjmp(estado)){ case 0:
#define CATCH(e) break; case e:
#define FIM_TRY } } while(0)
#define THROW(e) longjmp(estado, e)

    /* Tipos de exceções que este programa pode lançar */
typedef enum {EX1 = 1, EX2, EX3} tExcecao;

jmp_buf estado; /* Variável global */

    /* Alusões */
extern void F1(int);
extern void F2(int);
extern void F3(int);

void F1(int x)
{
    printf("\nExecutando F1()");
```

```

        if (!x)
            THROW(EX1);
        else if (x < 0)
            THROW(EX2);

        F2(x);
    }

void F2(int x)
{
    printf("\nExecutando F2()");

    F3(x);
}

void F3(int x)
{
    printf("\nExecutando F3()");

    if (x < 10)
        THROW(EX3);
}

int main(int argc, char** argv)
{
    TRY
    {
        printf("\nDentro do bloco try");
        F1(5);
    }
    CATCH(EX1)
    {
        printf("\nExcecao EX1 capturada\n");
    }
    CATCH(EX2)
    {
        printf("\nExcecao EX2 capturada\n");
    }
    CATCH(EX3)
    {
        printf("\nExcecao EX3 capturada\n");
    }
    FIM_TRY;
}

```



```
    return 0;
}
```

O entendimento do último programa requer um bom conhecimento de macros, que são completamente exploradas no **Volume I**. Mas existe um detalhe que merece ser discutido aqui: macros envolvem a instrução **switch** numa instrução **do-while**, de modo que os dois programas não são exatamente equivalentes, apesar de produzirem os mesmos resultados. O uso de **do-while** na definição de macros contendo múltiplas instruções é um jargão bastante utilizado na prática e também é discutido em detalhes no **Volume I** desta obra.

Comparada com as técnicas de tratamento de exceções implementadas em linguagens modernas, tais como C++ e Java, a técnica apresentada aqui é bastante rudimentar e expõe diversas fragilidades. Discutir os pontos fracos desta abordagem com relação às usadas por linguagens mais modernas requer uma discussão mais profunda de tratamento de exceções, o que está bem além do escopo deste livro. De qualquer modo, a simulação de tratamento de exceções proposta aqui supera, razoavelmente, muitos dos problemas decorrentes do tratamento ainda mais tosco descrito no início desta seção. Neste aspecto, essa simulação talvez seja o melhor que se consiga fazer em termos de tratamento de exceções em C e, como diz o adágio popular, quem não tem cão, caça com gato.

## 11.8 RELAÇÃO ENTRE SINAIS, INTERRUPÇÕES E EXCEÇÕES

Apesar de serem conceitos diferentes, existe uma estreita relação entre sinais, interrupções e exceções. Esta seção visa esclarecer algumas dúvidas associadas a esta relação.

Um programa pode causar uma **exceção de hardware** quando tenta, por exemplo, executar uma operação ilegal, tal como uma divisão por zero. Neste caso, a CPU envia um **pedido de interrupção** que é interceptado pelo sistema operacional. Então, o sistema operacional envia um **sinal** para o programa que causou o problema<sup>120</sup>. Se este programa tiver registrado um tratador para o respectivo sinal, ele será invocado; caso contrário, o tratador padrão para o mesmo sinal será executado, o que, caracteristicamente, causa o encerramento do programa. Conclui-se, portanto, que o envio de sinal causado por uma exceção de hardware é um **evento assíncrono** (v. **Seção 11.6**).

---

<sup>120</sup> Algumas vezes, o sistema operacional é capaz de lidar sozinho com um pedido de interrupção decorrente de uma exceção de hardware, como, por exemplo, quando a falha é decorrente da ausência de uma página de memória virtual.

O objetivo principal do tratamento de **exceções de alto nível** descrito na **Seção 11.7.4** é evitar que ocorram exceções de hardware (i.e., **exceções de baixo nível**). Exceções de alto nível são sempre enviadas de uma parte de um programa para outra parte do mesmo programa. Portanto, o lançamento de exceções de alto nível trata-se de um **evento síncrono**.

## 11.9 EXERCÍCIOS DE REVISÃO

1. Por que é preferível implementar uma macro similar à **assert()** (como exemplificado no **Capítulo 5** do **Volume I**) a usar a macro **assert()** definida no cabeçalho `<assert.h>`?
2. (a) Para que serve a variável global **errno**? (b) O padrão C99 especifica qual deve ser o tipo desta variável?
3. Quais são e para que servem as macros definidas no cabeçalho `<stdbool.h>`?
4. (a) Com que objetivo foi criado o cabeçalho `<iso646.h>`? (b) Que outra utilidade, além da primária, pode ter este cabeçalho?
5. O que especifica o padrão ISO 646?
6. A macro **NDEBUG** é definida na biblioteca padrão de C?
7. O que representam as seguintes macros?
  - (a) **EDOM**
  - (b) **EILSEQ**
  - (c) **ERANGE**
8. Que funções da biblioteca padrão de C utilizam as seguintes macros?
  - (a) **EDOM**
  - (b) **EILSEQ**
  - (c) **ERANGE**
9. (a) O que é tratamento de sinais? (b) Quais são as alternativas disponíveis para um programa tratar sinais em C?

10. Como um tratador de sinal é instalado em C?
11. O que são (a) sinais síncronos e (b) sinais assíncronos?
12. Que cuidados devem ser tomados no tratamento de sinais assíncronos para garantir a portabilidade de um programa?
13. O que é uma condição de exceção?
14. Por que sinais emitidos pela função **raise()** são considerados artificiais?
15. (a) Para que serve o tipo **sig\_atomic\_t**? (b) Em que situações ele deve ser usado?
16. Descreva os sinais representados pelas macros a seguir:
  - (a) **SIGFPE**
  - (b) **SIGILL**
  - (c) **SIGSEGV**
  - (d) **SIGABRT**
  - (e) **SIGTERM**
  - (f) **SIGINT**
17. (a) O que é um sinal irrecuperável? (b) Que sinais do exercício anterior são irrecuperáveis.
18. Descreva as macros a seguir:
  - (a) **SIG\_DFL**
  - (b) **SIG\_IGN**
19. Quando o sistema operacional envia um sinal do tipo sinal **SIGFPE** pode-se ter certeza de que ocorreu uma exceção de ponto flutuante. Esta afirmação é correta ou não? Explique.
20. Suponha que um programa tente executar uma instrução que requeira permissão que não lhe seja concedida. Que tipo de sinal, provavelmente, o sistema operacional, no qual o programa é executado, emitirá?
21. Qual é a diferença entre os sinais **SIGTERM** e **SIGKILL** emitidos por sistemas da família Unix?

22. Cite uma utilidade prática da função **raise()**.
23. (a) Descreva os parâmetro da função **signal()**. (b) O que retorna a função **signal()**?
24. Por que, frequentemente, a função **signal()** precisa ser chamada no corpo de uma função de tratamento de sinal?
25. Que funções podem ser seguramente chamadas no corpo de uma função de tratamento de sinal quando o sinal tratado é (a) síncrono e (b) assíncrono?
26. O que é um desvio generalizado?
27. A função **setjmp()** pode ser chamada sem uma chamada correspondente de **longjmp()** ou vice-versa? Explique.
28. Por que as funções **setjmp()** e **longjmp()** não são consideradas funções normais?
29. (a) Como é o retorno da função **longjmp()**? (b) Como é o retorno da função **setjmp()**?
30. Descreva o funcionamento das funções **setjmp()** e **longjmp()**.
31. (a) Para que serve o tipo **jmp\_buf**? (b) O que uma variável deste tipo armazena?
32. (a) Como são interpretados os parâmetros da função **longjmp()**? (b) O que ocorre quando esta função é chamada com o segundo parâmetro igual a zero?
33. Qual é o uso prático mais comum das funções **setjmp()** e **longjmp()**?
34. Descreva em linha gerais o tratamento de exceções usando as funções **setjmp()** e **longjmp()**.
35. Que relação existe entre:
  - (a) Sinais e interrupções
  - (b) Sinais e exceções
  - (c) Interrupções e exceções
36. (a) O que o programa a seguir imprime no meio de saída? (b) Qual é o problema com este programa? (c) Como este problema pode ser resolvido?

```
#include < setjmp.h >
```

```

main()
{
    jmp_buf estado;
    int i;

    i = setjmp(estado);
    printf("i = %d\n", i);

    longjmp(estado, 2);
    printf("Havera' impressao?\n");

    return 0;
}

```

### 37. O que imprime o programa a seguir?

```

#include <setjmp.h>
#include <stdio.h>

jmp_buf estado;

extern void f2(void);

int main(void)
{
    int i;

    printf("1\n");
    i = setjmp(estado);

    if(i == 0) {
        f2();
        printf("Teste\n");
    }

    printf("%d\n", i);

    return 0;
}

void f2(void)

```

```

{
    printf("2\n");
    longjmp(estado, 3);
}

```

38. Por que a instrução **printf()** chamada na função **main()** do programa a seguir nunca é executada?

```

#include <setjmp.h>
#include <stdio.h>

jmp_buf buffer;

void F(void)
{
    printf("2 ");
    longjmp(buffer, 3);
}

int main(void)
{
    int i;

    printf("1 ");
    i = setjmp(buffer);

    if(i == 0) {
        F();
        printf("Havera' impressao?");
    }

    printf("%d", i);

    return 0;
}

```

# *Capítulo 12*

---

*Miscelânea de tipos, funções e macros*

## 12.1 INTRODUÇÃO

Este capítulo apresenta os componentes dos cabeçalhos `<stdlib.h>` e `<stddef.h>`. Diferentemente do que ocorre com outros cabeçalhos da biblioteca padrão de C, os componentes dos cabeçalhos apresentados aqui não têm muitas afinidades entre si. Além disso, alguns componentes do cabeçalho `<stdlib.h>` poderiam, de modo mais coerente, fazer parte de outros cabeçalhos<sup>121</sup>. Por outro lado, o cabeçalho `<stddef.h>` é pouco utilizado, pois só contém dois componentes que não são declarados de modo redundante em outros cabeçalhos da biblioteca padrão.

## 12.2 CABEÇALHO `<stdlib.h>`

O cabeçalho `<stdlib.h>` contém uma miscelânea de tipos, macros e funções. Estes componentes podem ser classificados nas seguintes categorias:

- Controle de processos (v. **Seção 12.2.3**)
- Geração de números aleatórios (v. **Seção 12.2.4**)
- Alocação dinâmica de memória (v. **Seção 12.2.5**)
- Busca e ordenação de dados (v. **Seção 12.2.6**)
- Operações aritméticas inteiras (v. **Seção 12.2.7 e Capítulo 2**)
- Conversões de *strings* em números inteiros (v. **Seção 12.2.8 e Capítulo 6**)
- Conversões de *strings* em números reais (v. **Seção 12.2.9 e Capítulo 6**)
- Conversões entre caracteres extensos e multibytes (v. **Seção 12.2.10 e Capítulo 8**)

As funções declaradas em `<stdlib.h>` são apresentadas no texto de acordo com a categorização apresentada anteriormente. As funções que fazem parte das três últimas categorias são descritas em detalhes nos capítulos **2**, **6** e **8**, respectivamente, e serão apresentados aqui de forma bem resumida.

---

<sup>121</sup> `<stdlib.h>` é um dos mais antigos cabeçalhos da biblioteca padrão, e a única justificativa para a não inclusão de alguns componentes em outros cabeçalhos é simplesmente a não existência destes cabeçalhos.



## 12.2.1 TIPOS

*size\_t*

**Incluir:** `<time.h>`, `<string.h>`, `<wchar.h>`, `<stdlib.h>` ou `<std-def.h>`

**Descrição:** `size_t` é um tipo inteiro sem sinal definido em vários cabeçalhos (v. **Seção 1.7.1**).

*div\_t, ldiv\_t, lldiv\_t*

**Incluir:** `<stdlib.h>`

**Descrição:** Estes são os tipos das estruturas retornadas, respectivamente, pelas funções `div()`, `ldiv()` e `lldiv()`. Estes tipos são explorados na **Seção 2.6.1**.

*wchar\_t*

**Incluir:** `<wchar.h>` ou `<stddef.h>`

**Descrição:** O tipo `wchar_t` é utilizado para representar caracteres extensos e também é definido em `<wchar.h>` (v. **Seção 8.5.1**) e `<stddef.h>` (v. **Seção 12.3.1**).

## 12.2.2 MACROS

*EXIT\_FAILURE*

**Incluir:** `<stdlib.h>`

**Descrição:** A macro `EXIT_FAILURE` representa o valor retornado pela função `exit()` (v. **Seção 12.2.3**) quando o programa termina de modo malsucedido. Costuma-se retornar este valor da função `main()` em situação idêntica.

*EXIT\_SUCCESS*

**Incluir:** `<stdlib.h>`

**Descrição:** A macro **EXIT\_SUCCESS** representa o valor retornado pela função **exit()** quando o programa termina de modo bem-sucedido. Costuma-se retornar este valor da função **main()** em situação idêntica.

*MB\_CUR\_MAX*

**Incluir:** <stdlib.h>

**Descrição:** A macro **MB\_CUR\_MAX** representa o número máximo de caracteres que constituem um caractere multibyte na localidade corrente (v. **Capítulo 8**).

*RAND\_MAX*

**Incluir:** <stdlib.h>

**Descrição:** A macro **RAND\_MAX** representa o maior valor que pode ser retornado pela função **rand()** (v. **Seção 12.2.4**).

*NULL*

**Incluir:** <time.h>, <string.h>, <wchar.h>, <stdlib.h> ou <std-def.h>

**Descrição:** A macro **NULL** representa um ponteiro nulo e é definida em vários cabeçalhos (v. **Seção 1.7.2**).

### 12.2.3 FUNÇÕES DE CONTROLE DE PROCESSOS

*abort()*

**Incluir:** <stdlib.h>

**Descrição:** A função **abort()** causa uma terminação anormal do programa.

**Protótipo:**

void abort(void)
------------------

**Observações:**

- Esta função é usada quando o programa não é capaz de recuperar-se de um erro fatal.
- Esta função executa a chamada (v. **Seção 11.6.3**):  
`raise(SIGABRT);`
- Arquivos abertos poderão não ser fechados quando esta função for executada.

**Exemplo:**

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

void TratamentoDeSinal(int sinal)
{
    /* Esta função trata apenas o sinal SIGABRT */
    if (sinal != SIGABRT)
        return;

    printf("O programa sera' abortado agora. Tchau.");
}

int main()
{
    char c;

    printf( "Digite 1 para instalar a funcao de "
           "tratamento de sinal ou outro valor "
           "para nao instala-la: " );
    c = getchar();

    if (c == '1')
        signal(SIGABRT, TratamentoDeSinal);

    printf("Digite 1 para abortar o programa ou "
           "outro valor para termina-lo normalmente: ");

    getchar(); /* Limpa o buffer de entrada */
}
```

```

c = getchar();

if (c == '1')
    abort();

return 0;
}

```

### *atexit()*

**Incluir:** <stdlib.h>

**Descrição:** A função **atexit()** registra uma função a ser executada no encerramento do programa quando ele for finalizado normalmente [i.e., com uma chamada de **exit()** em qualquer parte do programa ou execução de **return** na função **main()**].

**Protótipo:**

```
int atexit(void (* funcao) (void))
```

**Parâmetro:** *funcao* – ponteiro para a função a ser registrada.

**Retorno:** Zero se **atexit()** consegue registrar a função; um valor diferente de zero, caso contrário.

**Observações:**

- Apenas um número limitado de funções pode ser registrado. Este limite depende da implementação de C, mas o padrão C99 recomenda que o mínimo seja 32.
- As funções são registradas numa pilha (i.e., a última função registrada será a primeira a ser chamada).
- Uma vez feito o registro de uma função, não existe nenhuma maneira de removê-lo.

**Exemplo:** Veja o exemplo da função **exit()**

## \_Exit() (C99)

**Incluir:** <stdlib.h>

**Descrição:** A função **\_Exit()** encerra a execução de um programa.

**Protótipo:**

```
void _Exit(int codigo)
```

**Parâmetro:** `codigo` – valor inteiro que será passado para o processo (usualmente, o sistema operacional) que causou a execução do programa.

**Observação:** Esta função é semelhante à **exit()**, mas, ao contrário desta última, não fecha necessariamente arquivos abertos pelo programa (mas pode fazê-lo, dependendo da implementação), nem chama funções registradas com **atexit()**.

**Exemplo:** O programa a seguir demonstra o uso das funções **\_Exit()** e **atexit()**.

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

void Despedida(void)
{
    printf("O programa sera' encerrado agora. Bye.\n");
}

int main()
{
    int opcao;

    /* Registra a função Despedida(). O que se */
    /* deseja mostrar aqui é que esta função */
    /* não será chamada, como ocorreria se */
    /* exit() fosse usada. */
    atexit(Despedida);

    printf( "Digite 1 para encerrar o programa "
           "usando _Exit ou outro valor para "
           "encerrar o programa normalmente: " );

    opcao = getchar();
```

```

if ( opcao == '1' ) {
    printf( "\nPrograma terminando com chamada "
           "de _Exit()...\n" );
    _Exit(EXIT_SUCCESS);
}

printf("\nPrograma terminando normalmente...\n");

return 0;
}

```

Se você executar o programa anterior, verá que a função `Despedida()` só é executada quando o programa termina normalmente.

*exit()*

**Incluir:** <stdlib.h>

**Descrição:** A função `exit()` encerra a execução do programa. Antes disso, todos os arquivos abertos pelo programa são fechados e, se houver funções registradas com `atexit()`, elas serão chamadas.

**Protótipo:**

```
void exit(int codigo)
```

**Parâmetro:** `codigo` – valor inteiro que será passado para o processo (usualmente, o sistema operacional) que causou a execução do programa.

**Observações:**

- Convencionalmente, utiliza-se zero como parâmetro de `exit()` para indicar o encerramento normal de um programa e um valor diferente de zero, caso contrário.
- O uso das constantes **EXIT\_SUCCESS** (encerramento normal) e **EXIT\_FAILURE** (encerramento anormal), definidas em <stdlib.h>, como parâmetro de `exit()`, é considerado mais legível (em inglês, obviamente...).

**Exemplo:** O programa a seguir demonstra o uso das funções **exit()** e **atexit()**.

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

void Despedida(void)
{
    printf("O programa sera' encerrado agora. Bye.\n");
}

int main()
{
    int opcao;

    /* Registra a função Despedida */
    atexit(Despedida);

    printf( "Digite 1 para encerrar o programa "
           "usando exit() ou outro valor para "
           "encerrar o programa normalmente: " );

    opcao = getchar();

    if ( opcao == '1' ) {
        printf( "\nPrograma terminando com "
               "chamada de exit()...\n" );
        exit(EXIT_SUCCESS);
    }

    printf("\nPrograma terminando normalmente...\n");

    return 0;
}
```

Se você executar o programa anterior verá que a função `Despedida()` é sempre executada porque as duas opções de encerramento são consideradas normais.

*getenv()*

**Incluir:** `<stdlib.h>`

**Descrição:** A função `getenv()` retorna um ponteiro para um *string* contendo o valor da variável de ambiente cujo nome é passado como argumento.

**Protótipo:**

```
char *getenv(const char *nome)
```

**Parâmetro:** `nome` – *string* que representa o nome de uma variável de ambiente.

**Retorno:** Um ponteiro para um *string* contendo o valor da variável de ambiente cujo nome é passado no argumento; **NULL**, se não for encontrada nenhuma variável de ambiente com este nome no hospedeiro onde o programa é executado.

**Observação:** O *string* retornado não pode ser utilizado para alterar o valor da variável de ambiente.

**Exemplo:**

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

int main(void)
{
    char *str;

    str = getenv("PATH");

    if (str)
        printf( "O conteudo da variavel de ambiente "
                "PATH e':\n\t\"%s\"\n", str);

    return 0;
}
```

*system()*

**Incluir:** `<stdlib.h>`

**Descrição:** A função `system()` passa um *string* representando um comando para o processador de comandos do sistema operacional onde o programa está sendo executado.



**Protótipo:**

```
int system(const char *comando)
```

**Parâmetro:** *comando* – *string* contendo um comando do sistema operacional ou **NULL** para consultar se existe um processador de comandos disponível.

**Retorno:**

- Se *comando* for **NULL**, um valor diferente de zero se existir um processador de comandos disponível; zero, caso contrário.
- Se *comando* não for **NULL**, o valor retornado será dependente de implementação. Por exemplo, nos sistemas Linux e DOS, se o compilador utilizado for gcc, a função retornará zero se não ocorrer nenhum erro, ou um valor diferente de zero se o processador de comandos não puder ser executado.

**Observações:**

- Este *string* pode ser um comando do sistema operacional, um arquivo *batch* ou um nome de programa executável. No caso de programa executável, ele deve residir no diretório corrente ou em um dos diretórios listados na variável de ambiente *PATH* (sistema DOS).
- O modo como o sistema operacional executa o comando (se este estiver disponível) depende de implementação.
- O comando pode causar o término do programa ou apresentar outro efeito não especificado.

**Exemplo:** O programa apresentado a seguir armazena num arquivo-texto o conteúdo do diretório corrente. Seu uso é limitado ao sistema DOS e àqueles da família Unix. Para compilá-lo no sistema Linux com o compilador gcc, utilize a opção `-DLINUX` para definir a macro `LINUX`.

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

#ifdef LINUX
#define COMANDO "ls -alt > Conteudo.txt"
#else
```

```

#define COMANDO "dir > Conteudo.txt"
#endif

int main()
{
    if (system(NULL)) { /* Testa se o processador de */
                        /* comandos está disponível */
        if(!system(COMANDO))
            printf("O conteudo do diretorio corrente "
                  "foi gravado no arquivo "
                  "\"Conteudo.txt\"\n");
        } else
            printf("Processador de comandos indisponivel.\n");

    return 0;
}

```

**Exemplo:** O programa a seguir testa a disponibilidade do processador de comandos e executa a calculadora do Windows:

```

#include <stdlib.h>

#ifdef LINUX
#error "Este programa nao funciona com Linux"
#endif

int main(void)
{
    if (system(NULL)) /* Verifica a disponibilidade */
                        /* do processador de comandos */
        /* Executa a calculadora do Windows */
        system("calc.exe");
    return 0;
}

```

Note que o último programa funciona apenas porque o programa `calc.exe` faz parte do caminho (`PATH`) do sistema. Se este não for o caso, ou se o programa não estiver no diretório onde o programa está sendo executado, deve-se especificar o caminho completo do programa invocado.

## 12.2.4 FUNÇÕES DE GERAÇÃO DE NÚMEROS ALEATÓRIOS

*rand()*

**Incluir:** <stdlib.h>

**Descrição:** A função **rand()** calcula um número pseudoaleatório no intervalo de zero a **RAND\_MAX** (v. **Seção 12.2.2**).

**Protótipo:**

```
int rand(void)
```

**Retorno:** Número pseudoaleatório no intervalo de zero a **RAND\_MAX**.

**Observações:**

- Para gerar números aleatórios entre M e N, sendo  $M < N$  e  $N \leq \text{RAND\_MAX}$ , utilize a fórmula:

```
rand() % (N - M + 1) + M
```

- Para não gerar os mesmos números aleatórios a cada execução do programa, a função **rand()** precisa ser alimentada com um valor arbitrário que não seja o mesmo em duas execuções do programa. Uma técnica comum usada para alimentar o gerador de números aleatórios consiste em usar o valor retornado pela função **time()** (v. **Seção 5.3.3**). Para tal, declare uma variável **t** do tipo **time\_t** (v. **Seção 5.3.1**) e chame a função **srand()** (v. adiante) para alimentar o gerador de números aleatórios como na instrução<sup>122</sup>:

```
srand((unsigned) time(&t));
```

**Exemplo:** O programa a seguir demonstra o uso das funções **rand()** e **srand()**.

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

/****
 *
 * Função LancaDados(): simula um número determinado
 *                       de lançamentos de um dado
 *
 * Argumentos: nLancamentos (entrada) - número de vezes
```

<sup>122</sup> A conversão explícita (unsigned) é necessária porque a função **time()** retorna um valor do tipo **time\_t**.

```

*                               que o dado será
*                               lançado
*
* Retorno: Nada
*
****/

void LancaDados(unsigned nLancamentos)
{
    unsigned i;

    for(i=0; i < nLancamentos; i++)
        printf("\t\t\t\t\t%d\n", rand()%6 + 1);
}

int main(void)
{
    time_t t;

    /* Alimenta o gerador de números aleatórios */
    srand((unsigned) time(&t));

    printf("\nPrimeira serie de lancamentos:\n");
    LancaDados(10);

    printf("\nSegunda serie de lancamentos:\n");
    LancaDados(10);

    return 0;
}

```

### *srand()*

**Incluir:** <stdlib.h>

**Descrição:** A função **srand()** inicia (ou **alimenta**) o gerador de números aleatórios.

**Protótipo:**

```
void srand(unsigned int n)
```

**Parâmetro:** *n* – valor que servirá como **semente** para o gerador de números aleatórios.

**Observações:**

- As sequências de números aleatórios retornadas por **rand()** serão idênticas se sementes idênticas forem utilizadas em chamadas sucessivas de **srand()**.
- É comum utilizar um valor retornado pela função **time()** como semente [v. descrição da função **rand()**].
- Consulte também a descrição de **rand()**.

**Exemplo:**

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

int main(void)
{
    /* Através de várias execuções deste programa */
    /* é possível perceber que os dois primeiros */
    /* valores impressos nunca são modificados */
    /* enquanto que os dois últimos valores são */
    /* modificados a cada execução do programa */

    time_t t;

    printf( "Primeira chamada de rand() SEM o "
           "inicializador srand(): %d\n", rand() );
    printf( "Segunda chamada de rand() SEM o "
           "inicializador srand(): %d\n", rand() );

    srand( (unsigned)time(&t) );

    printf( "Primeira chamada de rand() COM o "
           "inicializador srand(): %d\n", rand() );
    printf( "Segunda chamada de rand() COM o "
           "inicializador srand(): %d\n", rand() );

    return 0;
}
```

A importância do uso de **srand()** pode ser apreciada executando-se o programa anterior diversas vezes. Se você fizer isso, notará que os resultados apresentados pela função **rand()** são sempre os mesmo quando a função **srand()** não é utilizada.

### 12.2.5 FUNÇÕES DE ALOCAÇÃO DINÂMICA DE MEMÓRIA

Todas as funções de alocação de memória retornam **NULL** quando não conseguem alocar a quantidade de memória requerida. Portanto, é essencial testar se o valor do ponteiro retornado é **NULL** antes de tentar utilizá-lo para acessar uma porção de memória que não se tem certeza se foi realmente alocada.

O tipo de retorno das funções de alocação de memória é **void \***, que representa ponteiros genéricos. Um ponteiro genérico pode ser atribuído a qualquer ponteiro sem que seja necessário o uso de conversão explícita (v. **Volume I**).

*calloc()*

**Incluir:** <stdlib.h>

**Descrição:** A função **calloc()** aloca um array de blocos e zera seu conteúdo.

**Protótipo:**

```
void *calloc(size_t nElementos, size_t tamanho)
```

**Parâmetros:**

- `nElementos` – número de elementos (blocos) do array que serão alocados.
- `tamanho` – tamanho, em bytes, de cada bloco a ser alocado.

**Retorno:** O endereço do primeiro bloco alocado, quando for possível alocar o espaço necessário para conter os blocos requisitados ou **NULL**, quando a alocação não for possível.

**Observações:**

- Os blocos alocados são contíguos em memória.
- Todos os bits constituintes dos blocos são iniciados com zeros.
- Consulte também a função **malloc()**.

**Exemplo:**

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

#define NUM_ELEMENTOS 100u

int main(void)
{
    int      *blocos;
    size_t   i;

    /* Aloca um array com 100 elementos tipo */
    /* int. Todos os elementos serão zerados */
    blocos = calloc(NUM_ELEMENTOS, sizeof(int));

    /* Testa se foi possível */
    /* alocar o espaço requerido */
    if(blocos == NULL)
        printf("Array nao pode ser alocado\n");
    else{ /* Espaço alocado com sucesso */
        printf("Conteudo do array:\n");

        for (i = 0; i < NUM_ELEMENTOS; ++i)
            printf("\tblocos[%d] = %d\n", i, blocos[i]);

        free(blocos); /* Libera o espaço alocado */
    }

    return 0;
}
```

***malloc()*****Incluir:** <stdlib.h>**Descrição:** A função **malloc()** aloca um bloco de tamanho especificado.

```
void *malloc(size_t tamanho)
```

**Parâmetro:** *tamanho* – tamanho (i.e., número de bytes) do bloco de memória a ser alocado.

**Retorno:** O endereço inicial do bloco alocado, quando for possível alocar o espaço requisitado ou **NULL**, quando não for possível alocar este espaço.

### Observações:

- Os bytes que constituem o bloco de memória alocado não são iniciados<sup>123</sup>.
- Usualmente, o argumento `tamanho` envolve o uso do operador **sizeof** (v. **Volume I**), cujo uso é recomendado, principalmente, por questões de portabilidade.

### Exemplo:

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

#define NUM_ELEMENTOS 100u

int main(void)
{
    int    *blocos;
    size_t  i;

    /* Aloca um array com 100 elementos tipo */
    /* int. Nenhum elemento é iniciado.      */
    blocos = malloc(NUM_ELEMENTOS * sizeof(int));

    /* Testa se foi possível */
    /* alocar o espaço requerido */
    if(blocos == NULL)
        printf("Array nao pode ser alocado\n");
    else{ /* Espaço alocado com sucesso */
        printf("Conteudo do array:\n");

        for (i = 0; i < NUM_ELEMENTOS; ++i)
            printf("\tblocos[%d] = %d\n", i, blocos[i]);

        free(blocos); /* Libera o espaço alocado */
    }
}
```

---

<sup>123</sup> Se uma implementação desejar, ela pode codificar a função **malloc()** de modo que os bytes do bloco alocado sejam zerados. Versões recentes do compilador gcc procedem dessa maneira.



```
    return 0;
}
```

## *realloc()*

**Incluir:** <stdlib.h>

**Descrição:** A função **realloc()** recebe dois argumentos. O primeiro argumento deve ser um ponteiro para o início de um bloco de memória alocado utilizando **malloc()**, **calloc()** ou a própria função **realloc()**, e o segundo argumento especifica um novo tamanho desejado para o bloco.

### **Protótipo:**

```
void *realloc(void *ptrBloco, size_t novoTamanho)
```

### **Parâmetros:**

- **ptrBloco** – ponteiro para um bloco de memória alocado usando **malloc()**, **calloc()** ou **realloc()**.
- **novoTamanho** – novo tamanho para o bloco (pode ser maior ou menor do que o tamanho do bloco original, ou zero).

**Retorno:** Um ponteiro para o novo bloco, se a realocação ocorrer sem problemas; **NULL**, caso contrário (v. observações a seguir).

### **Observações:**

- A função **realloc()** é tipicamente utilizada para redimensionar blocos previamente alocados dinamicamente.
- Se o novo tamanho, especificado pelo segundo argumento de **realloc()**, for menor do que o tamanho atual do bloco, a diferença será retirada do final do bloco. Isto é, a porção final do bloco, cujo tamanho é a diferença entre o tamanho original e o novo tamanho, será liberada. O ponteiro retornado pela função apontará para o bloco original e a porção de memória que não foi liberada manterá seu conteúdo original.

- Se o novo tamanho for maior do que o tamanho atual do bloco, um novo bloco do tamanho especificado será alocado (se for possível), o conteúdo do bloco original será copiado para a porção inicial deste novo bloco, o bloco original será liberado e, finalmente, a função retornará um ponteiro para o novo bloco alocado. A porção final do novo bloco não será iniciada.
- Se o novo tamanho for maior do que o tamanho atual do bloco e não for possível alocar um novo bloco, o bloco antigo manterá seu conteúdo e seu endereço originais e a função retornará **NULL**.
- Se **NULL** for passado como primeiro argumento para a função **realloc()**, ela se comportará como **malloc()** com o tamanho do bloco especificado pelo segundo argumento.
- Se o novo tamanho for igual a zero e o ponteiro passado como primeiro argumento não for nulo, a chamada de **realloc()** vai se comportar como uma chamada da função **free()** (i.e., ela libera o espaço em memória apontado pelo ponteiro).
- Não se deve fazer referência ao bloco original depois que esta função obtiver êxito na realocação de um novo bloco.
- O funcionamento desta função é explicado em maiores detalhes no **Volume I**.

### Exemplo:

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

#define NUM_ELEMENTOS 100u

int main(void)
{
    char *bloco = NULL;

    /* Aloca um bloco com calloc() */
    bloco = calloc(NUM_ELEMENTOS, sizeof(char));

    if(bloco == NULL) {
        printf("Nao foi possivel alocar espaco\n");
        return 1;
    }
}
```

```

    } else
        printf( "Bloco alocado com o tamanho %d\n",
                NUM_ELEMENTOS );

    strcpy(bloco, "Isto é um teste.");

    /* Realoca o bloco de modo que ele */
    /* contenha exatamente o string      */
    bloco = realloc(bloco, strlen(bloco) + 1);

    if(bloco == NULL)
        printf("Nao foi possivel realocar o bloco\n");
    else
        printf( "Bloco realocado com o tamanho %d\n",
                strlen(bloco) + 1 );

    return 0;
}

```

## *free()*

**Incluir:** <stdlib.h>

**Descrição:** A função **free()** recebe como único argumento um ponteiro que aponta para um bloco de memória alocado utilizando **malloc()**, **calloc()** ou **realloc()** e libera este espaço em memória.

**Protótipo:**

```
void free(void *ptrBloco)
```

**Parâmetro:** `ptrBloco` – ponteiro para um bloco de memória alocado dinamicamente com **calloc()**, **malloc()** ou **realloc()**.

**Observações:**

- A função **free()** deve ser chamada apenas com **NULL** ou um ponteiro que esteja correntemente apontando para o início de um bloco de memória alocado com alguma das funções de alocação descritas aqui. Este ponteiro não deve ter sido previamente liberado ou passado como parâmetro para **realloc()**. Caso contrário, o resultado será imprevisível.

- Após uma chamada da função **free()**, não se deve mais usar o ponteiro utilizado nesta chamada para acessar o espaço de memória liberado; caso contrário, o programa poderá apresentar um comportamento indefinido.
- Apesar de um ponteiro ser considerado inválido após ser utilizado numa chamada da função **free()**, o sistema não é capaz de detectar esta situação. Assim, sugere-se que sempre se atribua **NULL** a um ponteiro logo após ele ser utilizado numa chamada da função **free()**. Em vez de chamar diretamente esta função, alguns programadores costumam definir uma macro que chame **free()** e, em seguida, atribua **NULL** ao ponteiro (v. exemplo a seguir).
- Se o ponteiro passado para **free()** for nulo, a função retornará sem executar nada.

### Exemplo:

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

#define TAMANHO 4096

/* A macro a seguir chama função free() e em */
/* seguida anula o ponteiro. Assim, uma */
/* tentativa de usar o ponteiro causa um erro */
/* em tempo de execução, em vez de deixar a */
/* execução prosseguir com um comportamento */
/* indefinido (erro lógico). */
#define FREE(x) do { free(x); x = NULL; } while(0)

int main()
{
    char *ptr;

    /* Tenta obter um bloco de 4096 bytes */
    ptr = malloc(TAMANHO * sizeof(char));

    if (!ptr) {
        fprintf( stderr, "Memoria insuficiente\n" );
        return 1;
    }

    strncpy( ptr, "Suponha que isto e' um string "
```

```

        "bem longo\n", TAMANHO );

    fputs(ptr, stdout);

    /* Para testar o programa, uma das instruções */
    /* a seguir deve ser comentada */
    free(ptr); /* A função */
/*    FREE(ptr); /* A macro */

    fputs(ptr, stdout);

    return 0;
}

```

Quando o programa anterior é executado no DOS com a chamada da macro `FREE()` comentada, a segunda chamada de `fputs()` causa uma impressão sem sentido. No sistema Linux, sob as mesmas condições, a segunda chamada de `fputs()` tem o mesmo efeito que a primeira chamada. Isto mostra que usar um ponteiro logo após utilizá-lo numa chamada de `free()` realmente provoca um comportamento indeterminado do programa. Nos sistemas DOS e Linux, se for removido o comentário da macro `FREE()`, e a chamada da função `free()` for comentada, o programa será abortado. Esta última situação pode parecer pior, mas, de fato, não é. Ter um programa abortado (i.e., apresentando um erro de execução) é bem melhor do que ter um programa com comportamento errático (i.e., apresentando um erro de lógica).

## 12.2.6 FUNÇÕES DE BUSCA E ORDENAÇÃO DE DADOS

### *bsearch()*

**Incluir:** <stdlib.h>

**Descrição:** A função `bsearch()` executa uma busca binária num array ordenado.

**Protótipo:**

```

void *bsearch( const void *chave, const void *array,
               size_t nElem, size_t tamanho,
               int (*FComp)(const void *e1, const void *e2)

```

**Parâmetros:**

- `chave` – ponteiro para a chave a ser procurada.
- `array` – array a ser pesquisado.
- `nElem` – número de elementos do array.
- `tamanho` – tamanho de cada elemento do array.
- `FComp` – ponteiro para uma função que compara os elementos do array. Esta função deve possuir dois argumentos do tipo **const void \***, cada um apontando para um elemento do array a ser pesquisado. A função deve comparar estes elementos e retornar um valor inteiro de acordo com os seguintes critérios de comparação:
  - o Um valor menor do que zero, se `*elem1 < *elem2`.
  - o Zero, se `*elem1 == *elem2`.
  - o Um valor maior do que zero, se `*elem1 > *elem2`.

**Retorno:** O endereço do primeiro elemento encontrado no array que tiver uma chave igual à chave de busca ou **NULL**, se a chave não for encontrada.

**Observações:**

- Busca binária é um método de busca de dados que se aplica a listas ordenadas em ordem crescente ou decrescente.
- Se o array não estiver ordenado, o resultado será indefinido.
- Como se trata de uma busca binária, o primeiro elemento encontrado não é necessariamente o primeiro elemento do array que casa com a chave procurada.

**Exemplo:**

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

/* Macro usada para calcular o número */
/* de elementos de um array          */
#define N_ELEMENTOS(ar) (sizeof(ar) / sizeof(ar[0]))
```

```

    /* Tipo do ponteiro de função usado */
    /* como último argumento de bsearch() */
typedef int (*tFPtr)(const void*, const void*);

/****
*
* Função Compara(): compara dois inteiros
*
* Argumentos: p1 (entrada) - ponteiro para o
*                  primeiro inteiro
*              p2 (entrada) - ponteiro para o
*                  segundo inteiro
*
* Retorno: 0, se os inteiros forem iguais
*          1, se o primeiro inteiro for maior
*          -1, se o primeiro inteiro for menor
*
****/

int Compara(const int *p1, const int *p2)
{
    return (*p1 - *p2);
}

/****
*
* Função TestaChave(): verifica se uma chave inteira
*                  faz parte de um array de inteiros
*
* Argumentos: chave (entrada) - a chave a ser testada
*              ar (entrada) - o array que será examinado
*
* Retorno: 1, se a chave fizer parte do array
*          0, se a chave não fizer parte do array
*
****/

int TestaChave(int chave, const int ar[])
{
    int *p;

    p = bsearch( &chave, ar, N_ELEMENTOS(ar),
                 sizeof(int), (tFPtr)Compara );

```

```

    return (p != NULL);
}

int main(void)
{
    int array[] = {123, 456, 789, 888, 900, 933};

    if (TestaChave(789, array))
        printf("789 foi encontrado no array\n");
    else
        printf("789 NAO foi encontrado no array\n");

    if (TestaChave(111, array))
        printf("111 foi encontrado no array\n");
    else
        printf("111 NAO foi encontrado no array\n");

    return 0;
}

```

### *qsort()*

**Incluir:** <stdlib.h>

**Descrição:** A função **qsort()** implementa um método de ordenação, denominado *quicksort*, e é capaz de ordenar arrays de elementos de quaisquer tipos.

### **Protótipo:**

```

void qsort( void *array, size_t nElementos,
            size_t tamanhoDoElemento,
            int (*FComp) (const void *, const void *))

```

### **Parâmetros:**

- **array** – um ponteiro para o início do array a ser classificado.
- **nElementos** – número de elementos no array a ser ordenado.
- **tamanhoDoElemento** – tamanho, em bytes, de cada elemento do array.



- **FComp** – ponteiro para uma função que compara os elementos do array. Esta função deve possuir dois argumentos do tipo **const void \***, cada um apontando para um elemento do array a ser ordenado. A função deve comparar estes elementos e retornar um valor inteiro de acordo com os seguintes critérios:
  - o Um valor menor do que zero, se `*elem1 < *elem2`.
  - o Zero, se `*elem1 == *elem2`.
  - o Um valor maior do que zero, se `*elem1 > *elem2`.

**Observação:** Os valores retornados pela função de comparação descrita anteriormente pressupõem que se deseje ordenar um array em ordem crescente. Se a ordem desejada for decrescente, bastará trocar `<` por `>` e vice-versa na definição da função.

### Exemplo:

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

/****
 *
 * Função Compara(): compara dois strings
 *
 * Argumentos: p1 (entrada) - ponteiro para o
 *                  primeiro strings
 *              p2 (entrada) - ponteiro para o
 *                  segundo strings
 *
 * Retorno:  0, se os strings forem iguais
 *           1, se o primeiro string for maior
 *          -1, se o primeiro string for menor
 *
 ****/

int Compara(const void *s1, const void *s2)
{
    const char *str1 = (char *)s1;
    const char *str2 = (char *)s2;

    return strcmp(str1, str2);
}
```

```

}

int main(void)
{
    char carros[4][10] = {"GM", "Ford", "Fiat", "Honda"};
    int i;

    qsort(carros, 4, sizeof(carros[0]), Compara);

    for (i = 0; i < 4; i++)
        printf("%s\n", carros[i]);

    return 0;
}

```

### 12.2.7 FUNÇÕES DE ARITMÉTICA INTEIRA

As funções que fazem parte desta categoria do cabeçalho `<stdlib.h>` são descritas em detalhes na **Seção 2.6** e são apresentadas aqui apenas para facilidade de referência.

FUNÇÃO	OPERAÇÃO	TIPO EM QUE ATUA
<b>abs()</b>	Valor absoluto	<b>int</b>
<b>labs()</b>	Valor absoluto	<b>long</b>
<b>llabs()</b> (C99)	Valor absoluto	<b>long long</b>
<b>div()</b>	Quociente e resto de divisão	<b>int</b>
<b>ldiv()</b>	Quociente e resto de divisão	<b>long</b>
<b>lldiv()</b> (C99)	Quociente e resto de divisão	<b>long long</b>

Tabela 12-1: Funções de aritmética inteira declaradas em `<stdlib.h>`.

### 12.2.8 FUNÇÕES DE CONVERSÃO DE *STRINGS* EM NÚMEROS INTEIROS

As funções que fazem parte desta categoria do cabeçalho `<stdlib.h>` são descritas em detalhes na **Seção 6.5.1** e são apresentadas aqui para facilidade de referência.

FUNÇÃO	CONVERTE UM STRING NUM VALOR DO TIPO...
<b>atoi()</b>	<b>int</b>
<b>atol()</b>	<b>long</b>
<b>atoll() (C99)</b>	<b>long long</b>
<b>strtol()</b>	<b>long</b>
<b>strtoll() (C99)</b>	<b>long long</b>
<b>strtoul()</b>	<b>unsigned long</b>
<b>strtoull() (C99)</b>	<b>unsigned long long</b>

Tabela 12-2: Funções de conversão de strings em números inteiros.

### 12.2.9 FUNÇÕES DE CONVERSÃO DE *STRINGS* EM NÚMEROS DE PONTO FLUTUANTE REAIS

As funções que fazem parte desta categoria do cabeçalho `<stdlib.h>` são descritas em detalhes na **Seção 6.5.2** e são apresentadas aqui para facilidade de referência.

FUNÇÃO	CONVERTE UM STRING NUM VALOR DO TIPO...
<b>atof()</b>	<b>double</b>
<b>strtod()</b>	<b>double</b>
<b>strtof() (C99)</b>	<b>float</b>
<b>strtold() (C99)</b>	<b>long double</b>

Tabela 12-3: Funções de conversão de strings em números de ponto flutuante reais.

### 12.2.10 FUNÇÕES DE CONVERSÃO ENTRE CARACTERES EXTENSOS E MULTIBYTES

As funções que fazem parte desta categoria do cabeçalho `<stdlib.h>` são descritas em detalhes na **Seção 8.4** e são apresentadas aqui para facilidade de referência.

FUNÇÃO	DESCRIÇÃO SUCINTA
<b>mblen()</b>	Calcula o número de bytes de um caractere multibyte.
<b>mbtowc()</b>	Retorna o caractere extenso correspondente a um caractere multibyte.

FUNÇÃO	DESCRIÇÃO SUCINTA
<b>mbstowcs()</b>	Converte um <i>string</i> multibyte num <i>string</i> extenso.
<b>wctomb()</b>	Retorna o caractere multibyte correspondente a um caractere extenso.
<b>wcstombs()</b>	Converte a <i>string</i> extenso num <i>string</i> multibyte.

Tabela 12-4: Funções de conversão entre caracteres extensos e multibytes.

## 12.3 MISCELÂNEA DE TIPOS E MACROS: <stddef.h>

O cabeçalho <stddef.h> define tipos e macros de uso geral. Este cabeçalho raramente precisa ser utilizado, visto que apenas um tipo, **ptrdiff\_t**, e uma macro, **offsetof()**, são definidos exclusivamente nele. Isto é, os demais componentes deste cabeçalho são também definidos em outros cabeçalhos.

### 12.3.1 TIPOS

*ptrdiff\_t*

**Incluir:** <stddef.h>

**Descrição:** **ptrdiff\_t** é um tipo inteiro com sinal que representa o tipo do resultado da subtração de dois ponteiros.

**Observação:** O uso deste tipo é recomendado por duas razões:

- (1) Legibilidade – o próprio nome do tipo indica que uma variável deste tipo deverá armazenar uma diferença entre ponteiros.
- (2) Portabilidade – o tipo do qual **ptrdiff\_t** é derivado pode variar entre implementações, mas o programador não precisa fazer alterações no programa para acomodá-lo a estas variações.

**Exemplo:** O fragmento de programa a seguir ilustra o uso do tipo **ptrdiff\_t**.

```
#include <stddef.h>

void F(void *p1, void *p2)
{
    ptrdiff_t pDiferenca;
    ...
}
```

```

    pDiferenca = p1 - p2;
    ...
}

```

### *size\_t*

**Incluir:** `<time.h>`, `<string.h>`, `<wchar.h>`, `<stdlib.h>` ou `<std-def.h>`

**Descrição:** `size_t` é um tipo inteiro sem sinal definido em vários cabeçalhos (v. **Seção 1.7.1**).

### *wchar\_t*

**Incluir:** `<wchar.h>`, `<stdlib.h>` ou `<stddef.h>`

**Descrição:** `wchar_t` é um tipo inteiro usado para representar caracteres extensos. Este tipo também é definido em `<wchar.h>` (v. **Seção 8.5.1**) e `<stdlib.h>` (v. **Seção 12.2.1**).

## 12.3.2 MACROS

### *NULL*

**Incluir:** `<time.h>`, `<string.h>`, `<wchar.h>`, `<stdlib.h>` ou `<std-def.h>`

**Descrição:** A macro `NULL` representa um ponteiro nulo e é definida em vários cabeçalhos (v. **Seção 1.7.2**).

### *offsetof()*

**Incluir:** `<stddef.h>`

**Descrição:** A macro `offsetof()` resulta no deslocamento (*offset*), em bytes, de um membro de uma estrutura partir do início dela mesma.

**Protótipo:**

```
size_t offsetof(tipoEstrutura, membro)
```

**Parâmetros:**

- `tipoEstrutura` – nome de um tipo de estrutura.
- `membro` – nome de um membro do mesmo tipo de estrutura representado pelo primeiro argumento.

**Observações:**

- É importante notar que o primeiro argumento é o nome de um tipo e não de uma variável.
- Consulte a **Seção 13.4** para outros exemplos de uso desta macro.

**Exemplo:**

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

typedef struct {
    char  membro1;
    int   membro2;
} tEstrutural;

typedef struct {
    int   membro1;
    char  membro2;
} tEstrutura2;

int main(void)
{
    printf( "Deslocamento de membro2 de tEstrutural: "
           "%d\n", offsetof(tEstrutural, membro2));
    printf( "Deslocamento de membro2 de tEstrutura2: "
           "%d\n", offsetof(tEstrutura2, membro2));

    return 0;
}
```

Quando compilado com Borland C++ 5.0 e executado no Windows XP, o último programa apresenta o seguinte resultado:

```
Deslocamento de membro2 de tEstrutura1: 1
Deslocamento de membro2 de tEstrutura2: 4
```

Por outro lado, quando compilado com gcc versões 3.4.2 (Windows XP) e 4.3.2 (Linux Ubuntu 8.10), o resultado obtido com o último programa é:

```
Deslocamento de membro2 de tEstrutura1: 4
Deslocamento de membro2 de tEstrutura2: 4
```

## 12.4 EXERCÍCIOS DE REVISÃO

1. (a) Como podem ser categorizadas as funções providas pelo cabeçalho `<stdlib.h>`? (b) Por que as funções declaradas em `<stdlib.h>` não são declaradas em cabeçalhos mais convenientes (i.e., mais específicos)?
2. Por que o cabeçalho `<stddef.h>` é raramente incluído?
3. Quais são os significados das macros (a) **EXIT\_FAILURE** e (b) **EXIT\_SUCCESS**?
4. (a) Em que diferem as funções **\_Exit()** e **exit()**? (b) Idem para as funções **\_Exit()** e **abort()**. (c) Idem para as funções **abort()** e **exit()**.
5. (a) Mencione duas utilidades práticas da função **system()**. (b) Por que nenhum uso desta função é portátil?
6. O que ocorre quando a função **system()** é chamada recebendo **NULL** como argumento?
7. (a) O que retorna a função **getenv()**? (b) Esta função pode ser usada de modo portátil?
8. (a) Para que serve a função **rand()**? (b) Cite três situações nas quais esta função é útil.
9. O que representa a macro **RAND\_MAX**?

10. (a) Como a função **srand()** deve ser chamada? (b) O que ocorre quando um programa chama a função **rand()** sem ter previamente chamado a função **srand()** convenientemente?
11. Suponha que você deseje gerar números aleatórios entre os números inteiros  $N_1$  e  $N_2$ . Escreva um trecho de programa em C que mostre como isto pode ser realizado.
12. O que retornam as funções de alocação dinâmica de memória quando (a) um bloco de memória é alocado e (b) um bloco de memória não pode ser alocado?
13. Descreva o funcionamento completo da função **realloc()**.
14. Para que servem as funções (a) **qsort()** e (b) **bsearch()**?
15. Como deve ser definida a função de comparação cujo endereço é passado como parâmetro para a função **bsearch()**?
16. Como deve ser definida a função de comparação cujo endereço é passado como parâmetro para a função **qsort()** quando a ordenação deve ser em ordem (a) crescente e (b) decrescente?
17. (a) O que deve armazenar uma variável do tipo **ptrdiff\_t**? (b) Que vantagens este tipo oferece em relação a um tipo inteiro primitivo?
18. Apresente um exemplo de uso da macro **offsetof()**.



# *Capítulo 13*

---

*Portabilidade de programas em C*

## 13.1 INTRODUÇÃO

De acordo com o padrão ISO 9126, **portabilidade** é um conjunto de atividades relacionadas com a transferência de um programa de um ambiente para outro. Idealmente, um **programa portátil** é aquele que pode ser transportado com pouco ou nenhum esforço para um ambiente diferente daquele para o qual o programa foi originalmente desenvolvido. Dentre os fatores que influenciam o **ambiente de um programa** podem ser citados:

- O sistema operacional, incluindo o fabricante, a versão e o sistema de arquivos utilizado.
- O hardware, incluindo CPU, memória e periféricos.
- O compilador, incluindo o fabricante e a versão.
- A interface do usuário (e.g., console, interface gráfica).
- A cultura do usuário, incluindo linguagem e convenções de apresentação de números, datas e horas<sup>124</sup>.

Com tantos fatores influenciando o ambiente de um programa, pode-se concluir que portabilidade de programas é um tópico bem complexo e não pode ser explorado num único capítulo. Portanto, aqui não se tem a pretensão de mencionar a maioria dos problemas associados a portabilidade de programas. Em vez disso, este capítulo lida com alguns problemas comuns relacionados com a escrita de programas portáteis em C, notadamente problemas associados a compiladores, sistemas operacionais e hardware.

Além das recomendações apresentadas neste capítulo, existem inúmeras outras distribuídas ao longo do **Volume I** e do presente volume que visam melhorar a portabilidade de programas escritos em C. O **Apêndice D** também descreve alguns erros de programação que são essencialmente associados a portabilidade.

## 13.2 PORTABILIDADE E PADRONIZAÇÃO

Quando uma linguagem é padronizada, como é o caso de C, tem-se a garantia de que um programa escrito de acordo com o padrão da linguagem pode ser compilado

---

<sup>124</sup> Este tópico está associado à localização ou internacionalização de programas (v. **Capítulo 5**), mas não deixa de ser também um aspecto de portabilidade.

em qualquer sistema operacional ou máquina que disponha do respectivo compilador<sup>125</sup>. Mas, mesmo que um programa observe estritamente todas as regras estabelecidas pelo padrão ISO de C, não há garantia de que ele será portátil. Isso ocorre porque este padrão deixa, deliberadamente, alguns aspectos da linguagem não especificados e permite que outros aspectos variem de acordo com a implementação.

De acordo com o padrão de C, não é portátil uma característica da linguagem ou da biblioteca padrão de C rotulada como:

- **Definida pela implementação.** Comportamento definido pela implementação refere-se a uma *construção legal* da linguagem ou de algum componente da biblioteca padrão que fabricantes de compiladores podem interpretar de maneiras diferentes. Neste caso, quem implementa o compilador ou a biblioteca tem liberdade para definir o que acontece quando uma dada construção da linguagem ou chamada de função classificada como tal é usada. Mas, o fabricante de compilador deve documentar todas as suas decisões que façam parte desta categoria. Para cada uma dessas características, existe apenas um número pequeno de possibilidades de escolhas. Por exemplo, o fato de o tipo **char** ser considerado **signed** ou **unsigned** é definido por implementação, mas há apenas estas duas possibilidades.
- **Não especificada.** Uma característica de C rotulada como tal representa algo que é *legal* e para o qual o padrão não impõe nenhuma restrição. A principal diferença entre comportamento definido pela implementação e comportamento não especificado é que, no segundo caso, o padrão não requer que a decisão do fabricante seja documentada. Exemplos: a ordem de avaliação de argumentos numa chamada de função e a representação de tipos de ponto flutuante não são especificadas.
- **Indefinida.** Este rótulo aplica-se ao resultado da execução de uma instrução ou chamada de função que *não* é considerada correta e para a qual o padrão de C não faz nenhuma especulação sobre o que pode ocorrer. Isto significa que se pode obter qualquer resultado (frequentemente, desagradável). Isto é, o compilador pode ignorar, emitir uma mensagem de erro, etc., pois o padrão não requer que essas operações sejam diagnosticadas durante a compilação. Isto significa que o comportamento preciso do que ocorrerá não é especificado pelo padrão de C, nem precisa ser documentado pela implementação. Na prática, o que ocorre em decorrência de algo classificado como comportamento indefinido é um erro

---

<sup>125</sup> No caso específico de C, que é uma linguagem madura e de uso bastante difundido, existe compilador para virtualmente qualquer sistema operacional ou máquina.

lógico ou de execução difícil de depurar. Portanto, mesmo que um programa não pretenda ser portátil, ele não deve apresentar nenhum tipo de comportamento considerado indefinido. Exemplo: O que acontece quando ocorre *overflow* numa operação inteira é indefinido (portanto, espere pelo pior...).

De acordo com o padrão ISO C99, um programa **estritamente em conformidade com o padrão** é aquele que:

- Usa apenas aspectos *especificados* pelo padrão.
- Não excede nenhum limite definido pela implementação.
- Não aguarda resultado que dependa de nenhum aspecto da linguagem C rotulado como *definido pela implementação, não especificado* ou *indefinido*.

**Exemplo 1:** O programa a seguir está estritamente em conformidade com o padrão ISO C99:

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

int main(void)
{
    (void) printf( "\nMenor valor do tipo long long: "
                  "%lld\n", LLONG_MIN );

    return 0;
}
```

**Exemplo 2:** Apesar de sua aparência ingênua, o programa a seguir não é portátil:

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

int main(void)
{
    (void) printf( "\nMenor valor do tipo int: "
                  "%d\n", INT_MIN );

    return 0;
}
```

De acordo com o padrão de C, um programa **em conformidade** (mas, não *estritamente*) com o padrão pode depender de aspectos não portáteis de uma implementação. Por exemplo, o último programa apresentado está em conformidade com o padrão apesar de não ser portátil. Assim, *conformidade* diz respeito a uma implementação específica, de modo que um programa pode estar em conformidade com uma implementação de C, mas não estar em conformidade com relação a outra. Um programa que usa extensões providas por uma dada implementação pode ainda ser considerado *em conformidade*, desde que estas extensões não alterem o comportamento de um programa *estritamente em conformidade*.

A razão pela qual o padrão de C deixa tantos aspectos considerados indefinidos ou não especificados é que, assim, os compiladores são capazes de gerar código executável mais eficiente para programas que usam aspectos bem definidos. Portanto, o padrão considera responsabilidade do programador evitar aquilo que é considerado comportamento indefinido ou não especificado.

Em resumo, um programa que pretende ser portátil deve observar as seguintes regras:

- Não depender de comportamento considerado definido pela implementação, não especificado ou indefinido.
- Não depender de extensões da linguagem C ou da biblioteca padrão de C.
- Tornar trechos inerentemente não portáteis fáceis de encontrar e substituir.

As recomendações apresentadas aqui são muito simples de entender, mas igualmente difíceis de seguir. O problema é que poucos programadores de C conhecem as características da linguagem C que são classificadas nas categorias consideradas inerentemente não portáteis.

## 13.3 ORDENAÇÃO DE BYTES (ENDIANESS)

Existem duas formas de representação de valores multibytes num computador<sup>126</sup>:

- **Representação *big-endian*** – os bytes são ordenados do byte *mais* significativo para o byte *menos* significativo. Isto é, o primeiro endereço em memória

---

<sup>126</sup> Existem vantagens e desvantagens em cada uma dessas representações. Discutir os prós e contras de cada uma dessas representações, além de não fazer parte do escopo deste livro, não é importante para a discussão que segue.

contém o byte *mais* significativo do valor representado. Processadores Motorola utilizados na linha de computadores Macintosh e muitos computadores de grande porte usam este tipo de representação.

- **Representação *little-endian*** – os bytes são ordenados do byte *menos* significativo para o *mais* significativo. Isto é, o primeiro endereço em memória contém o byte *menos* significativo do valor representado. Processadores Intel e similares usam este tipo de representação.

Essas duas formas de ordenação de bytes causam problemas de portabilidade quando dados escritos num arquivo dirigido a uma plataforma são lidos noutra plataforma<sup>127</sup>. Considere, como exemplo, o número inteiro 1234, que corresponde a 0x04D2 na base hexadecimal. Usando a representação *big-endian*, este número inteiro seria representado em quatro bytes como mostra a **Figura 13-1**.

00	00	04	D2
0	1	2	3

Figura 13-1: Representação *big-endian* de 1234 em hexadecimal.

Usando a representação *little-endian*, o mesmo número inteiro seria representado como mostrado na **Figura 13-2**.

D2	04	00	00
0	1	2	3

Figura 13-2: Representação *little-endian* de 1234 em hexadecimal.

Para facilitar o entendimento, considere novamente o valor inteiro 1234 do exemplo anterior, cuja representação binária em quatro bytes é:

00000000	00000000	00000100	11010010
----------	----------	----------	----------

Este número seria armazenado nas representações *big-endian* e *little-endian* conforme mostrado na **Tabela 13-1**.

<sup>127</sup> *Arquivo* aqui é usado no sentido amplo apresentado na **Seção 10.2**.

BYTE	ENDEREÇO RELATIVO	REPRESENTAÇÃO EM <i>BIG-ENDIAN</i> DE 1234	REPRESENTAÇÃO EM <i>LITTLE-ENDIAN</i> DE 1234
1	00	00000000	11010010
2	01	00000000	00000100
3	02	00000100	00000000
4	03	11010010	00000000

Tabela 13-1: Possíveis representações binárias de 1234.

Observe nos exemplos anteriores que a representação *big-endian* é mais amigável (pelo menos, para os povos ocidentais), pois ela corresponde exatamente ao modo como se costuma escrever números: do algarismo mais significativo para o algarismo menos significativo.

Programas que pretendem ser portáteis não podem, implícita ou explicitamente, fazer nenhuma suposição relativa à ordem de armazenamento dos bytes de uma variável multibyte. Por exemplo, o trecho de programa a seguir produz diferentes resultados em máquinas que utilizam diferentes ordenações de bytes:

```
int i = 15;
char c = *(char *) i;
```

As funções de entrada e saída da biblioteca padrão de C leem e escrevem bytes usando a ordenação nativa do sistema no qual o programa é executado, o que claramente causa problemas na leitura de arquivos quando as ordenações de bytes não coincidem na leitura e na escrita. Por exemplo, suponha que um programa em C executado numa máquina com representação *big-endian* escreva num arquivo binário o inteiro 1234 utilizando quatro bytes (v. exemplo anterior). Se outro programa executado numa máquina com representação *little-endian* ler este arquivo (novamente usando quatro bytes), ele interpretará o valor desse inteiro como sendo 3523477504.

**Exemplo:** O seguinte programa pode ser utilizado para determinar se um dado processador utiliza ordenação *big-endian* ou *little-endian*.

```
#include <stdio.h>

int main(void)
{
```

```

int x = 1;
char primeiroByte;

primeiroByte = *(char *)&x == 1;

if(primeiroByte)
    printf("Representacao em little-endian\n");
else
    printf("Representacao em big-endian\n");

return 0;
}

```

O programa apresentado anteriormente é independente do tamanho do tipo `int` e funciona da maneira descrita a seguir. A variável `x` recebe o valor 1, o que significa que ela será armazenada em memória tendo 1 em seu byte menos significativo e zero nos demais bytes que a compõem. Portanto, se a representação utilizada pelo processador for *big-endian*, o primeiro byte terá valor zero, enquanto se a representação utilizada pelo processador for *little-endian*, o primeiro byte terá valor 1. A expressão:

```
primeiroByte = *(char *)&x == 1;
```

atribui à variável `primeiroByte` o primeiro byte da variável `x`. Finalmente, a instrução `if` testa o valor da variável `primeiroByte` e imprime o resultado.

Uma solução para o problema de portabilidade de arquivos que podem ser lidos e escritos em plataformas com ordenações de bytes diferentes é ler e escrever arquivos binários utilizando sempre a mesma ordenação de bytes, como faz a linguagem de programação Java, que sempre lê e escreve bytes na representação *big-endian*, independentemente da representação nativa do sistema. Um algoritmo que segue esta abordagem para escrita de arquivo verifica inicialmente se a representação nativa do sistema é *big-endian*. Se este for o caso, o arquivo será escrito normalmente. Caso contrário, os bytes que compõem cada valor multibyte serão reordenados em *big-endian* antes de serem escritos no arquivo. A leitura de dados pode utilizar um algoritmo semelhante: se a representação nativa do sistema for *big-endian*, a leitura será feita normalmente; caso contrário, valores multibytes serão reordenados nesta representação<sup>128</sup>.

---

<sup>128</sup> Pode-se do mesmo modo adotar a abordagem de sempre ler e escrever arquivos binários na representação *little-endian*. Não existe nenhuma vantagem aparente entre uma escolha ou outra.



**Exemplo:** O programa a seguir contém as funções `EscreveBigEndian()` e `LeBigEndian()` que são semelhantes, respectivamente, às funções `fwrite()` e `fread()` (v. **Seção 10.7.5**). Mas, diferentemente destas funções de biblioteca, que realizam entrada e saída utilizando a ordenação de bytes nativa da plataforma corrente, as funções `EscreveBigEndian()` e `LeBigEndian()` sempre executam estas operações na ordenação *big-endian*.<sup>129</sup>

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

typedef enum {BIG_ENDIAN, LITTLE_ENDIAN} tOrdenacao;

/****
 *
 *  Função OrdenacaoDeBytes()
 *
 *  Descrição: Verifica se o sistema corrente é
 *             big ou little-endian
 *
 *  Parâmetros: Nenhum
 *
 *  Retorno: Valor indicando se o sistema corrente é
 *           big ou little-endian.
 *
 ****/

tOrdenacao OrdenacaoDeBytes(void)
{
    int    x = 1;
    char primeiroByte;

    primeiroByte = *(char *)&x == 1;

    return primeiroByte ? LITTLE_ENDIAN : BIG_ENDIAN;
}

/****
 *
```

---

<sup>129</sup> **Observação importante:** A função `LeBigEndian()`, que faz leitura de arquivos, assume que qualquer arquivo lido está em ordenação *big-endian* e ela não possui meios para verificar se este é realmente o caso.

```

* Função EscreveBigEndian()
*
* Descrição: Esta função é semelhante a fwrite(), mas
*             escreve sempre em big-endian.
*
* Parâmetros:
*     array (entrada): ponteiro para um array no qual
*                     os bytes serão lidos
*     tamanhoElemento (entrada): tamanho de cada
*                               elemento do array
*     nElementos (entrada): número de itens do array
*                           a ser escritos
*     stream(entrada): onde será feita a escrita
*
* Retorno: número de itens (não é o número de bytes!)
*           que foram realmente escritos no stream.
*
****/

size_t EscreveBigEndian(const void *array,
                        size_t tamanhoElemento,
                        size_t nElementos, FILE *stream)
{
    size_t i, j, retorno = 0;
    char  *copia, *buffer;

    /* Se a ordenação já é big-endian, basta chamar */
    /* a função fwrite() para realizar a tarefa.    */
    /* Se a ordenação de bytes desejada for little- */
    /* endian, esta é a única instrução que         */
    /* precisa ser alterada.                        */
    if (OrdenacaoDeBytes() == BIG_ENDIAN)
        return fwrite( array, tamanhoElemento, nElementos,
                       stream );

    /* O bloco apontado por copia conterá          */
    /* uma cópia de cada elemento do array.        */
    /* Este bloco não é essencial, mas facilita    */
    copia = calloc(1, tamanhoElemento);

    /* O bloco apontado por buffer                */
    /* conterá uma cópia de cada                  */
    /* elemento do array em big-endian            */

```

```

buffer = calloc(1, tamanhoElemento);

/* Se não for possível alocar os dois */
/* blocos, desiste, mas essa estratégia */
/* pode ser melhorada. */
if (!copia || !buffer)
    return 0;

/* Escreve um a um cada elemento do */
/* array com os bytes ordenados em */
/* formato big-endian */
for (i = 0; i < nElementos; ++i) {
    /* Primeiro faz um cópia de cada elemento. */
    /* Este passo não é essencial, mas ajuda. */
    memmove(copia, (char *)array + i*tamanhoElemento,
            tamanhoElemento);

    /* Armazena cada byte de cada elemento */
    /* do array original em ordenação big- */
    /* endian no array buffer. */
    for (j = 0; j < tamanhoElemento; ++j)
        buffer[tamanhoElemento - j - 1] = copia[j];

    /* A função fwrite() escreve os bytes no */
    /* arquivo na ordem em que eles se encontram */
    /* em memória. Como o bloco buffer contém os */
    /* bytes do elemento corrente do array em */
    /* ordenação big-endian assim serão escrito */
    /* os bytes no arquivo. */
    retorno += fwrite( buffer, tamanhoElemento, 1,
                      stream );

    /* Se ocorrer algum erro de escrita retorna */
    /* o número de bytes escritos até então. */
    if (ferror(stream))
        goto tchau; /* Um goto desses não faz mal */
}

tchau:
    free(copia);
    free(buffer);

```

```

        /* Se não ocorreu nenhum erro, o valor retornado */
        /* neste ponto deverá ser igual a nElementos      */
        return retorno;
    }

/****
*
* Função LeBigEndian()
*
* Descrição: Esta função é semelhante a fread(), mas
*             lê sempre em big-endian.
*
* Parâmetros:
*     array (entrada): ponteiro para um array no qual
*                     os bytes lidos serão depositados
*     tamanhoElemento (entrada): tamanho de cada
*                               elemento lido
*     nElementos (entrada): número de itens do array
*                           que serão lidos
*     stream(entrada): onde a leitura será feita
*
* Retorno: número de itens (não é o número de bytes!)
*          que foram realmente lidos no stream.
*
* OBSERVAÇÃO: É assumido que a ordenação do arquivo
*              é big-endian. Não há como esta função
*              verificar se isto é verdadeiro ou não
*
****/

size_t LeBigEndian( void *array, size_t tamanhoElemento,
                    size_t nElementos, FILE *stream)
{
    size_t i, j, retorno = 0;
    char  *copia, *buffer;

    /* Se a ordenação já é big-endian, basta chamar */
    /* a função fread() para realizar a tarefa.    */
    /* Se a ordenação de bytes desejada for little */
    /* -endian, esta é a única instrução que       */
    /* precisa ser alterada.                        */
    if (OrdenacaoDeBytes() == BIG_ENDIAN)
        return fread( array, tamanhoElemento, nElementos,

```

```

        stream );

    /* O bloco apontado por copia conterà      */
    /* uma cópia de cada elemento do array.      */
    /* Este bloco não é essencial, mas facilita */
    copia = calloc(1, tamanhoElemento);

    /* O bloco apontado por buffer              */
    /* conterà uma cópia de cada elemento      */
    /* do array em big-endian                    */
    buffer = calloc(1, tamanhoElemento);

    /* Se não for possível alocar os            */
    /* dois blocos, desiste, mas isso            */
    /* pode ser melhorado.                       */
    if (!copia || !buffer)
        return 0;

    /* Lê um a um cada elemento do array e      */
    /* ordena os bytes em formato big-endian    */
    for (i = 0; i < nElementos; ++i) {
        /* Lê cada elemento no bloco copia.      */
        retorno += fread( copia, tamanhoElemento, 1,
                           stream );

        /* Se ocorrer algum erro de leitura retorna */
        /* o número de bytes escritos até então    */
        if (ferror(stream)) {
            free(copia);
            free(buffer);
            return retorno;
        }

        /* Armazena cada byte de cada elemento    */
        /* do array original em ordenação big-    */
        /* endian no array buffer.                  */
        for (j = 0; j < tamanhoElemento; ++j)
            buffer[tamanhoElemento - j - 1] = copia[j];

        memmove( (char *)array + i*tamanhoElemento,
                  buffer, tamanhoElemento );
    }

```

```

    free(copia);
    free(buffer);

    /* Se não ocorreu nenhum erro, o */
    /* valor retornado neste ponto */
    /* deverá ser igual a nElementos */
    return retorno;
}

int main(void)
{
    double x = 2.54, y = 0, z = 0;
    FILE *stream = fopen("Arq.dat", "wb");

    if (!stream) {
        printf("Erro ao abrir arquivo para escrita\n");
        return 1;
    }

    if (1 != EscreveBigEndian(&x, sizeof(x), 1, stream)) {
        printf("Ocorreu um erro de escrita no arquivo\n");
        return 1;
    }

    fclose(stream);

    stream = fopen("Arq.dat", "rb");

    if (!stream) {
        printf("Erro ao abrir arquivo para leitura\n");
        return 1;
    }

    if (1 != fread(&y, sizeof(y), 1, stream)) {
        printf("Ocorreu um erro de leitura no arquivo\n");
        return 1;
    }

    /* É preciso retorna ao início do arquivo */
    rewind(stream);

    if (1 != LeBigEndian(&z, sizeof(z), 1, stream)) {

```

```

        printf("Ocorreu um erro de leitura no arquivo\n");
        return 1;
    }

    printf("Valor original:                %4.21E\n", x);
    printf("Valor lido com fread():        %4.21E\n", y);
    printf("Valor lido com LeBigEndian(): %4.21E\n", z);

    return 0;
}

```

O programa apresentado está repleto de comentários explicativos e espera-se que possa ser compreendido sem a ajuda de comentários adicionais. Este programa quando compilado e executado no Windows produz o seguinte resultado:

```

Valor original:                2.54E+000
Valor lido com fread():        3.07E+090
Valor lido com LeBigEndian(): 2.54E+000

```

Outra abordagem usada para resolver o problema de portabilidade de arquivos decorrente de ordenação de bytes consiste em armazenar no início de cada arquivo binário um número inteiro multibyte, denominado **número mágico**. Então, antes de iniciar-se o processamento propriamente dito de um arquivo, lê-se e testa-se este valor com constantes que representem o valor original e o valor reordenado do número mágico. De acordo com o resultado deste teste, o restante do arquivo será lido normalmente ou com a necessária reordenação de bytes. O esboço de programa a seguir ilustra esta abordagem:

```

#define NUMERO_MAGICO                0x11223344UL
#define NUMERO_MAGICO_REORDENADO 0x22114433UL

...
unsigned long  nMagico;
FILE          *stream = fopen("Arq.bin", "rb+");

fread(&nMagico, sizeof(nMagico), 1, stream);

if (nMagico == NUMERO_MAGICO)
    LeArquivo(stream); /* Arquivo lido normalmente */
else if (nMagico == NUMERO_MAGICO_REORDENADO)
    ConverteArquivo(stream); /* Arquivo será convertido */
else { /* Não era o tipo de arquivo esperado */
    fprintf(stderr, "Ocorreu um erro ao tentar ler arquivo");
}

```

```

    . . .
}

```

O mesmo problema relacionado a ordenação de bytes que acomete arquivos de dados também ocorre quando se enviam valores multibytes através de uma conexão de rede. Isto é, se os dados enviados através de uma rede podem ser lidos numa plataforma com ordenação diferente daquela de origem, eles devem ser escritos num formato que os permita ser recuperados sem ambiguidade.

Arquivos de texto que usam esquemas de codificação de caracteres com unidades de código de mais de um byte apresentam os mesmos problemas de portabilidade causados por ordenação de bytes descritos aqui. Neste caso, a abordagem mais comum para resolver o problema é o uso de uma marca de ordenação de bytes (v. **Seção 7.6**), que, essencialmente, é equivalente à última abordagem descrita.

## 13.4 ALINHAMENTO DE VARIÁVEIS E PREENCHIMENTO DE ESTRUTURAS

**Alinhamento de variáveis** refere-se ao fato de alguns processadores requererem que variáveis sejam alocadas em endereços específicos em memória dependendo dos tamanhos das variáveis.

Alguns processadores são intransigentes com relação a alinhamento de variáveis. Por exemplo, alguns processadores Motorola utilizados na linha de computadores Macintosh requerem que variáveis cujos tamanhos sejam maiores do que 8 bits sejam alinhados em endereços pares em memória. Neste caso, se houver uma tentativa de acesso a uma variável utilizando um endereço ímpar, ocorrerá uma condição de exceção e o programa será abortado.

Outros processadores podem permitir que variáveis sejam alocadas em posições não recomendadas, mas, neste caso, o desempenho do programa pode ser seriamente degradado. Por exemplo, alinhamento desempenha um papel crítico no desempenho de alguns processadores Intel.

Resumindo, as consequências de acesso a dados desalinhados variam de acordo com o processador e o sistema operacional. Ou seja, dependendo desses dois parâmetros, o acesso a dados desalinhados pode causar aborto do programa, ou não, ou acarretar grande ou pequena perda de desempenho. É inexorável, portanto, que, na melhor hipótese, haja queda de desempenho.



Tipicamente, variáveis de tipos escalares seguem as seguintes regras de alinhamento:

- Se uma variável ocupa apenas um byte e o processador permite endereçamento de bytes, ela pode ser alocada em qualquer endereço.
- Se uma variável ocupa  $n$  bytes, ela deve ser alocada num endereço divisível por  $n$ . Por exemplo, uma variável que ocupa 16 bits deve ser alocada em qualquer endereço par, uma variável que ocupa 32 bits deve ser alocada num endereço divisível por quatro, e assim por diante.

Se uma variável representa um array, basta alinhar o primeiro elemento do array seguindo as regras apresentadas aqui e os demais elementos estarão consequentemente alinhados, pois todos os elementos de um array são do mesmo tamanho. Alinhamento de uma união também não representa problema, pois, na prática, alocar uma união corresponde a alocar apenas uma variável escalar que corresponde ao maior campo da união.

Se um programa contém apenas variáveis dos tipos descritos até aqui, nesta seção, ele não apresentará nenhum problema de portabilidade devido a alinhamento de variáveis. Mas, se um programa utiliza estruturas, ele pode apresentar um problema de portabilidade.

O problema com estruturas é que nem sempre alinhar o primeiro membro resulta automaticamente no alinhamento dos demais campos, pois uma estrutura pode ter campos de tamanhos diferentes. Por exemplo, supondo que o tipo **short** ocupe 16 bits e o tipo **float**, 32 bits, considere a seguinte definição de estrutura:

```
typedef struct    {
    char    c1;
    short   s;
    char    c2;
    float   f;
} tEstrutural;

tEstrutural  umaEstrutura;
```

Supondo ainda que as regras de alinhamento para variáveis escalares mencionadas aqui sejam seguidas, o campo `c1` pode ser alocado em qualquer endereço. Mas, se o campo `c1` for alocado num endereço par, o campo `s`, que, hipoteticamente, ocupa 16 bits, não estará alinhado. Portanto, suponha que o compilador seja suficientemente *esperto* para alocar o campo `c1` num endereço ímpar, de modo que o campo `s` seja automaticamente alinhado. Com a escolha de um endereço ímpar para o campo `c1`,

o campo `c2` também será automaticamente alinhado, mas o campo `f`, que, hipoteticamente, ocupa 32 bits, não será alinhado. E, não há nenhuma outra escolha que o compilador possa fazer para a alocação do primeiro campo da estrutura de modo que todos os campos sejam alinhados em decorrência desta escolha. A solução adotada pela maioria dos compiladores em situações como essa é o **preenchimento de estruturas**.

Preencher uma estrutura significa acrescentar bytes convenientemente de modo que todos os campos da estrutura sejam alinhados de acordo com as exigências de um dado processador. No exemplo discutido anteriormente, para que o campo `f` da estrutura seja alinhado, é necessário um preenchimento de 3 bytes entre os campos `c2` e `f`, conforme mostrado na **Figura 13-3**.

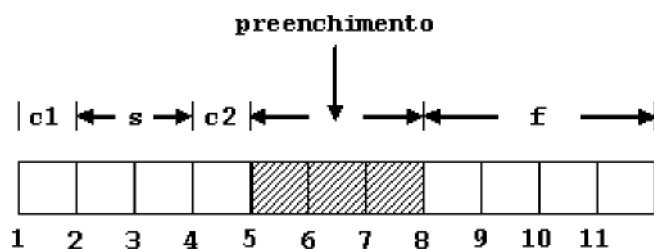


Figura 13-3: Estrutura com preenchimento.

Na **Figura 13-3**, cada pequeno retângulo representa um byte, e os números na parte inferior representam endereços de bytes em base decimal. Estes endereços foram escolhidos arbitrariamente de modo a facilitar a visualização do problema<sup>130</sup>.

O preenchimento de uma estrutura não causa nenhum transtorno em termos de acesso aos seus campos quando ela é mantida em memória principal. Ou seja, o acesso a um campo de estrutura utilizando os operadores `.` e `->` resulta sempre no mesmo conteúdo do campo, independentemente do fato de a estrutura ter sido preenchida ou não. No entanto, o preenchimento de estruturas causa dois problemas:

- *A estrutura ocupa mais espaço.* Isto é, o tamanho real de uma estrutura com preenchimento é maior do que a soma dos tamanhos dos campos da estrutura (sem incluir o preenchimento). Esse problema não é tão grave e pode ser ignorado se existirem poucas estruturas desse tipo.

<sup>130</sup> Qualquer escolha de endereços, que mantivesse a alocação do campo `c1` num endereço ímpar, serviria para esse propósito, pois a configuração final seria equivalente.

- *Operações de entrada e saída tendem a não ser portáteis.* Esse problema é sério e não deve ser subestimado.

Uma forma simples de eliminar ou, pelo menos reduzir, o preenchimento de estruturas de um certo tipo consiste na simples reordenação dos campos da estrutura do campo de maior largura para o campo de menor largura<sup>131</sup>. Por exemplo, utilizando esta técnica, a definição da estrutura `tEstrutura1` do exemplo apresentado antes resultaria em:

```
typedef struct {
    float  f;
    short  s;
    char   c1;
    char   c2;
} tEstrutura2;
```

Diferentemente de estruturas do tipo `tEstrutura1`, estruturas equivalentes do tipo `tEstrutura2` não requerem nenhum preenchimento para atender requisitos de alinhamento, como mostra a **Figura 13-4**.

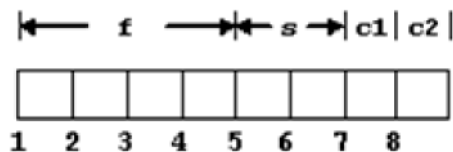


Figura 13-4: Estrutura reordenada sem preenchimento.

**Exemplo:** Para entender melhor os efeitos de preenchimento, considere o seguinte programa que utiliza as duas definições de tipos de estruturas já discutidas.

```
#include <stdio.h>

/* Diretiva pragma do compilador Borland C++ 5.0 */
/* que define alinhamento: */
/* -a1: nenhum alinhamento */
/* -a2: alinhamento em endereços pares */
/* -a4: alinhamento em endereços múltiplos de 4 */
```

<sup>131</sup> O padrão ISO da linguagem C não permite que um compilador faça reordenação de campos automaticamente. Assim, o programador deve fazer isso manualmente.

```

/* -a8: alinhamento em endereços múltiplos de 8 */
#pragma option -a8

typedef struct {
    char   c1;
    short  s;
    char   c2;
    float  f;
} tEstrutural;

typedef struct {
    float  f;
    short  s;
    char   c1;
    char   c2;
} tEstrutura2;

int main(void)
{
    printf( "Tamanho da estrutura tEstrutural: %u",
           sizeof(tEstrutural) );
    printf( "\nTamanho da estrutura tEstrutura2: %u",
           sizeof(tEstrutura2) );

    return 0;
}

```

Quando esse programa é compilado utilizando o compilador Borland C++ 5.0, que possui várias opções de alinhamento, ele produz diferentes resultados de acordo com a opção de alinhamento escolhida. As opções de alinhamento do programa podem ser alteradas por meio da diretiva **#pragma**:

```
#pragma option opção-de-alinhamento
```

que é exclusiva de compiladores da Borland e não tem efeito sobre outros compiladores<sup>132</sup>. As opções de alinhamento são descritas em forma de comentário no próprio programa e é interessante examinar os resultados apresentados pelo programa quando diferentes opções de alinhamento são utilizadas, conforme mostra a **Tabela 13-2**.

---

<sup>132</sup> Outros compiladores também possuem meios para controlar os modos de alinhamento. Por exemplo, o compilador gcc possui opções, tais como `-malign-double` e `-mno-align-double`, que podem ser utilizadas com essa finalidade. O compilador gcc ainda possui a opção `-fpack-struct` que faz com que ele não realize nenhum preenchimento de estruturas.

OPÇÃO DE ALINHAMENTO EM...	TAMANHO DE ESTRUTURAS DO TIPO <code>tEstrutura1</code>	TAMANHO DE ESTRUTURAS DO TIPO <code>tEstrutura2</code>
Qualquer endereço (i.e., sem alinhamento)	8	8
Endereços pares	10	8
Endereços múltiplos de 4	12	8
Endereços múltiplos de 8	12	8

Tabela 13-2: Diferentes tamanhos de uma mesma estrutura devido a alinhamento.

Observe na **Tabela 13-2** que a simples reordenação dos campos do tipo `tEstrutura1` que resulta no tipo `tEstrutura2` seguindo a técnica descrita aqui eliminou a necessidade de preenchimento em todas os três tipos de alinhamento disponíveis no compilador Borland. Mas esta técnica não é uma panacéia para o problema de preenchimento. Isto é, nem sempre a reordenação de campos de uma estrutura garante que não haja preenchimento da estrutura. Note ainda que os resultados apresentados na **Tabela 13-2** indicam quando o compilador realiza preenchimento, mas não proveem nenhuma informação sobre em qual endereço foi feito tal preenchimento. Para obter esse tipo de informação, pode-se utilizar a macro `offsetof()` definida no cabeçalho `<stddef.h>` (v. **Seção 12.3.2**).

**Exemplo:** A macro `offsetof()`, quando avaliada, resulta no deslocamento em bytes de um membro de uma estrutura a partir do seu início. O seguinte programa ilustra o uso da macro `offsetof()`.

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

#pragma option -a2

typedef struct {
    char   c1;
    short  s;
    char   c2;
    float  f;
} tEstrutura1;

typedef struct {
```

```

        float  f;
        short  s;
        char   c1;
        char   c2;
    } tEstrutura2;

int main(void)
{
    printf("\nTamanho de estruturas do tipo tEstrutural:"
           " %u", sizeof(tEstrutural));
    printf("\nDeslocamento de membros de tEstrutural:");
    printf( "\n\tDeslocamento de c1: %u",
             offsetof(tEstrutural, c1) );
    printf( "\n\tDeslocamento de s: %u",
             offsetof(tEstrutural, s) );
    printf( "\n\tDeslocamento de c2: %u",
             offsetof(tEstrutural, c2) );
    printf( "\n\tDeslocamento de f: %u",
             offsetof(tEstrutural, f) );

    printf( "\n\nTamanho de estruturas do tipo "
            "tEstrutura2: %u", sizeof(tEstrutura2) );
    printf("\nDeslocamento de membros de tEstrutura2:");
    printf( "\n\tDeslocamento de f: %u",
             offsetof(tEstrutura2, f) );
    printf( "\n\tDeslocamento de s: %u",
             offsetof(tEstrutura2, s) );
    printf( "\n\tDeslocamento de c1: %u",
             offsetof(tEstrutura2, c1) );
    printf( "\n\tDeslocamento de c2: %u\n",
             offsetof(tEstrutura2, c2) );

    return 0;
}

```

O resultado do programa do último exemplo depende do alinhamento de estruturas promovido pelo compilador. Por exemplo, quando este programa é compilado usando o compilador Borland C++ 5.0, com alinhamento em endereços pares obtido utilizando-se a diretiva:

```
#pragma option -a2
```

o programa produz como resultado:

```
Tamanho de estruturas do tipo tEstrutural1: 10
Deslocamento de membros de tEstrutural1:
    Deslocamento de c1: 0
    Deslocamento de s:  2
    Deslocamento de c2: 4
    Deslocamento de f:  6

Tamanho de estruturas do tipo tEstrutura2: 8
Deslocamento de membros de tEstrutura2:
    Deslocamento de f:  0
    Deslocamento de s:  4
    Deslocamento de c1: 6
    Deslocamento de c2: 7
```

Por outro lado, quando o mesmo programa é compilado usando o mesmo compilador, mas sem alinhamento (i.e., comentando a diretiva **#pragma** no início do programa), o resultado obtido é:

```
Tamanho de estruturas do tipo tEstrutural1: 8
Deslocamento de membros de tEstrutural1:
    Deslocamento de c1: 0
    Deslocamento de s:  1
    Deslocamento de c2: 3
    Deslocamento de f:  4

Tamanho de estruturas do tipo tEstrutura2: 8
Deslocamento de membros de tEstrutura2:
    Deslocamento de f:  0
    Deslocamento de s:  4
    Deslocamento de c1: 6
    Deslocamento de c2: 7
```

O compilador gcc também permite que se especifiquem alinhamentos alternativos de estruturas. A opção do compilador gcc que permite fazer isto tem o formato:

```
-fpack-struct [=n]
```

onde n deve ser uma potência de dois que especifique o máximo alinhamento de campos. Ou seja, campos com alinhamentos padronizados maiores do que este valor ficarão desalinhados, enquanto aqueles cujos alinhamentos padronizados são me-

nores que n estarão todos alinhados (mas não necessariamente com este valor). Se o valor de n for omitido, não haverá nenhum tipo de alinhamento<sup>133</sup>.

Conforme foi visto no **Capítulo 13 do Volume I**, campos de bits anônimos podem ser utilizados para forçar o alinhamento de variáveis, mas campos de bits (anônimos ou não) também são fontes potenciais de problemas de portabilidade.

Quando não é possível evitar o preenchimento de um tipo de estrutura, pode-se, pelo menos, impedir que o espaço de preenchimento seja armazenado num arquivo, escrevendo-se individualmente cada campo de uma estrutura, em vez de escrevê-los de uma vez<sup>134</sup>. Por exemplo, considerando as definições:

```
typedef struct {
    char    c1;
    short   s;
    char    c2;
    float   f;
} tEstrutura1;

tEstrutura1 estrutura;
```

A seguinte instrução de escrita não é recomendável:

```
fwrite(&estrutura, sizeof(estrutura), 1, stream);
```

Em vez de usar esta instrução, recomenda-se o uso das seguintes instruções:

```
fwrite(&estrutura.c1, sizeof(estrutura.c1), 1, stream);
fwrite(&estrutura.s, sizeof(estrutura.s), 1, stream);
fwrite(&estrutura.c2, sizeof(estrutura.c2), 1, stream);
fwrite(&estrutura.f, sizeof(estrutura.f), 1, stream);
```

Quando essa abordagem de escrita de estruturas é utilizada, as operações de leitura devem seguir o mesmo raciocínio; i.e., os campos também devem ser lidos individualmente.

---

<sup>133</sup> O uso desta opção de compilação não é recomendado pela documentação do compilador gcc.

<sup>134</sup> Se um campo consiste de uma estrutura aninhada, os campos desta estrutura também devem ser escritos individualmente e assim por diante.



Em plataformas com requisitos de alinhamento, pode ser desastroso fazer um ponteiro para um tipo que requeira alinhamento apontar para uma variável que requer alinhamento de outra natureza. Por exemplo, numa plataforma que alinha rigorosamente valores do tipo **int** em endereços pares<sup>135</sup>, poderá ocorrer um sério problema se um ponteiro para o tipo **char**, que não requer alinhamento, passar a apontar para uma variável do tipo **int** por meio de conversão explícita. Mais precisamente, neste exemplo, o erro ocorrerá se o ponteiro para **char** estiver apontando para um endereço não permitido para o tipo **int** no instante da conversão e tentar-se aplicar o operador de indireção ao mesmo.

**Exemplo:** O seguinte programa ilustra o que foi discutido no último parágrafo.

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

int ConverteStrEmInt(const char* s, int n)
{
    const char* aux = s + n;

    /* Conversão não-portável que pode causar o      */
    /* aborto do programa dependendo da plataforma */
    return *(int *) aux;
}

int main(void)
{
    const char str[] = "ABCDEFGH IJLMN";
    int      i, n, comprimento;

    comprimento = strlen(str);

    /* Dependendo da plataforma, o programa será */
    /* abortado antes de encerrar o laço a seguir */
    for (i = 0; i < comprimento; ++i) {
        n = ConverteStrEmInt(str, i);
        printf("i = %d\n", i);
    }

    return 0;
}
```

---

<sup>135</sup> Aqui, alinhar rigorosamente significa que qualquer tentativa de acesso a dados não alinhados provoca um erro de execução.

Dependendo da plataforma na qual é executado, o programa anterior será abortado devido à conversão de um ponteiro para **char** em ponteiro para **int** com a subsequente aplicação do operador de indireção.

## 13.5 ARITMÉTICA INTEIRA

### 13.5.1 TIPOS INTEIROS NÃO PORTÁVEIS

Conforme foi discutido no **Capítulo 2**, os únicos tipos inteiros primitivos portáteis de C são: (**signed**) **long long**, **unsigned long long**, **signed char** e **unsigned char**. Além destes tipos primitivos, os tipos inteiros definidos no cabeçalho `<stdint.h>` também são considerados portáteis. Entretanto, estes tipos são portáteis apenas no tocante à implementação. Quer dizer, o fato de estes tipos terem ou não sinal, bem como suas larguras não variarem entre implementações, como pode ocorrer com outros tipos inteiros.

Outro sério problema de portabilidade relacionado com qualquer número inteiro diz respeito a sua representação em memória. Isto inclui não apenas ordenação de bytes (v. **Seção 13.3**) como também o modo como números com sinal são representados; i.e., se a representação de inteiros usa complemento de um ou complemento de dois. Esses aspectos devem ser levados em consideração quando arquivos binários são criados em uma plataforma e lidos em outra.

### 13.5.2 OVERFLOW

*Overflow* ocorre quando o resultado de uma operação é um valor grande demais para caber numa variável do tipo do resultado. *Overflow* pode ocorrer em operações inteiras ou de ponto flutuante, mas lidar com *overflow* de inteiros é bem mais complicado do que com *overflow* de ponto flutuante (v. **Seção D.12.3**).

A ocorrência de *overflow* de inteiros pode ter consequências nefastas para um programa e o pior é que é impossível detectá-la após ter-se estabelecido. Em outras palavras, após a ocorrência de *overflow* de inteiros, o programa não tem como checar o resultado de uma operação para saber se ele é correto ou não. Assim, muitas vezes, o programa continua sua execução sob a suposição de que o cálculo esteja correto.

De acordo com o padrão ISO C99, a ocorrência de *overflow* de inteiros com sinal (i.e., **signed**) causa um comportamento indefinido (v. **Seção 13.2**). Por outro lado, *overflow* de inteiros sem sinal (i.e., **unsigned**) deve resultar num valor bem definido. Ou seja, neste último caso, o padrão afirma que o **resultado obtido** da operação é o resto da divisão do **resultado virtual** da operação pela soma do maior valor do tipo do resultado com um. Simbolicamente, isto significa que:

$$\text{resultado obtido} = \text{resultado virtual} \% (\text{maior valor do tipo} + 1)$$

Onde:

- *resultado obtido* é o valor **realmente** obtido após a ocorrência de *overflow*.
- *resultado virtual* é o valor que seria obtido se não houvesse ocorrido *overflow*.
- *maior valor do tipo* é o maior valor que pode ser contido numa variável do tipo do resultado da expressão<sup>136</sup>.

Por exemplo, se a avaliação de uma expressão cujo resultado é do tipo **unsigned int** resultar em *overflow*, o resultado obtido será o resultado virtual dividido por  $(\text{UINT\_MAX} + 1)$ , pois a macro **UINT\_MAX** representa o maior valor do tipo **unsigned int**. Assim, o verdadeiro resultado obtido com a soma:

```
UINT_MAX + 1
```

que evidentemente causa *overflow* é:

$$(\text{UINT\_MAX} + 1) \% (\text{UINT\_MAX} + 1) = 0$$

Em geral, somando-se um valor  $N$ , tal que  $0 < N < \text{UINT\_MAX}$ , ao maior valor do tipo **unsigned int** (i.e., **UINT\_MAX**), o resultado obtido deve ser  $N - 1$ , como mostrado a seguir<sup>137</sup>:

$$\begin{aligned} \text{resultado} &= (\text{UINT\_MAX} + N) \% (\text{UINT\_MAX} + 1) = \\ &= (\text{UINT\_MAX} + 1 + N - 1) \% (\text{UINT\_MAX} + 1) = \\ &= (\text{UINT\_MAX} + 1) \% (\text{UINT\_MAX} + 1) + (N - 1) \% (\text{UINT\_MAX} + 1) = \\ &= N - 1 \end{aligned}$$

Desse modo, tem-se, por exemplo:

$$\text{UINT\_MAX} + 25 = 24$$

<sup>136</sup> Este valor pode ser obtido usando-se uma das macros definidas em `<limits.h>` (v. **Seção 2.3**).

<sup>137</sup> Resultados semelhantes são obtidos quando o mesmo raciocínio é empregado para os demais tipos inteiros sem sinal.

**Exemplo:** Quando executado, o programa a seguir comprova o que foi exposto .

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

int main(void)
{
    unsigned umInt = UINT_MAX + 25;

    /* O resultado deve ser 25 - 1 = 24 */
    printf ("\numInt = %u\n", umInt);

    return 0;
}
```

Na representação de inteiros com complemento de 2, não há possibilidade de a alteração de sinal de um inteiro positivo causar *overflow*, mas existe esta possibilidade se o inteiro for negativo. Isso ocorre porque são permitidos mais valores negativos do que positivos nesta forma de representação. Por exemplo, em C, o valor **INT\_MIN** pode ser armazenado numa variável do tipo **int**, mas o valor **-INT\_MIN** não pode ser representado numa variável deste tipo. Obviamente, ambos os valores podem ser armazenados em variáveis de tipos inteiros mais largos do que **int**.

### 13.5.3 signed char e unsigned char

Algumas implementações consideram o tipo **char** como **signed char**, enquanto outras o veem como **unsigned char**. Nenhuma das interpretações pode ser considerada mais correta do que a outra simplesmente porque o padrão de C considera essa escolha dependente de implementação.

O fato de um valor do tipo **char** ser considerado **signed** ou **unsigned** é importante quando ele é convertido em **int**<sup>138</sup>. Se o valor convertido for considerado **unsigned**, os bits que restam para completar o resultado da conversão serão todos iguais a zero. Por outro lado, em algumas implementações, se o valor for considerado **signed**, os bits restantes poderão ser iguais ao bit de sinal. Isso é denominado **extensão** (ou **replicação**) de sinal. Por extensão de sinal, o bit de sinal é copiado para os bits superiores

---

138 Conforme foi visto na **Seção 1.5.2 do Volume I**, essa conversão de alargamento ocorre sempre que um valor do tipo **char** é usado numa expressão, numa chamada ou retorno de função.

do valor do tipo **int** resultante do alargamento do valor do tipo **char**. Portanto, se for importante garantir que o alargamento de uma variável do tipo **char** sempre ocorra sem extensão de sinal, deve-se declará-la com **unsigned char** (em vez de simplesmente **char**).

A atribuição de um valor negativo a uma variável de um tipo inteiro sem sinal é dependente de implementação. Em particular, não é portátil atribuir um valor negativo a uma variável do tipo **char**, uma vez que este tipo pode ser **signed** ou **unsigned**, dependendo da implementação. Por exemplo:

```
char c = 0 - 1; /* NÃO é portátil */
```

No exemplo anterior, se o tipo **char** for **signed**, a variável `c` receberá o valor `-1`, mas, se o tipo **char** for considerado **unsigned**, o valor armazenado em `c` será dependente de implementação. A instrução anterior não é portátil, mas poderá tornar-se portátil alterando-se o tipo da variável `c` para **signed char**:

```
signed char c = 0 - 1; /* 100% portátil: c receberá -1 */
/* em qualquer implementação */
```

## 13.6 CARACTERES E STRINGS

O conjunto de caracteres utilizado é dependente de implementação e cada implementação define como valores inteiros (i.e., pontos de código) são associados a caracteres. Portanto, se um programa utiliza (literalmente) constantes inteiras para representar caracteres, ele pode funcionar de modo diferente numa implementação que utiliza um código de caracteres diferente.

Suponha, por exemplo, que um programador deseje executar algum tipo de processamento usando todas as letras maiúsculas do alfabeto romano. Ele poderia, então, escrever um trecho de programa como o esboçado a seguir:

```
char c;
...
for (c = 65; c <= 90; ++c)
    ...
```

A intenção do programador é razoavelmente clara para qualquer conhecedor do código ASCII, pois 65 corresponde à letra *A* e 90 corresponde à letra *Z*. O uso de macros para evitar o uso de números mágicos (v. **Seção 6.6 do Volume I**), resolve o problema de legibilidade, mas não contribui para a solução do problema de portabilidade:

```

#define LETRA_A 65
#define LETRA_Z 90
...
char c;
...
for (c = LETRA_A; c <= LETRA_Z; ++c)
    ...

```

Este último trecho de programa é bem mais legível do que o anterior porque não requer conhecimento de ASCII para entendê-lo. Entretanto, o problema de portabilidade persiste porque não é o caso de que a letra *A* sempre corresponda a 65 ou a letra *Z* a 90 em qualquer código de caracteres. Além disso, existe outro problema de portabilidade mais sutil que aflige os trechos do programa anterior. Este problema será discutido a seguir.

Observe o seguinte trecho de programa:

```

for (p = string; *p; p++)
    if ((*p >= 'a') && (*p <= 'z'))
        *p += 'A' - 'a';

```

Com algum conhecimento do código ASCII (novamente ele), percebe-se que a intenção do programador é transformar as letras minúsculas de um *string* em letras maiúsculas. Ou seja, este trecho de programa é baseado na ideia de que cada par de letras maiúscula/minúscula de um conjunto de caracteres está a uma mesma distância, obtida por 'A' - 'a' (ou 'B' - 'b', 'C' - 'c', etc.) e que todas as letras aparecem em posições sucessivas na ordem natural do alfabeto romano. Infelizmente, alguns programadores não sabem que esta última suposição nem sempre é verdadeira. Existe pelo menos um código de caracteres em que esta suposição não é verdadeira: o código EBCDIC (v. **Seção 6.1.2**).

No código de caracteres EBCDIC, ainda em uso em mainframes IBM, os conjuntos de letras minúsculas e maiúsculas não são contíguos. Por exemplo, a letra *I* tem ponto de código 201 e *J* tem ponto de código 209; i.e., entre as letras *I* e *J* existem um caractere (que não é letra) e seis espaços vazios<sup>139</sup>. Apesar das descontinuidades entre letras neste código de caracteres, a distância entre cada par de respectivas letras minúscula e maiúscula é mantido constante. O problema é que nem todo caractere que está entre *a* e *z* ou entre *A* e *Z* é letra. Portanto, se o valor de um desses caracteres fosse somado à distância 'A' - 'a' no último trecho de programa, o resultado obtido não faria o menor sentido.

---

<sup>139</sup> No código EBCDIC, também há descontinuidade entre as letras *i* e *j*, *r* e *s*, e *R* e *S*.

Outro problema com o último trecho desse programa é que nem todo caractere considerado letra numa dada língua natural pertence ao intervalo entre 'a' e 'z' (ou entre 'A' e 'Z'). Por exemplo, se a codificação ISO 8859-1 (v. **Seção 6.1.2**) estiver sendo usada, o caractere 'ç' terá ponto de código 199 e, portanto, não estará entre 'a' e 'z'. Finalmente, quando se usa uma codificação multibyte (v. **Capítulo 7**), há ainda outro possível problema associado ao último trecho de programa: o uso de incremento e decremento para acessar caracteres de um *string* multibyte não produzi-rem o efeito desejado, porque alguns caracteres ocupam mais de um byte.

Concluindo, a melhor maneira de classificar e transformar caracteres de modo portátil é usando as funções declaradas nos cabeçalhos `<ctype.h>` e `<wctype.h>` (v. **Seções 6.2 e 8.6**). A manipulação de caracteres multibytes é feita de modo portátil usando-se funções declaradas nos cabeçalhos `<stdlib.h>` e `<wchar.h>` (v. **Capítulo 8**).

A seguir, são apresentados exemplos de instruções não portáteis decorrentes de suposições equivocadas com relação a códigos de caracteres que não são especificadas pelo padrão de C.

### Exemplo:

```
putchar(65); /* Não é portátil */
```

**Comentário:** O padrão ISO de C não especifica a qual caractere está associado o inteiro 65 ou qualquer outro valor. Aliás, o padrão sequer especifica que deva haver alguma associação entre um caractere e um inteiro específico. Por exemplo, no código ASCII, o caractere 'a' está associado a 97, enquanto, no código EBCDIC, este mesmo caractere está associado a 129.

**Solução:** Usar alguma forma portátil de representação de caractere se esta existir, ou uma macro para isolar a ausência de portabilidade. Por exemplo:

```
putchar('A'); /* 100% portátil: sempre imprimirá A */
```

### Exemplo:

```
unsigned char c;
...
if (c >= 'a' && c <= 'z')
...
```

**Comentário:** Este trecho de código provavelmente pretende testar se um dado caractere é uma letra minúscula, mas ele não é portátil, pois, conforme discutido antes, nem todos os pontos de códigos entre 'a' e 'z' representam letras minúsculas em qualquer código de caracteres. Por exemplo, no código EBCDIC, o ponto de código 0xA1 está entre 'a' e 'z' e não representa sequer uma letra.

**Solução:** Caracteres monobytes devem ser categorizados usando-se as funções declaradas em `<ctype.h>` (v. **Seção 6.2**). Funções declaradas neste cabeçalho levam em consideração informações sobre localidade, incluindo o código de caracteres utilizado e, portanto, são capazes de classificar caracteres adequadamente. Pelo mesmo motivo, programas que lidam com caracteres extensos devem usar funções declaradas em `<wctype.h>` (v. **Seção 8.6**) para classificar caracteres extensos. Por exemplo:

```
#include <ctype.h>
...
unsigned char c;
...
if (islower(c))
...
```

### Exemplo:

```
if ('5' + 1 == '6')
... /* Instruções correspondentes ao if */
else
... /* Instruções correspondentes ao else */
```

**Comentário:** O problema neste exemplo não é exatamente de portabilidade, mas há um erro decorrente do desconhecimento do padrão de C com relação a códigos de caracteres, assim como ocorre com os exemplos anteriores. Isto é, a instrução **if** simplesmente não faz sentido porque sua condição é sempre verdadeira; i.e., '5' + 1 sempre resulta em '6', de acordo com o padrão ISO de C. Portanto, a parte **else** jamais será executada.

**Solução:** Neste caso, a solução consiste apenas em extinguir as instruções da parte **else** e remover a instrução **if**, mantendo convenientemente as instruções necessárias.



## 13.7 COMPILADORES

O padrão C99 estipula limites mínimos a que uma implementação deve obedecer com relação a diversas construções da linguagem. A **Tabela 13-3** apresenta os limites mínimos a que um compilador, que segue o padrão C99, deve obedecer. Se você seguir boas normas de estilo de programação (v. **Capítulo 6** do **Volume I**), provavelmente não terá que se preocupar com os limites apresentados na **Tabela 13-3**, mas, de qualquer modo, é recomendável tê-la por perto para dirimir dúvidas.

CONSTRUÇÃO DA LINGUAGEM	LIMITE MÍNIMO SUPORTADO
Níveis de aninho de instruções	127
Níveis de aninho de diretivas condicionais	63
Número de declaradores numa declaração	12
Pares de parênteses numa declaração	63
Pares de parênteses numa expressão	63
Caracteres levados em consideração num identificador com ligação interna ou sem ligação	63
Caracteres levados em consideração num identificador com ligação externa	31
Identificadores com ligação externa num arquivo de programa	4095
Identificadores locais a um bloco (para cada bloco)	511
Macros simultaneamente definidas num arquivo de programa	4095
Parâmetros numa definição ou chamada de função	127
Parâmetros numa definição ou invocação de macro	127
Caracteres numa linha de código (sem incluir comentários)	4095
Caracteres num <i>string</i> constante	4095
Espaço ocupado em memória por uma variável (programas com hospedeiro)	65.535 bytes
Aninho de diretivas <b>#include</b>	15
Partes <b>case</b> de uma instrução <b>switch</b> (sem incluir aquelas que constituem uma instrução <b>switch</b> aninhada)	1023
Membros de uma estrutura ou união	1023

CONSTRUÇÃO DA LINGUAGEM	LIMITE MÍNIMO SUPORTADO
Constantes de uma enumeração	1023
Aninho de estruturas ou uniões	63

Tabela 13-3: Limites mínimos de compilação especificados pelo padrão C99.

É importante chamar a atenção para o fato de os limites apresentados na **Tabela 13-3** serem *limites inferiores*. Isto significa dizer que, embora algumas implementações excedam estes limites, uma dada implementação pode recusar-se a compilar um programa que exceda estes limites. Portanto, qualquer programa que ultrapasse os limites apresentados na **Tabela 13-3** estará sujeito a problemas de portabilidade. É importante observar ainda que, em outras versões de padronização de C, alguns dos limites apresentados na **Tabela 13-3** são bem inferiores.

Uma recomendação importante relativa a compiladores e ambientes de desenvolvimento é que não se devem utilizar extensões oferecidas por um dado compilador, incluindo funções de biblioteca que não façam parte da biblioteca padrão de C. Se o apelo para não seguir esta recomendação for irresistível, isole essas especificidades e documente-as bem.

O programa utilitário *make*, discutido no **Capítulo 4** do **Volume I**, pode facilitar a tarefa de portar um programa de um ambiente para outro, bem como de um compilador para outro, pois permite que opções que dependam de sistema operacional ou de compilador sejam facilmente ajustadas para se adaptarem a novas situações.

Diretivas **#pragma** são fontes de problemas de portabilidade, pois elas representam tarefas dependentes de implementações específicas (v. **Seção 5.7** do **Volume I**). Portanto, se um programa depende de uma característica específica de um compilador, que é determinada por uma diretiva **#pragma**, deve haver algum modo de adotar esta característica em outros compiladores. Em outras palavras, uma diretiva **#pragma** deve ser usada num programa apenas se o programa puder funcionar sem ela.

## 13.8 SISTEMAS OPERACIONAIS

O transporte de programas de um sistema operacional para outro envolve mais problemas de portabilidade do que o transporte entre processadores ou compiladores. Como exemplos de incompatibilidades devido à mudança de sistema operacional têm-se:

- Tamanhos máximos de nomes de arquivos.

- O modo como volumes são tratados.
- O uso de alias (i.e., *links* simbólicos, em Unix, e atalhos, em Windows).
- Sensibilidade ao uso de letras maiúsculas ou minúsculas (e.g, DOS não diferencia maiúsculas/minúsculas, enquanto que Unix faz essa distinção).
- O modo de lidar com a comunicação entre processos (e.g., o sistema operacional UNIX versão 7 possui 15 sinais, enquanto o padrão POSIX sugere o uso de 31 sinais).

Uma das maiores diferenças entre sistemas operacionais diz respeito a sistemas de arquivos. Até um mesmo fabricante pode apresentar sistemas de arquivos incompatíveis entre si. Por exemplo, a Microsoft desenvolveu os sistemas de arquivos FAT16, FAT32 e NTFS que são incompatíveis entre si.

Conforme foi visto na **Capítulo 12 do Volume I**, muitos problemas de portabilidade relacionados com sistemas operacionais são causados por funções de entrada e saída baseadas em sistema. Outra situação na qual o uso de funções não portáveis e dependentes de sistema operacional é inevitável é quando se desejam informações sobre arquivos, tais como tamanho e data de criação, pois a biblioteca padrão de C não possui funções que forneçam tais informações (v. **Capítulo 12 do Volume I**).

Trechos de programas que apresentam alguma dependência de sistema operacional ou compilador devem ser isolados e usados em conjunto com compilação condicional. O uso de compilação condicional pode aliviar alguns dos problemas aqui enumerados, mas não resolve todos.

## 13.9 REPRESENTAÇÕES DE QUEBRA DE LINHA

Uma representação de **quebra de linha** consiste em um caractere ou uma sequência de caracteres que representa o final de uma linha de um arquivo-texto. Representações de quebra de linha variam entre sistemas operacionais e as mais comuns são apresentadas na **Tabela 13-4**. Nesta tabela, LF e CR representam, respectivamente, acréscimo de linha (*line feed*, em inglês) e retorno de carro (*carriage return*, em inglês)<sup>140</sup>.

---

<sup>140</sup> Exceto por razões históricas, você não precisa conhecer o significados destas expressões, basta saber que LF e CR são denominações de dois caracteres não imprimíveis encontrados em qualquer código de caracteres utilizados com a finalidade descrita aqui.

SISTEMAS OPERACIONAIS	REPRESENTAÇÃO DE QUEBRA DE LINHA
Unix, Linux, AIX, Xenix, Mac OS X, Amiga, RISC OS	LF
CP/M, DOS, Windows, OS/2	CR+LF
Commodore, Mac OS (até versão 9)	CR

Tabela 13-4: Representações de quebra de linha em diferentes sistemas operacionais.

Uma consequência prática da existência de diferentes representações de quebra de linha em diferentes sistemas operacionais é que arquivos de texto gerados num sistema podem ser lidos incorretamente em outro sistema. Por exemplo, um arquivo de texto criado no sistema Unix pode aparecer de modo estranho quando lido num editor de texto do Windows e vice-versa<sup>141</sup>.

Para facilitar a criação de programas portáteis, a linguagem C provê a sequência de escape '`\n`' utilizada para denotar quebra de linha independentemente do sistema operacional utilizado. Quando uma função da biblioteca padrão de C escreve '`\n`' num arquivo em modo de texto, este caractere é traduzido para o caractere ou sequência de caracteres que representa quebra de linha no sistema operacional corrente. Quando é feita uma leitura em modo de texto, ocorre a tradução inversa: o caractere ou sequência de caracteres que representa quebra de linha é traduzida no caractere '`\n`'. Mas, quando um arquivo é aberto em modo binário, não ocorre tradução na leitura ou na escrita (v. **Capítulo 10**).

Como consequência da existência de diferentes representações de quebra de linha, o tamanho de um arquivo de texto (i.e., o número de bytes efetivamente armazenados nele) depende da representação de quebra de linha utilizada. Ou seja, num sistema operacional que usa dois caracteres para representar cada quebra de linha (e.g., Windows), o tamanho de um arquivo de texto é maior do que o tamanho de um arquivo de texto contendo a mesma informação, mas criado num sistema que use apenas um caractere para representar quebra de linha (e.g., Unix). Mais precisamente, se um arquivo de texto possui  $n$  linhas, se ele for criado, por exemplo, no Windows, ele ocupará  $n$  bytes a mais do que o arquivo de texto equivalente criado no Unix.

---

<sup>141</sup> Hoje, muitos editores de texto são suficientemente *espertos* para reconhecerem o problema e adaptarem automaticamente o texto para apresentá-lo corretamente.

## 13.10 ASPECTOS PRAGMÁTICOS DE PORTABILIDADE

Uma pergunta que frequentemente intriga os programadores de C é: *Há como um programa escrito em C ser totalmente portátil?* A resposta mais simples e óbvia para esta questão seria: *Sim, desde que o programa seja 100% escrito estritamente em conformidade com o padrão* (v. **Seção 13.2**). No entanto, esta resposta não tem muita utilidade prática, pois nem sempre é possível (ou melhor, desejável, do ponto de vista prático) escrever um programa seguindo rigorosamente o padrão de uma linguagem. Por exemplo, a biblioteca padrão da linguagem pode não oferecer funções para resolver muitos problemas comuns de programação. Este é o caso da biblioteca padrão de C, que não provê uma única função capaz de ler um caractere no meio de entrada padrão sem que o usuário digite [ENTER], e esta é uma situação bastante comum em programação.

A situação torna-se ainda mais crítica quando se deseja construir um programa contendo uma interface gráfica, como a maioria dos aplicativos atuais. Neste caso, ainda é possível escrever um programa portátil, mas isto simplesmente não é viável. Ou seja, a melhor solução é utilizar uma biblioteca específica para o sistema operacional em questão (e.g., a biblioteca MFC da Microsoft, no caso de construção de uma interface gráfica para sistemas operacionais da família Windows). Mas, se isto faz obviamente com que o programa deixe de ser portátil, por que é o mais recomendado? Existem duas justificativas principais para essa escolha. Primeiro, o tempo de desenvolvimento do programa torna-se muito menor (muito mesmo!) porque o programador não terá de codificar muitas operações que ele já encontra prontas na referida biblioteca. Em segundo lugar, o programa será mais confiável porque utilizará funções desenvolvidas por programadores bem qualificados e com profundo conhecimento do sistema operacional utilizado. Além disso, esta biblioteca provavelmente já terá sido utilizada e testada por milhares de outros programadores<sup>142</sup>.

Extensões da linguagem C são frequentemente necessárias, mas o funcionamento de um programa não deve depender exclusivamente delas. Além disso, é conveniente que aspectos que são dependentes de plataforma sejam isolados, idealmente, em módulos separados, para facilitar sua localização e alteração.

**Exemplo:** O fragmento de programa a seguir ilustra como isolar porções de um programa que apresenta características que não são portáteis.

---

<sup>142</sup> Esta segunda justificativa, obviamente, também se aplica ao uso da biblioteca padrão de C.

```

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

/** Dependências devem ser isoladas num módulo separado **/

#ifdef LINUX
#define LIMPA_TELA system("clear") /* Linux */
#else
#define LIMPA_TELA system("cls") /* DOS */
#endif

/** Final do módulo contendo as dependências **/

int main(void)
{
    int i;

    for (i = 1; i < 80; ++i)
        printf("\nIsto e' um teste");

    printf("\n\nDigite [ENTER] para limpar a tela");

    getchar();

    LIMPA_TELA;

    return 0;
}

```

No exemplo anterior, o trecho de programa entre os comentários:

```

/** Dependências devem ser isoladas num módulo separado **/

e

/** Final do módulo contendo as dependências **/

```

deve ser colocado num módulo do programa dedicado a conter dependências de plataforma<sup>143</sup>.

---

143 Conforme descrito em detalhes no **Volume I**, um módulo consiste em um arquivo de programa mais o respectivo arquivo de cabeçalho. No caso do último exemplo, apenas um arquivo de cabeçalho seria necessário, visto que existem apenas duas definições de macros que não geram código. Num caso mais geral, entretanto, definições de funções podem ser necessárias.

Se um programa for suficientemente complexo, pode ser ainda mais interessante isolar todas as dependências de uma dada plataforma num mesmo módulo. Assim, utilizar-se-ia compilação condicional para decidir qual módulo é carregado numa dada construção do programa.

Resumindo, é possível escrever um programa totalmente portátil, mas isto nem sempre é viável do ponto de vista prático. Contudo, o leitor é advertido para o fato de que a criação de um programa não trivial totalmente portátil e com utilidade prática é uma tarefa virtualmente utópica devido à inexorável dependência do programa com a plataforma utilizada. Apesar disso, se as recomendações apresentadas neste capítulo forem seguidas, muitos prováveis problemas de portabilidade poderão ser minimizados.

## 13.11 EXERCÍCIOS DE REVISÃO

1. O que você entende por *portabilidade de programas*?
2. De acordo com o padrão ISO de C, descreva o significado de uma característica da linguagem classificada como:
  - (a) Definida pela implementação
  - (b) Não especificada
  - (c) Indefinida
3. O que é um programa estritamente em conformidade com o padrão ISO de C?
4. Por que o padrão ISO de C não especifica de forma precisa muitos aspectos da linguagem C?
5. Por que, muitas vezes, não é viável se ter um programa estritamente em conformidade com o padrão ISO de C?
6. (a) Que classes de problemas de portabilidade os cabeçalhos `<stdint.h>` e `<inttypes.h>`, discutidos no **Capítulo 2**, são capazes de resolver? (b) Que problemas de portabilidade de inteiros são prováveis mesmo com o uso destes cabeçalhos?
7. (a) É possível testar se uma determinada operação inteira irá gerar *overflow* antes de sua execução? (b) É possível testar se uma determinada operação inteira resultou em *overflow*?

8. Por que o trecho de programa a seguir não é portátil?

```
char c;
...
c = -10;
```

9. Por que o fragmento de programa a seguir pode causar o aborto do programa que o contém?

```
char array[20];
*((int *) &array[7]) = 547;
```

10. Por que o seguinte trecho de programa não é portátil?

```
int x = 25;
char c = *(char *) &x;
```

11. (a) O que é ordenação de bytes? (b) Descreva os tipos de ordenações de bytes existentes. (c) Que consequências a ordenação de bytes traz para a portabilidade de um programa?

12. Suponha que o tipo **int** seja armazenado em dois bytes. Qual é o valor do primeiro byte do número  $0 \times 1234$  quando a plataforma em questão usa ordenação de bytes:

(a) *Big-endian*

(b) *Little-endian*

13. Por que o primeiro programa (**Exemplo 1**) apresentado na **Seção 13.2** é portátil, enquanto o segundo programa (**Exemplo 2**), apresentado na mesma seção, que é bem parecido com o primeiro, não é portátil?

14. (a) O que é alinhamento de bytes? (b) Um programa pode acessar dados não alinhados? (c) Quais são as possíveis consequências decorrentes de acesso a dados desalinhados?

15. (a) Por que muitos compiladores preenchem uma estrutura com espaços vazios? (b) Por que o mesmo não ocorre com arrays? (c) Por que não existe preenchimento de uniões?

16. Como é possível evitar que preenchimentos de estruturas (i.e., espaços vazios) sejam copiados para um arquivo?

17. Quando uma diretiva **#pragma** pode causar problemas de portabilidade para um programa?



18. Cite alguns problemas de portabilidade causados por peculiaridades de sistemas operacionais.
19. Quais são as formas de representação de quebra de linha comumente utilizadas?
20. Quais das seguintes instruções não são portáveis?
  - a) `i = ar[i++];`
  - b) `i = 7, i++, i++;`
  - c) `i = ++i + 1;`
  - d) `i = i + 1;`
21. Quando um arquivo de texto é escrito no sistema Linux, ele ocupa 300 bytes. Quando o mesmo conteúdo deste arquivo é escrito no sistema DOS, ele ocupa 320 bytes. Quantas linhas há no arquivo?
22. De acordo com os resultados apresentados na **Tabela 13-2 (Seção 13.4)**, estruturas do tipo `tEstrutural` têm tamanho real (i.e., sem preenchimento) de 8 bytes. Por outro lado, o programa que demonstra o uso da macro `offsetof()` apresentado na **Seção 13.4** indica que, quando um programa é compilado utilizando alinhamento par no compilador Borland C++ 5.0, são inseridos dois bytes de preenchimento em estruturas do tipo `tEstrutural`. Utilizando os resultados apresentados pelo programa de demonstração da macro `offsetof()`, descubra a posição, relativa aos demais campos de uma estrutura deste tipo, onde estes bytes de preenchimento são inseridos.
23. O programa a seguir pode ser utilizado para determinar se uma dada representação é *little-endian* ou *big-endian*. Entretanto, apesar de funcionar na maioria das implementações de C, ele não é totalmente portátil. Explique o funcionamento deste programa e informe por que ele não é portátil. [Curiosidade: Problemas decorrentes de ordenação de bytes são comumente conhecidos como problemas XINU, pois o programa seguinte imprime XINU em algumas máquinas, enquanto em outras imprime UNIX.]

```
#include <stdio.h>

int main(void)
{
    long int str[2] = {0x554E4958, 0x0};
```

```

        /* O primeiro elemento do array str contém */
        /* os valores hexadecimais dos caracteres */
        /* de 'UNIX' em ASCII */

        /* Imprime o array str como */
        /* se ele fosse um string */
printf ("%s\n", (char *)str);

printf( "\n\tSe voce ler \"XINU\" acima, a "
        "representacao e' little-endian."
        "\n\tSe voce ler \"UNIX\", a "
        "representacao e' big-endian.\n" );

return 0;
}

```

24. Qual das duas chamadas de **assert()** no programa a seguir causam o aborto do programa e por quê?

```

#include <string.h>
#include <stdint.h>
#include <assert.h>

int main(void)
{
    int16_t x = 0xFF0;
    char ar1[2] = {0x0F, 0xF0};
    char ar2[2] = {0xF0, 0x0F};

    assert(!memcmp(&x, ar1, 2));
    assert(!memcmp(&x, ar2, 2));

    return 0;
}

```

25. O programa a seguir foi escrito com o objetivo de contar o número de caracteres de um arquivo de texto.

```

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

int main()
{
    const char *const nomeArquivo = "Texto.txt";
    int contador = 0;

```

```

FILE                *stream;
int                 c; /* Caractere ou EOF */

stream = fopen(nomeArquivo, "r");

if (stream == NULL) {
    printf( "Nao foi possivel abrir %s\n",
           nomeArquivo );
    return 1;
}

while (1) {
    c = fgetc(stream);

    if (c == EOF)
        break;

    ++contador;
}

printf( "Numero de caracteres em %s: %d\n",
       nomeArquivo, contador );

fclose(stream);

return 0;
}

```

Crie um arquivo de texto contendo algumas linhas e, em seguida, compile e execute este programa sob as seguintes condições, explicando algum possível resultado inesperado.

- a) Em Unix/Linux abrindo o arquivo com o modo "r"
- b) Em Unix/Linux abrindo o arquivo com o modo "rb"
- c) Em DOS abrindo o arquivo com o modo "r"
- d) Em DOS abrindo o arquivo com o modo "rb"

26. Escreva uma função que recebe como argumentos um ponteiro genérico e o tamanho do objeto apontado pelo ponteiro, e imprime o conteúdo do objeto em formato hexadecimal nas representações *big-endian* e *little-endian*.

# *Apêndice A*

---

## *Construtores da linguagem C*

## A.1 INTRODUÇÃO

Este apêndice apresenta, em ordem alfabética, para fácil referência, todos os construtores usados correntemente pela linguagem C. Aqui, considera-se um **construtor** qualquer *token* que possa ser usado num programa em C e que tenha significado especial nesta linguagem. Estes construtores são categorizados como mostra a **Figura A-1**.

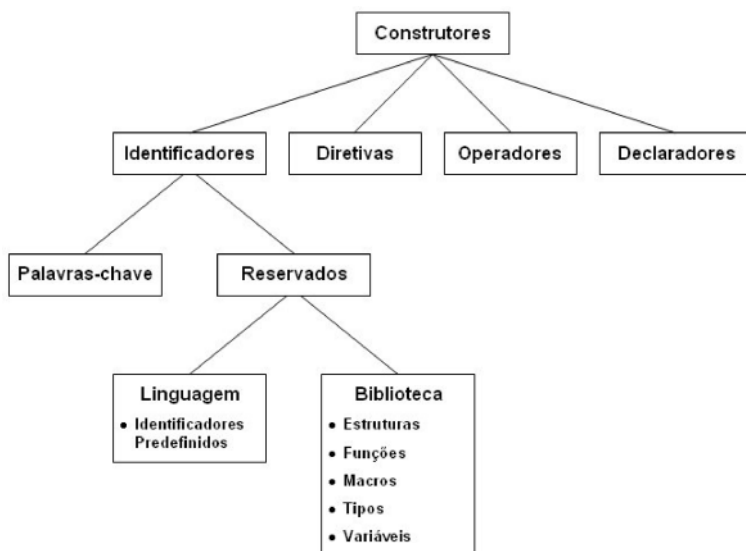


Figura A-1: Construtores da linguagem C.

Na **Figura A-1**, o quadro rotulado como *Diretivas* representa diretivas de pré-processamento, incluindo diretivas pragma definidas pela linguagem. Os identificadores predefinidos incluem macros predefinidas (e.g., `__DATE__`) e os identificadores `__func__` e `__VA_ARGS__` introduzidos pelo padrão C99 (v. **Volume I**). Espera-se que os demais componentes da **Figura A-1** sejam autoexplicativos<sup>144</sup>.

### A.1.1 PALAVRAS-CHAVE

Apenas os identificadores classificados como palavras-chave são rigorosamente proibidos de ser redefinidos em programas escritos em C.

<sup>144</sup> O construtor que se denomina, por simplicidade, *estrutura* neste apêndice trata-se na realidade de *rótulo de estrutura*.

A linguagem C possui apenas 37 palavras-chave, sendo que 32 delas foram definidas por C90 – o primeiro padrão ISO da linguagem C:

<b>auto</b>	<b>break</b>	<b>case</b>	<b>char</b>
<b>const</b>	<b>continue</b>	<b>default</b>	<b>do</b>
<b>double</b>	<b>else</b>	<b>enum</b>	<b>extern</b>
<b>float</b>	<b>for</b>	<b>goto</b>	<b>if</b>
<b>int</b>	<b>long</b>	<b>register</b>	<b>return</b>
<b>short</b>	<b>signed</b>	<b>sizeof</b>	<b>static</b>
<b>struct</b>	<b>switch</b>	<b>typedef</b>	<b>union</b>
<b>unsigned</b>	<b>void</b>	<b>volatile</b>	<b>while</b>

O padrão C99 inclui todas as palavras-chave definidas por C90 e acrescenta as seguintes:

- **\_Bool**
- **\_Complex**
- **\_Imaginary**
- **inline**
- **restrict**

## A.1.2 IDENTIFICADORES RESERVADOS

O programador deve abster-se de utilizar identificadores classificados como **reservados**, exceto em circunstâncias muito excepcionais (e.g., quando o programador escreve sua própria versão de uma função existente na biblioteca padrão de C). Identificadores reservados incluem não apenas aqueles expostos nos dois volumes desta obra como também inúmeros **identificadores reservados para uso futuro**. Os identificadores reservados correntemente em uso, de acordo com o padrão C99, são referenciados na **Seção A.2**, enquanto aqueles reservados para uso futuro são descritos na **Seção A.3**.

## A.2 REFERÊNCIAS DE CONSTRUTORES USADOS EM C

A tabela a seguir apresenta todos os construtores da linguagem C e suas colunas devem ser interpretadas da seguinte maneira:

- **Construtor** – os construtores da linguagem C em ordem alfabética.
- **Categoria** – a categoria de cada construtor, de acordo com a classificação apresentada na **Figura A-1**.
- **Definição** – em qual parte constituinte da linguagem cada construtor é definido; no caso de função ou variável global, representa o cabeçalho onde ela é declarada.
- **Referência** – a parte principal desta obra onde se pode obter a descrição de cada construtor.

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
-	Operador (binário)	Linguagem	<b>Vol. I - Seção 1.3.5</b>
-	Operador (unário)	Linguagem	<b>Vol. I - Seção 1.3.5</b>
--	Operador	Linguagem	<b>Vol. I - Seção 1.6.4</b>
!	Operador	Linguagem	<b>Vol. I - Seção 1.3.7</b>
!=	Operador	Linguagem	<b>Vol. I - Seção 1.3.6</b>
#	Diretiva	Pré-processador	<b>Vol. I - Seção 5.10</b>
#	Operador	Pré-processador	<b>Vol. I - Seção 5.3.5</b>
##	Operador	Pré-processador	<b>Vol. I - Seção 5.3.6</b>
<b>#define</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 5.3.1</b>
<b>#elif</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 5.4.1</b>
<b>#else</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 5.4.1</b>
<b>#endif</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 5.4.1</b>
<b>#error</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 5.6</b>
<b>#if</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 5.4.1</b>
<b>#ifdef</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 5.4.2</b>
<b>#ifndef</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 5.4.2</b>
<b>#include</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 2.2</b>
<b>#line</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 5.9</b>
<b>#pragma</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 5.7</b>
<b>#undef</b>	Diretiva	Pré-processador	<b>Vol. I - Seção 5.3.7</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>%</b>	Operador	Linguagem	<b>Vol. I - Seção 1.3.5</b>
<b>%=</b>	Operador	Linguagem	<b>Vol. I - Seção 1.6.3</b>
<b>&amp;</b>	Operador (binário)	Linguagem	<b>Vol. I - Seção 13.3.1</b>
<b>&amp;</b>	Operador (unário)	Linguagem	<b>Vol. I - Seção 3.2.1</b>
<b>&amp;&amp;</b>	Operador	Linguagem	<b>Vol. I - Seção 1.3.7</b>
<b>&amp;=</b>	Operador	Linguagem	<b>Vol. I - Seção 13.3.3</b>
<b>()</b>	Operador	Linguagem	<b>Vol. I - Seção 3.3.2</b>
<b>()</b>	Declarador	Linguagem	<b>Vol. I - Seção 3.3.1</b>
<b>(tipo)</b>	Operador	Linguagem	<b>Vol. I - Seção 1.6.1</b>
<b>*</b>	Operador (unário)	Linguagem	<b>Vol. I - Seção 3.2.3</b>
<b>*</b>	Declarador	Linguagem	<b>Vol. I - Seção 3.2.2</b>
<b>*</b>	Operador (binário)	Linguagem	<b>Vol. I - Seção 1.3.5</b>
<b>*=</b>	Operador	Linguagem	<b>Vol. I - Seção 1.6.3</b>
<b>,</b>	Operador	Linguagem	<b>Vol. I - Seção 1.9</b>
<b>/</b>	Operador	Linguagem	<b>Vol. I - Seção 1.3.5</b>
<b>/=</b>	Operador	Linguagem	<b>Vol. I - Seção 1.6.3</b>
<b>? :</b>	Operador	Linguagem	<b>Vol. I - Seção 1.8</b>
<b>[]</b>	Operador	Linguagem	<b>Vol. I - Seção 7.2.2</b>
<b>[]</b>	Declarador	Linguagem	<b>Vol. I - Seção 7.2.1</b>
<b>^</b>	Operador	Linguagem	<b>Vol. I - Seção 13.3.1</b>
<b>^=</b>	Operador	Linguagem	<b>Vol. I - Seção 13.3.3</b>
<b>__bool_true_ false_are_defined</b>	Macro	<stdbool.h>	<b>Vol. II - Seção 11.2</b>
<b>__DATE__</b>	Macro	Linguagem	<b>Vol. I - Seção 5.3.8</b>
<b>__FILE__</b>	Macro	Linguagem	<b>Vol. I - Seção 5.3.8</b>
<b>__func__</b>	Identificador predefinido	Linguagem	<b>Vol. I - Seção 8.6</b>
<b>__LINE__</b>	Macro	Linguagem	<b>Vol. I - Seção 5.3.8</b>
<b>__STDC__</b>	Macro	Linguagem	<b>Vol. I - Seção 5.3.8</b>
<b>__STDC_ HOSTED__</b>	Macro	Linguagem	<b>Vol. I - Seção 5.3.8</b>



CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>__STDC_IEC_559__</b>	Macro	Linguagem	<b>Vol. I - Seção 5.3.8</b>
<b>__STDC_IEC_559_COMPLEX__</b>	Macro	Linguagem	<b>Vol. I - Seção 5.3.8</b>
<b>__STDC_ISO_10646__</b>	Macro	Linguagem	<b>Vol. I - Seção 5.3.8</b>
<b>__STDC_VERSION__</b>	Macro	Linguagem	<b>Vol. I - Seção 5.3.8</b>
<b>__TIME__</b>	Macro	Linguagem	<b>Vol. I - Seção 5.3.8</b>
<b>__VA_ARGS__</b>	Identificador predefinido	Linguagem	<b>Vol. I - Seção 5.3.10</b>
<b>_Bool</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.4</b>
<b>_Complex</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.2.2</b>
<b>_Complex_I</b>	Macro	<complex.h>	<b>Vol. II - Seção 4.4.1</b>
<b>_Exit()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.3</b>
<b>_Imaginary</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.2.2</b>
<b>_Imaginary_I</b>	Macro	<complex.h>	<b>Vol. II - Seção 4.4.1</b>
<b>_IOFBF</b>	Macro	<stdio.h>	<b>Vol. II - Seção 10.7.2</b>
<b>_IOLBF</b>	Macro	<stdio.h>	<b>Vol. II - Seção 10.7.2</b>
<b>_IONBF</b>	Macro	<stdio.h>	<b>Vol. II - Seção 10.7.2</b>
<b>_Pragma</b>	Operador	Pré-processador	<b>Vol. I - Seção 5.8</b>
	Operador	Linguagem	<b>Vol. I - Seção 13.3.1</b>
	Operador	Linguagem	<b>Vol. I - Seção 1.3.7</b>
=	Operador	Linguagem	<b>Vol. I - Seção 13.3.3</b>
~	Operador	Linguagem	<b>Vol. I - Seção 13.3.1</b>
+	Operador (binário)	Linguagem	<b>Vol. I - Seção 1.3.5</b>
+	Operador (unário)	Linguagem	<b>Vol. I - Seção 1.3.5</b>
++	Operador	Linguagem	<b>Vol. I - Seção 1.6.4</b>
+=	Operador	Linguagem	<b>Vol. I - Seção 1.6.3</b>
<	Operador	Linguagem	<b>Vol. I - Seção 1.3.6</b>
<<	Operador	Linguagem	<b>Vol. I - Seção 13.3.2</b>
<<=	Operador	Linguagem	<b>Vol. I - Seção 13.3.3</b>
<=	Operador	Linguagem	<b>Vol. I - Seção 1.3.6</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
=	Operador	Linguagem	Vol. I - Seção 1.6.2
-=	Operador	Linguagem	Vol. I - Seção 1.6.3
==	Operador	Linguagem	Vol. I - Seção 1.3.6
>	Operador	Linguagem	Vol. I - Seção 1.3.6
->	Operador	Linguagem	Vol. I - Seção 9.3
>=	Operador	Linguagem	Vol. I - Seção 1.3.6
>>	Operador	Linguagem	Vol. I - Seção 13.3.2
>>=	Operador	Linguagem	Vol. I - Seção 13.3.3
•	Operador	Linguagem	Vol. I - Seção 9.3
abort()	Função	<stdlib.h>	Vol. II - Seção 12.2.3
abs()	Função	<stdlib.h>	Vol. II - Seção 2.6.2
acos()	Função	<math.h>	Vol. II - Seção 3.6.4
acos()	Macro	<tgmath.h>	Vol. II - Seção 4.5
acosf()	Função	<math.h>	Vol. II - Seção 3.6.3
acosh()	Macro	<tgmath.h>	Vol. II - Seção 4.5
acosh()	Função	<math.h>	Vol. II - Seção 3.6.5
acoshf()	Função	<math.h>	Vol. II - Seção 3.6.3
acoshl()	Função	<math.h>	Vol. II - Seção 3.6.3
acosl()	Função	<math.h>	Vol. II - Seção 3.6.3
and	Macro	<iso646.h>	Vol. II - Seção 11.3
and_eq	Macro	<iso646.h>	Vol. II - Seção 11.3
asctime()	Função	<time.h>	Vol. II - Seção 5.3.3
asin()	Macro	<tgmath.h>	Vol. II - Seção 4.5
asin()	Função	<math.h>	Vol. II - Seção 3.6.4
asinf()	Função	<math.h>	Vol. II - Seção 3.6.3
asinh()	Macro	<tgmath.h>	Vol. II - Seção 4.5
asinh()	Função	<math.h>	Vol. II - Seção 3.6.5
asinhf()	Função	<math.h>	Vol. II - Seção 3.6.3
asinhf()	Função	<math.h>	Vol. II - Seção 3.6.3
asinhf()	Função	<math.h>	Vol. II - Seção 3.6.3
asinhf()	Função	<math.h>	Vol. II - Seção 3.6.3
assert()	Macro	<assert.h>	Vol. II - Seção 11.4
atan()	Macro	<tgmath.h>	Vol. II - Seção 4.5
atan()	Função	<math.h>	Vol. II - Seção 3.6.4

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>atan2()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>atan2()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.4</b>
<b>atan2f()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>atan2l()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>atanf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>atanh()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.5</b>
<b>atanh()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>atanhf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>atanhl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>atanl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>atexit()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.3</b>
<b>atof()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 6.5.2</b>
<b>atoi()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 6.5.1</b>
<b>atol()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 6.5.1</b>
<b>atoll()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 6.5.1</b>
<b>auto</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 4.2.1</b>
<b>bitand</b>	Macro	<iso646.h>	<b>Vol. II - Seção 11.3</b>
<b>bitor</b>	Macro	<iso646.h>	<b>Vol. II - Seção 11.3</b>
<b>bool</b>	Macro	<stdbool.h>	<b>Vol. II - Seção 11.2</b>
<b>break</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.7.5</b>
<b>bsearch()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.6</b>
<b>btowc()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.3</b>
<b>BUFSIZ</b>	Macro	<stdio.h>	<b>Vol. II - Seção 10.7.2</b>
<b>cabs()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.6</b>
<b>cabsf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>cabsl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>cacos()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.3</b>
<b>cacosf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>cacosh()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.4</b>
<b>cacoshf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>cacoshl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>cacosl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>calloc()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.5</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>carg()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>carg()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.6</b>
<b>cargf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>cargl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>case</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.7.4</b>
<b>casin()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.3</b>
<b>casinf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>casinh()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.4</b>
<b>casinhf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>casinhl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>casinl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>catan()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.3</b>
<b>catanf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>catanh()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.4</b>
<b>catanhf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>catanhl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>catanl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>cbrt()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>cbrt()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>cbrtf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>cbrtl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>ccos()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.3</b>
<b>ccosf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>ccosh()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.4</b>
<b>ccoshf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>ccoshl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>ccosl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>ceil()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>ceil()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>ceilf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>ceill()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>cexp()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.5</b>
<b>cexpf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>cexpl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>char</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.2.2</b>
<b>CHAR_BIT</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>CHAR_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>CHAR_MIN</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>cimag()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.6</b>
<b>cimag()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>cimagf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>cimagl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>clearerr()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.12</b>
<b>clock()</b>	Função	<time.h>	<b>Vol. II - Seção 5.3.3</b>
<b>clock_t</b>	Tipo	<time.h>	<b>Vol. II - Seção 5.3.1</b>
<b>CLOCKS_PER_SEC</b>	Macro	<time.h>	<b>Vol. II - Seção 5.3.2</b>
<b>clog()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.5</b>
<b>clogf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>clogl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>compl</b>	Macro	<iso646.h>	<b>Vol. II - Seção 11.3</b>
<b>complex</b>	Macro	<complex.h>	<b>Vol. II - Seção 4.4.1</b>
<b>conj()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.6</b>
<b>conj()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>conjf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>conjl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>const</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 4.5.1</b>
<b>continue</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.7.5</b>
<b>copysign()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.12</b>
<b>copysign()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>copysignf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>copysignl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>cos()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>cos()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.4</b>
<b>cosf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>cosh()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.5</b>
<b>cosh()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>coshf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>coshl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>cosl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>cpow()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.5</b>
<b>cpowf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>cpowl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>cproj()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.6</b>
<b>cproj()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>cprojf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>cprojl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>creal()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>creal()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.6</b>
<b>crealf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>creall()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>csin()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.3</b>
<b>csinf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>csinh()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.4</b>
<b>csinhf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>csinhl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>csinl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>csqrt()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.5</b>
<b>csqrtf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>csqrtl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>ctan()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.3</b>
<b>ctanf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>ctanh()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.4</b>
<b>ctanhf()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>ctanhl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>ctanl()</b>	Função	<complex.h>	<b>Vol. II - Seção 4.4.2</b>
<b>ctime()</b>	Função	<time.h>	<b>Vol. II - Seção 5.3.3</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>CX_LIMITED_RANGE</b>	Pragma	Linguagem	<b>Vol. II - Seção 4.3</b>
<b>DBL_DIG</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>DBL_EPSILON</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>DBL_MANT_DIG</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>DBL_MAX</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>DBL_MAX_10_EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>DBL_MAX_EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>DBL_MIN</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>DBL_MIN_10_EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>DBL_MIN_EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>DECIMAL_DIG</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>default</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.7.4</b>
<b>difftime()</b>	Função	<time.h>	<b>Vol. II - Seção 5.3.3</b>
<b>div()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 2.6.2</b>
<b>div_t</b>	Tipo	<stdlib.h>	<b>Vol. II - Seção 2.6.1</b>
<b>do</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.7.3</b>
<b>double</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.2.2</b>
<b>double_t</b>	Tipo	<math.h>	<b>Vol. II - Seção 3.6.1</b>
<b>EDOM</b>	Macro	<errno.h>	<b>Vol. II - Seção 11.5.1</b>
<b>EILSEQ</b>	Macro	<errno.h>	<b>Vol. II - Seção 11.5.1</b>
<b>else</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.7.4</b>
<b>enum</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 9.12</b>
<b>EOF</b>	Macro	<stdio.h>	<b>Vol. II - Capítulo 10</b>
<b>ERANGE</b>	Macro	<errno.h>	<b>Vol. II - Seção 11.5.1</b>
<b>erf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.7</b>
<b>erf()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>erfc()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.7</b>
<b>erfc()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>erfcf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>erfcl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>erff()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>erfl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>errno</b>	Variável global	<errno.h>	<b>Vol. II - Seção 11.5.2</b>
<b>exit()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.3</b>
<b>EXIT_FAILURE</b>	Macro	<stdlib.h>	<b>Vol. II - Seção 12.2.2</b>
<b>EXIT_SUCCESS</b>	Macro	<stdlib.h>	<b>Vol. II - Seção 12.2.2</b>
<b>exp()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>exp()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>exp2()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>exp2()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>exp2f()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>exp2l()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>expf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>expl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>expm1()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>expm1()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>expm1f()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>expm1l()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>extern</b>	Palavra-chave	Linguagem	<b>Vol. I - Seções 3.3.6, 4.7.2</b>
<b>fabs()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.12</b>
<b>fabs()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>fabsf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>fabsl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>false</b>	Macro	<stdbool.h>	<b>Vol. II - Seção 11.2</b>
<b>fclose()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.1</b>
<b>fdim()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>fdim()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.9</b>
<b>fdimf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>fdiml()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>FE_ALL_EXCEPT</b>	Macro	<fenv.h>	<b>Vol. II - Seção 3.7.2</b>



CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>FE_DFL_ENV</b>	Macro	<fenv.h>	<b>Vol. II - Seção 3.7.2</b>
<b>FE_DIVBYZERO</b>	Macro	<fenv.h>	<b>Vol. II - Seção 3.7.2</b>
<b>FE_DOWNWARD</b>	Macro	<fenv.h>	<b>Vol. II - Seção 3.7.2</b>
<b>FE_INEXACT</b>	Macro	<fenv.h>	<b>Vol. II - Seção 3.7.2</b>
<b>FE_INVALID</b>	Macro	<fenv.h>	<b>Vol. II - Seção 3.7.2</b>
<b>FE_OVERFLOW</b>	Macro	<fenv.h>	<b>Vol. II - Seção 3.7.2</b>
<b>FE_TONEAREST</b>	Macro	<fenv.h>	<b>Vol. II - Seção 3.7.2</b>
<b>FE_TOWARDZERO</b>	Macro	<fenv.h>	<b>Vol. II - Seção 3.7.2</b>
<b>FE_UNDERFLOW</b>	Macro	<fenv.h>	<b>Vol. II - Seção 3.7.2</b>
<b>FE_UPWARD</b>	Macro	<fenv.h>	<b>Vol. II - Seção 3.7.2</b>
<b>feclearexcept()</b>	Função	<fenv.h>	<b>Vol. II - Seção 3.7.3</b>
<b>fegetenv()</b>	Função	<fenv.h>	<b>Vol. II - Seção 3.7.3</b>
<b>fegetexceptflag()</b>	Função	<fenv.h>	<b>Vol. II - Seção 3.7.3</b>
<b>fegetround()</b>	Função	<fenv.h>	<b>Vol. II - Seção 3.7.3</b>
<b>feholdexcept()</b>	Função	<fenv.h>	<b>Vol. II - Seção 3.7.3</b>
<b>FENV_ACCESS</b>	Pragma	Linguagem	<b>Vol. II - Seção 3.4.2</b>
<b>fenv_t</b>	Tipo	<fenv.h>	<b>Vol. II - Seção 3.7.1</b>
<b>feof()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.12</b>
<b>feraiseexcept()</b>	Função	<fenv.h>	<b>Vol. II - Seção 3.7.3</b>
<b>ferror()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.12</b>
<b>fesetenv()</b>	Função	<fenv.h>	<b>Vol. II - Seção 3.7.3</b>
<b>fesetexceptflag()</b>	Função	<fenv.h>	<b>Vol. II - Seção 3.7.3</b>
<b>fesetround()</b>	Função	<fenv.h>	<b>Vol. II - Seção 3.7.3</b>
<b>fetestexcept()</b>	Função	<fenv.h>	<b>Vol. II - Seção 3.7.3</b>
<b>feupdateenv()</b>	Função	<fenv.h>	<b>Vol. II - Seção 3.7.3</b>
<b>fexcept_t</b>	Tipo	<fenv.h>	<b>Vol. II - Seção 3.7.1</b>
<b>fflush()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.2</b>
<b>fgetc()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.3</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>fgetpos()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.9</b>
<b>fgets()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.4</b>
<b>fgetwc()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>fgetws()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>FILE</b>	Tipo	<stdio.h>	<b>Vol. II - Seção 10.4</b>
<b>FILENAME_</b> <b>MAX</b>	Macro	<stdio.h>	<b>Vol. II - Seção 10.7.1</b>
<b>float</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.2.2</b>
<b>float_t</b>	Tipo	<math.h>	<b>Vol. II - Seção 3.6.1</b>
<b>floor()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>floor()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>floorf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>floorl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>FLT_DIG</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>FLT_EPSILON</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>FLT_EVAL_</b> <b>METHOD</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>FLT_MANT_DIG</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>FLT_MAX</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>FLT_MAX_10_</b> <b>EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>FLT_MAX_EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>FLT_MIN</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>FLT_MIN_10_</b> <b>EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>FLT_MIN_EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>FLT_RADIX</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>FLT_ROUND</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>fma()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.12</b>
<b>fma()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>fmaf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>fmal()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>fmax()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>fmax()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.9</b>
<b>fmaxf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>fmaxl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>fmin()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>fmin()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.9</b>
<b>fminf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>fminl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>fmod()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>fmod()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.11</b>
<b>fmodf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>fmodl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>fopen()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.1</b>
<b>FOPEN_MAX</b>	Macro	<stdio.h>	<b>Vol. II - Seção 10.7.1</b>
<b>for</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.7.3</b>
<b>FP_CONTRACT</b>	Pragma	Linguagem	<b>Vol. II - Seção 3.4.1</b>
<b>FP_FAST_FMA</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>FP_FAST_FMAF</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>FP_FAST_FMAL</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>FP_ILOGB0</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>FP_ILOGBNAN</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>FP_INFINITE</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>FP_NAN</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>FP_NORMAL</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>FP_</b> <b>SUBNORMAL</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>FP_ZERO</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>fpclassify()</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.13</b>
<b>fpos_t</b>	Tipo	<stdio.h>	<b>Vol. II - Seção 10.4</b>
<b>fprintf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.7</b>
<b>fputc()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.3</b>
<b>fputs()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.4</b>
<b>fputwc()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>fputws()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>fread()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.5</b>
<b>free()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.5</b>
<b>freopen()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.1</b>
<b>frexp()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>frexp()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>frexpf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>frexpl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>fscanf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.6</b>
<b>fseek()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.9</b>
<b>fsetpos()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.9</b>
<b>ftell()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.9</b>
<b>fwide()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>fwprintf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>fwrite()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.5</b>
<b>fwscanf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>getc()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.3</b>
<b>getchar()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.3</b>
<b>getenv()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.3</b>
<b>gets()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.4</b>
<b>getwc()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>getwchar()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>gmtime()</b>	Função	<time.h>	<b>Vol. II - Seção 5.3.3</b>
<b>goto</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.7.5</b>
<b>HUGE_VAL</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>HUGE_VALF</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>HUGE_VALL</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>hypot()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.12</b>
<b>hypot()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>hypotf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>hypotl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>I</b>	Macro	<complex.h>	<b>Vol. II - Seção 4.4.1</b>
<b>if</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.7.4</b>
<b>ilogb()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>ilogb()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>ilogbf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>ilogbl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>imaginary</b>	Macro	<complex.h>	<b>Vol. II - Seção 4.4.1</b>
<b>imaxabs()</b>	Função	<inttypes.h>	<b>Vol. II - Seção 2.5.3</b>
<b>imaxdiv()</b>	Função	<inttypes.h>	<b>Vol. II - Seção 2.5.3</b>
<b>imaxdiv_t</b>	Tipo	<inttypes.h>	<b>Vol. II - Seção 2.5.1</b>
<b>INFINITY</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>inline</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 3.6</b>
<b>int</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.2.2</b>
<b>INT_FAST16_ MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_FAST16_ MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_FAST32_ MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_FAST32_ MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_FAST64_ MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_FAST64_ MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_FAST8_ MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_FAST8_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_LEAST16_ MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_LEAST16_ MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_LEAST32_ MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_LEAST32_ MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>INT_LEAST64_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_LEAST64_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_LEAST8_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_LEAST8_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>INT_MIN</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>INT16_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT16_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>int16_t</b>	Tipo	<stdint.h>	<b>Vol. II - Seção 2.4.1</b>
<b>INT32_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT32_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>int32_t</b>	Tipo	<stdint.h>	<b>Vol. II - Seção 2.4.1</b>
<b>INT64_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT64_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>int64_t</b>	Tipo	<stdint.h>	<b>Vol. II - Seção 2.4.1</b>
<b>INT8_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INT8_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>int8_t</b>	Tipo	<stdint.h>	<b>Vol. II - Seção 2.4.1</b>
<b>INTMAX_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INTMAX_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>intmax_t</b>	Tipo	<stdint.h>	<b>Vol. II - Seção 2.4.1</b>
<b>INTPTR_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>INTPTR_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>isalnum()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>isalpha()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>isblank()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>iscntrl()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>isdigit()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>isfinite()</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.13</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>isgraph()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>isgreater()</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.13</b>
<b>isgreaterequal()</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.13</b>
<b>isinf()</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.13</b>
<b>isless()</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.13</b>
<b>islessequal()</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.13</b>
<b>islessgreater()</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.13</b>
<b>islower()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>isnan()</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.13</b>
<b>isnormal()</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.13</b>
<b>isprint()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>ispunct()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>isspace()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>isunordered()</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.13</b>
<b>isupper()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>iswalnum()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswalpha()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswblank()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswcntrl()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswctype()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswdigit()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswgraph()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswlower()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswprint()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswpunct()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswspace()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswupper()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>iswxdigit()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>isxdigit()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.1</b>
<b>jmp_buf</b>	Tipo	<setjmp.h>	<b>Vol. II - Seção 11.7.1</b>
<b>L_tmpnam</b>	Macro	<stdio.h>	<b>Vol. II - Seção 10.7.10</b>
<b>labs()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 2.6.2</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>LC_ALL</b>	Macro	<locale.h>	<b>Vol. II - Seção 5.2.2</b>
<b>LC_COLLATE</b>	Macro	<locale.h>	<b>Vol. II - Seção 5.2.2</b>
<b>LC_CTYPE</b>	Macro	<locale.h>	<b>Vol. II - Seção 5.2.2</b>
<b>LC_MONETARY</b>	Macro	<locale.h>	<b>Vol. II - Seção 5.2.2</b>
<b>LC_NUMERIC</b>	Macro	<locale.h>	<b>Vol. II - Seção 5.2.2</b>
<b>LC_TIME</b>	Macro	<locale.h>	<b>Vol. II - Seção 5.2.2</b>
<b>lconv</b>	Estrutura	<locale.h>	<b>Vol. II - Seção 5.2.1</b>
<b>LDBL_DIG</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>LDBL_EPSILON</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>LDBL_MANT_DIG</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>LDBL_MAX</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>LDBL_MAX_10_EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>LDBL_MAX_EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>LDBL_MIN</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>LDBL_MIN_10_EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>LDBL_MIN_EXP</b>	Macro	<float.h>	<b>Vol. II - Seção 3.5</b>
<b>ldexp()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>ldexp()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>ldexpf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>ldexpl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>ldiv()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 2.6.2</b>
<b>ldiv_t</b>	Tipo	<stdlib.h>	<b>Vol. II - Seção 2.6.1</b>
<b>lgamma()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.10</b>
<b>lgamma()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>lgammaf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>lgammal()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>llabs()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 2.6.2</b>
<b>lldiv()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 2.6.2</b>
<b>lldiv_t</b>	Tipo	<stdlib.h>	<b>Vol. II - Seção 2.6.1</b>



CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>LLONG_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>LLONG_MIN</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>llrint()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>llrint()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>llrintf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>llrintl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>llround()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>llround()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>llroundf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>llroundl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>localeconv()</b>	Função	<locale.h>	<b>Vol. II - Seção 5.2.3</b>
<b>localtime()</b>	Função	<time.h>	<b>Vol. II - Seção 5.3.3</b>
<b>log()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>log()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>log10()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>log10()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>log10f()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>log10l()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>log1p()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>log1p()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>log1pf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>log1pl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>log2()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>log2()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>log2f()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>log2l()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>logb()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>logb()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>logbf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>logbl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>logf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>logl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>long</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.2.2</b>
<b>LONG_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>LONG_MIN</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>longjmp()</b>	Função	<setjmp.h>	<b>Vol. II - Seção 11.7.2</b>
<b>lrint()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>lrint()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>lrintf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>lrintl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>lround()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>lround()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>lroundf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>lroundl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>malloc()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.5</b>
<b>MATH_ERREXCEPT</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>math_errhandling</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>MATH_ERRNO</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>MB_CUR_MAX</b>	Macro	<stdlib.h>	<b>Vol. II - Seção 12.2.2</b>
<b>MB_LEN_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>mblen()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 8.4.2</b>
<b>mbrlen()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.3</b>
<b>mbrtowc()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.3</b>
<b>mbsinit()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.3</b>
<b>mbsrtowcs()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.3</b>
<b>mbstate_t</b>	Tipo	<wchar.h>	<b>Vol. II - Seção 8.5.1</b>
<b>mbstowcs()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 8.4.2</b>
<b>mbtowc()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 8.4.2</b>
<b>memchr()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.3</b>
<b>memcmp()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.3</b>
<b>memcpy()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.3</b>
<b>memmove()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.3</b>
<b>memset()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.3</b>
<b>mktime()</b>	Função	<time.h>	<b>Vol. II - Seção 5.3.3</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>modf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.12</b>
<b>modff()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>modfl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>NAN</b>	Macro	<math.h>	<b>Vol. II - Seção 3.6.2</b>
<b>nan()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.12</b>
<b>nanf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>nanl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>nearbyint()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>nearbyint()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>nearbyintf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>nearbyintl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>nextafter()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>nextafter()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>nextafterf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>nextafterl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>nexttoward()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>nexttoward()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>nexttowardf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>nexttowardl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>not</b>	Macro	<iso646.h>	<b>Vol. II - Seção 11.3</b>
<b>not_eq</b>	Macro	<iso646.h>	<b>Vol. II - Seção 11.3</b>
<b>NULL</b>	Macro	<stdlib.h> <stddef.h> <time.h> <string.h> <wchar.h>	<b>Vol. II - Seção 12.2.2</b> <b>Vol. II - Seção 12.3.2</b> <b>Vol. II - Seção 5.3.2</b> <b>Vol. II - Seção 6.3.2</b> <b>Vol. II - Seção 8.5.2</b>
<b>offsetof()</b>	Macro	<stddef.h>	<b>Vol. II - Seção 12.3.2</b>
<b>or</b>	Macro	<iso646.h>	<b>Vol. II - Seção 11.3</b>
<b>or_eq</b>	Macro	<iso646.h>	<b>Vol. II - Seção 11.3</b>
<b>perror()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.13</b>
<b>pow()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>pow()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>pow()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>powl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>PRId16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRId32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRId64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRId8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIdFAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIdFAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIdFAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIdFAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIdLEAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIdLEAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIdLEAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIdLEAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIdMAX</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIdPTR</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRi16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRi32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRi64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRi8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRiFAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRiFAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRiFAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRiFAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRiLEAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRiLEAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRiLEAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRiLEAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRiMAX</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRiPTR</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>printf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.7</b>
<b>PRIo16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>PRIo32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIo64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIo8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoFAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoFAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoFAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoFAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoLEAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoLEAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoLEAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoLEAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoMAX</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoPTR</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIo16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIo32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIo64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIo8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoFAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoFAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoFAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoFAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoLEAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoLEAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoLEAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoLEAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoMAX</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIoPTR</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIx16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIX16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIx32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIX32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIX64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIx64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>PRIx8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIX8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIXFAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIxFAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIXFAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIxFAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIxFAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIXFAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIXFAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIxFAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIXLEAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIxLEAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIXLEAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIxLEAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIxLEAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIXLEAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIxLEAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIXLEAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIXMAX</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIxMAX</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIXPTR</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PRIxPTR</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>PTRDIFF_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>PTRDIFF_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>ptrdiff_t</b>	Tipo	<stddef.h>	<b>Vol. II - Seção 12.3.1</b>
<b>putc()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.3</b>
<b>putchar()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.3</b>
<b>puts()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.4</b>
<b>putwc()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>putwchar()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>qsort()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.6</b>
<b>raise()</b>	Função	<signal.h>	<b>Vol. II - Seção 11.6.3</b>
<b>rand()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.4</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>RAND_MAX</b>	Macro	<stdlib.h>	<b>Vol. II - Seção 12.2.2</b>
<b>realloc()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.5</b>
<b>register</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 4.4</b>
<b>remainder()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>remainder()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.11</b>
<b>remainderf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>remainderl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>remove()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.10</b>
<b>remquo()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>remquo()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.11</b>
<b>remquof()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>remquol()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>rename()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.10</b>
<b>restrict</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 7.8</b>
<b>return</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 3.3.4</b>
<b>rewind()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.9</b>
<b>rint()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>rint()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>rintf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>rintl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>round()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>round()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>roundf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>roundl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>scalbln()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>scalbln()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>scalblnf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>scalblnl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>scalbn()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>scalbn()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>scalbnf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>scalbnl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>scanf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.6</b>
<b>SCHAR_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>SCHAR_MIN</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>SCNd16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNd32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNd64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNd8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNdFAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNdFAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNdFAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNdFAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNdLEAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNdLEAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNdLEAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNdLEAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNdMAX</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNdPTR</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNi16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNi32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNi64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNi8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNiFAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNiFAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNiFAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNiFAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNiLEAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNiLEAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNiLEAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNiLEAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNiMAX</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNiPTR</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNo16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>



CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>SCNo32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNo64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNo8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNoFAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNoFAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNoFAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNoFAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNoLEAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNoLEAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNoLEAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNoLEAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNoMAX</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNoPTR</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNu16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNu32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNu64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNu8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNuFAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNuFAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNuFAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNuFAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNuLEAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNuLEAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNuLEAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNuLEAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNuMAX</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNuPTR</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNx16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNx32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNx64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNx8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNxFAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNxFAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>SCNxFAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNxFAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNxLEAST16</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNxLEAST32</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNxLEAST64</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNxLEAST8</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNxMAX</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SCNxPTR</b>	Macro	<inttypes.h>	<b>Vol. II - Seção 2.5.2</b>
<b>SEEK_CUR</b>	Macro	<stdio.h>	<b>Vol. II - Seção 10.7.9</b>
<b>SEEK_END</b>	Macro	<stdio.h>	<b>Vol. II - Seção 10.7.9</b>
<b>SEEK_SET</b>	Macro	<stdio.h>	<b>Vol. II - Seção 10.7.9</b>
<b>setbuf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.2</b>
<b>setjmp()</b>	Função	<setjmp.h>	<b>Vol. II - Seção 11.7.2</b>
<b>setlocale()</b>	Função	<locale.h>	<b>Vol. II - Seção 5.2.3</b>
<b>setvbuf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.2</b>
<b>short</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.2.2</b>
<b>SHRT_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>SHRT_MIN</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>SIG_ATOMIC_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>SIG_ATOMIC_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>sig_atomic_t</b>	Tipo	<signal.h>	<b>Vol. II - Seção 11.6.1</b>
<b>SIG_DFL</b>	Macro	<signal.h>	<b>Vol. II - Seção 11.6.2</b>
<b>SIG_IGN</b>	Macro	<signal.h>	<b>Vol. II - Seção 11.6.2</b>
<b>SIGABRT</b>	Macro	<signal.h>	<b>Vol. II - Seção 11.6.2</b>
<b>SIGBREAK</b>	Macro	<signal.h>	<b>Vol. II - Seção 11.6.2</b>
<b>SIGFPE</b>	Macro	<signal.h>	<b>Vol. II - Seção 11.6.2</b>
<b>SIGILL</b>	Macro	<signal.h>	<b>Vol. II - Seção 11.6.2</b>
<b>SIGINT</b>	Macro	<signal.h>	<b>Vol. II - Seção 11.6.2</b>
<b>signal()</b>	Função	<signal.h>	<b>Vol. II - Seção 11.6.3</b>
<b>signbit()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.12</b>
<b>signed</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.2.2</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>SIGSEGV</b>	Macro	<signal.h>	<b>Vol. II - Seção 11.6.2</b>
<b>SIGTERM</b>	Macro	<signal.h>	<b>Vol. II - Seção 11.6.2</b>
<b>sin()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.4</b>
<b>sin()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>sinf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>sinh()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>sinh()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.5</b>
<b>sinhf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>sinhl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>sinl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>SIZE_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>size_t</b>	Tipo	<time.h> <string.h> <wchar.h> <stdlib.h> <stddef.h>	<b>Vol. II - Seção 5.3.1</b> <b>Vol. II - Seção 6.3.1</b> <b>Vol. II - Seção 8.5.1</b> <b>Vol. II - Seção 12.2.1</b> <b>Vol. II - Seção 12.3.1</b>
<b>sizeof</b>	Palavra-chave (operador)	Linguagem	<b>Vol. I - Seção 1.6.5</b>
<b>snprintf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.8</b>
<b>sprintf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.8</b>
<b>sqrt()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>sqrt()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.8</b>
<b>sqrtf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>sqrtl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>srand()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.4</b>
<b>sscanf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.8</b>
<b>static</b>	Palavra-chave	Linguagem	<b>Vol. I - Capítulo 4</b>
<b>stderr</b>	Variável global	<stdio.h>	<b>Vol. II - Seção 10.6</b>
<b>stdin</b>	Variável global	<stdio.h>	<b>Vol. II - Seção 10.6</b>
<b>stdout</b>	Variável global	<stdio.h>	<b>Vol. II - Seção 10.6</b>
<b>strcat()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strchr()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strcmp()</b>	Função	<string.h>	<b>Vol. II - Seção 6.4.2</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>strcoll()</b>	Função	<string.h>	<b>Vol. II - Seção 6.4.2</b>
<b>strcpy()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strcspn()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strerror()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strftime()</b>	Função	<time.h>	<b>Vol. II - Seção 5.3.3</b>
<b>strlen()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strncat()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strncpy()</b>	Função	<string.h>	<b>Vol. II - Seção 6.4.2</b>
<b>strncpy()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strpbrk()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strrchr()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strspn()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strstr()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strtod()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 6.5.2</b>
<b>strtof()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 6.5.2</b>
<b>strtoimax()</b>	Função	<inttypes.h>	<b>Vol. II - Seção 2.5.3</b>
<b>strtok()</b>	Função	<string.h>	<b>Vol. II - Seção 6.3.4</b>
<b>strtol()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 6.5.1</b>
<b>strtold()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 6.5.2</b>
<b>strtoll()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 6.5.1</b>
<b>strtoul()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 6.5.1</b>
<b>strtoull()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 6.5.1</b>
<b>strtoumax()</b>	Função	<inttypes.h>	<b>Vol. II - Seção 2.5.3</b>
<b>struct</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 9.2</b>
<b>strxfrm()</b>	Função	<string.h>	<b>Vol. II - Seção 6.4.2</b>
<b>switch</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.7.4</b>
<b>swprintf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>swscanf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>system()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 12.2.3</b>
<b>tan()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>tan()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.4</b>
<b>tanf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>tanh()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.5</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>tanh()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>tanhf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>tanhll()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>tanl()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>tgamma()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>tgamma()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.10</b>
<b>tgammaf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>tgammall()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>time()</b>	Função	<time.h>	<b>Vol. II - Seção 5.3.3</b>
<b>time_t</b>	Tipo	<time.h>	<b>Vol. II - Seção 5.3.1</b>
<b>tm</b>	Estrutura	<time.h> <wchar.h>	<b>Vol. II - Seção 5.3.1</b> <b>Vol. II - Seção 8.5.1</b>
<b>TMP_MAX</b>	Macro	<stdio.h>	<b>Vol. II - Seção 10.7.10</b>
<b>tmpfile()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.11</b>
<b>tmpnam()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.11</b>
<b>tolower()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.2</b>
<b>toupper()</b>	Função	<ctype.h>	<b>Vol. II - Seção 6.2.2</b>
<b>towctrans()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.4</b>
<b>tolower()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.4</b>
<b>towupper()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.4</b>
<b>true</b>	Macro	<stdbool.h>	<b>Vol. II - Seção 11.2</b>
<b>trunc()</b>	Macro	<tgmath.h>	<b>Vol. II - Seção 4.5</b>
<b>trunc()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.6</b>
<b>truncf()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>truncll()</b>	Função	<math.h>	<b>Vol. II - Seção 3.6.3</b>
<b>typedef</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 4.6</b>
<b>UCHAR_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>UINT_FAST16_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>UINT_FAST32_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>UINT_FAST64_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>UINT_FAST8_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>UINT_LEAST16_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>UINT_LEAST32_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>UINT_LEAST64_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>UINT_LEAST8_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>UINT_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>UINT16_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>uint16_t</b>	Tipo	<stdint.h>	<b>Vol. II - Seção 2.4.1</b>
<b>UINT32_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>uint32_t</b>	Tipo	<stdint.h>	<b>Vol. II - Seção 2.4.1</b>
<b>UINT64_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>uint64_t</b>	Tipo	<stdint.h>	<b>Vol. II - Seção 2.4.1</b>
<b>UINT8_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>uint8_t</b>	Tipo	<stdint.h>	<b>Vol. II - Seção 2.4.1</b>
<b>UINTMAX_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>uintmax_t</b>	Tipo	<stdint.h>	<b>Vol. II - Seção 2.4.1</b>
<b>UINTPTR_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>ULLONG_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>ULONG_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>
<b>ungetc()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.13</b>
<b>ungetwc()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>union</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 9.9</b>
<b>unsigned</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.2.2</b>
<b>USHRT_MAX</b>	Macro	<limits.h>	<b>Vol. II - Seção 2.3</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>va_arg()</b>	Macro	<stdarg.h>	<b>Vol. II - Seção 9.2.2</b>
<b>va_copy()</b>	Macro	<stdarg.h>	<b>Vol. II - Seção 9.2.2</b>
<b>va_end()</b>	Macro	<stdarg.h>	<b>Vol. II - Seção 9.2.2</b>
<b>va_list</b>	Tipo	<stdarg.h>	<b>Vol. II - Seção 9.2.1</b>
<b>va_start()</b>	Macro	<stdarg.h>	<b>Vol. II - Seção 9.2.2</b>
<b>vfprintf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.7</b>
<b>vfprintf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.6</b>
<b>vfwprintf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>vfwscanf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>void</b>	Palavra-chave	Linguagem	<b>Vol. I - Seções 3.3.1, 9.13, 11.4</b>
<b>volatile</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 4.5.2</b>
<b>vprintf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.7</b>
<b>vscanf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.6</b>
<b>vsnprintf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.8</b>
<b>vsprintf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.8</b>
<b>vsscanf()</b>	Função	<stdio.h>	<b>Vol. II - Seção 10.7.8</b>
<b>vswprintf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>vswscanf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>vwprintf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>vwscanf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>WCHAR_MAX</b>	Macro	<stdint.h> <wchar.h>	<b>Vol. II - Seção 2.4.2</b> <b>Vol. II - Seção 8.5.2</b>
<b>WCHAR_MIN</b>	Macro	<stdint.h> <wchar.h>	<b>Vol. II - Seção 2.4.2</b> <b>Vol. II - Seção 8.5.2</b>
<b>wchar_t</b>	Tipo	<wchar.h> <stdlib.h> <stddef.h>	<b>Vol. II - Seção 8.5.1</b> <b>Vol. II - Seção 12.2.1</b> <b>Vol. II - Seção 12.3.1</b>
<b>wcrtomb()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.3</b>
<b>wscat()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcschr()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcscmp()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcscoll()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>

CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>wscspy()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wscspn()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcsftime()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcslen()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcsncat()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcsncmp()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcsncpy()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcsprbrk()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcsrchr()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcsrtombs()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.3</b>
<b>wcsspn()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wcsstr()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wctod()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.6</b>
<b>wctof()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.6</b>
<b>wctoimax()</b>	Função	<inttypes.h>	<b>Vol. II - Seção 2.5.3</b>
<b>wctok()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wctol()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.6</b>
<b>wctold()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.6</b>
<b>wctoll()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.6</b>
<b>wctombs()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 8.4.2</b>
<b>wctoul()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.6</b>
<b>wctoull()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.6</b>
<b>wctoumax()</b>	Função	<inttypes.h>	<b>Vol. II - Seção 2.5.3</b>
<b>wcsxfrm()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.5</b>
<b>wctob()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.3</b>
<b>wctomb()</b>	Função	<stdlib.h>	<b>Vol. II - Seção 8.4.2</b>
<b>wctrans()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.4</b>
<b>wctrans_t</b>	Tipo	<wctype.h>	<b>Vol. II - Seção 8.6.1</b>
<b>wctype()</b>	Função	<wctype.h>	<b>Vol. II - Seção 8.6.3</b>
<b>wctype_t</b>	Tipo	<wchar.h> <wctype.h>	<b>Vol. II - Seção 8.5.1</b> <b>Vol. II - Seção 8.6.1</b>



CONSTRUTOR	CATEGORIA	DEFINIÇÃO	REFERÊNCIA
<b>WEOF</b>	Macro	<wchar.h> <wctype.h>	<b>Vol. II - Seção 8.5.2</b> <b>Vol. II - Seção 8.6.2</b>
<b>while</b>	Palavra-chave	Linguagem	<b>Vol. I - Seção 1.7.3</b>
<b>WINT_MAX</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>WINT_MIN</b>	Macro	<stdint.h>	<b>Vol. II - Seção 2.4.2</b>
<b>wint_t</b>	Tipo	<wchar.h> <wctype.h>	<b>Vol. II - Seção 8.5.1</b> <b>Vol. II - Seção 8.6.1</b>
<b>wmemchr()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.4</b>
<b>wmemcmp()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.4</b>
<b>wmemcpy()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.4</b>
<b>wmemmove()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.4</b>
<b>wmemset()</b>	Função	<wchar.h>	<b>Vol. II - Seção 8.5.4</b>
<b>wprintf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>wscanf()</b>	Função	<wchar.h>	<b>Vol. II - Seção 10.8</b>
<b>xor</b>	Macro	<iso646.h>	<b>Vol. II - Seção 11.3</b>
<b>xor_eq</b>	Macro	<iso646.h>	<b>Vol. II - Seção 11.3</b>

## A.3 IDENTIFICADORES RESERVADOS PARA USO FUTURO

Conforme já foi antecipado, o programador não deve usar palavras-chave (porque é expressamente proibido) nem as palavras reservadas apresentadas na **Seção A.2** (para evitar colisão de identificadores). Além disso, há identificadores que não são enumerados na **Seção A.2**, mas que são reservados pela linguagem C para uso futuro.

Embora algumas restrições apresentadas aqui tenham brechas que permitem que identificadores sejam reusados<sup>145</sup>, a sobrecarga de identificadores pode causar confusão e é sempre mais seguro evitar completamente o uso de identificadores declarados nos cabeçalhos da biblioteca padrão. O mesmo conselho aplica-se a identificadores reservados pelo padrão ISO para uso futuro.

<sup>145</sup> Por exemplo, num arquivo de programa que não inclui <string.h>, pode-se definir uma função estática denominada `strcpy()`.

São reservados para uso futuro os identificadores apresentados a seguir:

- Todos os identificadores que comecem com sublinha, seguida por outra sublinha ou uma letra maiúscula. Exemplos: `__x` e `_Maximo` são reservados.
- Todos os identificadores com escopo de arquivo que comecem com sublinha. Portanto, não se deve usar esses identificadores como nome de função ou variável global.
- Os seguintes nomes de funções: `cerf()`, `cerfc()`, `cexp2()`, `cexpm1()`, `clog10()`, `cloglp()`, `clog2()`, `clgamma()` e `ctgamma()`. Estes mesmos nomes com os sufixos *f* e *l* também são reservados [e.g. `cerff()`, `cerfl()`].
- Todos os nomes que comecem com *is* ou *to* seguidos por uma letra minúscula.
- Todos os nomes que comecem com *E* seguido por um dígito ou uma letra maiúscula.
- Nomes de funções começando com *str*, *mem* ou *wcs* seguidos por uma letra minúscula.

## A.4 ESTATÍSTICAS DA LINGUAGEM C

A título de curiosidade, uma implementação da linguagem C que segue o padrão C99 deve possuir, pelo menos, 1036 construtores, discriminados nas seções a seguir.

### A.4.1 LINGUAGEM C

A tabela a seguir apresenta todos os construtores que fazem parte da linguagem C, excluindo-se aqueles que fazem parte da biblioteca padrão, e que foram estudados no **Volume I** desta obra<sup>146</sup>.

CATEGORIA	TOTAL
Palavras-chave	37
Diretivas de pré-processamento	13

<sup>146</sup> A palavra-chave **sizeof** também é operador.

CATEGORIA	TOTAL
Identificadores predefinidos	2
Macros definidas pela linguagem	10
Operadores de pré-processamento	3
Operadores da linguagem	45
Declaradores da linguagem	3
<b>GERAL</b>	<b>112</b>

## A.4.2 BIBLIOTECA PADRÃO DE C

A tabela a seguir apresenta todos os construtores que fazem parte da biblioteca padrão de C<sup>147</sup>. Todos esses construtores foram estudados no presente volume.

CATEGORIA	TOTAL
Funções	465
Macros	418
Tipos derivados	32
Variáveis globais	4
Rótulos de estruturas	2
Pragmas	3
<b>GERAL</b>	<b>924</b>

---

<sup>147</sup> Diretivas **#pragma** não fazem parte da biblioteca padrão, mas, sim, da própria linguagem. Estas diretivas foram incluídas aqui por conveniência.

# *Apêndice B*

---

*Especificadores de formato de entrada  
e saída*

## B.1 INTRODUÇÃO

Este apêndice apresenta especificadores de formato que compõem *strings* de formatação utilizados pelas famílias de funções `printf` e `scanf` da biblioteca padrão de C (v. **Capítulo 10**).

Em cada uma dessas famílias, as funções utilizam os mesmos especificadores de formato. Por exemplo, o especificador `%d` é usado por funções da família `printf` para escrita de valores do tipo **int** na base decimal e por funções da família `scanf` para leitura de valores desse mesmo tipo. Funções das duas famílias, que possuem o caractere *w* em seus nomes, utilizam *strings* de formatação extensos.

As famílias `printf` e `scanf` utilizam especificadores de formato semelhantes, mas, apesar disso, existem importantes diferenças entre os dois conjuntos de especificadores. Por exemplo, o especificador de formato `%i` tem interpretações diferentes em *strings* de formatação das famílias `printf` e `scanf`. A **Seção B.4** chama a atenção para estas diferenças, que, frequentemente, confundem os programadores de C.

Este apêndice contém, deliberadamente, muita informação redundante. Espera-se com isso permitir consultas sob diversas perspectivas. Entretanto, é necessária alguma cautela no uso do material, pois, apesar de algumas informações serem muito parecidas, elas não são redundantes.

### B.1.1 RECOMENDAÇÕES DE USO

Tipicamente, o material contido neste apêndice é usado para responder à questão:

*Que especificador de formato deve ser empregado para ler ou escrever um determinado valor de um certo tipo usando tal e tal formato?*

Para obter a resposta desejada a esta pergunta sugere-se que o programador siga a seguinte sequência de passos:

1. Identifique a família da função que será usada.
  - 1.1 Se a função for da família `printf` (escrita), a seção principal de interesse é **B.2**.
  - 1.2 Se a função for da família `scanf` (leitura), a seção principal de interesse é **B.3**.

2. Estude o formato geral de um especificador de formato da família determinada no passo anterior. Os dois formatos gerais são descritos na **Seção B.2.1** (família printf) e na **Seção B.3.1** (família scanf).
3. Determine o tipo de especificador baseado no tipo do valor que será escrito ou lido. Os tipos de especificadores são apresentados de modo resumido na **Seção B.2.5** (família printf) e na **Seção B.3.4** (família scanf).
4. Localize a descrição completa do tipo de especificador determinado no passo anterior. Essas descrições encontram-se ordenadas pelos símbolos que constituem os tipos de especificadores na **Seção B.2.7** (família printf) e na **Seção B.3.6** (família scanf).
5. Se necessário, selecione os demais componentes do especificador de formato.
6. Certifique-se de que estará usando o especificador de formato correto consultando os exemplos que acompanham a descrição completa de cada tipo de especificador.

## B.1.2 NOTAÇÃO

A seguinte notação é adotada em muitos exemplos apresentados neste apêndice:

- $\mathfrak{h}$  denota um dígito hexadecimal com letras, se for o caso, em minúsculas.
- $\mathfrak{H}$  denota um dígito hexadecimal com letras, se for o caso, em maiúsculas.
- $\mathfrak{b}$  denota espaço em branco.
- $\mathfrak{d}$  denota um dígito em base decimal.

Também, aquilo que se encontra entre colchetes e em *itálico* num exemplo denota um comentário acrescentado para facilitar o entendimento do exemplo.

## B.2 ESPECIFICADORES DE FORMATO DA FAMÍLIA PRINTF

### B.2.1 FORMATO GERAL

Especificadores de formato da família printf possuem o seguinte formato geral:

```
%[Sinalizador][Largura][.Precisão][Prefixo]Tipo
```

Neste formato, apenas o caractere % e o componente *Tipo* são obrigatórios. Os significados destes componentes serão apresentados nas seções a seguir.

### B.2.2 SINALIZADORES

O componente *Sinalizador* é representado por caracteres que podem alterar a forma como o resultado é escrito. Existem quatro valores possíveis para sinalizadores que são apresentados juntamente com suas respectivas descrições na **Tabela B-1**.

CARACTERE SINALIZADOR	DESCRIÇÃO
–	Força o alinhamento da escrita à esquerda dentro da largura especificada. Este sinalizador afeta todos os tipos de especificadores de formato, exceto <i>n</i> , que não permite nenhum sinalizador.
+	Valores numéricos com sinal serão prefixados com +, se forem positivos, ou – se forem negativos.
<i>Espaço em branco</i>	Valores numéricos com sinal serão prefixados com espaço em branco, se forem positivos, ou – se forem negativos.
#	Este sinalizador tem vários efeitos: <ul style="list-style-type: none"> <li>• Valores inteiros na base octal são escritos começando com zero.</li> <li>• Valores inteiros na base hexadecimal são escritos começando com 0x ou 0X.</li> <li>• Números de ponto flutuante são sempre escritos com ponto decimal e zeros são removidos ao final da parte fracionária.</li> </ul>

Tabela B-1: Sinalizadores usados em especificadores de formato da família printf.

### B.2.3 LARGURA

O componente *Largura* corresponde ao número mínimo de colunas (ou campos) que serão, a princípio, utilizadas na escrita de um valor; i.e., o número mínimo de caracteres que serão escritos. Este componente pode ser ignorado dependendo do tipo de especificador utilizado e da largura do resultado formatado. Ou seja, se a largura do resultado formatado for menor do que este valor, haverá preenchimento com espaços em branco à esquerda, mas se a largura do resultado formatado for maior do que este valor, este componente será ignorado e a largura será aquela determinada pelo resultado.

### B.2.4 PRECISÃO

O componente *Precisão* deve ser precedido por ponto e representa o número de casas decimais com que um número de ponto flutuante será escrito. Mas este componente pode ter outras interpretações dependendo do tipo de especificador utilizado.

### B.2.5 TIPO DE ESPECIFICADOR

O componente *Tipo* é o principal componente de um especificador de formato. A **Tabela B-2** apresenta, resumidamente, os tipos de especificadores permitidos e seus respectivos significados.

TIPO DE ESPECIFICADOR	UTILIZADOS NA ESCRITA DE...
a, A (C99)	Números de ponto flutuante em formato hexadecimal
c	Caracteres
d, i	Números inteiros com sinal na base decimal
e, E	Números de ponto flutuante em notação científica
f	Números de ponto flutuante em notação convencional
F (C99)	Números de ponto flutuante em notação convencional



TIPO DE ESPECIFICADOR	UTILIZADOS NA ESCRITA DE...
g, G	Números de ponto flutuante em notação convencional ou científica
n	Nada (v. significado na <b>Seção B.2.7</b> )
o	Números inteiros sem sinal na base octal
p	Endereços na base hexadecimal
s	<i>Strings</i>
u	Números inteiros sem sinal na base decimal
x, X	Números inteiros sem sinal na base hexadecimal
%	O caractere ' % '

Tabela B-2: Tipos de especificadores de formato usados pela família printf.

## B.2.6 PREFIXO

O componente *Prefixo* especifica qual deve ser o tipo do parâmetro correspondente ao especificador de formato. A **Tabela B-3** apresenta os prefixos e os respectivos tipos de especificadores com os quais cada um deles pode ser utilizado. Esta tabela apresenta na terceira coluna como são interpretados os tipos dos parâmetros correspondentes ao especificador de formato constituído pelos respectivos componentes nas duas primeiras linhas. Por exemplo, o especificador composto do prefixo h com o tipo de especificador d é %hd e, de acordo com a **Tabela B-3**, o parâmetro correspondente a este especificador é interpretado como **short**.

PREFIXO	PODE SER USADO COM...	INTERPRETAÇÃO
h	d, i, n, o, u, x, X	<ul style="list-style-type: none"> <li>• <b>short*</b>, no caso de n</li> <li>• <b>short</b>, nos outros casos</li> </ul>
hh (C99)	d, i, n, o, u, x, X	<ul style="list-style-type: none"> <li>• <b>char*</b>, no caso de n</li> <li>• <b>char</b>, nos outros casos</li> </ul>
j (C99)	d, i, n, o, u, x, X	<ul style="list-style-type: none"> <li>• <b>intmax_t*</b>, no caso de n</li> <li>• <b>intmax_t</b>, nos outros casos</li> </ul>

PREFIXO	PODE SER USADO COM...	INTERPRETAÇÃO
l	c, s, d, i, n, o, u, x, X	<ul style="list-style-type: none"> <li>• Usado com c: <b>wchar_t</b></li> <li>• Usado com s: <b>wchar_t*</b></li> <li>• Usado com d, i, o, u, x, X: <b>long</b></li> <li>• Usado com n: <b>long*</b></li> </ul>
L	a, A, e, E, f, F, g, G	<b>long double</b>
ll (C99)	d, i, n, o, u, x, X	<ul style="list-style-type: none"> <li>• <b>long long*</b>, no caso de n</li> <li>• <b>long long</b>, nos outros casos</li> </ul>
t (C99)	d, i, n, o, u, x, X	<ul style="list-style-type: none"> <li>• <b>ptrdiff_t*</b>, no caso de n</li> <li>• <b>ptrdiff_t</b>, nos outros casos</li> </ul>
z (C99)	d, i, n, o, u, x, X	<ul style="list-style-type: none"> <li>• <b>size_t*</b>, no caso de n</li> <li>• <b>size_t</b>, nos outros casos</li> </ul>

Tabela B-3: Prefixos usados em especificadores de formato da família printf.

Quando prefixos não são usados, os parâmetros correspondentes aos tipos de especificadores são interpretados como mostra a **Tabela B-4**.

TIPO DE ESPECIFICADOR	INTERPRETAÇÃO
c, d, i, o, u, x, X	<b>int</b>
a, A, e, E, f, F, g, G	<b>double</b>
S	<b>const char *</b>
P	<b>void *</b>
N	<b>int *</b>
%	Nenhuma

Tabela B-4: Tipos de especificadores de formato sem prefixos da família printf.

Valores de ponto flutuante indefinidos (v. **Capítulo 3**) são escritos como mostra a **Tabela B-5**.

TIPOS DE ESPECIFICADORES	VALOR	ESCRITO COMO...
a, e, f, g	$+\infty$	<b>inf</b>
A, E, F, G	$+\infty$	<b>INF</b>
a, e, f, g	$-\infty$	<b>-inf</b>

TIPOS DE ESPECIFICADORES	VALOR	ESCRITO COMO...
A, E, F, G	$-\infty$	<code>–INF</code>
a, e, f, g	NaN	<code>nan</code>
A, E, F, G	NaN	<code>NAN</code>

Tabela B-5: Escrita de valores de ponto flutuante indefinidos.

## B.2.7 COMPOSIÇÕES DE ESPECIFICADORES DE FORMATO

Nesta seção, os tipos de especificadores serão descritos mais precisamente. Juntamente com cada tipo de especificador serão apresentados os outros componentes que podem ser utilizados com eles.

### *a* e *A* (C99)

Os tipos de especificadores *a* e *A* são usados para escrever números de ponto flutuante em formato hexadecimal (v. **Capítulo 1** do **Volume I**). Quando *a* é utilizado, o número é escrito no seguinte formato:

$$\pm 0xh.hhhp\pm dd$$

Quando *A* é utilizado, o número é escrito no formato:

$$\pm 0XH.HHHp\pm dd$$

Valores indefinidos são escritos como mostra a **Tabela B-5**.

Os demais componentes do formato geral de especificadores de formato da família `printf` afetam especificadores de formato com estes tipos da seguinte maneira:

- *Sinalizador*: Qualquer sinalizador é permitido e tem o efeito descrito na **Tabela B-1**.
- *Largura*: Conforme descrito na **Seção B.2.3**.
- *.Precisão*: Número de casas decimais com que o número de ponto flutuante será escrito.
- *Prefixo*:
  - Nenhum – o número será escrito como **double**.

- L – o número será escrito como **long double**.

**Observação:** Não há especificador para **float**; i.e., quando um valor a ser escrito é deste tipo, ele é convertido em **double**.

### Exemplos:

Instrução	Escreve...
<code>printf("%a", 30.0);</code>	<code>0xftp+1</code>
<code>printf("%.2A", 30.0);</code>	<code>0XF.00P+1</code>

### c

O tipo de especificador `c` causa a impressão de um único caractere. Os demais componentes do formato geral de especificador produzem o seguinte efeito:

- *Sinalizador*: Apenas o sinalizador – (alinhamento à esquerda) faz sentido.
- *Largura*: Se este valor for maior do que um, o caractere poderá ser alinhado à direita (padrão) ou à esquerda (utilizando o sinalizador –).
- *Precisão*: Ignorado.
- *Prefixo*:
  - l – o caractere será escrito como um caractere extenso se o *string* de formatação for um *string* extenso; caso contrário, o caractere será escrito como um caractere normal.

### Exemplos:

Instrução	Escreve...
<code>printf("%-2c", 'u');</code>	<code>u</code>
<code>printf("%2c", 'u');</code>	<code>bu</code>
<code>printf("%lc", L'u');</code>	<code>u [normal]</code>
<code>printf(L"%lc", L'u');</code>	<code>u [extenso]</code>
<code>wprintf(L"%c", 'u');</code>	<code>u [extenso]</code>

Instrução	Escreve...
<code>printf("&lt;%2c %-2c&gt;\n", 'x', 'y');</code>	<code>&lt; x y &gt;</code>

## *d e i*

Os tipos de especificadores *d* e *i* escrevem números inteiros com sinal na base decimal. Eis como os demais componentes do formato geral de especificadores afetam a escrita quando combinados com estes dois tipos de especificadores:

- *Sinalizador*: qualquer sinalizador é permitido e tem o efeito descrito na **Tabela B-1**.
- *Largura*: Conforme descrito na **Seção B.2.3**.
- *Precisão*: Especifica o número mínimo de dígitos que serão escritos. Zeros podem ser acrescentados à esquerda para completar o número de dígitos especificado.
- *Prefixo*: Quando os tipos de especificadores *d* e *i* são combinados com os prefixos permitidos obtêm-se as interpretações apresentadas na **Tabela B-6**.

PREFIXO	O NÚMERO SERÁ ESCRITO COMO...
Nenhum	<b>int</b>
h	<b>short</b>
hh	<b>char</b>
l	<b>long</b>
ll	<b>long long</b>

Tabela B-6: Prefixos usados com *d* e *i* (família `printf`).

**Observação:** Não há diferença entre usar *d* ou *i* em especificadores de formato de funções da família `printf`.

**Exemplos:**

Instrução	Escreve...
<code>printf("%d", 25);</code>	25
<code>printf("%.4i", -25);</code>	-0025
<code>printf("%4.2ld", -25L);</code>	b-25
<code>printf(" %4.2zi\n", (size_t)25);</code>	bbb25

*e, E*

Os tipos de especificadores *e* e *E* escrevem números de ponto flutuante em notação científica. Quando *e* é utilizado, um número é escrito no seguinte formato:

$$\pm d.ddd\pm dd$$

Quando *E* é utilizado, um número de ponto flutuante é escrito como:

$$\pm d.dddE\pm dd$$

Valores indefinidos são escritos de acordo com a **Tabela B-5**.

Os demais componentes do formato geral de especificador afetam especificadores com estes tipos da seguinte maneira:

- *Sinalizador*: qualquer sinalizador é permitido e tem o efeito descrito na **Tabela B-1**.
- *Largura*: Conforme descrito na **Seção B.2.3**.
- *Precisão*: Especifica o número casas decimais.
- *Prefixo*:
  - Nenhum – o número será escrito como **double**.
  - L – o número será escrito como **long double**.

**Observação:** Não há especificador para **float**; i.e., quando um valor a ser escrito é deste tipo, ele é convertido em **double**.

**Exemplos:**

<b>Instrução</b>	<b>Escreve...</b>
<code>printf("%e\n", 1.44);</code>	1.440000e+000
<code>printf("%E\n", 1.44);</code>	1.440000E+000
<code>printf("%-6.2e\n", 1.44);</code>	1.44e+000
<code>printf("%+6.2e\n", 1.44);</code>	+1.44e+000
<code>printf("%-6.2e\n", 1.44);</code>	1.44e+000
<code>printf("%-6.2e\n", -1.44);</code>	-1.44e+000
<code>printf("% 6.2e\n", 1.44);</code>	b1.44e+000
<code>printf("% 6.2e\n", -1.44);</code>	-1.44e+000
<code>printf("%#e\n", 1.44);</code>	1.440000e+000
<code>printf("%#e\n", -1.44);</code>	-1.440000e+000

*f*

O tipo de especificador `f` imprime números de ponto flutuante em formato tradicional. Valores indefinidos são escritos de acordo com a **Tabela B-5**. Os demais componentes do formato geral de especificador afetam especificadores que usam este tipo da seguinte maneira:

- *Sinalizador*: Qualquer sinalizador é permitido (v. **Tabela B-1**).
- *Largura*: Conforme descrito na **Seção B.2.3**.
- *Precisão*: Especifica o número de casas decimais que serão escritas.
- *Prefixo*:
  - Nenhum – o número será escrito como **double**.
  - L – o número será escrito como **long double**.

**Observação:** Não há especificador para **float**; i.e., quando um valor a ser escrito é deste tipo, ele é convertido em **double**.

**Exemplos:**

Instrução	Escreve...
<code>printf("%f\n", 1.44);</code>	1.440000
<code>printf("%F\n", 1.44);</code>	1.440000
<code>printf("%-6.2f\n", 1.44);</code>	1.44
<code>printf("%+6.2f\n", 1.44);</code>	<b>b</b> +1.44
<code>printf("%-6.2f\n", 1.44);</code>	1.44
<code>printf("%-6.2f\n", -1.44);</code>	-1.44
<code>printf("% 6.2f\n", 1.44);</code>	<b>b</b> b1.44
<code>printf("% 6.2f\n", -1.44);</code>	<b>b</b> -1.44
<code>printf("%#f\n", 1.44);</code>	1.440000
<code>printf("%#f\n", -1.44);</code>	-1.440000

*F (C99)*

A única diferença entre `f` e `F` é aquela apresentada na **Tabela B-5**. Em todos os outros aspectos, o tipo de especificador `F` tem exatamente o mesmo significado que `f`. O tipo de especificador `F` foi introduzido pelo padrão C99 e, por isso, nem todos os compiladores o reconhecem.

**Exemplos:**

Instrução	Escreve...
<code>printf("sqrt(-2) = %f\n", sqrt(-2));</code>	nan
<code>printf("sqrt(-2) = %F\n", sqrt(-2));</code>	NAN
<code>printf("\n1.0/0.0 = %f\n", 1.0/0.0);</code>	inf
<code>printf("\n1.0/0.0 = %F\n", 1.0/0.0);</code>	INF

*g e G*

Dependendo da precisão e do próprio valor a ser impresso, o tipo de especificador `g` funciona como `f` ou `e`. De modo semelhante, `G` corresponde ao uso de `F` ou `E`. Isto é, esses tipos de especificadores causam a escrita de um valor de ponto flutuante em notação convencional ou científica, dependendo de qual delas é considerada mais



compacta na presente situação. Os critérios para esta tomada de decisão são complicados demais para terem utilidade prática. Portanto, recomenda-se ao leitor que tente entender estes critérios comparando os exemplos apresentados a seguir, bem como fazendo seus próprios experimentos.

### Exemplos:

Instrução	Escreve...
<code>printf("%g\n", 1.44);</code>	1.44
<code>printf("%G\n", 1.44);</code>	1.44
<code>printf("%g\n", 0.00000144);</code>	1.44e-006
<code>printf("%6.2LG\n", 144000.0);</code>	1.4E+005
<code>printf("%-6.2g\n", 1.44);</code>	1.4
<code>printf("%+6.2g\n", 1.44);</code>	<del>bb</del> +1.4
<code>printf("%-6.2g\n", 1.44);</code>	1.4
<code>printf("%-6.2G\n", -1440.0);</code>	-1.4E+003
<code>printf("% 6.2g\n", 1.44);</code>	<del>bbb</del> 1.4
<code>printf("% 6.2g\n", -1.44);</code>	<del>bb</del> -1.4
<code>printf("%#g\n", 1.44);</code>	1.44000
<code>printf("%#G\n", -1.44);</code>	-1.44000

*n*

Diferentemente dos demais tipos de especificadores utilizados na família `printf`, `n` não causa escrita. Isto é, o tipo de especificador `n` armazena o número de caracteres escritos até então no endereço correspondente passado como parâmetro. O único componente adicional admitido na composição de um especificador de formato contendo `n` é um prefixo que pode ser um daqueles apresentados na **Tabela B-7**.

PREFIXO	O PARÂMETRO CORRESPONDENTE É INTERPRETADO COMO...
Nenhum	<b>int</b> *
h	<b>short</b> *
hh	<b>char</b> *
j	<b>intmax_t</b> *
l	<b>long</b> *

PREFIXO	O PARÂMETRO CORRESPONDENTE É INTERPRETADO COMO...
ll	<b>long long *</b>
t	<b>ptrdiff_t *</b>
z	<b>size_t *</b>

Tabela B-7: Prefixos usados com n (família printf).

**Observação:** O tipo de especificador n é semelhante a um tipo de especificador da família scanf e deve ter um endereço de variável correspondente onde o resultado será armazenado.

### Exemplo:

```
#include <stdio.h>

int main(void)
{
    int nColunasImpressas;

    /* Exemplos de %n */
    printf( "String com 24 caracteres%n\n",
           &nColunasImpressas );
    printf( "Numero de caracteres impressos: %d\n",
           nColunasImpressas );

    return 0;
}
```

Quando o programa do último exemplo é executado, é escrito o seguinte no meio de saída padrão:

*Numero de caracteres escritos: 24*

*o*

O tipo de especificador o serve para escrever números inteiros sem sinal em formato octal. Os demais componentes do formato geral de especificador afetam especificadores de formato contendo este tipo da seguinte maneira:

- *Sinalizador*: Qualquer sinalizador é permitido (v **Tabela B-1**).
- *Largura*: Conforme descrito na **Seção B.2.3**.
- *Precisão*: Especifica o número mínimo de dígitos a serem escritos. Zeros podem ser acrescentados à esquerda para completar o número de dígitos especificado.
- *Prefixo*: Quando o tipo de especificador `o` é combinado com os prefixos permitidos, obtêm-se as interpretações apresentadas na **Tabela B-8**.

PREFIXO	O NÚMERO SERÁ ESCRITO COMO...
Nenhum	<b>unsigned int</b>
h	<b>unsigned short</b>
hh	<b>unsigned char</b>
l	<b>unsigned long</b>
ll	<b>unsigned long long</b>

Tabela B-8: Prefixos usados com `o` (família `printf`).**Exemplos:**

Instrução	Escreve...
<code>printf("%o\n", 38);</code>	46
<code>printf("%-lo\n", 38L);</code>	46
<code>printf("%#o\n", 38);</code>	046
<code>printf("%6o\n", 38);</code>	bbb46

*p*

O tipo de especificador `p` é usado para escrever o valor armazenado num ponteiro do tipo **void \*** em formato hexadecimal. Os demais componentes do formato geral afetam especificadores que usam este tipo da seguinte maneira:

- *Sinalizador*: Qualquer sinalizador é permitido (v. **Tabela B-1**).
- *Largura*: Conforme descrito na **Seção B.2.3**.

- *Precisão*: Especifica o número mínimo de dígitos a serem escritos. Zeros podem ser acrescentados à esquerda para completar o número de dígitos especificado.
- *Prefixo*: Nenhum é permitido.

**Observação:** O resultado obtido com este tipo de especificador é dependente de implementação.

**Exemplo:** No exemplo a seguir, suponha que *x* seja uma variável do tipo **int**<sup>148</sup>:

Instrução	Escreve...
<code>printf("%p\n", &amp;x);</code>	0022ff74

*s*

O tipo de especificador *s* serve para escrever *strings*. Os demais componentes do formato geral de especificador afetam especificadores contendo este tipo da seguinte maneira:

- *Sinalizador*: Os únicos sinalizadores descritos na **Tabela B-1** que não têm efeito são *+* e *espaço em branco*.
- *Largura*: Especifica o número mínimo de caracteres que serão escritos. Se o comprimento do *string* for maior do que este valor e não for especificada nenhuma precisão, o escrito expande-se até o tamanho do *string*.
- *Precisão*: Especifica o número máximo de caracteres que serão escritos. Se o *string* a ser impresso tiver tamanho maior do que a especificação de largura, ele será truncado.
- *Prefixo*:

- *l* – o *string* será considerado extenso.

---

<sup>148</sup> O que a função **printf()** imprime é o endereço em memória onde a variável *x* é alocada. Portanto, se você executar um programa contendo as mesmas instruções dificilmente obterá o mesmo resultado apresentado aqui.

**Exemplos:**

Instrução	Escreve...
<code>printf("%s\n", "Bola");</code>	Bola
<code>printf("%.2s\n", "Bola");</code>	Bo
<code>wprintf(L"%s\n", "Bola");</code>	Bola (extenso)
<code>printf("%ls\n", L"Bola");</code>	Bola
<code>wprintf(L"%s\n", L"Bola");</code>	Bol (extenso)

***u***

O tipo de especificador **u** é usado para escrever números inteiros sem sinal na base decimal. Os demais componentes do formato geral de especificador afetam especificadores com este tipo da seguinte maneira:

- *Sinalizador*: Qualquer sinalizador é permitido (v. **Tabela B-1**).
- *Largura*: Conforme descrito na **Seção B.2.3**.
- *Precisão*: Especifica o número mínimo de dígitos a serem escritos. Zeros podem ser acrescentados à esquerda para completar o número de dígitos especificado.
- *Prefixo*: Quando o tipo de especificador **u** é combinado com os prefixos permitidos, obtêm-se as interpretações apresentadas na **Tabela B-9**.

PREFIXO	O NÚMERO SERÁ ESCRITO COMO...
Nenhum	<b>unsigned int</b>
h	<b>unsigned short</b>
hh	<b>unsigned char</b>
l	<b>unsigned long</b>
ll	<b>unsigned long long</b>

Tabela B-9: Prefixos usados com **u** (família printf).

**Exemplos:**

Instrução	Escreve...
<code>printf("%a", 30.0);</code>	0xfp+1
<code>printf("%.2A", 30.0);</code>	0XF.00P+1

*x e X*

Os tipos de especificadores *x* e *X* escrevem números inteiros sem sinal em formato hexadecimal. Se *x* for utilizado, as letras que compõem o número hexadecimal serão escritas em minúsculas; se *X* for utilizado, essas letras serão escritas em maiúsculas. Os demais componentes do formato geral afetam especificadores que usam estes tipos da seguinte maneira:

- *Sinalizador*: Qualquer sinalizador é permitido (v. **Tabela B-1**).
- *Largura*: Conforme descrito na **Seção B.2.3**.
- *Precisão*: Especifica o número mínimo de dígitos a serem escritos. Zeros podem ser acrescentados à esquerda para completar o número de dígitos especificado.
- *Prefixo*: Quando *x* e *X* são combinados com os prefixos permitidos, obtêm-se as interpretações apresentadas na **Tabela B-10**.

PREFIXO	O NÚMERO SERÁ ESCRITO COMO...
Nenhum	<b>unsigned int</b>
h	<b>unsigned short</b>
hh	<b>unsigned char</b>
l	<b>unsigned long</b>
ll	<b>unsigned long long</b>

Tabela B-10: Prefixos usados com *x* e *X* (família printf).**Exemplos:**

Instrução	Escreve...
<code>printf("%x\n", 9204);</code>	23f4

Instrução	Escreve...
<code>printf("%-lX\n", 9204L);</code>	23F4
<code>printf("%#x\n", 9204);</code>	0x23f4
<code>printf("%6x\n", 9204);</code>	bb23f4

%

O tipo de especificador % é utilizado na escrita do caractere '%' e não utiliza nenhum outro componente do formato geral de especificadores.

### Exemplo:

```
#include <stdio.h>

int main(void)
{
    /* Quanto símbolos % serão impressos? */
    /* Resposta: a metade! */
    printf("%%%%%%%%");

    return 0;
}
```

## B.2.8 EXEMPLOS ADICIONAIS

**Exemplo:** A seguinte chamada de `printf()` demonstra o efeito do componente *.Precisão*, especificado como um ponto seguido de um inteiro.

```
printf("|%5.4d| |%-5.4s| |%5.0f|\n", 123, "um string",
      45.6);
```

Esta última chamada produz como saída:

```
| 0123| |um s | | 46|
```

Quando utilizada na impressão de inteiros, a precisão indica o número mínimo de dígitos que será apresentado. Isto significa que o número pode ser precedido por zeros, se for necessário. Com *strings*, a precisão especifica o número máximo de ca-

racteres do *strings* que será apresentado<sup>149</sup>. No caso, de números de ponto flutuante, a precisão especifica o número de casas decimais; i.e., o número é arredondado para satisfazer o número de casas decimais especificado. Ainda neste caso, quando o número de casas decimais for zero, o número será arredondado para o inteiro mais próximo e, conseqüentemente, será apresentado sem ponto decimal<sup>150</sup>.

**Exemplo:** A chamada de `printf()` a seguir:

```
printf("|%+5d| |%+5d| |% 5d| |% 5d|\n", 123,
      -123, 123, -123);
```

produz como resultado:

```
| +123| | -123| | 123| | -123|
```

Neste último exemplo, o sinalizador `+` faz com que um sinal sempre seja apresentado. Por outro lado, o uso de espaço como sinalizador faz com que o sinal positivo seja substituído por espaço em branco.

**Exemplo:** A chamada de `printf()` a seguir:

```
printf("|%#5o| |%#5x| |%#5.0f|\n", 15, 15, 15.0);
```

produz como saída:

```
| 017| | 0xf| | 15.0|
```

O uso do sinalizador `#` neste último exemplo faz com que cada número seja apresentado usando a notação de constantes da linguagem C; i.e., começando com zero se o número for octal, começando com `0x` se o número for hexadecimal, etc. (v. **Capítulo 1 do Volume I**). Neste caso, todos os algarismos significativos serão apresentados, mesmo se o tamanho do campo for excedido.

**Exemplo:** O seguinte trecho de programa ilustra o uso de asterisco como especificador de formato:

```
#define LARGURA 80
. . .
printf("%*.*s\n", LARGURA, LARGURA, string);
```

<sup>149</sup> Note neste exemplo que o string foi alinhado à esquerda dentro do seu campo devido ao uso do sinalizador `-`.

<sup>150</sup> Se for desejada a apresentação de ponto decimal, mas sem casas decimais, utiliza-se o sinalizador `#`.



Quando esta última chamada de **printf()** é executada, o *string* recebido como argumento é alinhado à direita no meio de saída padrão.

## B.2.9 RESUMO DE ESPECIFICADORES DE FORMATO

Para facilidade de referência, a **Tabela B-11** a seguir apresenta tipos de especificadores juntamente com seus prefixos. Nesta tabela, as colunas devem ser interpretadas do seguinte modo:

- **Tipo de valor** – como o parâmetro correspondente ao especificador é interpretado<sup>151</sup>.
- **Especificador** – um tipo de especificador precedido por um prefixo; todas as combinações possíveis de tipos de especificadores e prefixos são apresentadas na tabela.
- **Conversão** – a conversão que é aplicada ao parâmetro antes de ele ser impresso.
- **Base** – a base numérica na qual os valores numéricos são escritos; não faz sentido para valores não numéricos.

TIPO DE VALOR	ESPECIFICADOR	CONVERSÃO	BASE
<code>const char *</code>	<code>%s</code>	<code>(const char *)</code>	
<code>const wchar_t *</code>	<code>%ls</code>	<code>(const wchar_t *)</code>	
<code>double</code>	<code>%a (C99)</code>	<code>(double)</code>	10
<code>double</code>	<code>%A (C99)</code>	<code>(double)</code>	10
<code>double</code>	<code>%e</code>	<code>(double)</code>	10
<code>double</code>	<code>%E</code>	<code>(double)</code>	10
<code>double</code>	<code>%f</code>	<code>(double)</code>	10
<code>double</code>	<code>%F (C99)</code>	<code>(double)</code>	10
<code>double</code>	<code>%g</code>	<code>(double)</code>	10
<code>double</code>	<code>%G</code>	<code>(double)</code>	10
<code>int *</code>	<code>%n</code>		

<sup>151</sup> A única exceção é o tipo de especificador `n`, que, conforme foi visto anteriormente, não é usado em escrita.

TIPO DE VALOR	ESPECIFICADOR	CONVERSÃO	BASE
int *	%hhn (C99)		
int	%c	(unsigned char)	
int	%d	(int)	10
int	%hd	(short)	10
int	%hhd (C99)	(signed char)	10
int	%i	(int)	10
int	%hi	(short)	10
int	%hhi (C99)	(signed char)	10
int	%o	(unsigned int)	8
int	%ho	(unsigned short)	8
int	%hho (C99)	(unsigned char)	8
int	%u	(unsigned int)	10
int	%hu	(unsigned short)	10
int	%hhu (C99)	(unsigned char)	10
int	%x	(unsigned int)	16
int	%hx	(unsigned short)	16
int	%hhx (C99)	(unsigned char)	16
int	%X	(unsigned int)	16
int	%hX	(unsigned short)	16
int	%hhX (C99)	(unsigned char)	16
intmax_t *	%jn (C99)		
intmax_t	%jd (C99)	(intmax_t)	10
intmax_t	%ji (C99)	(intmax_t)	10
intmax_t	%jo (C99)	(uintmax_t)	8
intmax_t	%ju (C99)	(uintmax_t)	10
intmax_t	%jx (C99)	(uintmax_t)	16
intmax_t	%jX (C99)	(uintmax_t)	16
long *	%ln		
long double	%La (C99)	(long double)	10
long double	%LA (C99)	(long double)	10
long double	%Le	(long double)	10
long double	%LE	(long double)	10
long double	%Lf	(long double)	10

TIPO DE VALOR	ESPECIFICADOR	CONVERSÃO	BASE
long double	%LF (C99)	(long double)	10
long double	%Lg	(long double)	10
long double	%LG	(long double)	10
long long *	%lln (C99)		
long long	%lld (C99)	(long long)	10
long long	%lli (C99)	(long long)	10
long long	%llo (C99)	(unsigned long)	8
long long	%llu (C99)	(unsigned long)	10
long long	%llx (C99)	(unsigned long)	16
long long	%lX (C99)	(unsigned long)	16
long	%ld	(long)	10
long	%li	(long)	10
long	%lo	(unsigned long)	8
long	%lu	(unsigned long)	10
long	%lx	(unsigned long)	16
long	%lX	(unsigned long)	16
Nenhum	%%	(char) '%'	
ptrdiff_t *	%tn (C99)		
ptrdiff_t	%td (C99)	(ptrdiff_t)	10
ptrdiff_t	%ti (C99)	(ptrdiff_t)	10
ptrdiff_t	%to (C99)	(size_t)	8
ptrdiff_t	%tu (C99)	(size_t)	10
ptrdiff_t	%tx (C99)	(size_t)	16
ptrdiff_t	%tX (C99)	(size_t)	16
short *	%hn		
size_t *	%zn (C99)		
size_t	%zd (C99)	(ptrdiff_t)	10
size_t	%zi (C99)	(ptrdiff_t)	10
size_t	%zo (C99)	(size_t)	8
size_t	%zu (C99)	(size_t)	10
size_t	%zx (C99)	(size_t)	16
size_t	%zX (C99)	(size_t)	16

TIPO DE VALOR	ESPECIFICADOR	CONVERSÃO	BASE
void *	%p	(void *)	
wint_t	%lc	(wchar_t)	

Tabela B-11: Tipos e prefixos permitidos em especificadores de formato da família printf.

## B.3 ESPECIFICADORES DE FORMATO DA FAMÍLIA SCANF

Os membros da família de funções scanf usam especificadores de formato que descrevem qual é o formato esperado de dados que deverão ser lidos. Como essas funções utilizam listas de argumentos variáveis, elas não teriam meios para saber quais são os tipos dos demais parâmetros se não existissem os especificadores de formato.

A maioria dos exemplos apresentados nesta seção usa a função **sscanf()** (v. **Seção 10.7.8**), pois, devido à praticidade, esta função provê a melhor maneira de testar especificadores da família scanf. Quer dizer, como **sscanf()** lê dados num *string*, não há preocupação em descrever como os dados devem ser organizados num arquivo ou introduzidos por um usuário pela entrada padrão.

### B.3.1 FORMATO GERAL

Especificadores de formato da família scanf possuem o seguinte formato geral:

<code>%[*][Largura][Prefixo]Tipo</code>
---

Do mesmo modo que ocorre com os especificadores de formato da família printf, apenas o caractere % e o componente *Tipo* são obrigatórios. Nas seções a seguir, os significados destes componentes serão apresentados.

### B.3.2 ASTERISCO

Quando asterisco é utilizado, o valor lido correspondente é descartado. Considere o seguinte programa como exemplo:

```
#include <stdio.h>

int main(void)
{
    int x = 0, y = 0;

    sscanf("-2 25", "%*d %d", &x, &y);
    printf("\nx = %d \t y = %d", x, y);

    sscanf("-2 25", "%d %*d", &x, &y);
    printf("\nx = %d \t y = %d", x, y);

    return 0;
}
```

O programa imprime o seguinte no meio de entrada padrão:

```
x = 25    y = 0
x = -2    y = 0
```

Para entender o resultado apresentado por esse programa, note que os caracteres que compõem a entrada de dados são os mesmos nas duas chamadas de **sscanf()**. Observe ainda que, nas duas chamadas de **sscanf()**, nenhum valor é atribuído à variável *y*. Isto ocorre porque, nas duas situações, solicita-se à função **sscanf()** que leia os dois valores, mas descarte o primeiro valor lido. Portanto, como só existe um valor a ser atribuído, ele será atribuído à primeira variável que, no caso, é *x*. A variável recebe valores diferentes em cada caso porque o asterisco causa o descarte ora do primeiro valor, ora do segundo valor.

É importante observar que, quando faz parte de um especificador de formato, o asterisco representa a leitura de um valor do tipo especificado que não é atribuído a nenhuma variável. Portanto, se o valor correspondente encontrado na entrada não puder ser convertido no tipo especificado, a função da família *scanf* irá falhar do mesmo modo que faria se o valor não tivesse que ser desprezado. Por exemplo, o trecho de programa a seguir:

```
int x = 0, y = 0;

sscanf("z 25", "%*d %d", &x, &y);
printf("\nx = %d \t y = %d", x, y);
```

causaria a impressão de:

```
x = 0      y = 0
```

O resultado apresentado por esse trecho de programa indica que o primeiro valor que deveria ser lido como um inteiro e, em seguida, descartado não pode ser convertido. Portanto, a função **sscanf()** encerra a leitura sem armazenar nenhum valor.

### B.3.3 LARGURA

A largura especifica o número máximo de caracteres que são levados em consideração durante a leitura. O número de caracteres realmente consumidos pode ser menor do que este valor, mas jamais será maior.

### B.3.4 TIPO DE ESPECIFICADOR

A **Tabela B-12** apresenta, resumidamente, os tipos de especificadores permitidos e seus respectivos significados.

TIPO DE ESPECIFICADOR	UTILIZADO NA LEITURA DE...
a, A (C99)	Números de ponto flutuante
c	Caracteres
d	Números inteiros com sinal na base decimal
e, E (Alteração/C99)	Números de ponto flutuante
f (Alteração/C99), F (C99)	Números de ponto flutuante
g, G (Alteração/C99)	Números de ponto flutuante
i	Números inteiros com sinal em qualquer base
n	Nada (v. significado na <b>Seção B 3.6</b> )
o	Números inteiros sem sinal na base octal
p	Ponteiro na base hexadecimal
s	<i>String</i>
u	Números inteiros sem sinal na base decimal
x, X	Números inteiros sem sinal na base hexadecimal
[ <i>caracteres</i> ]	<i>String</i>

TIPO DE ESPECIFICADOR	UTILIZADO NA LEITURA DE...
%	O caractere ' % '

Tabela B-12: Tipos de especificadores de formato usados pela família scanf.

### B.3.5 PREFIXO

O componente *Prefixo* especifica qual deve ser o tipo da variável que receberá o valor lido e convertido. A **Tabela B-13** mostra os prefixos que podem ser utilizados. Esta tabela apresenta ainda os respectivos tipos com os quais cada prefixo pode ser usado e qual é a interpretação em cada caso.

PREFIXO	PODE SER USADO COM...	TIPO DA VARIÁVEL
h	d, i, n, o, u, x, X	<b>short</b>
hh (C99)	d, i, n, o, u, x, X	<b>char</b>
j (C99)	d, i, n, o, u, x, X	<b>intmax_t</b>
l	c, s, d, i, n, o, u, x, X, a, A, e, E, f, F, g, G	<ul style="list-style-type: none"> <li>• Usado com c: <b>wchar_t</b></li> <li>• Usado com s: <b>wchar_t*</b></li> <li>• Usado com d, i, n, o, u, x, X: <b>long</b></li> <li>• Usado com a, A, e, E, f, F, g, G: <b>double</b></li> </ul>
L	a, A, e, E, f, F, g, G	<b>long double</b>
ll (C99)	d, i, n, o, u, x, X	<b>long long</b>
t (C99)	d, i, n, o, u, x, X	<b>ptrdiff_t</b>
z (C99)	d, i, n, o, u, x, X	<b>size_t</b>

Tabela B-13: Prefixos usados em especificadores de formato da família scanf.

### B.3.6 COMPOSIÇÕES DE ESPECIFICADORES DE FORMATO

Nesta seção, os tipos de especificadores serão descritos em detalhes. Juntamente com cada especificador serão apresentados ainda os outros componentes que podem acompanhá-lo.

*a e A (C99)*

Os tipos de especificadores *a* e *A* são utilizados na leitura de valores de ponto flutuante e não há diferença entre *a* e *A*. Esses tipos de especificadores aceitam os prefixos mostrados na **Tabela B-14**.

PREFIXO	OS CARACTERES LIDOS SÃO CONVERTIDOS EM...	FUNÇÃO DE CONVERSÃO
Nenhum	<b>float</b>	<b>strttof()</b>
<b>l</b>	<b>double</b>	<b>strttd()</b>
<b>L</b>	<b>long double</b>	<b>strtold()</b>

Tabela B-14: Prefixos e conversões usados com *a* e *A* (família *scanf*).

O valor lido no meio de entrada pode incluir:

- Sinal
- Ponto
- *e* ou *E* (notação científica)
- *+INF* ou *-INF*, representando, respectivamente, *+∞* ou *-∞*
- *+NAN* ou *-NAN* (representando um valor não numérico)
- *0x* ou *0X* mais uma sequência de dígitos hexadecimais (C99)

**Exemplos:**

```
double x;
```

Instrução	Armazena...
<code>sscanf("254E-2", "%a", &amp;x);</code>	2.54 em <i>x</i>

*c*

O tipo de especificador *c* é utilizado na leitura de um (se a largura não for especificada) ou mais caracteres (se a largura for maior do que um). Nenhum espaço em branco inicial é saltado.

Na ausência do prefixo *l*, o respectivo argumento deve ser o endereço de um array de caracteres com capacidade suficiente para conter os caracteres lidos. Se o *stream*



onde a leitura é realizada tiver orientação extensa, os caracteres extensos lidos serão convertidos em caracteres multibytes por meio de chamadas da função **wcrtomb()**, começando no estado inicial de conversão (v. **Capítulo 10**). A função de leitura não acrescenta nenhum caractere nulo; i.e., o resultado da leitura não será um *string*.

Se o prefixo **l** estiver presente e o *stream* onde a leitura é realizada tiver orientação por byte, os dados de entrada serão interpretados como caracteres multibytes começando no estado inicial de mudança (v. **Seção 8.2**). Neste caso, cada caractere multibyte será convertido em caractere extenso por meio de chamadas da função **mbrtowc()**. O respectivo argumento deve ser o endereço de um array de caracteres extensos com capacidade suficiente para conter os caracteres extensos resultantes das operações de leitura e conversão. Se o *stream* onde a leitura é realizada tiver orientação extensa, simplesmente não haverá conversão. Novamente, a função de leitura não acrescenta nenhum caractere nulo; i.e., o resultado da leitura não será um *string*.

### Exemplos:

```
char    c;
char    ar[5];
wchar_t ce;
wchar_t arExt[5];
```

Instrução	Armazena...
<code>sscanf("abc", "%c", &amp;c);</code>	'a' em c
<code>sscanf("abc", "%2c", ar);</code>	'a' e 'b' em ar[]
<code>swscanf(L"abc", L"%c", &amp;c);</code>	'a' em c
<code>sscanf("129E-2", "%lc", &amp;ce);</code>	L'a' em ce
<code>sscanf("abc", "%2lc", arExt);</code>	L'a' e L'b' em arExt[]
<code>swscanf(L"abc", L"%lc", &amp;ce);</code>	L'a' em ce

### d

O tipo de especificador **d** é utilizado na leitura de inteiros em base decimal e aceita os prefixos mostrados na **Tabela B-15**.

PREFIXO	OS CARACTERES LIDOS SÃO CONVERTIDOS EM...	FUNÇÃO DE CONVERSÃO <sup>152</sup>
Nenhum	<b>int</b>	<b>strtol()</b>
h	<b>short</b>	<b>strtol()</b>
hh	<b>char</b>	<b>strtol()</b>
j (C99)	<b>intmax_t</b>	<b>strtoumax()</b>
l	<b>long</b>	<b>strtol()</b>
ll (C99)	<b>long long</b>	<b>strtoll()</b>
t (C99)	<b>ptrdiff_t</b>	<b>strtoumax()</b>
z (C99)	<b>size_t</b>	<b>strtoumax()</b>

Tabela B-15: Prefixos e conversões usados com d (família scanf).<sup>152</sup>

**Exemplos:**

```
short s;  
long l;  
int i;
```

Instrução	Armazena...
<code>sscanf("-210", "%3d", &amp;i);</code>	-21 em i
<code>sscanf("-210", "%hd", &amp;s);</code>	-210 em s
<code>sscanf("-210abc", "%ld", &amp;l);</code>	-210 em l

*e, E*

Os tipos de especificadores e e E funcionam exatamente do mesmo modo que a e A descritos anteriormente.

*f*

O tipo de especificador f funciona exatamente do mesmo modo que a, A, e ou E descritos anteriormente.

<sup>152</sup> A base utilizada na chamada de cada função é 10 (v. Seção 6.5.1).

## *F (C99)*

O tipo de especificador *F* tem o mesmo significado que *f*, *a*, *A*, *e* ou *E* descritos anteriormente. O tipo de especificador *F* foi introduzido pelo padrão C99.

## *g e G*

Os tipos de especificadores *g* e *G* são equivalentes a *f*, *F*, *a*, *A*, *e* ou *E* descritos anteriormente.

## *i*

O tipo de especificador *i* é utilizado na leitura de inteiros em qualquer base e pode ser usado com os prefixos mostrados na **Tabela B-16**.

PREFIXO	OS CARACTERES LIDOS SÃO CONVERTIDOS EM...	FUNÇÃO DE CONVERSÃO <sup>153</sup>
Nenhum	<b>int</b>	<b>strtol()</b>
<b>h</b>	<b>short</b>	<b>strtol()</b>
<b>hh</b>	<b>char</b>	<b>strtol()</b>
<b>j (C99)</b>	<b>intmax_t</b>	<b>strtoumax()</b>
<b>l</b>	<b>long</b>	<b>strtol()</b>
<b>ll (C99)</b>	<b>long long</b>	<b>strtoll()</b>
<b>t (C99)</b>	<b>ptrdiff_t</b>	<b>strtoumax()</b>
<b>z (C99)</b>	<b>size_t</b>	<b>strtoumax()</b>

Tabela B-16: Prefixos e conversões usados com *i* (família scanf).

### Observações:

- A principal diferença entre os tipos de especificadores *d* e *i* é que, quando *d* é usado, a função de leitura espera que o valor a ser lido seja representado na base 10, por outro lado, quando *i* é usado, esta função espera que a base utilizada possa ser deduzida de acordo com o formato do valor lido. Isto é, se o número começar

<sup>153</sup> A base utilizada na chamada de cada função é zero. Isto significa que a base será determinada pelo formato do valor lido (v. **Seção 6.5.1**).

com 0x ou 0X, ele será considerado hexadecimal; se ele começar com zero, a base octal será assumida e, em outros casos, a base será considerada decimal.

- Se algum dígito ou letra não corresponder à base presumida, a leitura será interrompida.

**Exemplo:** O programa a seguir realça a diferença entre os tipos de especificadores de i.

```
#include <stdio.h>

#define IMPRIME(x) printf("\tValor de %s = %ld\n", #x, x)

int main(void)
{
    int x, y, z;

    /* Exemplos de uso de d */
    sscanf("125", "%d", &x);
    sscanf("031", "%d", &y);
    sscanf("0x2a", "%d", &z);

    printf("\nUsando o especificador %%d:\n");
    IMPRIME(x);
    IMPRIME(y);
    IMPRIME(z);

    /* Exemplos de uso de i */
    sscanf("125", "%i", &x);
    sscanf("031", "%i", &y);
    sscanf("0x2a", "%i", &z);

    printf("\nUsando o especificador %%i:\n");
    IMPRIME(x);
    IMPRIME(y);
    IMPRIME(z);

    return 0;
}
```

Quando executado, esse programa imprime o seguinte no meio de saída padrão:

*Usando o especificador %d:*

```
Valor de x = 125
Valor de y = 31
Valor de z = 0
```

*Usando o especificador %i:*

```
Valor de x = 125
Valor de y = 25
Valor de z = 42
```

*n*

O tipo de especificador *n* é utilizado para armazenar na variável correspondente o número de caracteres do meio de entrada consumidos até então e aceita os prefixos apresentados na **Tabela B-17**.

PREFIXO	O PARÂMETRO CORRESPONDENTE É INTERPRETADO COMO...
Nenhum	<b>int *</b>
h	<b>short *</b>
hh	<b>char *</b>
j (C99)	<b>intmax_t *</b>
l	<b>long *</b>
ll (C99)	<b>long long *</b>
t (C99)	<b>ptrdiff_t *</b>
z (C99)	<b>size_t *</b>

Tabela B-17: Prefixos usados com *n* (família scanf).

### Exemplo:

```
#include <stdio.h>

int main(void)
{
    float f = 0;
    int    caracteresLidos;
```

```

        /* Exemplo de uso de n */
        sscanf( "-2 25.3abc", "%*d %f %n", &f,
                &caracteresLidos );
        printf( "\nCaracteres consumidos na leitura: %d",
                caracteresLidos );

        return 0;
    }

```

O programa do exemplo anterior imprime o seguinte no meio de saída padrão:

```
Caracteres consumidos na leitura: 7
```

A explicação para o resultado apresentado por esse programa é baseada na interpretação sequencial do *string* de formatação e é a seguinte:

- O asterisco no especificador de formato "%\*d" solicita à função **sscanf()** que leia um inteiro e descarte o valor lido. Portanto, são consumidos os caracteres '-' e '2'.
- O especificador de formato "%f" solicita que a função **sscanf()** leia um valor do tipo **float**.
- O próximo caractere na sequência é ' ' (espaço em branco), que é lido e desprezado.
- Os próximos caracteres, '2', '5', '.' e '3' são lidos, convertidos em 25.3 e este valor é atribuído à variável *f*. O caractere 'a' causa o encerramento da leitura de dados, uma vez que ele não faz parte do valor convertido.
- O especificador de formato %n solicita que a função **sscanf()** armazene o número de caracteres consumidos na variável *caracteresLidos*.

*o*

O tipo de especificador *o* é utilizado na leitura de inteiros sem sinal em base octal e aceita os prefixos mostrados na **Tabela B-18**.

PREFIXO	OS CARACTERES LIDOS SÃO CONVERTIDOS EM...	FUNÇÃO DE CONVERSÃO <sup>154</sup>
Nenhum	<b>int</b>	<b>strtoul()</b>
h	<b>short</b>	<b>strtoul()</b>
hh	<b>char</b>	<b>strtoul()</b>
j (C99)	<b>intmax_t</b>	<b>strtoumax()</b>
l	<b>long</b>	<b>strtoul()</b>
ll (C99)	<b>long long</b>	<b>strtoull()</b>
t (C99)	<b>ptrdiff_t</b>	<b>strtoumax()</b>
z (C99)	<b>size_t</b>	<b>strtoumax()</b>

Tabela B-18: Prefixos e conversões usados com o (família scanf).

**Exemplos:**

```
short s;
long l;
int i;
```

Instrução	Armazena... <sup>155</sup>
<code>sscanf("210", "%2o", &amp;i);</code>	021 em i
<code>sscanf("-210", "%ho", &amp;s);</code>	0177570 em s
<code>sscanf("219", "%lo", &amp;l);</code>	021 em l

*p*

O tipo de especificador *p* é utilizado na leitura de inteiros sem sinal em base hexadecimal que podem ser armazenados em ponteiros do tipo **void \*** (i.e., ponteiros genéricos) e não aceita nenhum prefixo.

**Exemplos:**

```
long *p1, *p2;
```

---

<sup>154</sup> A base 8 é utilizada na chamada de cada função (v. **Seção 6.5.1**).

<sup>155</sup> Os valores são apresentados na base octal.

Instrução	Armazena...
<code>sscanf("fa210", "%3p", &amp;p1);</code>	0xFA2 em p2
<code>sscanf("21A-3", "%p", &amp;p2);</code>	0x21A em p2

**s**

O tipo de especificador *s* é utilizado na leitura de caracteres que serão armazenados num array. O caractere terminal de *string* também é armazenado como último caractere no array. Se a largura não for especificada, a leitura encerra quando for encontrado o primeiro espaço em branco. Este tipo de especificador aceita apenas o prefixo *l*, que indica que os caracteres lidos serão interpretados como multibytes e convertidos, por meio da função **mbrtowc()** (v. **Seção 8.5.3**), em caracteres extensos. O resultado será armazenado em forma de *string* extenso.

Na ausência do prefixo *l*, o respectivo argumento deve ser o endereço de um array de caracteres com capacidade suficiente para conter os caracteres lidos mais um (correspondente ao caractere nulo). Se o *stream* onde a leitura é realizada tiver orientação extensa, os caracteres extensos lidos serão convertidos em caracteres multibytes por meio de chamadas da função **wcrtomb()**, começando no estado inicial de conversão (v. **Capítulo 10**). A função de leitura acrescenta um caractere nulo, de modo que o resultado da leitura será um *string*.

Se o prefixo *l* estiver presente e o *stream* onde a leitura é realizada tiver orientação por byte, os dados de entrada serão interpretados como caracteres multibytes começando no estado inicial de mudança (v. **Seção 8.2**). Neste caso, cada caractere multibyte será convertido em caractere extenso pela função de leitura por meio de chamadas da função **mbrtowc()**. O respectivo argumento deve ser o endereço de um array de caracteres extensos com capacidade suficiente para conter os caracteres extensos resultantes das operações de leitura e conversão, incluindo um caractere extenso nulo. Se o *stream* onde a leitura é realizada tiver orientação extensa, não haverá conversão. A função de leitura acrescenta um caractere nulo e, portanto, o resultado da leitura será um *string* extenso.

**Exemplos:**

```
char    s[10];
wchar_t sExt[10];
```



Instrução	Armazena...
<code>sscanf("abcdef", "%s", s);</code>	"abcdef" em s
<code>sscanf("abcdef", "%3s", s);</code>	"abc" em s
<code>sscanf("abcdef", "%ls", sExt);</code>	L"abcdef" em sExt
<code>swscanf(L"abcdef", L"%ls", sExt);</code>	L"abcdef" em sExt

## u

O tipo de especificador **u** é utilizado na leitura de inteiros sem sinal em base decimal e aceita os prefixos mostrados na **Tabela B-19**.

PREFIXO	OS CARACTERES LIDOS SÃO CONVERTIDOS EM...	FUNÇÃO DE CONVERSÃO <sup>156</sup>
Nenhum	<b>int</b>	<b>strtoul()</b>
h	<b>short</b>	<b>strtoul()</b>
hh	<b>char</b>	<b>strtoul()</b>
j (C99)	<b>intmax_t</b>	<b>strtoumax()</b>
l	<b>long</b>	<b>strtoul()</b>
ll (C99)	<b>long long</b>	<b>strtoull()</b>
t (C99)	<b>ptrdiff_t</b>	<b>strtoumax()</b>
z (C99)	<b>size_t</b>	<b>strtoumax()</b>

Tabela B-19: Prefixos e conversões usados com u (família scanf).<sup>156</sup>

## Exemplos:

```
unsigned short us;
unsigned long ul;
unsigned int ui;
```

Instrução	Armazena...
<code>sscanf("-210", "%u", &amp;ui);</code>	4294967086 em ui <sup>157</sup>
<code>sscanf("210", "%2hu", &amp;us);</code>	21 em us
<code>sscanf("210abc", "%lu", &amp;ul);</code>	210 em ul

<sup>156</sup> A base 10 é utilizada na chamada de cada função (v. **Seção 6.5.1**).

<sup>157</sup> Este valor corresponde a -210 convertido em **unsigned int** e é dependente de implementação.

*x e X*

Os tipos de especificadores `x` e `X` são utilizados na leitura de inteiros sem sinal em base hexadecimal e aceitam os prefixos mostrados na **Tabela B-18**.

PREFIXO	OS CARACTERES LIDOS SÃO CONVERTIDOS EM...	FUNÇÃO DE CONVERSÃO <sup>158</sup>
Nenhum	<code>int</code>	<code>strtoul()</code>
<code>h</code>	<code>short</code>	<code>strtoul()</code>
<code>hh</code>	<code>char</code>	<code>strtoul()</code>
<code>j</code> (C99)	<code>intmax_t</code>	<code>strtoumax()</code>
<code>l</code>	<code>long</code>	<code>strtoul()</code>
<code>ll</code> (C99)	<code>long long</code>	<code>strtoull()</code>
<code>t</code> (C99)	<code>ptrdiff_t</code>	<code>strtoumax()</code>
<code>z</code> (C99)	<code>size_t</code>	<code>strtoumax()</code>

Tabela B-20: Prefixos e conversões usados com `x` e `X` (família `scanf`).

**Observações:**

- Não há diferença entre usar `x` ou `X`.
- Não faz diferença se o valor lido utiliza a notação para constantes hexadecimais de C (e.g., `0x2a19`) ou não (e.g, `2a19`).

**Exemplos:**

```
short s;  
long l;  
int i;
```

Instrução	Armazena...
<code>sscanf("fa210", "%3x", &amp;i);</code>	<code>0xFA2</code> em <code>i</code>
<code>sscanf("210A", "%hx", &amp;s);</code>	<code>0x210A</code> em <code>s</code>
<code>sscanf("0X2a19", "%lX", &amp;l);</code>	<code>0x2A19</code> em <code>l</code>
<code>sscanf("2a19", "%lX", &amp;i);</code>	<code>0x2A19</code> em <code>i</code>

<sup>158</sup> A base hexadecimal é utilizada na chamada de cada função (v. **Seção 6.5.1**).

### [*caracteres*]

O tipo de especificador com formato [*caracteres*] solicita à função de leitura que leia no meio de entrada todos os caracteres entre colchetes e os armazene no endereço representado pelo argumento correspondente. A leitura é encerrada quando a função encontra o primeiro caractere que não faz parte do conjunto de caracteres entre colchetes.

Quando os caracteres entre colchetes são iniciados com o caractere '^', a função de leitura considera todos os caracteres que não se encontram entre colchetes<sup>159</sup>. Neste caso, a leitura é encerrada quando a função encontra o primeiro caractere que faz parte do conjunto de caracteres entre colchetes.

Este tipo de especificador aceita apenas o prefixo `l`, que indica que os caracteres lidos serão considerados multibytes e convertidos, por meio da função `mbrtowc()` (v. **Seção 8.5.3**), em caracteres extensos.

Na ausência do prefixo `l`, o respectivo argumento deve ser o endereço de um array de caracteres com capacidade suficiente para conter os caracteres lidos mais um (correspondente ao caractere nulo). Se o *stream* onde a leitura é realizada tiver orientação extensa, os caracteres extensos lidos serão convertidos em caracteres multibytes por meio de chamadas da função `wcrtomb()`, começando no estado inicial de conversão (v. **Capítulo 10**). A função de leitura acrescenta um caractere nulo e, assim, o resultado da leitura será um *string*.

Se o prefixo `l` estiver presente e o *stream* onde a leitura é realizada tiver orientação por byte, os dados de entrada serão interpretados como caracteres multibytes começando no estado inicial de mudança (v. **Seção 8.2**). Neste caso, cada caractere multibyte será convertido em caractere extenso por intermédio de chamadas de `mbrtowc()`. O respectivo argumento deve ser o endereço de um array de caracteres extensos com capacidade suficiente para conter os caracteres extensos resultantes das operações de leitura e conversão, incluindo o caractere `L'\0'`. Se o *stream* onde a leitura é realizada tiver orientação extensa, não haverá conversão. A função de leitura acrescenta um caractere extenso nulo. Portanto, o resultado da leitura será um *string* extenso.

Dependendo da implementação, conjuntos de caracteres podem ser especificados utilizando uma notação de intervalo por meio do caractere '-'. Por exemplo:

```
"%[a-g]"
```

---

<sup>159</sup> Ou seja, o caractere '^' informa à função de leitura que o complemento do conjunto representado pelos caracteres que não se encontram entre colchetes deve ser levado em consideração, em vez do próprio conjunto entre colchetes.

significa, em muitas implementações, que a função de leitura deve considerar os caracteres que se encontram no intervalo de 'a' a 'g', inclusive.

**Exemplos:**

```
char    s[10];  
wchar_t sExt[10];
```

Instrução	Armazena...
sscanf("abcdef", "%[ecab]", s);	"abc" em s
sscanf("abcdef", "%[^uzec]", s);	"ab" em s
swscanf(L"abcdef", L"%[ecab]", s);	"abc" em s
sscanf("abcdef", "%l[ecab]", sExt);	L"abc" em sExt
swscanf(L"abcdef", L"%l[ecab]", s);	"abc" em s

%

O tipo de especificador % casa com o símbolo de percentagem (' % '). Este tipo de especificador não admite composição com nenhum componente do formato geral.

**Exemplo:**

```
int i;
```

Instrução	Armazena...
sscanf("% -5", "% %d", &i);	-5 em i

**B.3.7 RESUMO DE ESPECIFICADORES DE FORMATO**

A **Tabela B-21** a seguir apresenta tipos de especificadores juntamente com os prefixos permitidos. Nesta tabela, as colunas devem ser interpretadas do seguinte modo:

- **Especificador** – Um tipo de especificador acompanhado de um prefixo. Todas as combinações possíveis de tipos de especificadores com prefixos são apresentados na tabela.

- **Tipo da variável** – O tipo da variável cujo endereço é passado como parâmetro que deve casar com a respectiva combinação de tipo de especificador com prefixo.
- **Função de conversão** – A função de conversão chamada para converter os caracteres lidos num valor do tipo desejado. A função **mbrtowc()** é descrita na **Seção 8.5.3** e as demais funções, que convertem *strings* em números, são apresentadas na **Seção 6.5**.
- **Base** – A base com a qual os valores numéricos são convertidos. Quando este valor for zero, a base será determinada pelo formato do número (v. **Seção 6.5**). Não faz sentido para valores não numéricos.

TIPO DA VARIÁVEL	ESPECIFICADOR	FUNÇÃO DE CONVERSÃO	BASE
char <sup>10</sup>	%c		
char	%hhd (C99)	strtol()	10
char	%hhi (C99)	strtol()	0
char * [array]	%s		
char * [array]	%[...]		
double	%la (C99), %lA (C99), %le, %lE, %lf, %lF (C99), %lg, %lG	strtod()	10
float	%a (C99), %A (C99), %e (Alteração/C99), %E (Alteração/C99), %f (Alteração/C99), %F (C99), %g (Alteração/C99), %G (Alteração/C99)	strtof()	10
int	%d	strtol()	10
int	%i	strtol()	0
int	%n		
int	%hhn (C99)		
intmax_t	%jd (C99)	strtoumax()	10
intmax_t	%ji (C99)	strtoumax()	0
intmax_t	%jn (C99)		

TIPO DA VARIÁVEL	ESPECIFICADOR	FUNÇÃO DE CONVERSÃO	BASE
long	%ld	strtol()	10
long	%li	strtol()	0
long	%ln		
long double	%La (C99), %LA (C99), %Le (Alteração/C99), %LE (Alteração/C99), %Lf (Alteração/C99), %LF (C99), %Lg (Alteração/C99), %LG (Alteração/C99)	strtold()	10
long long	%lld (C99)	strtoll()	10
long long	%lli (C99)	strtoll()	0
long long	%lln (C99)		
ptrdiff_t	%td (C99)	strtoimax()	10
ptrdiff_t	%ti (C99)	strtoimax()	0
ptrdiff_t	%tn (C99)		
ptrdiff_t	%to (C99)	strtoumax()	8
ptrdiff_t	%tu (C99)	strtoumax()	10
ptrdiff_t	%tx (C99)	strtoumax()	16
ptrdiff_t	%tX (C99)	strtoumax()	16
short	%hd	strtol()	10
short	%hi	strtol()	0
short	%hn		
size_t	%zd (C99)	strtoimax()	10
size_t	%zi (C99)	strtoimax()	0
size_t	%zn (C99)		
size_t	%zo (C99)	strtoumax()	8
size_t	%zu (C99)	strtoumax()	10
size_t	%zx (C99)	strtoumax()	16
size_t	%zX (C99)	strtoumax()	16
uintmax_t	%jo (C99)	strtoumax()	8
uintmax_t	%ju (C99)	strtoumax()	10

TIPO DA VARIÁVEL	ESPECIFICADOR	FUNÇÃO DE CONVERSÃO	BASE
uintmax_t	%jx (C99)	strtoumax()	16
uintmax_t	%jX (C99)	strtoumax()	16
unsigned int	%o	strtoul()	8
unsigned int	%hho (C99)	strtoul()	8
unsigned int	%u	strtoul()	10
unsigned int	%hhu (C99)	strtoul()	10
unsigned int	%x	strtoul()	16
unsigned int	%hhx (C99)	strtoul()	16
unsigned int	%X	strtoul()	16
unsigned int	%hhX (C99)	strtoul()	16
unsigned long	%lo	strtoul()	8
unsigned long	%lu	strtoul()	10
unsigned long	%lx	strtoul()	16
unsigned long	%lX	strtoul()	16
unsigned long long	%llo (C99)	strtoull()	8
unsigned long long	%llu (C99)	strtoull()	10
unsigned long long	%llx (C99)	strtoull()	16
unsigned long long	%llX (C99)	strtoull()	16
unsigned short	%ho	strtoul()	8
unsigned short	%hu	strtoul()	10
unsigned short	%hx	strtoul()	16
unsigned short	%hX	strtoul()	16
void **x	%p		
wchar_t <sup>160</sup>	%lc		
wchar_t x[]	%ls		
wchar_t x[]	%l[...]		

Tabela B-21: Tipos e Prefixos permitidos em especificadores de formato da família scanf.

<sup>160</sup> Se desejar ler apenas um caractere o programador pode passar o endereço de uma variável do tipo **wchar\_t**. Caso contrário, ele deve passar o endereço de um array de elementos do tipo **wchar\_t** com capacidade suficiente para conter os caracteres extensos lidos.

## B.4 DIFERENÇAS ENTRE ESPECIFICADORES DE FORMATO DAS FAMÍLIAS PRINTF E SCANF

Existem inúmeras semelhanças entre os especificadores de formato das famílias de funções printf e scanf. Ao mesmo tempo, algumas diferenças entre especificadores de formato usados nos dois casos são tão sutis que, muitas vezes, o programador é induzido ao erro por não percebê-las. Nesta seção serão apresentadas diferenças fundamentais, resumidas na **Tabela B-22**, que, às vezes, passam despercebidas para o programador.

FAMÍLIA PRINTF	FAMÍLIA SCANF
Formato geral: % [ <i>Sinalizador</i> ] [ <i>Largura</i> ] [ <i>.Precisão</i> ] [ <i>Prefixo</i> ] <i>Tipo</i>	Formato geral: % [*] [ <i>Largura</i> ] [ <i>Prefixo</i> ] <i>Tipo</i>
A largura especifica o número mínimo de caracteres que serão escritos.	A largura especifica o número máximo de caracteres que serão levados em consideração durante a leitura.
Os tipos de especificadores d e i são usados na escrita de inteiros na base decimal	O tipo de especificador d é usado na leitura de inteiros na base decimal, mas, com i, podem ser lidos inteiros em qualquer base.
Não existe o tipo de especificador % [ <i>caracteres</i> ].	Existe o tipo de especificador % [ <i>caracteres</i> ].
O prefixo l pode ser usado apenas com c, s, d, i, n, o, u, x, X.	O prefixo l pode ser usado com c, s, d, i, n, o, u, x, X, a, A, e, E, f, g, G.
Os tipos de especificadores a, A, e, E, f, F, g, G, x e X têm significados distintos.	Todos estes tipos de especificadores têm o mesmo significado.
Não existe especificador de formato para o tipo <b>float</b> .	Qualquer um dos especificadores de formato: %a, %A, %e, %E, %f, %F, %g e %G pode ser usados com <b>float</b> .
Não existe asterisco em especificação de formato.	O asterisco pode fazer parte de um especificador de formato.

Tabela B-22: Diferenças entre especificadores de formato das famílias printf e scanf.



# *Apêndice C*

---

*Especificadores de formato de datas e  
horas*

Este apêndice apresenta uma tabela-resumo contendo os especificadores de formato de datas e horas que podem ser usados com a função **strftime()** (v. **Seção 5.3.3**) ou **wcsftime()** (v. **Seção 8.5.5**). Aqui também é apresentado um exemplo que demonstra o uso da maioria destes especificadores de formatos em localidades brasileira ("**pt\_BR.utf8**"), americana ("**en\_US.utf8**") e padrão ("**C**").

ESPECIFICADOR	CAMPOS DE ESTRUTURAS <b>tm</b> AFETADOS	DESCRIÇÃO
%a	tm_wday	Nome abreviado do dia da semana (e.g., Sex).
%A	tm_wday	Nome completo do dia da semana (e.g., sexta).
%b	tm_mon	Nome abreviado do mês (e.g., Jan)
%B	tm_mon	Nome completo do mês (e.g., janeiro).
%c	Todos	Representação preferencial de data e hora na localidade corrente (e.g., Sex 12 Jan 2007 19:58:51 BRT).
%C (C99)	tm_year	Ano/100 (e.g., 07).
%d	tm_mday	Dia do mês (01-31).
%D (C99)	tm_mday tm_mon tm_year	Data no formato mês/dia/ano (e.g., 01/12/07).
%e (C99)	tm_mday	Dia do mês semelhante à %d, mas com espaço em branco substituindo zero.
%F (C99)	tm_mday tm_mon tm_year	Ano-mês-dia no formato de data ISO 8601 (e.g., 2007-01-12).
%g (C99)	tm_wday tm_yday tm_year	Ano de acordo com o padrão ISO 8601 com o ano representado por dois dígitos a partir de 00 (e.g., 07).

ESPECIFICADOR	CAMPOS DE ESTRUTURAS <code>tm</code> AFETADOS	DESCRIÇÃO
%G (C99)	<code>tm_wday</code> <code>tm_yday</code> <code>tm_year</code>	Semelhante a %g, mas com o ano representado por quatro dígitos (e.g., 2007).
%h (C99)	<code>tm_mon</code>	O mesmo que %b (e.g., Jan).
%H	<code>tm_hour</code>	Hora (apenas) em formato de 24 horas, a partir de 00 (e.g., 19).
%I	<code>tm_hour</code>	Hora (apenas) em formato de 12 horas, a partir de 01 (e.g., 07).
%j	<code>tm_yday</code>	Dia do ano, a partir de 001.
%m	<code>tm_mon</code>	Número de dois dígitos que corresponde ao mês do ano, a partir de 01.
%M	<code>tm_min</code>	Minutos passados da hora (00-59).
%n (C99)	Nenhum	Caractere de quebra de linha '\n'.
%p	<code>tm_hour</code>	Indicador AM/PM ou um <i>string</i> equivalente, de acordo com a localidade corrente.
%r (C99)	<code>tm_sec</code> <code>tm_min</code> <code>tm_hour</code>	Hora em formato AM/PM (e.g., 07:58:51).
%R (C99)	<code>tm_min</code> <code>tm_hour</code>	Hora:minuto em formato de 24 horas (e.g., 19:58).
%S	<code>tm_sec</code>	Segundos passados do minuto.
%t (C99)	Nenhum	Caractere de tabulação horizontal '\t'.
%T (C99)	<code>tm_sec</code> <code>tm_min</code> <code>tm_hour</code>	Hora:minuto:segundo em formato de 24 horas (e.g., 19:58:51).
%u (C99)	<code>tm_wday</code>	Número que corresponde ao dia da semana de acordo com o padrão ISO 8601 (Segunda-feira = 1).

ESPECIFICADOR	CAMPOS DE ESTRUTURAS tm AFETADOS	DESCRIÇÃO
%U	tm_wday tm_yday	Número que corresponde à semana do ano (00-53), começando com o primeiro domingo como primeiro dia da semana 01 (cf. %W).
%V (C99)	tm_wday tm_yday tm_year	Número que corresponde à semana do ano de acordo com o padrão ISO 8601 (01-53), onde a primeira semana correspondente tem pelo menos quatro dias no ano corrente, com segunda-feira sendo o primeiro dia da semana.
%w	tm_wday	Número que corresponde ao dia da semana (0-6, com Domingo = 0).
%W	tm_wday tm_yday	Número que corresponde à semana do ano (00-53) começando com a primeira segunda-feira como o primeiro dia da semana 01 (cf. %U).
%x	Todos	Representação preferencial de data (sem a hora) na localidade corrente (e.g., 12-01-2007).
%X	Todos	Hora a partir de 00:00:00.
%y	tm_year	Ano do século com dois dígitos a partir de 00 (e.g., 07).
%Y	tm_year	Ano com quatro dígitos a partir de 0000 (e.g., 2007).
%z (C99)	tm_isdst	Fuso horário (se houver) (e.g., -0300).
%Z	tm_isdst	Nome da zona de fuso horário (se houver) (e.g., BRT).
%%	Nenhum	Caractere ' % '.

ESPECIFICADOR	CAMPOS DE ESTRUTURAS <code>tm</code> AFETADOS	DESCRIÇÃO
<code>%Ec (C99)</code>	Todos	Representação alternativa de data e hora (como <code>%c</code> ) provida pela localidade corrente (e.g., Sex 12 Jan 2007 19:58:51 BRT).
<code>%EC (C99)</code>	<code>tm_mday</code> <code>tm_mon</code> <code>tm_year</code>	Representação alternativa de ano/100 (como <code>%C</code> ) provida pela localidade corrente (e.g., 20).
<code>%Er (C99)</code>	<code>tm_sec</code> <code>tm_min</code> <code>tm_hour</code> <code>tm_mday</code> <code>tm_mon</code> <code>tm_year</code>	Representação alternativa de data e hora no formato de 12 horas provida pela localidade corrente (e.g., 07:58:51).
<code>%Ex (C99)</code>	Todos	Representação alternativa de data (como <code>%x</code> ) provida pela localidade corrente (e.g., 12-01-2007).
<code>%EX (C99)</code>	Todos	Representação alternativa de hora (como <code>%X</code> ) provida pela localidade corrente (e.g., 19:58:51).
<code>%Ey (C99)</code>	<code>tm_mday</code> <code>tm_mon</code> <code>tm_year</code>	Representação alternativa de ano do século (como <code>%y</code> ) provida pela localidade corrente (e.g., 07).
<code>%EY (C99)</code>	<code>tm_mday</code> <code>tm_mon</code> <code>tm_year</code>	Representação alternativa de ano (como <code>%Y</code> ) provida pela localidade corrente (e.g., 2007).

Tabela C-1: Especificadores de formato de datas e horas usados com `strftime()`.

Algumas convenções especificadas pelo padrão ISO 8601 e utilizadas em especificadores de formato da função **`strftime()`** apresentados na **Tabela C-1** são complexas demais para serem discutidas neste livro. O leitor interessado nesses formatos de data e hora encontrarão farta documentação sobre o assunto na internet.

**Exemplo:** O programa a seguir demonstra o uso da função **strftime()** com a maioria dos especificadores de formato descritos na **Tabela C-1**.

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

#define MAX 80

/* As localidades a seguir funcionam */
/* no Linux Ubuntu 8.10 */
#define LOCALIDADE_BR "pt_BR.utf8"
#define LOCALIDADE_US "en_US.utf8"

int main(void)
{
    struct tm *t;
    time_t segundos;
    char s[MAX] = {'\0'};
    int op;
    char *local;

    printf("Escolha a localidade:\n");
    printf( "\t1 = Brasileira\n\t2 = Americana\n"
           "\tOutro valor = C\n" );
    printf("Sua opcao: ");
    scanf("%d", &op);

    if (op == 1)
        local = setlocale(LC_ALL, LOCALIDADE_BR);
    else if (op == 2)
        local = setlocale(LC_ALL, LOCALIDADE_US);
    else
        local = setlocale(LC_ALL, NULL);

    if (!local) {
        printf("\nNao foi possivel alterar localidade");
        return 1;
    }

    printf("\nA localidade corrente e': %s\n\n", local);

    time(&segundos);
```

```

t = localtime(&segundos);

strftime(s, MAX, "Usando especificador %%a: %a", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%A: %A", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%b: %b", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%B: %B", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%c: %c", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%C (C99): "
           "%C", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%d: %d", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%D (C99): "
           "%D", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%e (C99): "
           "%e", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%F (C99): "
           "%F", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%g (C99): "
           "%g", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%G (C99): "
           "%G", t);
printf("%s\n", s);

```

```

strftime(s, MAX, "Usando especificador %%h (C99): "
        "%h", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%H: %H", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%I: %I", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%j: %j", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%m: %m", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%M: %M", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%p: %p", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%r (C99): "
        "%r", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%R (C99): "
        "%R", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%S: %S", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%T (C99): "
        "%T", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%u (C99): "
        "%u", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%U: %U", t);

```



```

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

strftime(s, MAX, "Usando especificador %%V (C99): "
          "%V", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%w: %w", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%W: %W", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%x: %x", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%X: %X", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%y: %y", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%Y: %Y", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%z (C99): "
          "%z", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%Z: %Z", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%Ec (C99): "
          "%Ec", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%EC (C99): "
          "%EC", t);
printf("%s\n", s);

strftime(s, MAX, "Usando especificador %%Er (C99): "
          "%Er", t);
printf("%s\n", s);

```

```

    strftime(s, MAX, "Usando especificador %%Ex (C99): "
              "%Ex", t);
    printf("%s\n", s);

    strftime(s, MAX, "Usando especificador %%EX (C99): "
              "%EX", t);
    printf("%s\n", s);

    strftime(s, MAX, "Usando especificador %%Ey (C99): "
              "%Ey", t);
    printf("%s\n", s);

    strftime(s, MAX, "Usando especificador %%EY (C99): "
              "%EY", t);
    printf("%s\n", s);

    return 0;
}

```

Resultado da execução do programa no Linux usando a localidade "pt\_BR.utf8" (português do Brasil):

*Escolha a localidade:*

*1 = Brasileira*

*2 = Americana*

*Outro valor = C*

*Sua opcao: 1*

*A localidade corrente e': pt\_BR.utf8*

*Usando especificador %a: Sex*

*Usando especificador %A: sexta*

*Usando especificador %b: Jan*

*Usando especificador %B: janeiro*

*Usando especificador %c: Sex 12 Jan 2007 19:58:51 BRT*

*Usando especificador %C (C99): 20*

*Usando especificador %d: 12*

*Usando especificador %D (C99): 01/12/07*

*Usando especificador %e (C99): 12*

*Usando especificador %F (C99): 2007-01-12*

```

Usando especificador %g (C99): 07
Usando especificador %G (C99): 2007
Usando especificador %h (C99): Jan
Usando especificador %H: 19
Usando especificador %I: 07
Usando especificador %j: 012
Usando especificador %m: 01
Usando especificador %M: 58
Usando especificador %p:
Usando especificador %r (C99): 07:58:51
Usando especificador %R (C99): 19:58
Usando especificador %S: 51
Usando especificador %T (C99): 19:58:51
Usando especificador %u (C99): 5
Usando especificador %U: 01
Usando especificador %V (C99): 02
Usando especificador %w: 5
Usando especificador %W: 02
Usando especificador %x: 12-01-2007
Usando especificador %X: 19:58:51
Usando especificador %y: 07
Usando especificador %Y: 2007
Usando especificador %z (C99): -0300
Usando especificador %Z: BRT
Usando especificador %Ec (C99): Sex 12 Jan 2007 19:58:51 BRT
Usando especificador %EC (C99): 20
Usando especificador %Er (C99): 07:58:51
Usando especificador %Ex (C99): 12-01-2007
Usando especificador %EX (C99): 19:58:51
Usando especificador %Ey (C99): 07
Usando especificador %EY (C99): 2007

```

Resultado da execução do programa no Linux usando a localidade "en\_US.utf8" (inglês dos Estados Unidos):

```

Escolha a localidade:
1 = Brasileira
2 = Americana
Outro valor = C

```

Sua opção: 2

A localidade corrente e': en\_US.utf8

```

Usando especificador %a: Fri
Usando especificador %A: Friday
Usando especificador %b: Jan
Usando especificador %B: January
Usando especificador %c: Fri 12 Jan 2007 08:00:26 PM BRT
Usando especificador %C (C99): 20
Usando especificador %d: 12
Usando especificador %D (C99): 01/12/07
Usando especificador %e (C99): 12
Usando especificador %F (C99): 2007-01-12
Usando especificador %g (C99): 07
Usando especificador %G (C99): 2007
Usando especificador %h (C99): Jan
Usando especificador %H: 20
Usando especificador %I: 08
Usando especificador %j: 012
Usando especificador %m: 01
Usando especificador %M: 00
Usando especificador %p: PM
Usando especificador %r (C99): 08:00:26 PM
Usando especificador %R (C99): 20:00
Usando especificador %S: 26
Usando especificador %T (C99): 20:00:26
Usando especificador %u (C99): 5
Usando especificador %U: 01
Usando especificador %V (C99): 02
Usando especificador %w: 5
Usando especificador %W: 02
Usando especificador %x: 01/12/2007
Usando especificador %X: 08:00:26 PM
Usando especificador %y: 07
Usando especificador %Y: 2007
Usando especificador %z (C99): -0300
Usando especificador %Z: BRT
Usando especificador %Ec (C99): Fri 12 Jan 2007 08:00:26 PM
BRT
Usando especificador %EC (C99): 20
Usando especificador %Er (C99): 08:00:26 PM
Usando especificador %Ex (C99): 01/12/2007

```

```

Usando especificador %EX (C99): 08:00:26 PM
Usando especificador %Ey (C99): 07
Usando especificador %EY (C99): 2007

```

Resultado da execução do programa no Linux usando a localidade padrão "C":

*Escolha a localidade:*

*1 = Brasileira*

*2 = Americana*

*Outro valor = C*

*Sua opcao: 3*

*A localidade corrente e': C*

```

Usando especificador %a: Fri
Usando especificador %A: Friday
Usando especificador %b: Jan
Usando especificador %B: January
Usando especificador %c: Fri Jan 12 20:01:18 2007
Usando especificador %C (C99): 20
Usando especificador %d: 12
Usando especificador %D (C99): 01/12/07
Usando especificador %e (C99): 12
Usando especificador %F (C99): 2007-01-12
Usando especificador %g (C99): 07
Usando especificador %G (C99): 2007
Usando especificador %h (C99): Jan
Usando especificador %H: 20
Usando especificador %I: 08
Usando especificador %j: 012
Usando especificador %m: 01
Usando especificador %M: 01
Usando especificador %p: PM
Usando especificador %r (C99): 08:01:18 PM
Usando especificador %R (C99): 20:01
Usando especificador %S: 18
Usando especificador %T (C99): 20:01:18
Usando especificador %u (C99): 5
Usando especificador %U: 01
Usando especificador %V (C99): 02
Usando especificador %w: 5
Usando especificador %W: 02

```

```
Usando especificador %x: 01/12/07
Usando especificador %X: 20:01:18
Usando especificador %y: 07
Usando especificador %Y: 2007
Usando especificador %z (C99): -0300
Usando especificador %Z: BRT
Usando especificador %Ec (C99): Fri Jan 12 20:01:18 2007
Usando especificador %EC (C99): 20
Usando especificador %Er (C99): 08:01:18 PM
Usando especificador %Ex (C99): 01/12/07
Usando especificador %EX (C99): 20:01:18
Usando especificador %Ey (C99): 07
Usando especificador %EY (C99): 2007
```

# *Apêndice D*

---

*Erros comuns de programação em C*

## D.1 INTRODUÇÃO

Este apêndice apresenta erros que, de acordo com o julgamento e a experiência do autor, são comuns em programação usando C. Um uso prático da coletânea de erros apresentada aqui é como uma lista de verificação para inspeção de programas durante a fase de teste e depuração (v. **Seção 6.10** do **Volume I**).

## D.2 OPERADORES

### D.2.1 USO DE ATRIBUIÇÃO EM VEZ DE IGUALDADE OU VICE-VERSA

Algumas vezes, o programador usa distraidamente o operador de comparação `==` quando a intenção é usar o operador de atribuição `=`, ou vice-versa. Uma precaução contra este tipo de erro é, sempre que possível, usar uma constante no lado esquerdo de uma expressão de comparação. Por exemplo, ao analisar uma expressão como a que faz parte da instrução **if** a seguir:

```
int x;
...
if (x = 10)
    ...
```

*alguns* compiladores podem emitir uma mensagem de advertência, mas isto não impede o programa de ser compilado. Entretanto, se o programador tivesse escrito o trecho de programa como:

```
int x;
...
if (10 = x)
    ...
```

*qualquer* compilador indicaria a ocorrência de erro.

Quando se juntam atribuição e teste numa única expressão, o programador deve tornar explícita sua intenção. Por exemplo, é melhor escrever o teste da instrução **if** a seguir como:

```
if ( (p = malloc(sizeof(double))) != NULL )
```



...

do que escrevê-la como:

```
if ( p = malloc(sizeof(double)) )
...
```

## D.2.2 USO INCORRETO DE REGRAS DE PRECEDÊNCIA E ASSOCIATIVIDADE

Regras de precedências e associatividade que regem a ordem de aplicação de operadores são facilmente esquecidas. Além disso, algumas escolhas de precedência de operadores de C não são intuitivas. Por exemplo, `==` tem maior precedência que `&`, de modo que a expressão `x & 1 == 0` é interpretada como `x & (1 == 0)` [e não como `(x & 1) == 0`, como provavelmente era esperado] que resulta em zero, independentemente do valor de `x`.

A seguir, serão apresentados exemplos de erros frequentemente cometidos devido a suposições incorretas sobre ordem de aplicação de operadores.

### Exemplo:

```
if (sinalizadores & MASCARA != 0)
```

Provavelmente, nesta instrução, a intenção do programador é testar se algum sinalizador está ligado (v. **Seção 13.7.1 do Volume I**). Acontece, porém, que o operador `!=` tem maior precedência que o operador `&`, de modo que a expressão que acompanha a instrução **if** é interpretada como:

```
sinalizadores & (MASCARA != 0)
```

Neste caso, para obter a correta interpretação, o programador deve usar parênteses:

```
if ((sinalizadores & MASCARA) != 0)
```

### Exemplo:

```
while (c = getc(stream) != EOF)
```

Novamente, a intenção do programador parece óbvia: encerrar o laço **while** quando a função **getc()** retornar **EOF**. Mas, provavelmente, ele esqueceu que o operador **!=** é mais forte que o operador **=**, de modo que a expressão que acompanha o laço **while** é interpretada como:

```
c = (getc(stream) != EOF)
```

de modo que **c** receberá sempre receberá zero ou 1. Mais uma vez, a correção é obtida por meio do uso de parênteses:

```
while ((c = getc(stream)) != EOF)
```

### D.2.3 USO DE && EM VEZ DE || OU VICE-VERSA

Este problema na maioria das vezes é decorrente do uso incorreto das leis de De Morgan ensinadas em Lógica Matemática:

EXPRESSÃO NEGADA	EXPRESSÃO EQUIVALENTE
!(A && B)	!A    !B
!(A    B)	!A && !B

**Exemplo:**

```
if ( !(x < 0 && y <= 10) )
...
```

é equivalente a:

```
if ( !(x < 0) || !(y <= 10) ) /* OK */
...
```

ou:

```
if ( x >= 0 || y > 10 ) /* OK */
...
```

Mas não é equivalente a:

```
if ( !(x < 0) && !(y <= 10) ) /* INCORRETO! */
...
```

nem a:

```
if ( x >= 0 && y > 10 ) /* INCORRETO! */
...
```

## D.2.4 USO DE & EM VEZ DE &&

Os operadores && e & não são intercambiáveis. Mais precisamente, && é um operador lógico, mas & *não é operador lógico em C*<sup>153</sup>. Exemplos que mostram diferenças entre estes dois operadores são apresentados no **Seção 13.3.1 do Volume I**.

## D.2.5 USO DE | EM VEZ DE ||

Os operadores || e | não são intercambiáveis: || é um operador lógico, enquanto | *não é operador lógico em C*<sup>154</sup>. Exemplos que mostram diferenças entre estes dois operadores são apresentados no **Seção 13.3.1 do Volume I**.

## D.2.6 OPERADORES LÓGICOS DE C NÃO SÃO COMUTATIVOS

Diferentemente do que ocorre em Lógica Matemática, os operadores lógicos && e || de C não são comutativos. Isto significa que a ordem com a qual seus operandos são escolhidos é importante<sup>155</sup>. Considere, por exemplo, a seguinte expressão lógica:

```
x > 0 || ++y < 10
```

Nesta expressão, devido ao curto-circuito, a subexpressão ++y < 10 será avaliada apenas quando x ≤ 0. Consequentemente, apenas nesta situação a variável y será incrementada. Por outro lado, na expressão a seguir, obtida da expressão anterior com a inversão dos operandos do operador ||:

```
++y < 10 || x > 0
```

a variável y será sempre incrementada.

Como outro exemplo da importância da escolha da ordem dos operandos de um operador lógico, considere o seguinte fragmento de programa:

```
if (x > 0 && y%x)
    ...
```

153 Em Java, há dois operadores lógicos de conjunção, & e &&, que se distinguem, respectivamente, pela ausência ou presença de curto-circuito, mas, em C, apenas && é operador lógico.

154 Em Java, há dois operadores lógicos de disjunção, | e ||, que se distinguem, respectivamente, pela ausência ou presença de curto-circuito, mas, em C, apenas || é operador lógico.

155 Note que a discussão aqui não tem nada a ver com ordem de avaliação de operandos apresentada na **Seção 1.3.3 do Volume I**. Os dois operadores && e || têm ordem de avaliação de operando bem definida, que é primeiro avaliar o primeiro operando e, então, *se for preciso*, avaliar o segundo operando. O que se está discutindo aqui é quais serão o primeiro e o segundo operandos escolhidos pelo programador.

Naquilo que diz respeito à avaliação da expressão que acompanha a instrução **if**, o fragmento de programa anterior funciona perfeitamente bem. No entanto, se os operandos do operador **&&** forem invertidos:

```
if (y%x && x > 0)
    ...
```

o programa será abortado quando *x* for igual a zero.

## D.2.7 SEPARAÇÃO INDEVIDA DE SÍMBOLOS

Alguns operadores são constituídos por múltiplos *tokens* que podem ser separados, mas outros são constituídos por um único *token*. Neste último caso, os caracteres que constituem o operador não podem ser separados. Por exemplo, o operador **+=** é constituído por dois *tokens* (**+** e **=**) e, assim, ele pode ser escrito com espaço entre eles:

```
x +      = y; /* É feio, mas é legal */
```

Por outro lado, o operador **->** consiste em um único *token* e, portanto, seus caracteres constituintes não podem ser separados. Por exemplo, a expressão a seguir é ilegal:

```
y = p -  > x; /* ILEGAL: -> é indivisível */
```

Concluindo, é aconselhável que o programador nunca separe os caracteres constituintes de um operador, em vez de preocupar-se em saber quais deles são separáveis ou não.

## D.2.8 SUPOSIÇÕES SOBRE ORDEM DE AVALIAÇÃO DE OPERANDOS

Em C, apenas quatro operadores possuem ordem de avaliação de operandos definida: **&&**, **||**, **?:** e **,** (operador vírgula). Nenhum outro operador possui esta propriedade. Esta última categoria inclui construções que sequer são considerados operadores em outras linguagens de programação, como **()** (operador de chamada de função) e **[]** (operador de acesso a elemento de array).

Considere o seguinte exemplo:

```
int i = 0, arA[NUM_ELEMENTOS], arB[NUM_ELEMENTOS];
...
while (i < NUM_ELEMENTOS)
    arA[i] = arB[i++];
```

O problema com o trecho de programa anterior é que o operador de atribuição não possui ordem de avaliação de operandos definida. Apenas se o primeiro operando da atribuição for avaliado primeiro, a provável intenção do programador será satisfeita. Uma solução para o problema apresentado é escrever o laço **while** do último trecho de programa como:

```
while (i < NUM_ELEMENTOS) {
    arA[i] = arB[i];
    ++i;
}
```

É importante notar que essa categoria de problemas descrita aqui não pode ser resolvida por meio do uso de parênteses, pois esses problemas não são decorrentes de precedência ou associatividade.

## D.3 ESTRUTURAS DE CONTROLE

### D.3.1 USO INDEVIDO DE PONTO E VÍRGULA

Observe o seguinte trecho de programa:

```
int i = 10;
...
while(i > 0);
    --i;
```

A indentação usada pelo programador indica que sua provável intenção era que a instrução:

```
--i;
```

constituísse o corpo do laço **while**. Entretanto, o ponto e vírgula no final da linha contendo **while** termina prematuramente essa instrução.

Outro exemplo:

```
if (x > max);
    max = x;
```

Neste último exemplo, a instrução:

```
max = x;
```

será sempre executada, independentemente do resultado da avaliação da expressão:

```
x > max
```

Uma prevenção contra esse tipo de erro que termina uma estrutura de controle prematuramente é usar sempre um abre-chaves ao final da linha inicial da estrutura de controle (v. **Seção 6.5** do **Volume I**). Obviamente, este abre-chaves deve ter um correspondente fecha-chaves ao final do corpo da estrutura de controle.

### D.3.2 INSTRUÇÃO SWITCH-CASE SEM BREAK

Em linguagens como C e suas descendentes (e.g., C++ e Java), a estrutura de controle de seleção (i.e., **switch-case** em C e linguagens descendentes) possui *efeito cascata*. Ou seja, uma vez que um caso é selecionado, todas as instruções correspondentes a este caso e aos casos subsequentes são executadas. Muito raramente, quando se usa uma instrução **switch-case**, este é o efeito desejado. Isto é, o que o programador deseja, em geral, é executar exatamente as instruções correspondentes a um dado caso, e apenas estas instruções. Este efeito é obtido com o uso da instrução **break** (v. **Seção 1.7.4** do **Volume I**). Portanto, para cada instrução **switch-case** verifique se não está faltando **break** ou se o efeito cascata é de fato o desejado. Na segunda situação, que é rara, é recomendável incluir um comentário que explicita que não houve esquecimento de **break**. Por exemplo, a instrução **switch-case** esquematizada a seguir pode deixar dúvida com relação à ausência de **break** após o primeiro caso:

```
switch (opcao) {
    case 1:
        x = y - z; /* Falta de break acidental ou proposital? */
    case 2:
        ...
    default:
        ...
}
```

Se o efeito cascata, que poucas vezes é desejado, for realmente o que se pretende, esta intenção deve ser esclarecida por meio de comentário:

```
switch (opcao) {
    case 1:
```

```

        x = y - z; /* Segue-se efeito cascata */
    case 2:
        ...
    default:
        ...
}

```

### D.3.3 INSTRUÇÃO DO-WHILE CONFUNDIDA COM REPEAT-UNTIL

Em muitas linguagens de programação (e.g., Pascal, Basic, Modula-2) a estrutura de controle equivalente a **do-while** em C repete a execução de seu corpo *até que* uma dada condição seja satisfeita, em vez de *enquanto* uma condição é satisfeita. Nestas linguagens, o programador deve especificar qual é a condição que deve ser satisfeita para que o laço pare. Desse modo, programadores acostumados a programar nessas linguagens tendem a pensar que a estrutura **do-while** de C funciona da mesma maneira, mas este não é o caso<sup>156</sup>.

Parece que a dificuldade de raciocinar corretamente com relação a **do-while** aumenta quando a condição a ser especificada consiste em uma conjunção ou disjunção de expressões. Suponha, a título de ilustração, que se tenham duas variáveis,  $x$  e  $y$ , e que se deseje encerrar um laço quando  $x < 0$  e  $y > 10$ . Então, um programador de Pascal poderia escrever<sup>157</sup>:

```

REPEAT
...
UNTIL (x < 0) AND (y > 10);

```

Deparado com a mesma situação, um programador inexperiente em C, mas habituado a escrever programas em Pascal (ou Modula-2, Delphi, Basic, etc.) seria levado a escrever a seguinte instrução **do-while**:

```

do {
...
} while ((x < 0) && (y > 10));

```

<sup>156</sup> Este problema não aflige o uso do laço **while**, pois, na maioria das linguagens mais usadas, este laço possui equivalentes que funcionam do mesmo modo.

<sup>157</sup> Mesmo que você nunca tenha programado em Pascal, provavelmente não deve apresentar dificuldade de entender esta estrutura de controle. Ela apenas instrui o computador a executar um conjunto de instruções até que a condição especificada no final seja verdadeira.

Acontece que os raciocínios empregados nessas duas instruções não são equivalentes. Isto é, a instrução `REPEAT-UNTIL`, em Pascal, instrui o computador a parar a execução do laço quando a conjunção  $x < 0$  e  $y > 10$  for verdadeira, e a instrução **do-while** em C diz que o laço deve continuar sendo executado enquanto a conjunção  $x < 0$  e  $y > 10$  for verdadeira. Ou seja, o laço **do-while** deve parar quando a negação da conjunção  $x < 0$  e  $y > 10$  for verdadeira. Portanto, os dois laços não podem ser equivalentes.

Concluindo, uma boa estratégia para determinar a condição de parada de um laço **do-while** é a seguinte:

1. Encontre a expressão lógica que determine quando o laço deve terminar. Exemplos: o laço deve terminar quando o final do arquivo for atingido, o laço deve terminar quando  $x < 0$  e  $y > 10$ , etc.
2. Obtenha uma expressão resultante da negação daquela determinada no passo anterior e utilize-a como condição de término do laço **do-while**. Se a expressão encontrada no passo anterior consistir em uma conjunção ou disjunção, utilize uma das leis de De Morgan que devem ter sido aprendidas em algum curso de Lógica Matemática (v. **Seção D.2.3**).

Por exemplo, suponha que um laço de repetição **do-while** deva terminar quando  $x < 0$  e  $y > 10$  (**Passo 1**). Em C, esta expressão condicional é escrita como  $x < 0 \ \&\& \ y > 10$ , e a negação desta expressão é:  $!(x < 0 \ \&\& \ y > 10)$  (**Passo 2**). De acordo com uma das leis de De Morgan, esta última expressão é equivalente a:  $!(x < 0) \ || \ !(y > 10)$ . Portanto, a instrução **do-while** em questão deve ser escrita como:

```
do {
    ...
} while (!(x < 0) || !(y > 10));
```

### D.3.4 ELSE QUE NÃO CORRESPONDE AO IF DESEJADO

Conforme se afirmou na **Seção 1.7.4 do Volume I**, uma parte **else** sempre corresponde à parte **if** da estrutura de controle **if-else** mais próxima que ainda não possui um **else** correspondente. Infelizmente, esta regra é tão fácil de apreender quando de ser esquecida. Por exemplo, considere o seguinte trecho de programa:

```
if (x == 0)
    if (y == 0)
        ++y;
```



```
else
    --x;
```

Pelo modo como endentou esse trecho de programa, aparentemente o programador gostaria que a parte **else** correspondesse à primeira instrução **if**. Entretanto, devido à regra de casamento de ifs e elses, a parte **else** casa com a segunda instrução **if**. O problema pode ser corrigido envolvendo-se a segunda instrução **if** entre chaves, como mostrado a seguir:

```
if (x == 0) {
    if (y == 0)
        ++y;
} else
    --x;
```

## D.4 DEFINIÇÕES INCORRETAS DE FUNÇÕES

Esta seção apresenta alguns problemas associados a definições incorretas de funções. Evidentemente, chamadas de funções também podem apresentar problemas. Esta última categoria de problemas será discutida na **Seção D.6**.

### D.4.1 RECURSÃO SEM FIM

O programador deve garantir que, para quaisquer que sejam os parâmetros de uma função recursiva, um caso não recursivo seja atingido (v. **Seção 3.5 do Volume I**). Por exemplo:

```
int Fatorial(int n)
{
    if(n == 0 || n == 1)
        return 1;
    return n*Fatorial(n - 1);
}
```

Essa função funcionará bem se for usada dentro de seu domínio (i.e., se  $n \geq 0$ ), mas haverá recursão infinita se ela for chamada, inadvertidamente, com um valor negativo como parâmetro. A melhor solução para a função `Fatorial()` anterior é apresentada a seguir<sup>158</sup>:

```
unsigned Fatorial(int n)
{
    if(n < 0)
        return 0; /* Zero indica erro */

    if(n == 0 || n == 1)
        return 1;

    return n*Fatorial(n - 1);
}
```

## D.4.2 RETORNO DE ZUMBIS

Uma função não deve jamais retornar o endereço de uma variável local de duração automática, conforme discutido em detalhes na **Seção 7.5 do Volume I**. Por exemplo:

```
char* UmaFuncao1(const char *str)
{
    char aux[80];
    ...
    return aux; /* Zumbi retornado */
}
```

Para resolver o problema apresentado por essa função, existem duas abordagens:

1. Usar uma variável de duração fixa (v. **Seção 4.2 do Volume I** ou **Seção 12.2.5** no presente volume). Por exemplo:

```
char* UmaFuncao2(const char *str)
{
```

---

<sup>158</sup> A solução mais simples para este problema parece ser usar **unsigned** como tipo de parâmetro da função, pois, assim, a função nunca receberia um valor negativo. Mas, esta não é uma boa ideia. A principal justificativa para não adotar esta solução é que, caso seja passado um valor negativo como parâmetro desta função, ela estará retornando um resultado incorreto, pois fatoriais de números negativos simplesmente não existem.

```

static char aux[80];
...
return aux; /* OK. Não é Zumbi */
}

```

Um possível problema com essa abordagem é que, na próxima chamada dessa função, o conteúdo do array `aux[]` poderá ser alterado. Portanto, talvez seja necessário fazer uma cópia dele na função que faz a chamada.

2. Usar alocação dinâmica de memória (v. **Capítulo 11** do **Volume I**). Por exemplo:

```

char* UmaFuncao3(const char *str)
{
    char *aux = malloc(80*sizeof(char));
    ...
    return aux; /* OK. Não é Zumbi */
}

```

Esta solução é melhor do que a anterior, mas é necessário que o programador se lembre de incluir uma chamada de **free()** no corpo da função que faz a chamada da função `UmaFuncao3()` para liberar o espaço alocado.

É importante chamar a atenção para o fato de o efeito zumbi só se manifestar quando a pilha de execução é alterada, como demonstra o exemplo seguinte.

```

#include <stdio.h>

#define TAMANHO_ZUMBI 5

int *RetornaZumbi(void)
{
    int i, zumbi[TAMANHO_ZUMBI] = {1, 2, 3, 4, 5};

    printf( "\nConteúdo do array zumbi em "
            "RetornaZumbi():\n" );

    for (i = 0; i < TAMANHO_ZUMBI; ++i)
        printf("\tzumbi[%d] = %d\n", i, zumbi[i]);

    return zumbi;
}

int main(void)

```

```

{
    int *pInt, i, soma = 0;

    pInt = RetornaZumbi();

    /* Até aqui está tudo bem, pois */
    /* a pilha ainda não foi alterada */
    /* e o zumbi continua vivo. */
    for (i = 0; i < TAMANHO_ZUMBI; ++i)
        soma += pInt[i]; /* Ainda está OK */

    /* A chamada da função printf() */
    /* altera o conteúdo da pilha */
    printf("\nResultado da soma em main(): %d\n", soma);

    /* Outra alteração no conteúdo da pilha */
    printf("\nConteudo do array zumbi em main():\n");

    /* Como o conteúdo da pilha foi alterado, o */
    /* zumbi morreu e reencarnou com outro valor */
    for (i = 0; i < TAMANHO_ZUMBI; ++i)
        printf("\tpInt[%d] = %d\n", i, pInt[i]);

    return 0;
}

```

Quando esse programa é compilado e executado no Windows XP, ele produz o seguinte resultado:

*Conteudo do array zumbi em RetornaZumbi():*

```

zumbi[0] = 1
zumbi[1] = 2
zumbi[2] = 3
zumbi[3] = 4
zumbi[4] = 5

```

*Resultado da soma em main(): 15*

*Conteudo do array zumbi em main():*

```

pInt[0] = 1
pInt[1] = 2009332896
pInt[2] = 2009209023
pInt[3] = 32
pInt[4] = 0

```

Observe que, conforme previsto, o efeito zumbi só se manifestou na impressão do array no corpo de **main()** após a alteração do conteúdo da pilha, que ocorreu com as duas chamadas da função **printf()**<sup>159</sup>. Antes disso, o cálculo da soma dos elementos do array foi perfeitamente correto.

### D.4.3 CHAMADAS SEM O DEVIDO RETORNO

Considere a função **F()** definida no programa a seguir.

```
#include <stdio.h>

int F(int arg)
{
    if (arg > 0)
        return arg;
}

int main(void)
{
    printf( "\nValor retornado por F(10): %d\n",
           F(10) );
    printf( "Valor retornado por F(-10): %d\n",
           F(-10) );

    return 0;
}
```

De acordo com o cabeçalho da função **F()**, ela deve *sempre* retornar um valor do tipo **int**. Mas, conforme se pode notar, esta função retorna um valor apenas quando o argumento é positivo. O que ocorre, então, quando esta função é chamada com um argumento negativo? De acordo com o padrão ISO de C, o resultado é indefinido (v. **Seção 13.2**).

Quando o último programa é executado no Windows XP, ele imprime:

```
Valor retornado por F(10): 10
Valor retornado por F(-10): -1
```

---

<sup>159</sup> O resultado da segunda impressão do array poderá ser diferente se for utilizado outro compilador ou sistema operacional, mas provavelmente não será aquilo que é esperado.

Mas, este resultado poderia ser diferente, dependendo do compilador ou sistema operacional utilizado.

### D.4.4 DESCONHECIMENTO DE REGRAS DE ESCOPO

Observe o seguinte fragmento de programa:

```
int x; /* Variável global */

void F()
{
    float x = 2.54;
    ...
    printf("Valor corrente de x: %d");
}
```

O especificador de formato (%d) usado com **printf()** revela que provavelmente o programador desejava imprimir o valor da variável global `x`. Entretanto, como, inadvertidamente, foi declarada uma variável local à função com o mesmo nome que a variável global, pelas regras de escopo de C, é a variável local que tem validade no corpo da função.

Considere agora o seguinte programa:

```
#include <stdio.h>

int i;

void F()
{
    printf("\nFuncao F() chamada\n", i);

    for (i = 5; i > 0; --i)
        printf("\ti = %d\n", i);
}

int main(void)
{
    for (i = 0; i <= 5; ++i) {
        F();
    }
}
```

```
    return 0;
}
```

O último programa nunca termina, a não ser que o usuário solicite seu encerramento ao sistema operacional (e.g., digitando [CTRL]+C). O problema aqui também tem a ver com escopo, mas o caso aqui é o oposto do exemplo precedente. Isto é, uma solução seria simplesmente definir uma variável local no corpo de `F()` denominada `i` de modo a impedir que esta função utilizasse a variável global de mesmo nome.

## D.4.5 VAZAMENTO DE MEMÓRIA

Toda memória alocada de forma dinâmica deve ser liberada explicitamente por meio de chamadas da função `free()`. Se uma função, logo antes de retornar, tiver um ponteiro local apontando para um espaço em memória alocado dinamicamente, este espaço jamais poderá ser acessado de novo<sup>160</sup>. Assim, se esta função for chamada muitas vezes, a capacidade do *heap* poderá ser esgotada bem antes do previsto devido ao vazamento de memória causado pela função. Por exemplo:

```
void UmaFuncao(const char *str)
{
    char *p = malloc(strlen(str) + 1);
    ... /* Executa algum processamento */
        /* usando o bloco alocado          */
}
```

Se a função `free()` não for usada antes do retorno da função do exemplo anterior, cada chamada dela causará vazamento de memória<sup>161</sup>.

## D.4.6 USO INCORRETO DE VARIÁVEIS LOCAIS DE DURAÇÃO FIXA

Considere o seguinte programa que demonstra o uso incorreto de uma variável local de duração fixa.

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

<sup>160</sup> A não ser, obviamente, que a função retorne um ponteiro para o espaço alocado dinamicamente.

<sup>161</sup> Entretanto, se uma função aloca memória e retorna um ponteiro para o espaço alocado, ela não deve chamar `free()` antes de retornar (v. **Seção D.10.1**).

```

#define BYTES_POR_INT 8 /* Pode não ser o caso */
#define TAM_PREFIXO_OX 2 /* Caracteres em "0x" */
#define TAM_ARRAY      BYTES_POR_INT + TAM_PREFIXO_OX + 1

/****
 *
 * IntEmHex(): Retorna um string correspondente à
 *              representação hexadecimal de um int
 *
 * Argumento: valor (entrada) - o valor cuja
 *              representação será
 *              determinada
 *
 * Retorno: string contendo a representação
 *              hexadecimal do inteiro; NULL se a
 *              representação não couber no string
 *
 ****/

char *IntEmHex(int valor)
{
    static char strHexadecimal[TAM_ARRAY];

    /* Verifica se o resultado da */
    /* representação hexadecimal */
    /* caberá no array */
    if (sizeof(valor) > BYTES_POR_INT)
        return NULL; /* Não vai caber no array */

    /* Copia a representação hexadecimal */
    /* do parâmetro no array */
    sprintf(strHexadecimal, "0x%X", valor);

    return strHexadecimal;
}

int main(void)
{
    printf("\nRepresentacao hexadecimal de %d: \"%s\""
           "\nRepresentacao hexadecimal de %d: \"%s\"\n",
           1, IntEmHex(1), 2, IntEmHex(2));

    return 0;
}

```



Quando esse programa é compilado e executado no Windows XP, obtém-se o seguinte resultado no meio de saída padrão:

```
Representacao hexadecimal de 1: "0x1"
Representacao hexadecimal de 2: "0x1"
```

A primeira linha impressa pelo programa está correta, mas a segunda evidentemente não está. Isto ocorre porque a função **printf()** precisa, ao mesmo tempo, usar os dois *strings* retornados pela função `IntEmHex()`. Acontece, porém, que o *string* retornado por esta função é armazenado numa variável local de duração fixa, de modo que, quando é feita a segunda chamada desta função, o conteúdo do primeiro *string* é sobrescrito.

Há duas soluções para esse problema. A mais simples delas consiste em substituir a chamada de **printf()** na função **main()** por duas chamadas dessa função, como mostrado a seguir:

```
printf( "\nRepresentacao hexadecimal de %d: \"%s\"",
        1, IntEmHex(1) );

printf( "\nRepresentacao hexadecimal de %d: \"%s\"",
        2, IntEmHex(2) );
```

Esta solução funciona porque, quando a chamada da função `IntEmHex()` sobrescrever o *string* retornado na primeira chamada, este *string* não mais é necessário.

A segunda solução é um pouco mais complicada e requer alterar o código da função `IntEmHex()` para que ela use alocação dinâmica de memória em vez de uma variável local de duração fixa. Esta solução é adequada se for realmente necessário processar mais de um *string* retornado por essa função ao mesmo tempo. Uma nova implementação da função `IntEmHex()` usando alocação dinâmica de memória poderia ser:

```
char *IntEmHex2(int valor)
{
    char *strHexadecimal;

    /* Verifica se o resultado da */
    /* representação hexadecimal */
    /* caberá no array */
    if (sizeof(valor) > BYTES_POR_INT)
        return NULL; /* Não vai caber no array */

    strHexadecimal = malloc(TAM_ARRAY);
```

```

        /* Copia a representação hexadecimal */
        /* do parâmetro no array */
        sprintf(strHexadecimal, "0x%X", valor);

        return strHexadecimal;
    }

```

**Exercício:** Baseado no resultado impresso pelo programa apresentado no início desta seção, o que pode ser concluído com relação à ordem com que o programa avalia argumentos numa chamada de função?

## D.5 ENTRADA E SAÍDA

### D.5.1 USO INCORRETO DE `scanf()`

Um erro bastante comum entre iniciantes em programação em C é passar uma variável como argumento para **`scanf()`**, em vez de passar o endereço da própria variável. Como o padrão de C não requer que compiladores de C chequem a esperada correspondência entre especificadores de formato e respectivos endereços de variáveis em chamadas da função **`scanf()`**, este erro manifesta-se apenas durante execução do programa. Por exemplo:

```

int x;
...
scanf("%d", x); /* Deveria ser scanf("%d", &x); */

```

Apesar de parecer um erro ingênuo, suas consequências podem ser desastrosas. Isto é, este erro pode acarretar em aborto do programa ou em sua evolução para um erro lógico. Bons compiladores (e.g., gcc) ou programas *lint* apontam este tipo de irregularidade (v. **Seção 6.11 do Volume I**).

Agora, observe o seguinte programa:

```

#include <stdio.h>

int main(void)
{
    int i;
    char c;

```

```

    for (i = 0; i < 5; ++i) {
        fprintf(stderr, "Digite um inteiro: ");
        scanf("%d", &c);
        printf ("Valor de i: %d\n", i);
    }

    return 0;
}

```

Quando esse programa é compilado com DevC++ 4.9.9.2 e executado no Windows XP, pode-se obter o seguinte<sup>162</sup>:

```

Digite um inteiro: 1
Valor de i: 0
Digite um inteiro: 2
Valor de i: 0
Digite um inteiro: 3
Valor de i: 0
Digite um inteiro: 4
Valor de i: 0
Digite um inteiro: 5
Valor de i: 0
Digite um inteiro: 6
Valor de i: 0
Digite um inteiro: 7
Valor de i: 0
Digite um inteiro: 8
Valor de i: 0
Digite um inteiro: 9
Valor de i: 0
Digite um inteiro: ^C

```

Como pode ser observado, quando o programa anterior é executado, ele entra em repetição infinita. Isto ocorre devido ao uso de um argumento na função **scanf()** que não casa com o respectivo especificador de formato. Mais especificamente, a chamada de **scanf()** contém o especificador de formato **%d** que faz com que esta função espere ler um *string* que possa ser convertido num valor do tipo **int**. Finalmente, este valor deve ser armazenado numa variável do tipo **int**. Acontece, porém, que o segundo argumento desta chamada de **scanf()** é o endereço de uma variável do tipo **char**, e uma variável deste tipo não é capaz de conter qualquer valor do tipo **int** (v. **Seção 1.2 do Volume I**). Portanto, os bytes que sobram serão armazenados no espaço em memória que segue esta variável. Como a implementação particular utilizada escolheu

---

<sup>162</sup> Negrito representa entrada de dados do usuário.

armazenar a variável `i` que controla o laço **for** exatamente nesta posição, esses bytes que sobram são armazenados no espaço reservado à variável `i`.

**Exercício:** Execute o programa anterior e, quando instado pelo programa, introduza um valor inteiro bem grande. Se o valor introduzido for suficientemente grande, você verá que o programa se encerrará imediatamente. Então, explique por que, no exemplo de execução apresentado anteriormente, o programa entrou em repetição sem fim.

Outro erro comum relacionado ao uso de **scanf()** é não checar o valor retornado por esta função. Consulte a **Seção 2.6 do Volume I** para aprender como usar corretamente o retorno de **scanf()**.

## D.5.2 USO INCORRETO DE printf()

O uso incorreto da função **printf()** para imprimir um *string* pode conduzir a um sério e famoso problema de segurança, popularmente conhecido como *bug de printf*. Por exemplo, o programa a seguir simplesmente imprime o *string* que o usuário digita na linha de comando seguindo o nome do programa quando este é executado<sup>163</sup>:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    if (2 != argc) {
        printf( "\nEste programa deve ser usado assim:\n"
               " %s <string>\n", argv[0]);
        return 1;
    }

    printf(argv[1]);

    return 0;
}
```

Os problemas com esse programa começam a surgir quando o usuário digita *strings* contendo especificadores de formato, pois, na ausência de argumentos adicionais correspondentes, a função **printf()** irá imprimir o conteúdo da pilha de execução do programa. Por exemplo, se um usuário digitar o seguinte na linha de comando do Windows XP<sup>164</sup>:

163 Nem o programa nem os ataques de hackers envolvidos em casos reais são tão simples quanto o programa apresentado aqui. Mas espera-se que o exemplo seja suficiente para ilustrar o problema.

164 Aqui, supõe-se que o nome do programa executável é `printfBug.exe`.

```
D:>printfBug "%X %X %X %X %X %X % X %X %X %X"
```

ele poderá obter o seguinte como resultado impresso na tela:

```
3E2480 3E2AE0 4012B5 77C1AEAD 1A 0 22FFA8 10 3 22FFB0
```

No exemplo de execução anterior, o suposto *hacker* usa vários especificadores `%X` para obter o conteúdo de parte da pilha de execução em formato hexadecimal. Desse modo, ele poderia obter informações importantes (e.g., nome de usuário, senha) e a segurança do sistema poderia ser violada. Além disso, um usuário malicioso poderia também escrever na pilha (usando o especificador `%n`), corrompendo-a ou forçando o programa a executar instruções malignas.

Todo o problema descrito é consequência do uso incorreto de **printf()** para imprimir um *string*. Isto é, a simples substituição da instrução:

```
printf(argv[1]);
```

por:

```
printf("%s", argv[1]);
```

resolve o problema.

Concluindo, a única maneira segura de imprimir um *string* usando **printf()** e outras funções da família `printf` é:

```
printf("%s", str); /* Seguro */
```

e nunca:

```
printf(str); /* PERIGO PARA A SEGURANÇA!!! */
```

## D.5.3 STRING DE FORMATAÇÃO INCORRETO

Erros bastante frequentes relacionados a especificadores de formato das famílias de funções `printf` e `scanf` decorrem da confusão entre os especificadores usados para os tipos de ponto flutuante, como mostra a tabela a seguir.

ESPECIFICADOR DE FORMATO PARA O TIPO...		
Família	float	double
printf	–	f
scanf	f	lf

Outros especificadores frequentemente confundidos são aqueles usados para impressão de valores dos tipos **unsigned long** e **unsigned long long**. Os especificado-

res corretos para estes tipos são, respectivamente, `%lu` (e não `%ul`) e `%llu` (e não `%ull`). O uso incorreto de `%ul` ou `%ull` como especificador de formato em chamadas de funções da família `printf` conduz a erros relativamente difíceis de perceber, pois o número é impresso como se fosse **unsigned int** e tem uma (no caso de uso de `%ul`) ou duas letras *l* (no caso de uso de `%ull`) acrescentadas ao final do valor impresso. Assim, como, dependendo da fonte utilizada na impressão, *l* (ele) parece com *l* (um), torna-se difícil enxergar o erro, como mostra o exemplo a seguir:

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

int main(void)
{
    unsigned long      umLong = ULONG_MAX;
    unsigned long long umLongLong = ULLONG_MAX;

    /* ERRADO: Não existem os      */
    /* especificadores %ul ou %ull */
    printf ("\numLong = \t%ul\n", umLong);
    printf ("umLongLong = \t%ull\n", umLongLong);

    /* CORRETO */
    printf ("\numLong = \t%lu\n", umLong);
    printf ("umLongLong = \t%llu\n", umLongLong);

    return 0;
}
```

Quando o último programa é compilado e executado no Linux, obtém-se o seguinte resultado escrito no meio padrão de saída<sup>165</sup>:

```
umLong =      4294967295l
umLongLong =   4294967295ll

umLong =      4294967295
umLongLong =   18446744073709551615
```

---

<sup>165</sup> O compilador `gcc` costuma emitir mensagens de advertência relacionadas ao uso incorreto de especificadores de formato, o que realmente faz neste caso. Portanto, o programador vai obter este resultado apenas se não der atenção a estas mensagens ou estiver usando outro compilador que não as emita, visto que o padrão de C não requer que compiladores ajam assim.

Observe como é difícil visualizar o erro quando o meio de saída utiliza a fonte Courier ou alguma outra fonte que representa graficamente a letra ele do mesmo modo que o dígito um.

Antes de usar um argumento variável para uma função da família `printf` ou `scanf`, deve-se ter certeza que ele casa com o respectivo especificador de formato. Por exemplo, suponha que haja uma constante inteira positiva, denominada `CONST_INT`, definida no início de um arquivo-fonte. Então, as seguintes chamadas de **`printf()`** serão incorretas:

```
printf("Valor da constante: %u", CONST_INT); /* Errado */
printf("Valor da constante: %lu", CONST_INT); /* Errado */
printf("Valor da constante: %llu", CONST_INT); /* Errado */
```

O problema com essas chamadas de **`printf()`** é que, apesar de o padrão ISO de C especificar como o tipo da constante deve ser determinado (v. **Seção 1.2.3 do Volume I**), a interpretação depende da largura de cada tipo inteiro e esta não é especificada pelo padrão ISO<sup>166</sup>. Portanto, se, por exemplo, a constante for interpretada como sendo do tipo **long**, a primeira chamada estará incorreta (ou, pelo menos não é portátil) porque este tipo talvez seja mais largo do que o esperado pelo especificador de formato `%u`. Raciocínio análogo pode ser usado para justificar a afirmação de que as demais chamadas de **`printf()`** são incorretas.

Uma forma correta de resolver o problema é, em primeiro lugar, garantir que a constante seja interpretada com algum tipo **unsigned** por meio de uso do sufixo `u` ou `U`. Então, pode-se escrever a chamada de **`printf()`** como:

```
printf( "Valor da constante: %llu",
        (unsigned long long) CONST_INT );
```

Nesta última chamada de **`printf()`**, a conversão explícita da constante garante que esse argumento será sempre interpretado como **unsigned long long** e, portanto, sempre irá casar com o especificador `%llu`.

## D.5.4 LIXO NO BUFFER ASSOCIADO À ENTRADA PADRÃO

Diversos problemas que aparecem num programa podem ser decorrentes do uso incorreto do meio de entrada padrão (usualmente, o teclado). A principal causa desses problemas é deixar caracteres remanescentes de uma leitura no buffer associado à

---

<sup>166</sup> Isto é, supondo que a constante tenha sido definida sem o sufixo `u` ou `U` e dependendo do valor da constante, o tipo dela pode ser interpretado como `int`, `unsigned int`, `long`, etc. Esta interpretação dependerá dos intervalos de valores (i.e., das larguras) dos tipos inteiros primitivos da implementação.

entrada padrão e não removê-los antes de iniciar uma nova leitura. A **Seção 2.6** do **Volume I** discute esses problemas e mostra como eles podem ser resolvidos. (Consulte também a **Seção D.14.2**.)

### D.5.5 NÃO EXISTE USO CORRETO PARA `gets()`

Certamente `gets()` é a função mais condenada da biblioteca padrão de C (v. **Seção 8.4.1** do **Volume I**). O problema com esta função é que ela não permite limitar o número de caracteres lidos no meio de entrada padrão, como mostra o fragmento de programa a seguir:

```
char  str[UINT_MAX]; /* Não importa o tamanho do array. */
...           /* O usuário sempre pode digitar */
gets(str);      /* um caractere a mais! */
```

O uso de `fgets()` é recomendado em substituição a `gets()`, mas lembre que, diferentemente de `gets()`, a função `fgets()` não descarta o caractere `'\n'` (v. **Seção 10.7.4**).

### D.5.6 USO INCORRETO DE EOF

A macro `EOF` é retornada por diversas funções de entrada ou saída declaradas em `<stdio.h>` para indicar uma condição de exceção, principalmente quando há uma tentativa de executar uma operação de leitura além do final de um arquivo. Conforme foi afirmado na **Seção 12.8.1** do **Volume I**, esta macro deve ser usada apenas para arquivos de texto. Mas, mesmo quando esta regra é seguida, muitos programadores não sabem como usar essa macro corretamente para determinar quando o final de um arquivo foi atingido. Por exemplo, suponha que o seguinte trecho de programa tenha sido escrito com o objetivo de ler cada caractere de um arquivo de texto e imprimi-lo no meio de saída padrão:

```
char  c;
FILE  *stream;
...
while ((c = fgetc(stream)) != EOF)
    putchar(c);
```

O problema com o trecho desse programa é que ele não funcionará se o arquivo lido contiver um byte com valor 255, como será explicado em seguida.



Em muitas implementações, o tipo **char** é **signed** (v. **Seção 13.5.3**). Também, em muitas implementações, a macro **EOF** é definida como `-1`, que corresponde a `0xFFFF`, supondo que o tipo **int** ocupe 16 bits<sup>167</sup>. Portanto, quando o valor 255 (`0xFF`) é lido e armazenado numa variável do tipo **char**, a comparação desta variável com **EOF** resulta em igualdade, porque, quando a variável do tipo **char** é comparada com **EOF**, seu valor é alargado para o tipo **int** resultando, por extensão de sinal (v. **Seção 13.5.3**), em `0xFFFF`. Deste modo, o laço do trecho desse programa pode ser encerrado antes de a função **fgetc()** retornar **EOF**.

A solução para o problema apresentado pelo trecho desse programa é simples: basta definir a variável `c` como **int**, em vez de **char**. Deste modo, quando um caractere com valor 255 for lido, ele será armazenado como `0x00FF` e a comparação com **EOF** não resultará em igualdade.

Como deve ter sido assimilado nesta seção, o uso da macro **EOF** não é tão trivial quanto aparenta. Portanto, é mais aconselhável usar as funções **feof()** e **ferror()** para testar se houve tentativa de leitura além do final de um arquivo ou se ocorreu algum outro tipo de erro, como mostra o fragmento de programa a seguir:

```
FILE *streamEntrada;
char c;
...
while (1) {
    c = fgetc(streamEntrada);

    /* Testa se o final do arquivo de entrada foi */
    /* atingido ou se ocorreu algum erro de leitura */
    if (feof(streamEntrada) || ferror(streamEntrada))
        break;
    ...
}
```

## D.5.7 USO INCORRETO DE feof()

A função **feof()** retorna um valor diferente de zero após uma tentativa de leitura além do final do *stream* que ela recebe como argumento, mas, algumas vezes, é usada incorretamente. Um exemplo de uso incorreto da função **feof()** é o trecho de programa a seguir:

```
while ( !feof(streamEntrada) )
```

---

<sup>167</sup> Um raciocínio semelhante aplica-se se o tipo **int** tiver uma largura maior.

```
putc(getc(streamEntrada), streamSaida);
```

Esse problema com o laço **while** é que a função **putc()** irá copiar indevidamente o valor retornado por **getc()** quando esta última função tentar ler além do final do arquivo. Maiores detalhes sobre este problema e sua solução são discutidos na **Seção 12.8.1** do **Volume I**.

## D.5.8 PROMPTS QUE O USUÁRIO NÃO LÊ

Quando um programa produz uma extensa saída de dados na tela, o usuário pode redirecionar a saída para um arquivo. O programador pode prever esse comportamento do usuário e enviar prompts para **stderr** em vez de **stdout**. Por exemplo, em vez de escrever:

```
printf("Digite o valor de x: ");
```

o programador deve escrever:

```
fprintf(stderr, "Digite o valor de x: ");
```

Outra situação na qual o usuário pode não ser capaz de ler aquilo que o programa escreve na saída padrão é quando o programa é abortado sem que haja tempo de descarregar o buffer associado a **stdout**. Neste caso, novamente, o programador deve usar **stderr** em vez de **stdout**, conforme exposto anteriormente, ou usar a função **fflush()** como mostra o exemplo desta função na **Seção 10.7.2**.

## D.6 CHAMADAS INCORRETAS DE FUNÇÕES

### D.6.1 SUPOSIÇÕES SOBRE ORDEM DE AVALIAÇÃO DE PARÂMETROS

A linguagem C não especifica em que ordem os parâmetros passados para uma função são avaliados. Portanto, não se deve fazer nenhuma suposição com relação a esta ordem, pois ela é dependente de implementação (v. **Seção 3.3.3** do **Volume I**). Na prática, isto significa que, se o resultado esperado numa chamada de função depende da ordem na qual os parâmetros são avaliados, partes do programa precisam ser alteradas.

Cuidado especial deve ser tomado quando chamadas de funções contêm operadores com efeitos colaterais. Por exemplo, o programa a seguir demonstra o efeito da ordem de empilhamento de parâmetros numa chamada de função:

```

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

void ImprimeInts(int x, int y)
{
    printf("\n\tPrimeiro valor: %d", x);
    printf("\n\tSegundo valor: %d", y);
}

int main(void)
{
    int x = 10;

    ImprimeInts( x, ++x );
    putchar('\n');

    return 0;
}

```

Quando o programa é compilado com empilhamento de parâmetros da direita para a esquerda, sua execução produz como resultado:

```

Primeiro valor: 11
Segundo valor: 11

```

Quando o empilhamento de parâmetros é efetuado da esquerda para a direita, o resultado do programa é:

```

Primeiro valor: 10
Segundo valor: 11

```

## D.6.2 ARGUMENTOS INCORRETOS

Quando o compilador desconhece o protótipo de uma função, é possível chamá-la usando números e tipos de parâmetros incorretos sem que o compilador perceba. Então, um erro numa chamada de tal função só seria detectado pelo *linker*, o que tornaria a correção do erro mais difícil (v. **Seção 3.3.6 do Volume I**).

Funções com listas de argumentos variáveis [e.g., **printf()**] também não permitem ao compilador checar se os parâmetros são usados corretamente<sup>168</sup>. Assim, esta categoria de funções requer cuidados especiais por parte do programador para garantir que os parâmetros passados para a função realmente correspondem aos esperados.

### D.6.3 OMISSÃO DE TESTE DE CONDIÇÃO DE EXCEÇÃO

Toda função que retorna um valor indicando ocorrência de uma condição de exceção deve ter seu valor de retorno testado antes de ser usado. Funções notáveis desta categoria são as funções de alocação dinâmica de memória e a função **fopen()**. Todas estas funções retornam **NULL** quando não conseguem atingir seus objetivos. Portanto, por exemplo, em vez de usar:

```
char *p = malloc(strlen(str) + 1);

strcpy(p, str);

use:

char *p;

if ((p = malloc(strlen(str) + 1)) != NULL)
    strcpy(p, str);
```

Em vez de usar:

```
FILE *stream = fopen("UmArquivo.txt", "r");

fgets(array, sizeof(array), stream);

use:

FILE *stream;
```

---

<sup>168</sup> O compilador gcc é capaz de emitir mensagens de advertência quando o tipo de um argumento não corresponde ao respectivo especificador de formato em chamadas de algumas funções das famílias printf e scanf, mas nem todo compilador faz isso.

```
if ((stream = fopen("UmArquivo.txt", "r")) != NULL)
    fgets(array, sizeof(array), stream);
```

Algumas funções – notadamente aquelas declaradas em `<math.h>` – sinalizam a ocorrência de erro por meio da variável global **errno** (v. **Seção 11.5**) e é igualmente importante checar se o valor desta variável foi alterado após a chamada de uma dessas funções. Por exemplo, o programador não deve escrever um trecho de programa como o mostrado a seguir:

```
double x, y;
...
y = sqrt(x); /* E se x for negativo? */
printf("Valor de y = %f", y);
```

Em vez do trecho anterior, o programador deve escrever:

```
double x, y;
...
errno = 0; /* É preciso zerar errno antes */

y = sqrt(x);

if (!errno) /* Não ocorreu erro */
    printf("Valor de y = %f", y);
```

Evidentemente, nos dois últimos fragmentos de programa poder-se-ia simplesmente testar o valor do argumento `x` e evitar a chamada de **sqrt()** quando este valor fosse negativo. Assim, a alternativa apresentada é mais viável quando o argumento passado para esta função é derivado de um cálculo complexo e deseja-se evitar efetuarlo mais de uma vez ou armazená-lo numa variável auxiliar.

O uso judicioso de macros pode ajudar na tarefa de checar a ocorrência de erros, conforme demonstrado no programa a seguir<sup>169</sup>:

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

---

<sup>169</sup> Para não correr riscos com o uso de macros, bem como melhorar a legibilidade do programa, esta macro pode facilmente ser transformada numa função.

```

#define ASSEGURA_SQRT(y, x, msg) \
    do { \
        errno = 0; \
        y = sqrt(x); \
        if (errno) {\
            fprintf(stderr, msg);\
            exit(1); \
        } \
    } while (0)

int main()
{
    double      raiz, x = 1, y = -1;
    const char* msgErro = "Ocorreu um erro ao calcular "
                          "uma raiz em main()\n";

    /* Em vez de usar as      */
    /* instruções a seguir: */
    errno = 0;
    raiz = sqrt(x);

    if (errno) {
        fprintf(stderr, msgErro);
        return 1;
    }

    printf("\nRaiz de %f = %f\n", x, raiz);

    /* Use a macro ASSEGURA_SQRT: */
    ASSEGURA_SQRT( raiz, y, msgErro);

    printf("Raiz de %f = %f\n", y, raiz);

    return 0;
}

```

## D.6.4 OMISSÃO DE TESTE EM ALOCAÇÃO DINÂMICA DE MEMÓRIA

Este tópico é um caso particular daquele apresentado na **Seção D.6.3** que merece destaque. Alguns programas fazem uso intensivo de alocação dinâmica de memória e, como sempre se deve testar se a alocação foi bem sucedida antes de prosseguir, o programador pode tornar-se entediado e simplesmente abandonar esta prática. O uso de uma macro pode facilitar o trabalho do programador se for sempre desejado que o programa reaja de certa maneira quando não é possível alocar espaço dinamicamente. A definição e uso da macro `ASSEGURA_MALLOC()` apresentada no programa a seguir ilustra este ponto.

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

#define ASSEGURA_MALLOC(p, tam, msg) \
    if (!(p = malloc(tam))) {\
        fprintf(stderr, msg);\
        exit(1); \
    }

int main()
{
    int *p;

    /* Em vez de usar as */
    /* instruções a seguir: */
    p = malloc(1000*sizeof(int));

    if (!p) {
        printf( "\nNao foi possivel alocar "
               "memoria em main()" );
        exit(1);
    }

    *p = 10;

    printf("\n*p = %d\n", *p);

    /* Use a macro ASSEGURA_MALLOC: */
}
```

```

ASSEGURA_MALLOC( p, 1000*sizeof(int),
                  "\nNao foi possivel alocar "
                  "memoria em main()" );

*p = 100;

printf("\n*p = %d\n", *p);

return 0;
}

```

### D.6.5 IMPLEMENTAÇÕES INCORRETAS DA BIBLIOTECA PADRÃO

Algumas implementações de funções da biblioteca padrão de C não seguem as especificações do padrão ISO. E isto ocorre até mesmo com implementações bem notáveis, como algumas versões da biblioteca GNU glib. Exemplos de funções implementadas em desacordo com o padrão de C em versões relativamente recentes da biblioteca glib incluem **snprintf()** e **vsnprintf()**<sup>170</sup>.

Quando utilizar uma função de alguma biblioteca cujo comportamento não coincide com o que é especificado pelo padrão ISO de C, consulte FAQs, newsgroups, etc. na internet para verificar se essa função foi realmente implementada de acordo com o padrão ISO de C.

### D.6.6 CHAMADAS DE FUNÇÕES SEM PARÂMETROS

Em algumas linguagens de programação (e.g., Pascal, Basic), uma função sem parâmetros é chamada usando-se apenas o nome da função. Em C, não é ilegal usar isoladamente o nome de uma função definida com ou sem parâmetros, mas o significado tem interpretação bem diferente daquele de algumas outras linguagens. Por exemplo:

```

void UmaFuncaoSemParametros()
{

```

---

<sup>170</sup> A documentação da biblioteca glib reconhece e classifica como broken a implementação de uma dada função que não esteja em conformidade com o padrão.



```

}
...
UmaFuncaoSemParametros;

```

Essa última instrução é perfeitamente legal em C, mas desprovida de significado prático. Quer dizer, em C, a instrução:

```
UmaFuncaoSemParametros;
```

significa que se está avaliando o endereço da função `UmaFuncaoSemParametros()` e simplesmente desprezando o resultado (v. **Seção 10.5** do **Volume I**). Em algumas outras linguagens de programação, uma instrução equivalente a essa seria interpretada como uma chamada de função, mas, em C, não é isso que ocorre.

## D.6.7 ALUSÕES E PONTEIROS PARA FUNÇÕES SEM PROTÓTIPOS

Funções devem ser declaradas usando-se protótipos de modo a permitir que o compilador seja capaz de checar passagens de parâmetros. A mesma recomendação é válida com relação a ponteiros de funções. Por exemplo:

```
extern void Troca(int *, int *);
```

é melhor do que:

```
extern void Troca();
```

De modo semelhante:

```
int (*pf)(float);
```

é melhor do que:

```
int (*pf)();
```

Nos dois casos, para evitar interpretação ambígua, utilize **void** para indicar ausência de parâmetros.

## D.7 PRÉ-PROCESSADOR

### D.7.1 ARQUIVOS DE PROGRAMA NÃO DEVEM SER INCLUÍDOS

A inclusão de arquivos de programa (i.e., aqueles com extensão `.c`) em outros arquivos pode até não causar nenhum problema, mas não faz absolutamente nenhum sentido. Os únicos arquivos que devem ser incluídos em outros arquivos que constituem um programa multiarquivo são arquivos de cabeçalho (v. **Capítulo 4** do **Volume I**).

### D.7.2 INCLUSÃO MÚLTIPLA DE ARQUIVOS

A inclusão múltipla de um arquivo de cabeçalho pode causar conflito de identificadores ou, pelo menos, retardar a compilação de um programa. Para evitar este tipo de problema, use diretivas de compilação condicional, como mostrado na **Seção 5.5** do **Volume I**.

### D.7.3 INCLUSÃO RECURSIVA DE ARQUIVOS

Inclusão recursiva de arquivos ocorre quando um arquivo de cabeçalho inclui outro arquivo de cabeçalho e este segundo arquivo inclui o primeiro. Por exemplo:

Conteúdo de `arquivo1.h`:

```
#include arquivo2.h
...
```

Conteúdo de `arquivo2.h`:

```
#include arquivo1.h
...
```

O que ocorre numa situação como essa é que o compilador (ou, mais precisamente, o pré-processador) terminaria desistindo de incluir os arquivos após algumas tentativas e o programa não seria compilado. Ao final dessas tentativas, o compilador poderá emitir uma mensagem como:

```
#include nested too deeply
```

Se o programa usar diretivas de compilação condicional, como aquelas descritas na **Seção 5.5** do **Volume I**, o problema será resolvido.

## D.7.4 DEFINIÇÕES DE TIPOS USANDO #define

Um problema decorrente do uso de **#define** em vez de **typedef** para definição de tipos é ilustrado a seguir:

```
#define tPonteiroParaChar char *
...
tPonteiroParaChar p1, p2;
```

No trecho de programa anterior, após a expansão da macro `tPonteiroParaChar`, a última linha do trecho do programa ficaria assim:

```
char *p1, p2;
```

Muito provavelmente, o resultado obtido não era aquele esperado pelo programador. Para evitar este tipo de problema, use **typedef** para definir tipos e nunca use **#define** com este propósito (v. **Seção 4.6** do **Volume I**).

## D.7.5 DEFINIÇÕES INCORRETAS DE MACROS

A **Seção 5.3.11** do **Volume I** discute em detalhes os erros frequentemente cometidos em definições de macros. Resumidamente, são eles:

- Uso de ponto e vírgula para terminar uma definição de macro
- Falta de parênteses em torno de cada argumento da macro
- Falta de parênteses em torno de uma expressão que constitui o corpo da macro
- Uso repetido de um argumento no corpo da macro

## D.7.6 CHAMADAS INCORRETAS DE MACROS

Uma macro não deve ser invocada usando um argumento que sofra a ação de um operador com efeito colateral, pois o efeito pode ser aplicado mais de uma vez no argumento (v. **Seção 5.3.11** do **Volume I**).

O uso incorreto de uma macro pode ocorrer também por falta de conhecimento de que uma dada função de biblioteca pode ter sido implementada como macro (v. **Seção 1.4** neste volume).

## D.8 PONTEIROS

### D.8.1 PONTEIROS NÃO INICIADOS

Existem vários sintomas que um programa pode apresentar em decorrência do uso de um ponteiro não iniciado:

- O programa pode ser abortado enquanto executa uma função que com certeza não contém *bugs* (e.g., uma função de biblioteca). Neste caso, um ponteiro não iniciado pode ter corrompido o código da função.
- Uma função é chamada, mas nunca inicia sua execução ou nunca retorna. Pode ser que um ponteiro não iniciado tenha corrompido a pilha de execução.
- O programa ora funciona corretamente, ora não funciona. Talvez um ponteiro não iniciado esteja corrompendo aleatoriamente partes do programa, dependendo do valor aleatório recebido pelo ponteiro.

Para evitar erros decorrentes de ponteiros não iniciados, o programador deve, antes de usar qualquer ponteiro, perguntar a si mesmo: *Para qual variável esse ponteiro está apontando?* Por exemplo, antes de chamar a função **strcpy()** como no trecho de programa a seguir:

```
strcpy(str, "bola");
```

pergunte-se: *Para onde o ponteiro str está apontando?*

Se você não souber responder a esta pergunta, provavelmente o ponteiro não deve estar apontando para um endereço válido.

### D.8.2 PONTEIROS ÓRFÃOS

Um **ponteiro órfão** é um ponteiro que aponta para uma posição válida em memória, mas cujo espaço associado está livre para alocação a qualquer instante. Em outras palavras, um ponteiro órfão aponta para um zumbi (v. **Seção D.4.2**).

É importante notar que zumbis também ocorrem em situações que não envolvem retorno de funções. Como mostra o exemplo a seguir:

```
int *p = NULL;

...
{
    int x = 10; /* x é uma variável local a este */
                /* bloco e tem duração automática */
    p = &x; /* Até aqui, tudo bem */
}
```

Nesse trecho de programa, o problema aparece a partir do fecha-chaves, pois, neste ponto, a memória alocada para `x` é liberada, o que faz com que o ponteiro `p` passe a apontar para um zumbi.

Observe agora o seguinte exemplo:

```
int p1 = malloc(sizeof(int));
int p2 = p1;
...
free(p1);
```

Neste último fragmento de programa, deve parecer óbvio para qualquer programador de C que, após a chamada de `free()`, `p1` torna-se um ponteiro inválido. Mas talvez menos óbvio seja o fato de `p2` também tornar-se um ponteiro inválido.

Na **Seção 11.2.4** do **Volume I**, foi sugerido o uso de uma macro, denominada `FREE()`, em substituição do uso direto da função `free()`. Esta macro simplesmente chama a função `free()` e, em seguida, torna nulo o ponteiro usado como argumento. Deste modo, o objetivo desta macro é transformar um possível erro lógico num erro de execução<sup>171</sup>. Infelizmente, para o caso apresentado no exemplo mais recente, o uso desta macro seria eficaz apenas se houvesse tentativa de uso do ponteiro `p1`, mas este não seria o caso com o ponteiro `p2`. Por exemplo, a última instrução do trecho de programa a seguir:

```
int p1 = malloc(sizeof(int));
```

---

<sup>171</sup> Conforme foi discutido no **Capítulo 6** do **Volume I**, erros de execução são mais fáceis de ser resolvidos do que erros lógicos.

```
int p2 = p1;
...
FREE(p1);
*p1 = 10;
```

causaria o aborto imediato do programa, visto que a macro `FREE()` atribui **NULL** a `p1`. Entretanto, se a instrução a seguir fosse acrescentada ao final do último trecho de programa:

```
*p2 = 10;
```

ela seria considerada perfeitamente legal, mas poderia causar um erro lógico mais adiante no programa.

É importante salientar ainda que, quando se têm dois ponteiros apontando para um mesmo espaço alocado dinamicamente, como foi o caso do último exemplo, não se deve chamar a função **free()** [nem a macro `FREE()`] para cada ponteiro. Isto é, esta função (ou macro) deve ser chamada apenas para um deles. Aliás, é importante lembrar aqui que o ponteiro passado como parâmetro para a função **free()** [ou a macro a `FREE()`] deve estar apontando para o *início do bloco alocado*.

### D.8.3 PONTEIRO INCREMENTADO PASSA A APONTAR PARA OUTRO ENDEREÇO

Poucas afirmações são tão óbvias mesmo para um programador iniciante em C quanto o título desta seção. Infelizmente, até programadores mais experientes se esquecem dessa evidente assertiva. Observe a seguinte função que tenta imitar a função **strcpy()** (v. **Seção 6.3.4**) da biblioteca padrão:

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

/****
 *
 * CopiaString(): Esta função deveria imitar strcpy()
 *                mas está implementada incorretamente
 *
 * Argumento: destino (saída) - o array que recebe
 *                a cópia
 *                origem (entrada) - o string que é copiado
 *
 * Retorno: string contendo a cópia (i.e., deveria ser)
```

```

*
****/
char *CopiaString(char *destino, const char *origem)
{
    while (*destino++ = *origem++)
        ; /* Intencionalmente vazio */

    /* A cópia do string foi perfeita,      */
    /* mas o retorno é um desastre porque */
    /* o ponteiro 'destino' não está       */
    /* apontando para o string copiado no  */
    /* array (primeiro parâmetro).         */
    return destino;
}

int main()
{
    char *p, str[30];

    /* Usar a função CopiaString() */
    /* como a seguir não apresenta */
    /* nenhum problema:             */
    CopiaString(str, "bola");

    printf("\nString copiado: \"%s\"", str);

    /* Usar a função CopiaString() */
    /* como a seguir representa    */
    /* um problema:                 */
    p = CopiaString(str, "problema");

    printf("\nString copiado: \"%s\"", p);

    /* O pior ocorre agora: o resultado da */
    /* concatenação caberia confortavelmente */
    /* no array str se p apontasse para o   */
    /* início do array, mas não é o caso    */
    strcat(p, "com Tiranossauro Rex");

    printf("\nString concatenado: \"%s\"\n", p);

    return 0;
}

```

Quando compilado e executado no Windows XP, o programa anterior imprime o seguinte no meio de saída padrão<sup>172</sup>:

```
String copiado: "bola"
String copiado: "    Ò--+wŧ@-wP$>"
String concatenado: "F+w"
```

O problema nesse programa é que a função `CopiaString()` incrementa o ponteiro recebido como primeiro argumento e, ao final, retorna este ponteiro que, neste instante, está apontando para o próximo byte além do final do *string* copiado. A correção deste problema consiste simplesmente em guardar o endereço inicial do array que recebe a cópia, como mostrado a seguir:

```
char *CopiaString2(char *destino, const char *origem)
{
    char *inicio = destino;

    while (*destino++ = *origem++)
        ; /* Intencionalmente vazio */

    return inicio;
}
```

Outro exemplo de retorno incorreto de um ponteiro incrementado é o seguinte programa, que contém uma função que duplica um *string*:

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

/****
 *
 * DuplicaString(): Cria um clone de um string
 *
 * Argumento: destino (saída) - o array que recebe
 *                  a cópia
 *             origem (entrada) - o string que é copiado
 *
 * Retorno: ponteiro para o clone ou NULL se for
 *           impossível alocar o espaço necessário
```

---

<sup>172</sup> A segunda e a terceira linhas impressas podem ser diferentes em diferentes execuções do programa. O importante é notar que elas simplesmente não fazem nenhum sentido.



```

*
****/
char *DuplicaString(const char *origem)
{
    char *destino;

    if ((destino = malloc(strlen(origem) + 1)) == NULL)
        return NULL;

    while (*destino++ = *origem++)
        ; /* Intencionalmente vazio */

    return destino;
}

int main()
{
    char *p;

    /* O ponteiro retornado pela função */
    /* DuplicaString() não aponta para */
    /* o início da duplicata           */
    p = DuplicaString("bola");

    /* Deve imprimir lixo, mas */
    /* não aborta o programa    */
    printf("\nString clonado: \"%s\"", p);

    /* Desastre total: para liberar */
    /* um bloco alocado dinamicamente */
    /* deve-se passar para free() um */
    /* ponteiro para o início do bloco */
    free(p);

    return 0;
}

```

Quando compilado e executado no Windows XP, esse programa imprime o seguinte no meio de saída padrão:

```
String clonado: ""
```

Quando compilado e executado no Linux Ubuntu 8.10, esse programa, além de imprimir o mesmo resultado, é abortado.

O problema apresentado pela função `DuplicaString()` do último exemplo é idêntico àquele apresentado pela função `CopiaString()` do exemplo anterior. Contudo, o problema decorrente do uso do ponteiro retornado por `DuplicaString()` na função `main()` é diferente, pois ele é passado para a função `free()` sem estar apontando para o início de um bloco alocado dinamicamente.

A diferença de resultado nos dois sistemas operacionais nos quais o programa anterior foi testado mostra que se tem em mãos um problema de portabilidade. Afinal, o padrão ISO de C afirma que, quando a função `free()` é usada com um ponteiro que não aponta para o início de um bloco alocado dinamicamente, o resultado é indefinido (v. **Seção 13.2**). Quando o programa é executado no Windows XP, ele não é abortado, mas, com certeza, a função `free()` não libera nenhum espaço como seria desejado.

## D.8.4 INDIREÇÃO DE PONTEIRO NULO

Aplicar o operador de indireção a um ponteiro nulo causa o aborto de um programa e todo programador de C deve conhecer este fato. O que alguns programadores não percebem é que algumas funções da biblioteca padrão aplicam este operador a um ponteiro recebido como argumento sem antes testá-lo para saber se é nulo. Por exemplo, o programa a seguir mostra o que ocorre quando a função `strcpy()` é chamada com um ponteiro nulo:

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

int main(void)
{
    char *p = NULL, ar[10];

    strcpy(ar, p);

    return 0;
}
```

O programa é abortado porque a função `strcpy()` aplica o operador `*` a ambos os ponteiros recebidos como argumentos sem antes testar se algum deles é nulo.

## D.9 ARRAYS E STRINGS

### D.9.1 DESRESPEITO AOS LIMITES DE ARRAYS

Desrespeitar os limites de um array ocorre com frequência em laços de repetição, como mostra o próximo exemplo :

```
#include <stdio.h>

int main(void)
{
    int ar[5], i;

    for (i = 1; i <= 5; ++i) {
        ar[i] = 1;
        printf("\tar[%d] = %d\n", i, ar[i]);
    }

    return 0;
}
```

O erro no exemplo é provavelmente derivado do fato de o programador, intuitivamente, imaginar que a indexação de arrays em C começa em 1 (como seria mais natural) e não em zero (porque é mais eficiente). Em algumas implementações, o programa anterior poderá entrar em repetição sem fim se a variável *i* for armazenada na próxima posição em memória seguindo o array *ar*[].

É importante notar que simplesmente consultar um valor que está além das fronteiras de um array não é problemático, apesar de não fazer sentido. O problema ocorre quando se altera tal valor, como mostra o seguinte trecho de programa:

```
int *p, ar[5] = {1, 2, 3, 4, 5};

p = ar + 10; /* p aponta para além dos limites do array */

/* A instrução a seguir não faz sentido, */
/* mas não há maiores consequências para */
/* o funcionamento do programa.          */
printf("Conteudo apontado por p = %d\n", *p);

*p = 0; /* Catastrófico */
```

*Strings* também são arrays e o mesmo cuidado deve ser tomado para que as fronteiras de um *string* não sejam ultrapassadas. Considere, por exemplo, a seguinte função `RemoveBranços()` definida no programa a seguir e cujo objetivo é remover os espaços em branco finais de um *string*.

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

/****
 *
 * RemoveBranços(): Remove espaços em brancos
 *                  no final de um string
 *
 * Argumento: str (saída) - o string cujos espaços
 *                  serão removidos
 *
 * Retorno: o string sem espaços em branco finais
 *
 ****/
char *RemoveBranços(char *str)
{
    char *p = strchr(str, '\0');

    /* p está apontando para o          */
    /* caractere terminal do string */

    /* Faz p apontar para o último caractere */
    p--;

    /* Substitui cada caractere considerado */
    /* espaço em branco por '\0'           */
    while(isspace(*p))
        *p-- = '\0';

    return str;
}

int main(void)
{
    char s1[] = "bola      ";
    char s2[] = "          ";
    char s3[] = "          ";
```

```

printf("String \"%s\" sem espacos finais: ", s1);
printf("\"%s\"\n", RemoveBrancos(s1) );

printf("String \"%s\" sem espacos finais: ", s2);
printf("\"%s\"\n", RemoveBrancos(s2) );

/* O programa será certamente abortado */
/* antes de chegar aqui.                */

printf("String \"%s\" sem espacos finais: ", s3);
printf("\"%s\"\n", RemoveBrancos(s3) );

return 0;
}

```

A função `RemoveBrancos()` funciona bem quando o *string* recebido como argumento contém caracteres seguidos de espaços em branco [e.g., o *string* `s1` definido em `main()`], mas pode causar o aborto do programa se o *string* recebido contiver apenas espaços em branco [e.g., o *string* `s2` definido em `main()`] ou for vazio [e.g., o *string* `s3` definido em `main()`]. O problema nestes dois últimos casos é que a função `RemoveBrancos()` poderá alterar bytes localizados antes do início do *string* recebido como argumento.

**Exercício:** Reescreva a função `RemoveBrancos()` de tal modo que, se ela receber um *string* contendo apenas espaços em branco ou for vazio, ela retorne um *string* vazio.

Quando uma variável definida logo antes ou depois de um array assume valores inesperados, deve-se suspeitar de corrupção de memória causada por acesso indevido além dos limites do array. Por exemplo, considerando as definições de variáveis a seguir:

```

int    i;
float  ar[10];
double d

```

Se as variáveis `i` e `d` aqui definidas assumirem eventualmente valores estranhos, um possível suspeito será o acesso ao array `ar[]`. Isto é, se a variável `i` estiver assumindo valores imprevistos, talvez posições em memória que antecedem o limite inferior do array estejam sendo alteradas. Por outro lado, se o fato ocorre com a variável `d`, talvez o limite superior do array esteja sendo ultrapassado e alterado.

## D.9.2 STRINGS CONSTANTES DEVEM SER CONSIDERADOS CONSTANTES

O programador deveria respeitar *strings* constantes assim como respeita (compulsoriamente) constantes numéricas. Infelizmente, erros decorrentes do desrespeito a essa regra ocorrem por mera ignorância por parte do programador.

Não custa nada repetir o que foi afirmado na **Seção 8.2** do **Capítulo I**: muitas implementações armazenam *strings* constantes em porções de memória usadas apenas para leitura. Portanto, tentar alterar o conteúdo de um *string* constante pode causar o aborto do programa.

Para permitir que o compilador detecte um provável erro antes que ele ocorra durante a execução do programa, acostume-se a declarar ponteiros para *strings* constantes com **const**, como mostrado a seguir:

```
const char *s = "Bola";
...
*s = 'C'; /* O compilador aponta o erro */
```

No exemplo, o compilador aponta o erro na instrução que tenta alterar o conteúdo do *string* constante devido ao uso de **const** na declaração da variável *s*. Sem o uso de **const**, talvez o programa fosse abortado e o programador nem imaginaria por qual razão.

## D.9.3 COMPARAÇÃO INCORRETA DE STRINGS

Considere o seguinte trecho de programa:

```
char *s1 = "bola";
char s2[] = "bola";

if (s1 == s2)
    printf("Os strings s1 e s2 sao iguais");
```

A chamada de **printf()** nesse trecho de programa só seria executada se, antes da instrução **if**, o ponteiro *s1* estivesse apontando para o *string* *s2*. Em outras palavras, a instrução **if** testa se dois ponteiros apontam para a mesma posição em memória, e não se o conteúdo dos *strings* apontados são iguais. Portanto, se o objetivo for determinar

se dois *strings* de caracteres monobytes são iguais ou diferentes, deve-se usar uma das seguintes funções declaradas em `<string.h>` (v. **Seção 6.3**)<sup>173</sup>:

- **strcmp()**
- **strncmp()**
- **memcmp()**

Considerando situação semelhante com *strings* extensos, devem-se usar as funções correspondentes declaradas em `<wchar.h>` e descritas na **Seção 8.5**.

Se a comparação tiver como objetivo ordenar *strings*, a melhor escolha é uma das funções a seguir (v. **Seção 6.4**):

- **strcoll()**
- **strxfrm()** [em conjunto com **strcmp()**]

Se, neste último caso, forem usados *strings* extensos, deve-se usar **wscoll()** ou **wcsxfrm()** em conjunto com **wscmp()** (v. **Seção 8.5**).

## D.9.4 STRINGS CONSTANTES SEM ACESSIBILIDADE

Quando se atribui um *string* constante a um ponteiro, este torna-se o único meio de acesso ao *string*. Se, subsequentemente, o mesmo ponteiro for associado a outro endereço, perde-se este meio de acesso apesar de o *string* ainda encontrar-se armazenado em memória. Assim, tem-se mais um caso de vazamento de memória, pois o *string* continuará ocupando espaço em memória inutilmente, já que jamais ele poderá ser acessado novamente. Por exemplo:

```
char *p = "pitomba"; /* p torna-se o único meio de */
                      /* acesso ao string "pitomba" */

...

p = "caju"; /* p deixa de apontar para "pitomba" */
           /* e passa a apontar para "caju"      */
```

---

<sup>173</sup> No caso do exemplo apresentado, a função mais adequada seria **strcmp()**.

No trecho de programa anterior, após a segunda atribuição feita ao ponteiro `p`, o *string* "pitomba" continua armazenado em memória, mas não poderá mais ser acessado, gerando, assim, um desperdício de memória.

## D.9.5 USO DE `sizeof` EM VEZ DE `strlen()`

O operador **`sizeof`** não deve jamais ser usado em substituição a **`strlen()`** para calcular o comprimento de um *string*. Por exemplo:

```
char *p = "bola";
char s1[80] = "bola";
char s2[] = "bola";
size_t t;

t = sizeof(p); /* Resulta no número de bytes num ponteiro */
t = sizeof(s1); /* Resulta no número de bytes no array s1 */
t = sizeof(s2); /* Resulta no número de bytes no array s2 */
```

Em nenhuma das instruções anteriores, o resultado é aquele esperado (i.e., 4). Portanto, nunca use **`sizeof`** para calcular o número de caracteres visíveis num *string*.

## D.9.6 FUNÇÕES QUE NÃO LIMITAM O NÚMERO DE CARACTERES ESCRITOS

Muitas funções declaradas nos cabeçalhos `<string.h>` [e.g., **`strcpy()`**] e `<wchar.h>` [e.g., **`wcscpy()`**] não têm capacidade de limitar o número de caracteres escritos num array. Mas existem funções equivalentes nesses mesmos cabeçalhos [e.g., **`strncpy()`**, **`wcsncpy()`**] que possuem essa capacidade. Para evitar possível corrupção de memória, o programador deve dar preferência a funções de processamento de *strings* que limitam o número de caracteres escritos no array que armazena o resultado da operação. Por exemplo, em vez de usar a função **`strcpy()`**, como no trecho de programa:

```
char s[10];
...
strcpy(s, "um longo string que corrompe memoria");
```

use a função **`strncpy()`**:



```
char s[10];
...
strncpy( s, "um longo string que corrompe memoria",
        sizeof(s) );
```

O resultado da execução do último trecho de programa pode até não satisfazer o programador, mas, pelo menos, não haverá corrupção de memória, como ocorre com o trecho de programa precedente.

Algumas funções usadas em formatação em memória (v. **Seção 10.7.8**) declaradas em `<stdio.h>` [e.g., **sprintf()**] também apresentam a mesma possibilidade de corromper memória. Novamente, estas funções possuem equivalentes que limitam o número de caracteres escritos [e.g., **snprintf()**].

## D.9.7 FUNÇÕES QUE NEM SEMPRE PRODUZEM STRINGS

Algumas funções declaradas nos cabeçalhos `<string.h>` [e.g., **strncpy()**] e `<wchar.h>` [e.g., **wcsncpy()**] nem sempre incluem o caractere terminal de *string* no resultado das operações. Portanto, para garantir que o resultado destas operações seja realmente um *string*, pode ser necessário inserir um caractere `'\0'` no resultado da operação. Por exemplo, considere o programa a seguir:

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

#define TAMANHO 12

int main()
{
    char str[TAMANHO];

    strncpy(str, "Um string grande", TAMANHO);

    printf( "String apos chamada de strncpy():"
           "\n\t \"%s\"\n", str );

    return 0;
}
```

Quando este programa é executado, ele pode imprimir:

```
String apos chamada de strncpy():
    "Um string gr; "
```

A segunda linha impressa pelo programa pode variar entre uma execução e outra do programa. Mas, de qualquer modo, é importante notar que os caracteres após *gr*, ao final desta linha, representam bytes aleatórios encontrados em memória e que a impressão de caracteres continua até que seja encontrado um byte nulo. Isso ocorre porque o resultado armazenado no array *str* pela função **strncpy()** não contém o caractere terminal `'\0'` e, portanto, o array *str* não contém um *string*. Este problema pode ser resolvido acrescentando-se a instrução:

```
str[TAMANHO - 1] = '\0';
```

logo após a chamada de **strncpy()**.

## D.9.8 STRINGS CONSTANTES VERSUS CARACTERES CONSTANTES

A **Seção 8.3** do **Volume I** discute alguns enganos frequentes cometidos por iniciantes em C quando lidam com caracteres isolados e *strings*. Alguns exemplos apresentados naquela seção são:

```
char *p;
...
*p = "A"; /* ERRADO: *p deveria receber um caractere e */
          /* não um ponteiro para um string constante */

p = 'A'; /* ERRADO: p deveria receber um ponteiro */
          /* para char e não um valor do tipo char */
```

Confusões entre iniciação e atribuição de *strings* também são comuns entre iniciantes em C:

```
char *p = "String"; /* OK: Quem está recebendo */
                    /* um valor é p e não *p */

...
*p = "String"; /* ERRADO: Quem está recebendo */
               /* um valor é *p e não p */
```

## D.9.9 ALOCAÇÃO DE ESPAÇO INSUFICIENTE PARA CONTER UM STRING

Conforme foi enfatizado na **Seção 8.4 do Volume I**, arrays usados para armazenar o resultado de alguma operação com *strings* devem ter capacidade suficiente para conter todos os caracteres resultantes da operação (inclusive '`\0`', se for o caso). Infelizmente, muitas vezes, o programador se esquece desta regra. Considere como exemplo:

```
char *p;
...
strcpy(p, "Bola");
```

Este é o pior erro de todos dessa categoria, pois o local (aleatório) para onde o ponteiro `p` aponta em memória não pode receber um único caractere sequer. Mesmo assim, este tipo de erro é muito comum entre alunos de programação.

Outro exemplo:

```
char *p = malloc(strlen("Bola"));
...
strcpy(p, "Bola");
```

Este erro, aparentemente, não é tão mal quanto o anterior, porque, aqui, o programador se esqueceu apenas de alocar espaço para o caractere terminal de *string* ('`\0`').

De qualquer modo, qualquer dos exemplos apresentados pode causar o aborto do programa devido à corrupção de memória.

## D.10 ALOCAÇÃO DINÂMICA DE MEMÓRIA

### D.10.1 ZUMBIS TAMBÉM ASSOMBRAM O HEAP

Se uma função aloca memória e retorna um ponteiro para o espaço alocado, ela não deve chamar **free()** antes de retornar. Ou seja, a responsabilidade de liberar o espaço alocado passa a ser da função que faz a chamada. Por exemplo:

```

char *Funcao1(const char *str)
{
    char *p = malloc(strlen(str) + 1);
    ... /* Executa algum processamento */
        /* usando o bloco alocado          */

    free(p); /* ERRADO: gestação de um zumbi */

    return p; /* Retorno de zumbi */
}

char *Funcao2(const char *str)
{
    char *pAux = Funcao2("Boi");

    ...

    free(pAux); /* CORRETO: quem chama uma função que aloca      */
                /* espaço dinamicamente tem a responsabilidade */
                /* de liberar o espaço alocado.                  */
}

```

Diferentemente do que ocorre quando se usa uma variável de duração automática cujo endereço é retornado por uma função (v. **Seção D.4.2**), quando se continua usando um espaço alocado dinamicamente após o mesmo ter sido liberado com **free()**, o efeito pode manifestar-se antes mesmo de o conteúdo do *heap* ser alterado explicitamente, como mostra o exemplo a seguir.

```

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

#define TAMANHO_ZUMBI 5

int *RetornaZumbi2(void)
{
    int i, *zumbi = malloc(TAMANHO_ZUMBI*sizeof(int));

    if (!zumbi)
        return NULL;

    for (i = 0; i < TAMANHO_ZUMBI; ++i)
        zumbi[i] = i + 1;
}

```

```

    printf( "\nConteudo do array zumbi em "
            "RetornaZumbi2():\n" );

    for (i = 0; i < TAMANHO_ZUMBI; ++i)
        printf("\tzumbi[%d] = %d\n", i, zumbi[i]);

    free(zumbi);

    return zumbi;
}

int main(void)
{
    int *pInt, i, *p, soma = 0;

    pInt = RetornaZumbi2();

    /* Até aqui está tudo bem, pois */
    /* o heap ainda não foi alterado */
    /* e o zumbi continua vivo.      */
    for (i = 0; i < TAMANHO_ZUMBI; ++i)
        soma += pInt[i]; /* Ainda está OK */

    printf("\nResultado da soma em main(): %d\n", soma);

    p = calloc(500000000, sizeof(int));

    printf("\nConteudo do array zumbi em main():\n");
    for (i = 0; i < TAMANHO_ZUMBI; ++i)
        printf("\tpInt[%d] = %d\n", i, pInt[i]);

    return 0;
}

```

O resultado da execução desse programa quando compilado e executado no Windows XP é:

```

Conteudo do array zumbi em RetornaZumbi2():
    zumbi[0] = 1
    zumbi[1] = 2
    zumbi[2] = 3

```

```

zumbi[3] = 4
zumbi[4] = 5

```

*Resultado da soma em main(): 14*

*Conteúdo do array zumbi em main():*

```

pInt[0] = 0
pInt[1] = 2
pInt[2] = 3
pInt[3] = 4
pInt[4] = 5

```

Note que o resultado do último programa mostra que, diferentemente do que se possa imaginar, o efeito zumbi manifesta-se sem sequer ter havido nova alocação de memória no *heap*<sup>174</sup>.

O programador inexperiente poderia imaginar que depurar o erro apresentado pelo programa anterior seria mais fácil do que aquele exibido pelo último programa apresentado na **Seção D.4.2**. Afinal, os resultados mostrados aqui são tão próximos do esperado. Por outro lado, um programador mais experiente certamente acharia mais fácil prever a causa do erro apresentado pelo programa da **Seção D.4.2**, porque erros tão grotescos quanto aqueles certamente são causados por ponteiros órfãos.

## D.10.2 USO INCORRETO DE `free()`

Existem alguns erros comuns decorrentes do uso incorreto da função **`free()`**:

- O ponteiro usado como argumento de **`free()`** não aponta para o início de um bloco alocado dinamicamente com **`malloc()`**, **`calloc()`** ou **`realloc()`** (v. exemplo na **Seção D.8.3**).
- O ponteiro usado como argumento de **`free()`** já foi usado numa chamada anterior desta função sem que houvesse uma nova alocação entre as duas chamadas de **`free()`**. Por exemplo:

---

<sup>174</sup> Alguns textos sobre programação em C afirmam que o efeito indesejado só se manifesta quando há uma nova alocação, mas o padrão de C apregoa que o comportamento é indefinido. Portanto, o problema pode aparecer a qualquer instante.

```
int p1 = malloc(sizeof(int));
int p2 = p1;
...
free(p1);
free(p2);
```

O problema com o trecho do programa exemplificado é que tanto `p1` quanto `p2` apontam para o mesmo bloco alocado dinamicamente. Assim, a primeira chamada de **free()** está perfeitamente correta, mas a segunda chamada não está porque o bloco apontado pelo ponteiro `p2` já foi liberado na primeira chamada.

Em qualquer dos casos resumidos aqui, de acordo com o padrão ISO de C, o resultado é indefinido (v. **Seção 13.2**).

## D.10.3 LISTA ENCADEADA NÃO É ARRAY

Os elementos de uma lista encadeada não são necessariamente adjacentes<sup>175</sup>. Portanto, eles não podem ser acessados por meio de índices ou incrementos de ponteiros como ocorre com arrays. Logo, não caia na tentação de substituir uma instrução como<sup>176</sup>:

```
p = p->proximo; /* Modo correto de acessar o próximo */
               /* elemento de uma lista encadeada   */
```

por:

```
p++; /* Modo ERRADO de acessar o próximo elemento da lista */
```

ou:

```
++p; /* Modo ERRADO de acessar o próximo elemento da lista */
```

ou:

```
p[2]; /* Modo ERRADO de acessar o próximo elemento da lista */
```

## D.11 OPERAÇÕES INTEIRAS

### D.11.1 OVERFLOW

A maneira mais simples e portátil de determinar se uma operação inteira resultará

---

<sup>175</sup> Se você ainda não sabe: é exatamente por isso que cada elemento armazena o endereço do próximo elemento da lista.

<sup>176</sup> Supõe-se aqui que `p` seja um ponteiro para um elemento de uma lista encadeada (v. **Capítulo 11** do **Volume I**).

em *overflow* é testar se, considerando os operandos como **unsigned**, o resultado da operação é um valor negativo. Por exemplo, se *x* e *y* são duas variáveis do tipo **int**, a instrução **if** a seguir mostra como este teste de *overflow* pode ser realizado numa operação de adição:

```
if ((int) ((unsigned)x + (unsigned)y) < 0)
    printf("Ocorrera' overflow na soma x + y");
```

Além de adição, outras operações aritméticas também podem causar *overflow*, como mostra o seguinte programa:

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

int main(void)
{
    int umInt1, umInt2;

    umInt1 = 0x40000000;

    printf("\numInt1 = %d (0x%X)\n", umInt1, umInt1);

    umInt2 = umInt1 + 0xC0000000;
    printf( "umInt1 + 0xC0000000 = %d (0x%X)\n",
            umInt2, umInt2 );

    umInt2 = umInt1 * 0x4;
    printf( "umInt1 * 0x4 = %d (0x%X)\n",
            umInt2, umInt2 );

    umInt2 = umInt1 - (signed)UINT_MAX;
    printf( "umInt1 - (signed)UINT_MAX = %d (0x%X)\n",
            umInt2, umInt2 );

    return 0;
}
```

Quando esse programa é compilado e executado no Windows XP, ele produz o seguinte resultado:



```

umInt1 = 1073741824 (0x40000000)
umInt1 + 0xC0000000 = 0 (0x0)
umInt1 * 0x4 = 0 (0x0)
umInt1 - (signed)UINT_MAX = 1073741825 (0x40000001)

```

## D.11.2 ERRO DE SINAL

Como os tipos inteiros de C, com exceção de **char**, são considerados **signed** na ausência de especificação explícita com o uso de **unsigned**, uma ocorrência de *overflow* pode causar mudança de sinal, como mostra o exemplo a seguir:

```

#include <stdio.h>
#include <limits.h>

int main(void)
{
    int umInt = INT_MAX;

    printf( "\numInt = %d (0x%X)\n",
            umInt, umInt );
    printf( "umInt + 1 = %d (0x%X)\n",
            umInt + 1 , umInt + 1);

    return 0;
}

```

Quando compilado e executado no Windows XP, o programa anterior produz o seguinte resultado:

```

umInt = 2147483647 (0x7FFFFFFF)
umInt + 1 = -2147483648 (0x80000000)

```

No último exemplo, a variável `umInt` é iniciada com **INT\_MAX**, que é o maior valor que uma variável do tipo **signed int** pode conter. Assim sendo, quando esta variável é acrescida de um, seu bit mais significativo, que representa o sinal, passa a ser 1 e seu valor é interpretado como negativo.

Um **erro de sinal** ocorre quando uma variável com sinal é interpretada como uma variável sem sinal ou vice-versa. Algumas situações comuns que podem causar erros de sinal são:

- Uso de inteiros com sinal em comparações.

- Uso de inteiros com sinal em operações aritméticas.
- Comparações entre inteiros com sinal e inteiros sem sinal.
- Atribuição de um valor com sinal a uma variável sem sinal ou vice-versa.

**Exemplo:** O programa a seguir apresenta um erro de sinal.

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

int main(void)
{
    int    i;
    short  nBytes;
    char   ar[256];

    /* Para tornar o exemplo mais realista,          */
    /* suponha que o valor 40000 é introduzido        */
    /* por um usuário do programa e que este         */
    /* valor não cabe numa variável do tipo short */
    nBytes = 40000;
    printf("\nnBytes = %d\n", nBytes);

    if(nBytes > 256) {
        fprintf( stderr, "Numero de bytes a ser alterados"
                  " e' grande demais" );
        return 1;
    }

    printf( "Numero de bytes que serao alterados:"
            " %zu\n", (size_t) nBytes );

    memset(ar, 'X', nBytes);
    ar[nBytes - 1] = '\0';

    printf("%s\n", ar);

    return 0;
}
```

Quando compilado e executado no Linux, esse programa produz o seguinte resultado:

```
nBytes = -25536
Numero de bytes que serao alterados: 4294941760
Segmentation fault
```

Após a impressão da última linha, o programa é abortado porque a função **memset()** tenta alterar posições em memória que estão bem além do final do array `ar[]`.

**Exemplo:** O programa a seguir apresenta mais um caso de erro de sinal.

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

#define TAM_ARRAY 80

char array[TAM_ARRAY];

char* CopiaString(const char *str, int nBytes)
{
    if(nBytes > TAM_ARRAY)
        return NULL;

    return memcpy(array, str, nBytes);
}

int main(void)
{
    char *p, str[TAM_ARRAY] = "Um string";

    p = CopiaString(str, -10);

    if (p)
        printf("\n%s\n", p);
    else
        printf("\nCopiaString() retornou NULL\n");

    return 0;
}
```

Quando esse programa é executado no Linux, ele apresenta um erro do tipo *segmentation fault* e é abortado.

**Exercício:** Explique por que esse último programa é abortado.

## D.12 OPERAÇÕES DE PONTO FLUTUANTE

### D.12.1 ARREDONDAMENTOS

Uma dada operação de ponto flutuante pode não resultar no valor esperado devido a arredondamento. Como exemplo de erro de arredondamento, considere o seguinte programa<sup>177</sup>:

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

int main(void)
{
    double      x;
    long double y;

    /* O valor atribuído a x certamente será zero */
    x = pow(10, 40) +
        700          -
        pow(10, 40) +
        pow(10, 45) +
        500          -
        pow(10, 45);

    printf( "\nPrimeira expressao:\n\tValor de x "
           "(double) = %f\n", x );

    /* Alterando-se a ordem das parcelas pode-se */
    /* obter o resultado correto. Por exemplo: */

    x = pow(10, 40) -
```

---

<sup>177</sup> Este programa é adaptado de um exemplo proposto em Dolenc, A. *et alii* (v. **Bibliografia**).

```

        pow(10, 40) +
        pow(10, 45) -
        pow(10, 45) +
        700          +
        500;

    printf( "\nSegunda expressao:\n\tValor de x "
           "(double) = %f\n", x );

    /* Utilizar long double não altera o resultado */
    y = powl(10, 40) +
        700          -
        powl(10, 40) +
        powl(10, 45) +
        500          -
        powl(10, 45);

    printf("\nValor de y (long double) = %Lf\n", y);

    return 0;
}

```

Quando executado no sistema Windows XP, o programa produz o seguinte resultado no meio de saída padrão:

*Primeira expressao:*

*Valor de x (double) = 0.000000*

*Segunda expressao:*

*Valor de x (double) = 1200.000000*

*Valor de y (long double) = 0.000000*

Analisando-se o resultado do último programa, nota-se que, apesar de o erro absoluto ser grande (1200), na realidade, o erro relativo é pequeno. Mesmo assim, percebe-se que o simples rearranjo de operandos numa expressão aritmética de ponto flutuante é capaz de produzir resultados bem discrepantes.

## D.12.2 COMPARAÇÕES

Devido ao modo aproximado como são armazenados, números de ponto flutuante não devem ser comparados usando o operador relacional de igualdade (v. **Seção 1.3.6 do Volume I**). Em vez disso, deve-se testar se a diferença entre os dois números se comparados é suficientemente pequena para que eles sejam considerados iguais. Por exemplo:

Em vez de usar:

```
double x, y;
...
if (x == y)
    ...
```

Use:

```
#define DELTA 0.000001
...
double x, y;
...
if ( fabs(x - y) <= DELTA )
    ...
```

Em vez de usar os operadores `==` e `!=` que, conforme mostrado anteriormente, podem produzir resultados indesejados, pode-se ainda usar uma das funções apresentadas no programa a seguir<sup>178</sup>:

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

/* O valor da constante LIMITE depende: */
/* - Do tipo de ponto flutuante utilizado */
/* - Da arquitetura da máquina utilizada */
/* - Da precisão utilizada na obtenção dos */
/*     valores utilizados */
/*                                     */
/* Aqui, o valor de LIMITE é arbitrário, mas */
/* na prática, ele deve ser determinado */
/* experimentalmente. */
#define LIMITE 0.001
#define LIMITE2 0.00001
```

---

<sup>178</sup> Esse programa é adaptado de um exemplo proposto em Dolenc, A. *et alii* (v. **Bibliografia**).

```

unsigned SaoIguais1(float exp1, float exp2, float limite)
{
    if (fabs(exp1 - exp2) <= limite)
        return 1;

    return 0;
}

unsigned SaoIguais2(float exp1, float exp2, float limite)
{
    float maior, menor;

    maior = (exp1 >= exp2) ? exp1 : exp2;
    menor = (exp1 >= exp2) ? exp2 : exp1;

    if (!maior)
        return SaoIguais1(exp1, exp2, limite);

    if (fabs(fabs(menor/maior) - 1) <= limite)
        return 1;

    return 0;
}

int main(void)
{
    float x = 2.5556,
          y = 2.555678,
          limite = LIMITE2;

    printf( "\n*** Limiar de erro utilizado: %f ***\n",
            limite );

    if (SaoIguais1(x, y, limite))
        printf("\nDe acordo com SaoIguais1(), "
               "%f e %f sao iguais\n", x, y);
    else
        printf("\nDe acordo com SaoIguais1(), "
               "%f e %f NAO sao iguais\n", x, y);

    if (SaoIguais2(x, y, limite))
        printf("De acordo com SaoIguais2(), "

```

```

        "%f e %f sao iguais\n", x, y);
    else
        printf("De acordo com SaoIguais2(), "
               "%f e %f NAO sao iguais\n", x, y);

    return 0;
}

```

Executando-se esse programa no Windows XP, obtém-se o seguinte resultado:

```
*** Limiar de erro utilizado: 0.001000 ***
```

```
De acordo com SaoIguais1(), 2.555600 e 2.555678 sao iguais
De acordo com SaoIguais2(), 2.555600 e 2.555678 sao iguais
```

```
*** Limiar de erro utilizado: 0.000010 ***
```

```
De acordo com SaoIguais1(), 2.555600 e 2.555678 NAO sao
iguais
```

```
De acordo com SaoIguais2(), 2.555600 e 2.555678 NAO sao
iguais
```

## D.12.3 OVERFLOW E UNDERFLOW

Diferentemente do que ocorre com *overflow* em operações inteiras, exceções de *overflow* e *underflow* podem ser detectadas após ocorrerem em operações de ponto flutuante. Por exemplo:

```

#include <stdio.h>
#include <float.h>
#include <math.h>

int main(void)
{
    double umDouble = 2*DBL_MAX;

    if (umDouble >= HUGE_VAL)
        printf ("\nOcorreu overflow\n");

    printf ("Valor de umDouble: %f\n", umDouble);
}

```



```
    return 0;
}
```

No programa exemplificado, foi propositalmente atribuído à variável `umDouble` um valor que não pode ser contido numa variável do tipo **double**. Como resultado, essa variável recebeu o valor **HUGE\_VAL** indicando a ocorrência de *overflow*.

Outras informações sobre como detectar condições de *overflow* ou *underflow* em operações de ponto flutuante são apresentadas no **Capítulo 3**.

## D.13 COMENTÁRIOS

### D.13.1 COMENTÁRIOS NÃO TERMINADOS

Quando o programador se esquece de fechar um comentário apropriadamente, trechos do seu programa podem ser *engolidos* pelo comentário. Por exemplo:

```
a = b; /* Este comentário não termina onde deveria e...
c = d; /* a instrução c = d está dentro do comentário */
```

Erros decorrentes de uso incorreto de comentários são comuns, mas muito fáceis de encontrar se for utilizado um editor de programas razoável. Isto é, um editor que empregue uma convenção gráfica que diferencie comentários de outras construções da linguagem, facilitando a inspeção visual de programas. Por exemplo, o editor do ambiente DevC++ apresenta comentários em fonte azul e em itálico que facilitam sua identificação visual <sup>179</sup>.

### D.13.2 COMENTÁRIOS MAL POSICIONADOS

Considere o seguinte fragmento de programa:

```
int a, b, *p;
...
a = b/*p; /* p aponta para tal e tal valor */
```

---

<sup>179</sup> Qualquer bom editor de programas permite que o usuário altere essa e outras configurações gráficas. Mas aconselha-se que o programador não faça isto com frequência para que se acostume a identificar visualmente os diversos componentes de um programa com facilidade.

O problema com esta última instrução é que a combinação de caracteres `/*` entre `b` e `p` é interpretada como abertura de comentário. Neste caso, ocorre um erro de compilação, mas poderia ter sido convertido num erro lógico se o programador tivesse colocado ponto e vírgula no final da linha:

```
int a, b, *p;
...
a = b/*p;  /* p aponta para tal e tal valor */;
```

Novamente, como foi dito na **Seção D.12.6**, usando-se um bom editor de programas, torna-se relativamente fácil identificar este tipo de erro visualmente. Por exemplo, usando o editor de programas do ambiente DevC++, a última linha do primeiro trecho de programa anterior apareceria como (note o uso de itálico):

```
a = b/*p;  /* p aponta para tal e tal valor */
```

O uso de itálico e de uma coloração diferente indica que tudo que segue `a = b` nesta linha é interpretado como comentário.

A correção desse erro é trivial: basta colocar `*p` entre parênteses ou usar espaço entre `/` e `*` antes de `p`.

### D.13.3 COMENTÁRIOS ANINHADOS

Não se devem aninhar comentários de um mesmo tipo, pois nem todo compilador aceita este tipo de construção. Se precisar remover temporariamente um trecho de programa contendo comentários, use compilação condicional, conforme mostrado a seguir:

```
#ifndef MACRO_NUNCA_DEFINIDA
... /* Trecho que se deseja excluir temporariamente */
#endif
```

## D.14 ERROS DE PORTABILIDADE

Um erro de portabilidade pode não impedir um programa de ser executado normalmente numa implementação de C específica, mas não há garantia de que ele vá funcionar noutra implementação. Erros comuns de programação associados a portabilidade são discutidos no **Capítulo 13** deste volume, mas existem três erros de

portabilidade tão comuns que merecem destaque especial neste apêndice. Estes erros serão descritos nas subseções a seguir.

### D.14.1 void main()

Conforme foi discutido no **Capítulo 8** do **Volume I**, a função **main()** pode ter dois protótipos diferentes:

```
int main(void)
```

```
int main(int argc, char *argv[])
```

Em nenhum dos protótipos da função **main()**, o padrão ISO de C especifica que o tipo de retorno desta função seja diferente de **int**. Portanto, apesar de muitas implementações permitirem o uso de **void** como tipo de retorno de **main()**, essa tolerância é uma extensão da linguagem C e, assim, não é portátil.

### D.14.2 fflush(stdin)

A função **fflush()** serve para descarregar apenas buffers associados a *streams* de saída (v. **Seção 10.7.2**). A biblioteca padrão de C não provê nenhuma função para descarregar *streams* de entrada (como é o caso de **stdin**), mas é fácil implementar uma função para limpeza do buffer associado a um stream de entrada, como é o caso da função `LimpaBuffer()` apresentada na **Seção 2.6** do **Volume I**.

### D.14.3 conio.h

O cabeçalho "`conio.h`" é uma extensão bastante comum entre as implementações de C (v. **Capítulo 12** do **Volume I**). Neste cabeçalho, são declaradas funções que realizam operações de entrada e saída que não são abrangidas pela biblioteca padrão de C. Provavelmente, as três funções deste cabeçalho mais utilizadas são:

- `getch()` – esta função lê um caractere em **stdin** sem uso de *buffering*.
- `getche()` – esta função faz o mesmo que `getch()` e ainda ecoa o caractere digitado na tela [`getch()` não ecoa].

- `clrscr()` – esta função limpa a tela.

Nem as funções declaradas em `"conio.h"`, nem este próprio cabeçalho são portáteis.

## D.15 OUTROS ERROS COMUNS

### D.15.1 USO DE VARIÁVEIS NÃO INICIADAS

Um erro bastante frequente é o uso do conteúdo de uma variável antes de ela assumir um valor conhecido; i.e., antes de ela ser iniciada. Uma variável também é considerada não iniciada quando ela é *iniciada* com uma variável não iniciada. Por exemplo, se `x` e `y` são variáveis locais, ambas serão consideradas não iniciadas no trecho de programa a seguir:

```
int x;      /* x não foi iniciada... */
int y = x; /* consequentemente, y também não é iniciada */
```

Conforme mencionado anteriormente, atribuir a uma variável um valor proveniente de variáveis não iniciadas é, obviamente, equivalente a não iniciar a variável, como mostra o exemplo a seguir:

```
#include <stdio.h>

int main()
{
    int x, y;
    int soma = x + y;

    printf("Digite dois numeros para somar: ");
    scanf("%d %d");
    printf("A soma dos numeros e' igual a: %f\n", soma);

    return 0;
}
```

Quando este programa é executado, o resultado pode ser o seguinte ou qualquer outro resultado sem sentido.

```
Digite dois numeros para somar: 1 3  
A soma dos numeros e' igual a: -1393
```

Lembre-se de que apenas variáveis de duração fixa são iniciadas automaticamente com zero e que variáveis de duração automática não são iniciadas automaticamente (v. **Seção 4.2 do Volume I**). Assim, quando uma variável assume um valor aparentemente aleatório, ela deve ser checada com cuidado para que fique assegurado de que foi devidamente iniciada.

## D.15.2 CONVERSÕES INADEQUADAS DE TIPOS

Considere a seguinte iniciação de variável:

```
double x = 1/2;
```

Se a intenção do programador é confundir os leitores de seu programa, está tudo bem com a iniciação anterior. Mas, se ele deseja que seu programa seja legível e, ao mesmo tempo, demonstrar conhecimento da linguagem C, é melhor ele substituir aquela iniciação por:

```
double x = 0;
```

Nos dois casos, o efeito é o mesmo.

Conforme foi discutido na **Seção 1.5 do Volume I**, há cinco situações nas quais as conversões automáticas podem ocorrer em C:

- Numa atribuição (incluindo iniciação de variáveis) envolvendo tipos diferentes.
- Quando os tipos inteiros **char** e **short** são usados em expressões, chamadas ou retornos de funções (conversão de alargamento).
- Numa expressão aritmética quando tipos diferentes são misturados.

- Numa passagem de parâmetro quando o tipo do parâmetro real não coincide com o tipo do parâmetro formal.
- Num retorno de função quando o tipo do valor retornado não coincide com o tipo de retorno declarado no cabeçalho da função.

Em qualquer das situações enumeradas aqui podem ocorrer erros devidos a conversões automáticas<sup>180</sup>. A **Seção 1.5** do **Volume I** apresenta diversos exemplos de erros de programação decorrentes de misturas de tipos e conversões automáticas.

Resumindo, o melhor conselho que se pode oferecer é que se deve evitar, sempre que possível, as conversões automáticas, tomando-se o cuidado de nunca misturar valores de tipos diferentes.

### D.15.3 AMBIGUIDADES EM DEFINIÇÕES E ALUSÕES DE VARIÁVEIS GLOBAIS

Conforme foi apresentado na **Seção 4.7** do **Volume I**, a ausência de **extern** numa alusão de variável global ou a ausência de iniciação numa definição de variável global conduz à ambiguidade. Portanto, para evitar que o compilador interprete uma alusão ou definição de variável global de modo diferente do pretendido, use sempre **extern** em alusão e acrescente uma iniciação explícita a uma definição de variável global. Exemplos:

```
int          global1; /* Definição ou alusão??? */
extern int   global2; /* Alusão com certeza */
int          global3 = 0; /* Definição com certeza */
```

### D.15.4 NOMES DE IDENTIFICADORES TROCADOS

Erros decorrentes da troca inadvertida de um identificador por outro podem ocorrer quando se utilizam identificadores muito parecidos uns com os outros ou quando os nomes usados como identificadores são desprovidos de significado. A adoção de um estilo de programação que promova a representatividade de identificadores minimiza a possibilidade de ocorrência deste tipo de erro (v. **Capítulo 6** do **Volume I**).

---

<sup>180</sup> Erro devido à conversão de alargamento só ocorre em decorrência de *bugs* no compilador utilizado.

**Exemplo:**

```
int x, y; /* É fácil confundir x com y */  
  
int idade, matricula; /* É mais difícil confundir */  
                        /* estas duas variáveis      */
```

**D.15.5 COLISÕES DE IDENTIFICADORES**

Nunca utilize o mesmo identificador para uma variável e uma função, pois o não cumprimento desta regra pode acarretar erros de ligação difíceis de ser resolvidos.

De um modo geral, erros devido a colisões de identificadores não são fáceis de detectar. Portanto, tente evitá-los seguindo as recomendações de estilo apresentadas no **Capítulo 6** do **Volume I**.

# *Bibliografia*

---



Blunden, B., *Software Exorcism – A Handbook for Debugging and Optimizing Legacy Code*, Apress, 2003.

Crawford, T. e Prinz, P., C: *In a Nutshell*, O'Reilly, 2005.

Dinkumware, Ltd, *Dinkum C Library Reference Manual*, publicado em <http://www.dinkumware.com/manuals/>, acessado em 23/11/2006.

Dolenc, A. et alii, *Notes On Writing Portable Programs In C (Nov 1990, 8th Revision)*, publicado em [www.literateprogramming.com/portableC.pdf](http://www.literateprogramming.com/portableC.pdf), acessado em 27/09/2007.

Gillam, R., *Unicode Demystified*, Addison Wesley, 2002.

Goldberg, D., *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, Association for Computing Machinery, março, 1991.

Hyde, R., *Write Great Code – Volume 2: Thinking Low-Level, Writing High-Level*, No Starch Press, 2006.

*International Components for Unicode Home Page*, publicado em <http://www.icu-project.org/>, acessado em 17/09/2008.

International Organization for Standardization, *ISO/IEC 14651: Information Technology – International String Ordering and Comparison – Method for Comparing Character Strings and Description of the Common Template Tailorable Ordering*, 3.<sup>a</sup> edição, International Organization for Standardization, 2007.

International Organization for Standardization, *ISO 8601(E): Data Elements and Interchange Formats – Information Interchange – Representation of Dates and Times*, 3.<sup>a</sup> edição, International Organization for Standardization, 2004.

International Organization for Standardization, *ISO/IEC 9126:1991(E): Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their Use*, International Organization for Standardization, 1991.

International Organization for Standardization, *ISO/IEC 9899:1990 – Programming Languages – C*, International Organization for Standardization, 1990.

International Organization for Standardization, *ISO/IEC 9899/AMD1:1995 – Programming Languages – C (Amendment 1)*, International Organization for Standardization, 1995.

International Organization for Standardization, *ISO/IEC 9899:1999 – Programming Languages – C*, International Organization for Standardization, 1999.

International Organization for Standardization, *ISO/IEC 9899:1999 Cor. 1:2001 – Programming Languages – C (Technical Corrigendum 1)*, International Organization for Standardization, 2001.

International Organization for Standardization, *ISO/IEC 9899:1999 Cor. 2:2004 – Programming Languages – C (Technical Corrigendum 2)*, International Organization for Standardization, 2004.

International Organization for Standardization, *Rationale for International Standard – Programming Languages – C*, Revisão 5.10, International Organization for Standardization, Abril, 2003.

Jones, D. M., *New C Standard – An Economic & Cultural Commentary*, publicado em <http://www.knosof.co.uk/cbook/cbook.html>, acessado em 21/12/2008.

Jönsson, A., *Calling Conventions on the x86 Platform*, publicado em <http://www.angelcode.com/dev/callconv/callconv.html>, acessado em 29/09/2007.

Kernighan, B. W. e Pike, R., *The Practice of Programming*, Addison-Wesley, 1999.

Kernighan, B. W. e Ritchie, D. M., *The C Programming Language*, Second Edition, Prentice Hall, 1988.

Koenig, A., *C Traps and Pitfalls*, Addison-Wesley, 1989.

Korpela, J. K., *Unicode Explained*, O'Reilly, 2006.

Koziol, J. et alii, *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, Wiley Publishing, 2004.

Kulish, U. W. e Miranker W. L., *The Arithmetic of the Digital Computer: A New Approach*, SIAM Review, 28(1):1-40, março, 1986.

Van der Linden, P., *Expert C Programming: Deep C Secrets*, Prentice Hall, 1994.

Matloff, N., *An Interactive Guide to Faster, Less Frustrating Debugging*, publicado em <http://heather.cs.ucdavis.edu/~matloff/UnixAndC/C/CLanguage/Debug.html>, acessado em 23/11/2006.

McConnell, S., *Code Complete: A Practical Handbook of Software Development*, Microsoft Press, 1993.

De Oliveira, U., *Introdução À Programação*, Editora Universitária/UFPB, 2000.

De Oliveira, U., *Programando em C: Volume I – Fundamentos*, Editora Ciência Moderna, 2008.

Oualline, S., *Practical C Programming*, 3.<sup>a</sup> edição, O'Reilly, 1997.

Robbins, J., *Debugging Applications*, Microsoft Press, 2000.

Schildt, H., *C/C++ Programmer's Reference*, Third Edition, McGraw-Hill/Osborne, 2003.

Schotland, T. e Petersen, P., *Exception Handling in C without C++*, Dr. Dobbs's Journal, novembro, 2000.

Sedgewick, R., *Algorithms in C*, Addison-Wesley, 1990.

Spencer, H. et alii, *Recommended C Style and Coding Standards* (versão atualizada de *Indian Hill C Style and Coding Standards*), Rev. 6.0, 1990, publicado em <http://www.chris-lott.org/resources/cstyle/indhill-cstyle.html>, acessado em 05/06/2002.

Stallman, R., et alii, *Debugging with GDB – The Gnu Source-Level Debugger*, Ninth Edition, GNU Press, 2002.

Sun Microsystems, *International Language Environments Guide*, Sun Microsystems, 2005.

SunSoft, *Developer's Guide to Internationalization*, Sun Microsystems, 1995.

Sutter, H., *The String Formatters of Manor Farm*, C/C++ Users Journal, 19(11), novembro, 2001.

Tenenbaum, A. M., Langsam, Y. e Augenstein, M. J. *Estruturas de Dados Usando C*, Makron Books, 1995.

Unicode Inc., *Unicode Collation Algorithm*, publicado em <http://www.unicode.org/unicode/reports/tr10/>, acessado em 23/11/2006.

Viega, J. e Messier, M., *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Input Validation & More*, O'Reilly, 2003.

Wissink, C. e Kaplan, M., *Sorting It All Out: An Introduction To Collation*, Twenty-first International Unicode Conference, 14 a 17 de maio de 2002.

Wolf, T., *Getting Interactive Input in C*, publicado em [http://home.data-comm.ch/t\\_wolf/tw/c/getting\\_input.html](http://home.data-comm.ch/t_wolf/tw/c/getting_input.html), acessado em 13/11/2004.

Zeller, A., *Debugging with DDD – User’s Guide and Reference Manual*, Free Software Foundation, Inc., 2004.