CAPÍTULO 12

PROCESSAMENTO DE ARQUIVOS

12.1 Introdução

Até aqui foram apresentados apenas programas que recebiam dados via entrada padrão (tipicamente, o teclado) e escreviam resultados na saída padrão (tipicamente, a tela). Em geral, processamento de entrada e saída consiste em copiar dados entre a memória principal e dispositivos externos, como um disco rígido, por exemplo. Uma operação de entrada copia dados de um dispositivo de entrada para a memória e uma operação de saída copia dados da memória principal para um dispositivo de saída.

A linguagem C não faz nenhuma distinção entre fontes ou destinos de dados. Isto é, em C, assim como no sistema operacional Unix, o termo ***arquivo*** refere-se não apenas a arquivos armazenados propriamente ditos, como também a qualquer dispositivo que possa ser utilizado como repositório de dados para um programa.

A linguagem C faz distinção entre dois tipos de processamento de arquivos:

(1) Processamento de arquivos baseado em *streams*. Arquivos baseados em ***streams*** são mais fáceis de processar e utilizam uma abordagem razoavelmente portável. Este tipo de processamento é realizado por meio de um conjunto de funções da biblioteca padrão de C, a maioria das quais encontra-se no módulo stdio.

(2) Processamento de arquivos baseado em sistemas. Este tipo de processamento de arquivos utiliza funções de entrada e saída providas pelo sistema operacional em uso e, portanto, pode ser mais eficiente do que processamento de arquivos baseado em ***streams***. Por outro lado, este tipo de processamento não possui portabilidade.

Qualquer dispositivo de entrada e saída (i.e., qualquer arquivo no sentido de C e Unix) pode ser associado a um ***stream***, e as funções da biblioteca padrão de C permitem acessar tais dispositivos usando este conceito de ***stream***. Esta abordagem permite que o programador processe entrada e saída de dados com independência de dispositivo. Aqui, apenas arquivos baseados em ***streams*** serão estudados.

Todas as funções do módulo stdio que ainda não foram apresentadas em capítulos precedentes serão vistas neste capítulo. Outras funções da biblioteca padrão de C para processamento de ***streams*** são dedicadas à entrada e saída de caracteres e ***strings*** extensos e encontram-se no módulo wchar. Estas funções serão apresentadas no **Volume II**.

12.2 Streams

12.2.1 O Conceito de Stream

Enquanto um arquivo está sendo processado (i.e., lido ou escrito), a linguagem C não faz nenhuma distinção entre dispositivos de armazenamento ou de entrada/saída. Por exemplo, é irrelevante para um programa escrito em C se um arquivo está sendo lido de uma unidade de disco rígido ou por meio de uma conexão de rede. Em qualquer situação, arquivos são processados utilizando o conceito de ***stream***.

Um ***stream***[1] consiste em uma seqüência de bytes, similar a um array unidimensional de bytes. ***Stream*** é um conceito importante em programação

[1] A palavra ***stream*** significa ***corrente*** ou ***fluxo*** em português e é derivada da analogia existente entre o escoamento de um fluido e o ***escoamento de dados*** entre um dispositivo de entrada e saída e o programa em C que o utiliza. Em ambas as situações, o fluido ou o fluxo de dados é continuamente renovado como se estivesse escoando.

em C e outras linguagens, pois ele provê uma interface lógica comum a quaisquer dispositivos de entrada e saída. Do modo como C considera o conceito de arquivo, ele pode referir-se a um arquivo armazenado em disco rígido, ao monitor de vídeo, ao teclado, a uma porta de comunicação etc. O conceito de ***stream*** permite tratar todos estes dispositivos (arquivos) do mesmo modo e o programador não precisar preocupar-se com as diversas diferenças entre eles.

É importante salientar que o conceito de ***stream*** é importante apenas para deixar claro que se está processando arquivos sem levar em consideração fonte ou origem de dados e sem utilizar funções específicas de um dado sistema operacional. Quando o tipo de processamento realizado pode ser deduzido por meio do contexto, os termos ***processamento de streams*** e ***processamento de arquivos*** podem ser usados indiferentemente sem ambigüidade.

12.2.2 Implementação de Streams em C: Estruturas do Tipo FILE

O conceito de ***stream*** é implementado em C por meio de ponteiros para um tipo de estrutura denominada **FILE**, cuja definição encontra-se no arquivo `<stdio.h>`. Assim, antes de processar um arquivo utilizando o conceito de ***stream***, deve-se primeiro declarar um ponteiro para esta estrutura, como, por exemplo:

```
FILE *meuStream;
```

Freqüentemente, este ponteiro é denominado **ponteiro para *stream*** ou simplesmente ***stream*** e o identificador utilizado para rotulá-lo envolve o uso da palavra ***stream***. Muitas funções de biblioteca da biblioteca padrão de C usadas em processamento de arquivos utilizam originalmente esta denominação em suas listas de parâmetros.

Depois de criar um ponteiro para a estrutura **FILE**, que representa um ***stream***, deve-se associá-lo a um repositório de dados, que, em C, recebe o nome genérico de ***arquivo***. Isto é realizado utilizando a função **fopen()**, conforme será visto na **Seção 12.5**.

Uma variável do tipo **FILE** armazena informações sobre o estado de um ***stream***, que incluem:

• Um **indicador de erro** – a este campo algumas funções da biblioteca padrão atribuem um valor diferente de zero quando encontram erro de escrita ou leitura no *stream* (v. **Seção 12.8**).

• Um **indicador de final de arquivo** – a este campo algumas funções da biblioteca padrão atribuem um valor diferente de zero quando encontram o final do arquivo durante uma operação de leitura (v. **Seção 12.8**).

• Um **apontador de posição** – especifica o próximo byte a ser lido ou escrito, se o arquivo suportar acesso direto (v. **Seção 12.9**).

• Um **indicador de estado** – especifica se o *stream* permite leitura e/ou escrita e o modo do *stream* (v. **Volume II**).

• Um **buffer** – especifica o tamanho e o endereço de um buffer utilizado em operações de entrada e saída (v. **Seção 12.3**).

A estrutura **FILE** é uma estrutura complexa, cujos campos contêm informações sobre arquivos. A implementação destes campos depende muito do sistema operacional no qual o programa é executado. Portanto, eles não devem ser acessados diretamente por um programa. Ao invés disso, o programa deve utilizar apenas ponteiros para *streams* em conjunto com funções do módulo stdio. Por exemplo, quando desejar mover o apontador de posição de um arquivo, o programador deve utilizar uma função de biblioteca apropriada para este objetivo, ao invés de tentar alterar seu valor diretamente.

12.3 Buffering

Um **buffer** é uma área de memória onde dados provenientes de um arquivo ou que se destinam a um arquivo são armazenados temporariamente. O uso de buffers permite que o acesso aos dispositivos de entrada/saída, que é relativamente lento se 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**. Neste tipo de ***buffering***, o sistema armazena caracteres até que um caractere de quebra de linha, representado pela seqüência de escape `'\n'`, seja encontrado, ou até que o buffer esteja cheio. Este tipo de ***buffering*** é utilizado, por exemplo, quando dados são lidos do teclado. Neste último caso, os dados são armazenados num buffer até que um caractere de quebra de linha seja introduzido (por exemplo, por meio da digitação de `[ENTER]` ou `[RETURN]`). Quando isto acontece, os caracteres digitados são enviados para o programa.

• **Buffering de bloco**. Usando ***buffering*** de bloco, o sistema armazena bytes até que um bloco inteiro seja preenchido (independentemente de o sistema encontrar o caractere '`\n`' ou não). O tamanho padrão de um bloco é tipicamente definido de acordo com o sistema operacional utilizado.

Como padrão, todos os ***streams*** associados a arquivos armazenados têm ***buffering*** de bloco, enquanto ***streams*** associados a um terminal de computador (por exemplo, teclado e tela) têm ***buffering*** de linha ou não têm ***buffering*** (v. adiante), dependendo da implementação.

Tanto em ***buffering*** de linha quanto em ***buffering*** de bloco, pode-se explicitamente **descarregar** a área de buffer associada com ***streams de saída***, forçando o sistema a enviar o conteúdo desta área para o meio de saída, por meio de uma chamada da função de biblioteca **fflush()**, cujo protótipo é:

```
int fflush(FILE *stream)
```

Por exemplo, a chamada:

```
fflush(stdout);
```

força o sistema a descarregar a área de buffer associada com o ***stream*** de saída padrão **stdout** (v. **Seção 12.6**), enviando seu conteúdo para o meio de saída correspondente.

A função **fflush()** serve para descarregar apenas buffers associados com ***streams*** de saída. Ou seja, não existe nenhuma função na biblioteca padrão de C que descarregue ***streams*** de entrada[2]. A função **fflush()** retorna o valor da constante **EOF**, definida em `<stdio.h>`, se ocorrer algum erro durante sua execução; caso contrário, ela retorna 0.

Embora ***buffering*** proveja um modo mais eficiente de processamento de ***streams*** do que o processamento individual de cada caractere, o uso de ***buffering*** é insatisfatório quando se deseja processar cada caractere à medida que ele é introduzido pelo meio de entrada ou deva ser imediatamente escrito no meio de saída. Por exemplo, num processador de texto, os caracteres aparecem na tela logo após serem digitados; i.e., você não tem que digitar `[ENTER]` ou `[RETURN]` após cada caractere digitado. A biblioteca padrão de C contém uma função que permite que se estabeleça o tamanho de um buffer com qualquer valor desejado, inclusive zero (v. **Seção 12.10.5**). Quando o tamanho de um buffer for igualado a zero, nenhum buffer será utilizado com o respectivo ***stream*** e obtém-se, assim, um ***stream* sem *buffering***.

12.4 Formatos de Arquivos

O **formato** de um arquivo refere-se ao modo como as seqüências de bytes que o compõem são interpretadas. Existem dois tipos de formatos para arquivos: (1) **arquivos de texto** (ou **arquivos formatados**) e (2) **arquivos binários** (ou **arquivos não-formatados**).

12.4.1 Arquivos de Texto

Os bytes que constituem um arquivo de texto são interpretados como caracteres que são organizados em linhas, cada uma delas terminada pelo caractere de quebra de linha `'\n'`. O número mínimo de caracteres numa linha que um compilador deve suportar, de acordo com o padrão C99, é 254.

---

[2] Algumas implementações de C permitem que a função **fflush()** seja utilizada para descarga de ***buffers*** de entrada (por exemplo, `fflush(stdin)`), mas este uso da função **fflush()** não é portável, uma vez que o padrão ISO não especifica que esta função possa ser utilizada com streams de entrada.

Quando um arquivo de texto é escrito, ocorre um mapeamento do conteúdo armazenado em memória para caracteres que irão fazer parte do arquivo. Quando um arquivo de texto é lido, ocorre o mapeamento inverso; i.e., seqüências de caracteres são interpretadas antes de ser armazenadas em memória. Este mapeamento depende do sistema operacional utilizado e até mesmo caracteres podem ser mapeados deste modo. Por exemplo, no sistema operacional DOS/Windows, o caractere `'\n'` é mapeado numa seqüência de dois caracteres (CR-LF), enquanto em sistemas operacionais da família Unix o caractere `'\n'` é mapeado num único caractere (LF).

Para entender melhor esse formato de arquivo, suponha que uma variável do tipo **int**, cujo valor corrente é `15`, seja gravado num arquivo em formato de texto. Então, a seqüência de bits escrita no arquivo corresponderia à representação dos caracteres `'1'` e `'5'` no código de caracteres utilizado pelo sistema. Neste caso, se você abrir o arquivo com um editor de texto, será capaz de visualizar o número `15`. A interpretação de conteúdo realizada durante a transferência do valor desta variável para o arquivo pode ser visualizada conforme mostrado na **Figura 40**.

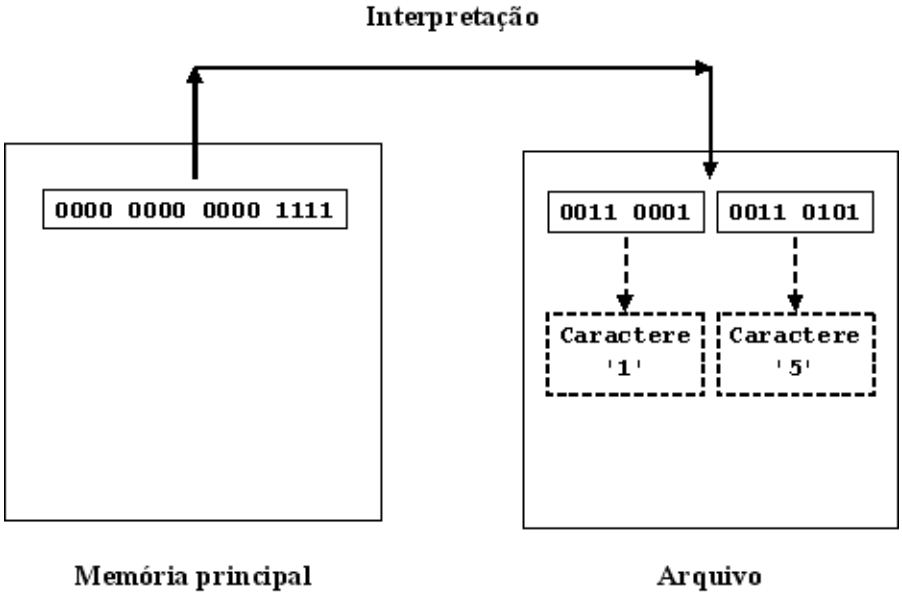


Figura 40: Arquivo de texto: interpretação de conteúdo

Um arquivo de texto pode ser criado de duas maneiras alternativas: (1) por meio de um editor de texto ou (2) por meio de um programa que escreve no arquivo.

12.4.2 Arquivos Binários

Em arquivos binários não existe interpretação de conteúdo. Em outras palavras, num arquivo binário, cada bit é lido ou escrito exatamente como ele aparece no arquivo. Arquivos binários são indicados quando é importante preservar o conteúdo sendo processado.

Para entender melhor esse formato de arquivo, suponha que o inteiro `15` seja gravado num arquivo em formato binário. Neste caso, a seqüência de bits escrita no arquivo corresponderia exatamente à representação de `15` na memória do computador. Se você abrir este arquivo com um editor de texto, verá uma seqüência de caracteres que não correspondem ao número `15`. Assim, a transferência do valor desta variável para o arquivo pode ser visualizada conforme mostrado na **Figura 41**.

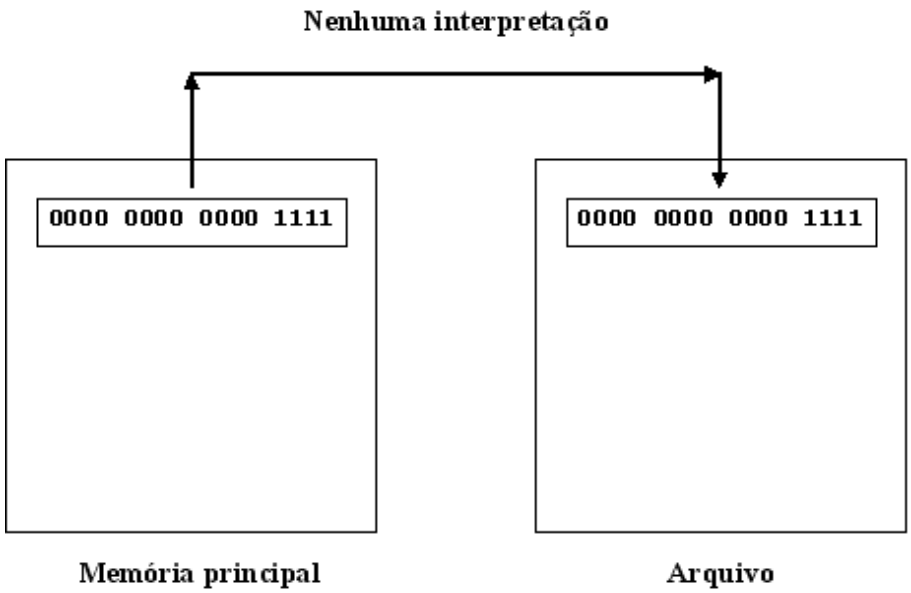


Figura 41: Arquivo binário: nenhuma interpretação de conteúdo

12.4.3 Comparação entre Arquivos de Texto e Arquivos Binários

Num arquivo de texto, qualquer informação é armazenada em forma de texto. Na maioria das vezes, a representação binária de caracteres é a mesma representação textual, mas para outros tipos de dados esta representação é bem diferente. Por exemplo, armazenar o valor `-1.24E03` num arquivo de texto significa armazenar os oito caracteres que o compõem. Isto requer que a representação binária interna do número seja traduzida para o formato legível de texto. Por outro lado, se este mesmo número fosse representado num arquivo binário, não haveria nenhuma tradução e o número seria armazenado exatamente do mesmo modo como ele é armazenado na memória do computador.

Cada formato de arquivo tem suas vantagens e desvantagens, conforme mostrado na **Tabela 37**.

FORMATO	VANTAGENS	DESVANTAGENS
Texto	· Legibilidade · Pode ser editado com editor de texto · O arquivo pode ser facilmente transferido de uma plataforma para outra	· A representação numérica pode não ser precisa · Pode ocupar muito mais espaço
Binário	· Não existe erro de conversão de valores numéricos · A leitura ou escrita de arquivos é mais rápida, pois não há conversão · Dependendo da natureza dos dados, pode ocupar menos espaço	· A transferência para outra plataforma pode ser problemática se as plataformas usarem representações internas de valores diferentes

Tabela 37: Comparação entre arquivos de texto e binários

12.5 Abrindo e Fechando um Arquivo

Um ***stream*** é associado a um arquivo por meio de uma operação de abertura e dissociado por meio de uma operação de fechamento do arquivo.

12.5.1 Abrindo um Arquivo: Função fopen()

Após declarar um ponteiro de ***stream***, conforme foi visto na **Seção 12.2**, deve-se associá-lo a um arquivo por meio da função de biblioteca **fopen()**, cujo protótipo é:

```
FILE *fopen(const char *nome, const char *modo)
```

A função **fopen()** recebe dois argumentos: o primeiro argumento é um nome de arquivo, especificado de acordo com as regras do sistema operacional utilizado e o segundo argumento é um **modo de acesso** (ou **modo de abertura**). Ambos os argumentos devem ser ***strings***.

Existem dois conjuntos de modos de acesso: 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 38**.

MODO DE ACESSO	DESCRIÇÃO
"r"	Abre um arquivo de texto existente apenas para leitura. A leitura começa no início do arquivo.
"w"	Cria um novo arquivo de texto apenas para escrita. Se o arquivo já existir, seu conteúdo será destruído. O indicador de posição do arquivo aponta inicialmente para o início do arquivo.
"a"	Abre um arquivo de texto existente para acréscimo. É permitido escrever apenas no final do arquivo, mesmo que o indicador de posição do arquivo seja movido para outra posição antes do final do arquivo. Se o arquivo cujo nome foi especificado não existir, um novo arquivo com este nome será criado.

"r+"	Abre um arquivo de texto existente para leitura e escrita. O indicador de posição do arquivo aponta inicialmente para o início do arquivo.
"w+"	Cria um novo arquivo de texto para leitura e escrita. Se o arquivo já existir, seu conteúdo será destruído. O indicador de posição do arquivo aponta inicialmente para o início do arquivo.
"a+"	Abre um arquivo de texto existente ou cria um novo arquivo para leitura e acréscimo. Pode-se ler dados em qualquer parte do arquivo, mas dados podem ser escritos apenas no final do arquivo.

Tabela 38: Modos de acesso para arquivos de texto

A única diferença entre os especificadores de modo de acesso para arquivos binários e aqueles apresentados na **Tabela 38** para arquivos de texto é que os especificadores para arquivos binários têm o acréscimo da letra `b`. Arquivos de texto também podem ter acrescida a letra `t` em seus modos de abertura, como mostra a **Tabela 39**.

MODO DE ACESSO PARA ARQUIVOS BINÁRIOS	EQUIVALENTE AO SEGUINTE MODO PARA ARQUIVOS DE TEXTO
"rb"	"r" ou "rt"
"wb"	"w" ou "wt"
"ab"	"a" ou "at"
"r+b"	"r+" ou "r+t"
"w+b"	"w+" ou "w+t"
"a+b"	"a+" ou "a+t"

Tabela 39: Modos de acesso para arquivos binários

A função **fopen()** cria dinamicamente uma estrutura do tipo **FILE**, preenche os campos desta estrutura com informações específicas do arquivo cujo nome é recebido como argumento e, finalmente, retorna um ponteiro para esta estrutura**.** Este ponteiro pode, então, ser posteriormente utilizado

no programa para processar o arquivo. Se não for possível, por algum motivo, abrir o arquivo especificado, **fopen()** retorna um ponteiro nulo.

A razão pela qual a abertura de um arquivo falha depende do modo de abertura e do sistema operacional utilizado. Por exemplo, quando o modo de abertura é "r" e o arquivo não existe ou o programa não tem permissão para acessá-lo, a tentativa de abertura não obtém êxito.

É sempre importante, antes de tentar processar um arquivo, 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 teste.dat para leitura apenas:

```
#include  <stddef.h>
#include  <stdio.h>
   ...
   FILE *ptrStream;

       /* Tenta abrir o arquivo especificado */
   ptrStream = fopen("teste.dat", "r");

  if (ptrStream == NULL) { /* O arquivo não pode ser aberto */
    /* Escreva aqui o trecho de programa que será     */
    /* executado quando o arquivo não pode ser aberto */
      ...
   } else { /* Abertura ocorreu sem problemas */
    /* Aqui o arquivo pode ser processado normalmente */
      ...
   }
```

Pode-se ter mais de um arquivo aberto ao mesmo tempo num programa, mas o número de arquivos que podem estar simultaneamente abertos varia de acordo com o sistema operacional utilizado. Um programa pode checar se este limite foi atingido comparando o número de arquivos correntemente abertos com a constante **FOPEN_MAX**, definida em <stdio.h>, que representa o número máximo de arquivos que podem estar simultaneamente abertos no sistema operacional corrente.

Outra constante importante definida em <stdio.h> é **FILENAME_MAX**, que representa o tamanho máximo do ***string*** que pode ser utilizado para representar um nome de arquivo no sistema operacional em uso.

O nome de um arquivo deve ser especificado de acordo com as regras do sistema operacional utilizado, mas alguns nomes de arquivos funcionam na maioria dos sistemas, como, por exemplo, `cccccccc.ccc`, onde `c` representa uma letra ou um dígito. Por outro lado, nomes de arquivos, tais como `/tmp/temp01` e `c:\Temp\temp01`, são específicos de alguns sistemas.

O nome de arquivo usado como parâmetro de **fopen()** pode especificar um dispositivo de entrada e saída (por exemplo, uma impressora) se o sistema operacional usado prover uma denominação em forma de ***string*** para tal dispositivo. Por exemplo, no sistema operacional DOS/Windows, uma impressora pode ser denominada `"LPT1"`, enquanto em sistemas da família Unix/Linux, esta denominação pode ser `"/dev/lp0"`.

É importante ressaltar que o que determina se um arquivo será considerado um arquivo de texto ou um arquivo binário é seu modo de abertura. Ou seja, as funções do módulo stdio funcionam de modo diferente nestes dois modos de abertura (v. exemplo apresentado na **Seção 12.8.1**).

Os modos de abertura `"r+"`, `"w+"` e `"a+"` devem ser usados com cuidado, pois sobrescrever partes de um arquivo de texto pode ter resultado indesejável. No modo binário, o número de caracteres escritos ou lidos é sempre o mesmo e não há este problema.

12.5.2 Fechando um Arquivo: Função fclose()

Quando um programa não precisa mais processar um arquivo, deve-se fechá-lo utilizando a função de biblioteca **fclose()**. Esta função recebe como único argumento um ponteiro de ***stream*** associado a um arquivo aberto pela função **fopen()**. A função **fclose()** tem o seguinte o protótipo:

```
int fclose(FILE *stream)
```

Por exemplo, considerando o trecho de programa do exemplo apresentado na **Seção 12.5.1**, pode-se fechar o arquivo `teste.dat` associado ao ***stream*** `ptrStream` utilizando a seguinte chamada de **fclose()**:

```
fclose(ptrStream)
```

Um erro freqüente entre os iniciantes em C é utilizar o nome do arquivo como argumento, em vez do ponteiro de ***stream***, numa chamada de **fclose()** [por exemplo, `fclose("teste.dat")`]. Isto certamente trará um sério problema durante a execução do programa, pois apesar de um ***string*** ser interpretado como um ponteiro, este ponteiro, obviamente, não é compatível com um ponteiro para a estrutura **FILE**.

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. Antes de liberar este espaço, quando se trata de um ***stream*** de saída com ***buffering***, a função **fclose()** descarrega o conteúdo da área de buffer para o meio externo de destino. No caso de um arquivo aberto apenas para leitura que utilize ***buffering***, o conteúdo do buffer é simplesmente descartado.

Qualquer sistema operacional fecha todos os arquivos abertos quando o programa que os manipula termina normalmente e muitos sistemas os fecham mesmo quando o programa é abortado. Entretanto, não se deve confiar que este último caso aconteça sempre. Portanto, se o programador pode prever que, em determinada situação, o programa pode ser abortado, ele deve fechar todos os arquivos abertos por precaução.

12.6 Streams Padrão: stdin, stdout e stderr

Existem três ***streams*** padrão em C que são automaticamente abertos no início da execução de qualquer programa. Eles são todos ***streams*** de texto e são denominados **stdin**, **stdout** e **stderr**. Uma descrição sumária destes ***streams*** é apresentada a seguir.

- **stdin** – representa a entrada padrão de dados e, no caso de microcomputadores, tipicamente, é associado ao teclado.
- **stdout** – representa a saída padrão de dados e, no caso de microcomputadores, tipicamente, é associado à tela do computador.
- **stderr** – representa a saída padrão de mensagens de erro e, tipicamente, está associado ao mesmo dispositivo que **stdout**.

As funções de entrada e saída vistas até aqui [por exemplo, **scanf()** e **printf()**] usam, respectivamente, os *stream*s padrão **stdin** e **stdout**.

Muitos sistemas operacionais permitem que estes *streams* sejam redirecionados. Por exemplo, o programador pode desejar que as mensagens de erro sejam escritas num arquivo em vez de na tela do computador (v. **Seção 12.7**).

12.7 Redirecionamento de Entrada e Saída Padrão

A facilidade de redirecionamento de entrada e saída oferecida pelos sistemas operacionais DOS e Unix permite que entrada e saída padrão de um programa sejam redirecionadas para arquivos de texto. Esta facilidade é bastante útil principalmente quando a entrada de dados do programa é demorada ou tediosa e quando se deseja ter uma cópia da saída do programa num arquivo.

12.7.1 Redirecionamento para Arquivos de Texto

Uma maneira de introduzir dados para um programa, que, de outro modo, teria que recebê-los via entrada padrão, é por meio de redirecionamento de entrada. Tanto sistemas operacionais da família Unix quanto o sistema DOS permitem redirecionamento de entrada de modos similares.

Suponha que o programa para o qual você deseje redirecionar entrada seja denominado `MeuProg.exe` no DOS ou `MeuProg` no Unix e que este programa esteja localizado no diretório corrente. Suponha ainda que você tenha um arquivo de texto, denominado `Entrada.txt`, localizado no mesmo diretório e contendo os dados para seu programa. Então, simplesmente digite na linha de comando do DOS:

```
MeuPrograma < Entrada.txt
```

ou

```
./MeuPrograma < Entrada.txt
```

na linha de comando do Unix.

Usando redirecionamento de entrada, o programa lê dados de um arquivo de texto do mesmo modo que o faria se os dados fossem introduzidos no meio de entrada padrão. Este artifício é bastante conveniente quando um programa requer dados complicados de digitar ou quando se está testando o programa e deseja-se garantir que ele receba sempre os mesmos dados em cada teste de execução.

De modo análogo, pode-se redirecionar o meio de saída padrão. Supondo que você deseje que a saída resultante do seu programa seja escrita num arquivo de texto denominado `Saida.txt`, digite na linha de comando do DOS:

```
MeuPrograma > Saida.txt
```

ou

```
./MeuPrograma > Saida.txt
```

na linha de comando do Unix.

Se o arquivo para onde a saída é direcionada não existir, ele será criado. Mas, cuidado, se o arquivo para onde você estiver redirecionando a saída já existir, ele será sobrescrito sem nenhum aviso prévio. O arquivo resultante do redirecionamento da saída é um arquivo de texto e pode ser tratado assim para todos os efeitos (você pode, por exemplo, abri-lo com um editor de texto).

Redirecionamento de saída permite que o programa escreva num arquivo de texto aquilo que seria escrito no meio de saída padrão e é útil quando a saída do programa é muito longa ou quando o programador deseja analisar mais detalhadamente os resultados do programa.

Entrada e saída padrão podem ser redirecionadas simultaneamente, digitando, por exemplo:

```
MeuPrograma < Entrada.txt > Saida.txt
```

no prompt do DOS ou:

```
./MeuPrograma < Entrada.txt > Saida.txt
```

no prompt do Unix.

Para redirecionar a saída padrão de erros para um arquivo denominado `Erros.txt`, digite o seguinte na linha de comando do DOS:

```
MeuPrograma 2> Erros.txt
```

ou

```
./MeuPrograma 2> Erros.txt
```

no prompt do Unix[3].

Redirecionamento de saída de mensagens de erros é útil quando há muitas mensagens de erro intercaladas com a impressão de resultados normais do programa.

Finalmente, todos os ***streams*** padrão podem ser redirecionados ao mesmo tempo. Por exemplo, no DOS você digitaria:

```
MeuPrograma < Entrada.txt > Saida.txt 2> Erros.txt
```

enquanto no Unix você digitaria o seguinte na linha de comando:

```
./MeuPrograma < Entrada.txt > Saida.txt 2> Erros.txt
```

12.7.2 Redirecionamento para Impressora

Para imprimir a saída resultante do seu programa você pode, conforme foi visto anteriormente, redirecionar a saída para um arquivo e imprimir este arquivo normalmente como qualquer arquivo de texto. Mas, se preferir, também pode redirecionar a saída do seu programa diretamente para uma impressora. Supondo que uma impressora esteja conectada na porta `LPT1`, você pode imprimir a saída do programa no sistema operacional DOS digitando o seguinte na linha de comando:

```
MeuProg > LPT1
```

Em sistemas operacionais da família Unix, supondo que uma impressora esteja conectada no dispositivo `/dev/lp0`, você pode imprimir a saída de um programa digitando o seguinte na linha de comando:

```
MeuProg > /dev/lp0
```

[3] Este comando funciona nas interfaces sh e bash, mas não funciona com csh.

Deve-se ressaltar que não há nenhuma garantia de que os exemplos apresentados acima irão funcionar, pois eles dependem dos nomes associados às impressoras pelo sistema operacional.

12.7.3 Redirecionamento Utilizando a Função freopen()

Os ***streams*** padrão podem ser redirecionados utilizando a função **freopen()**, cujo protótipo é:

```
FILE *freopen( const char *nomeDoArquivo,
               const char *modo, FILE *stream )
```

onde os parâmetros são interpretados como:

- `nomeDoArquivo` – ***string*** contendo o nome do arquivo a ser aberto
- `modo` – ***string*** contendo o modo de abertura (v. **Seção 12.5.1**)
- `stream` – ponteiro para a estrutura **FILE** de um arquivo aberto

A função **freopen()** associa um arquivo com um ***stream*** vinculado a um arquivo já aberto e é freqüentemente utilizada para permitir que um ***stream*** padrão (**stdin**, **stdout** ou **stderr**) seja associado com um arquivo específico. Quando obtém êxito, esta função retorna o parâmetro `stream` recebido como argumento. Caso a operação falhe, a função retornará **NULL**.

Como exemplo de uso da função **freopen()**, considere o seguinte trecho de programa:

```
FILE *arquivo;
...
arquivo = freopen("arq1.dat", "r", stdin);
```

Após a execução do trecho de programa acima, funções que realizavam leitura na entrada de dados padrão [por exemplo, **scanf()**] passarão a ler dados no arquivo `arq1.dat` [se a chamada de **freopen()** for bem-sucedida, obviamente].

12.8 Processamento Seqüencial de Arquivos

Uma vez que um arquivo tenha sido aberto conforme foi descrito na **Seção 12.5.1**, pode-se usar o ponteiro de *stream* que o representa para processá-lo.

Ler dados num arquivo faz com que os dados armazenados nele sejam transferidos para variáveis armazenadas na memória principal. Por outro lado, escrever dados num arquivo faz com que os dados contidos em variáveis armazenadas na memória principal sejam transferidos para ele.

De acordo com o tamanho dos objetos manipulados de cada vez, o processamento de arquivos pode ser dividido em três categorias:

- **Um caractere por vez** – neste tipo de processamento, as funções utilizadas lêem ou escrevem um caractere ou byte de cada vez.

- **Uma linha por vez** – as funções utilizadas neste tipo de processamento lêem ou escrevem uma linha de cada vez; é apropriado apenas para arquivos de texto.

- **Um bloco por vez** – neste tipo de processamento, as funções utilizadas lêem ou escrevem um array de bytes de tamanho arbitrário de cada vez.

O processamento de arquivos pode também ser classificado de acordo com o tipo de acesso:

- **Acesso seqüencial** – 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 precedem sejam necessariamente acessados. Nem todo arquivo permite este tipo de acesso.

Tipicamente, em acesso direto, são utilizadas funções que processam um bloco de cada vez. Isto é, funções que processam caractere ou linhas podem também ser utilizadas em acesso direto, mas isso não é usual. Nesta

seção serão apresentadas funções para processamento de arquivos de acordo com o tamanho dos objetos manipulados de cada vez. Os exemplos apresentados nesta seção referem-se a acesso seqüencial. O acesso direto será apresentado na **Seção 12.9**.

12.8.1 Processando Caracteres: Funções fgetc(), fputc(), getc() e putc()

Existem quatro funções, duas das quais podem ser implementadas como macros, para processamento de um caractere por vez num ***stream***. Estas funções são brevemente descritas na **Tabela 40**.

FUNÇÃO	DESCRIÇÃO
getc()	Função que lê um caractere do ***stream***; pode ser implementada como macro.
fgetc()	Função que lê um caractere do ***stream***.
putc()	Função que escreve um caractere no ***stream***; pode ser implementada como macro.
fputc()	Função que escreve um caractere no ***stream***.

Tabela 40: Funções para processamento de caracteres

As funções apresentadas na **Tabela 40** lêem ou escrevem um caractere (byte) no ***stream*** recebido como argumento. Antes de retornarem, estas funções movem o apontador de posição do ***stream*** para o próximo caractere a ser lido ou escrito. Os protótipos destas funções são apresentados na **Tabela 41**.

FUNÇÃO	PROTÓTIPO
getc()	`int getc(FILE *stream)`
fgetc()	`int fgetc(FILE *stream)`
putc()	`int putc(int caractere, FILE *arquivo)`
fputc()	`int fputc(int caractere, FILE *arquivo)`

Tabela 41: Protótipos de funções para processamento de caracteres

As macros[4] **getc()** e **putc()** são normalmente mais eficientes do que as funções correspondentes **fgetc()** e **fputc()** (v. comparação entre macros e funções na **Seção 5.3.3**). Entretanto, nem sempre estas macros podem ser utilizadas com segurança em substituição às funções respectivas (novamente, v. **Seção 5.3.3**).

Conforme foi afirmado na **Seção 12.5.2**, as funções do módulo stdio funcionam de modo diferente de acordo com o fato de o arquivo ser aberto em modo de texto ou binário. Considere como exemplo o seguinte programa que ilustra este fato:

```
#include <stdio.h>

int main(void)
{
   FILE   *streamSaida; /* Stream de saída */

            /*** Trecho 1 ***/

  streamSaida = fopen("ModoTexto.txt", "w"); /* Modo texto */

   if (!streamSaida)
      return 1;     /* Arquivo não pode ser aberto */

   for (int i = 0; i < 100; ++i)
      fputc('\n', streamSaida);

   fclose(streamSaida);

            /*** Trecho 2 ***/

  streamSaida = fopen("ModoBin.txt", "wb"); /* Modo binário */

   if (!streamSaida)
      return 1;     /* Arquivo não pode ser aberto */

   for (int i = 0; i < 100; ++i)
      fputc('\n', streamSaida);

   fclose(streamSaida);
   return 0;
}
```

[4] Aqui, supõe-se que **getc()** e **putc()** são implementadas como macros. O padrão ISO de C sugere, mas não requer isso.

O programa acima é dividido em dois trechos, em cada um dos quais é criado um arquivo. As instruções utilizadas em cada trecho do programa são idênticas. A diferença, além dos nomes dados aos arquivos, é que, no primeiro trecho do programa, o arquivo é aberto no modo texto, enquanto no segundo trecho ele é aberto no modo binário. Quando este programa é executado no sistema operacional DOS, o arquivo produzido pelo primeiro trecho ocupa 100 bytes, enquanto aquele produzido pelo segundo trecho ocupa 200 bytes[5]. Isto mostra apenas que a função **fputc()** funciona de maneira diferente nos modos texto e binário, mas esta afirmação pode ser generalizada para outras funções do módulo stdio.

A função a seguir utiliza as macros **getc()** e **putc()** para copiar um arquivo para outro, byte a byte:

```
/****
 *
 * Função CopiaArquivoPorCaractere(): Copia um arquivo byte a
byte.
 *
 * Parâmetros:
  *  arquivoDeEntrada  (entrada):  nome  do  arquivo  que  será
                                                        copiado
  *  arquivoDeSaida  (entrada):  nome  do  arquivo  que  receberá
                                                    a cópia
 *
 * Retorno: 1, se não houver erro; zero, em caso contrário.
 *
 ****/

unsigned CopiaArquivoPorCaractere(const char *arquivoDeEntrada,
                           const char  *arquivoDeSaida )
{
    FILE  *ptrEntrada, *ptrSaida; /* Streams de entrada e saída
*/
   char  c;

   ptrEntrada = fopen(arquivoDeEntrada, "rb");

   if (ptrEntrada == NULL)
  return 0; /* Arquivo de entrada não pode ser aberto */
```

[5] Se este programa for executado num sistema operacional da família Unix, os dois arquivos criados terão o mesmo tamanho (100 bytes).

```
    /*  Neste  ponto  o  arquivo  de  entrada  foi  aberto  com
                                                    sucesso  */
        /* Se o arquivo de saída não puder ser aberto, deve-se
                                                       fechar */
      /* o arquivo de entrada antes de retornar    */
   if ((ptrSaida = fopen(arquivoDeSaida, "wb")) == NULL) {
        fclose(ptrEntrada);   /* Arquivo de saída não pode ser
                                                       aberto */
      return 0;
   }

   while (1) {
      c = getc(ptrEntrada);
     if (feof(ptrEntrada)) /* Testa se final do arquivo */
      break;              /* de entrada foi atingido  */
      putc(c, ptrSaida);
   }

    /* Processamento terminado. É necessário fechar os arquivos
*/
   fclose(ptrEntrada);
   fclose(ptrSaida);

   return 1;
}
```

Note que ambos os arquivos foram abertos em modo binário, uma vez que se está lendo e escrevendo os mesmos bytes sem preocupação em interpretá-los. Se o arquivo sendo copiado for um arquivo de texto, ele poderá também ser aberto em modo texto sem problemas, apesar de, neste caso, a operação ser menos eficiente. Entretanto, se o arquivo for um arquivo binário e você tentar copiá-lo abrindo-o em modo texto, esta operação não terá portabilidade. Quer dizer, a operação de cópia poderá funcionar num sistema operacional (por exemplo, no Unix) mas não funcionar em outro (por exemplo, no DOS).

A função **feof()** utilizada como teste para saída do **while** da função do último exemplo retorna um valor diferente de zero após o final do ***stream*** que ela recebe como argumento ter sido atingido. O protótipo desta função é:

```
int feof(FILE *stream)
```

A função **feof()** é uma fonte de grande confusão e causa de erro em programas escritos por iniciantes em C. Esta confusão é provavelmente derivada da expectativa de esta função funcionar do mesmo modo que uma função similar [denominada EOF()] existente em outras linguagens de programação (como Pascal, por exemplo). Ocorre, porém, que a função EOF() de Pascal ***antecipa*** o encontro do final do arquivo, enquanto a função **feof()** de C sinaliza o final do arquivo apenas quando há uma tentativa de cruzá-lo. Ou seja, a função EOF() de Pascal retorna ***true*** quando a ***próxima*** operação de leitura irá ultrapassar o final do arquivo, mas a função **feof()** de C retorna um valor diferente de zero apenas quando já houve uma tentativa de leitura além do final do arquivo. Com o raciocínio utilizado na linguagem Pascal em mente, um programador inexperiente em C seria tentado a escrever o laço **while** da função do último exemplo como:

```
while ( !feof(ptrEntrada) )
    putc(getc(ptrEntrada), ptrSaida);
```

Mas, utilizando este laço, a função do último exemplo copiaria indevidamente o valor retornado pela macro **getc()** quando ela tentasse ler além do final do arquivo (v. a seguir).

Quando a macro **getc()** tenta ler além do final de um arquivo, ela retorna o valor de uma constante simbólica (macro) denominada **EOF** e definida em `<stdio.h>`. Esta constante indica que o final do arquivo foi atingido[6]. Entretanto, a constante **EOF** não pode ser utilizada com arquivos binários. Por exemplo, não se pode utilizar o valor retornado pela macro **getc()** para testar se ele é igual a **EOF** e concluir que, nesta situação, o final do arquivo foi atingido. Isto é, a substituição da instrução **while** do exemplo anterior por:

[6] Um outro engano bastante freqüente, que aflige até mesmo programadores com alguma experiência, é imaginar que todo arquivo possui uma marca **EOF** gravada em seu interior para informar onde o arquivo termina, mas este raciocínio é simplesmente falso. A constante **EOF** é retornada pela macro **getc()** (e por outras funções e macros de entrada e saída) para indicar que houve uma tentativa de ir além do final de um arquivo, bem como outras condições de exceção. Esta constante não é lida no arquivo.

```
while (1) {
    c = getc(ptrEntrada);
    if (c == EOF)      /* Testa se final do arquivo */
        break;        /* de entrada foi atingido   */
    putc(c, ptrSaida);
}
```

provavelmente não produzirá o resultado desejado. O problema aqui é que o laço **while** será terminado quando o primeiro caractere que tiver um valor inteiro igual ao da constante **EOF** for encontrado, o que pode não necessariamente representar o final do arquivo. Por exemplo, freqüentemente, a constante **EOF** é definida como -1, que não corresponde ao valor numérico de nenhum caractere. Logo, **EOF** pode ser identificada inequivocamente num arquivo que contém apenas caracteres (i.e., num arquivo de texto), pois como **EOF** é um valor negativo do tipo **int**, este valor nunca será igual a um caractere (**unsigned char**). No entanto, este valor pode ser encontrado (talvez várias vezes) num arquivo contendo bytes que não correspondem necessariamente a caracteres.

Em resumo, num ***stream*** de texto, pode-se usar tanto a constante **EOF** quanto a função **feof()** para testar se o final dele foi atingido. Por outro lado, num ***stream*** de texto, pode-se utilizar apenas a função **feof()** com este objetivo.

12.8.2 Processando Linhas: Funções fgets() e fputs()

O processamento de arquivo linha por linha é conveniente apenas para arquivos de texto. Existem duas funções do módulo stdio que lêem e escrevem uma linha num dado ***stream*** de texto, respectivamente **fgets()** e **fputs()**.

A função **fgets()** tem o seguinte protótipo:

```
char *fgets(char *s, int n, FILE *stream)
```

onde:

- `s` é um ponteiro para o primeiro elemento de um array de caracteres para onde os caracteres lidos serão armazenados
- `n` representa o número máximo de caracteres que serão armazenados no array `s`

• `stream` representa o ***stream*** onde será feita a leitura

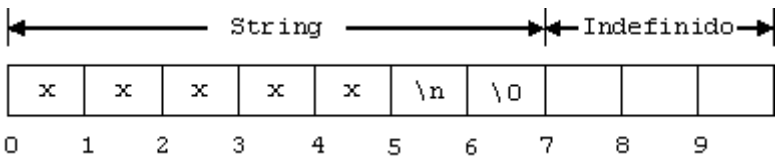
A função **fgets()** lê caracteres até atingir um caractere de quebra de linha (`'\n'`), o final do arquivo ou o número máximo de caracteres especificado (i.e., `n - 1`). Esta função escreve automaticamente um caractere nulo `'\0'` após o último caractere armazenado no array `s`. Quando o final do arquivo é atingido durante uma chamada de **fgets()** antes de ela armazenar qualquer caractere no array, ela retorna **NULL**. Caso contrário, esta função retorna o argumento `s`.

É importante notar que existem diferenças entre **fgets()** e a **gets()**, a função que lê ***strings*** na entrada padrão (v. **Seção 8.4.1**). Ambas as funções acrescentam um caractere nulo após o último caractere armazenado no array. Entretanto, **gets()** não armazena nenhum caractere de quebra de linha no array, enquanto **fgets()** escreve este caractere no array se ele for encontrado. Também, **fgets()** permite que se especifique o número máximo de caracteres a ser lidos, enquanto a função **gets()** não permite isto.

Algumas vezes a função **fgets()** inclui o caractere `'\n'` no array de entrada e é necessário remover este caractere apropriadamente. Por exemplo, suponha que esta função seja chamada como no seguinte trecho de programa:

```
char ar[10];
...
fgets(ar, 10, stdin);
```

e o usuário introduza apenas 5 caracteres seguidos de `[ENTER]`. Então, a função **fgets()** irá encontrar o caractere `'\n'` antes de ler os 9 caracteres que lhe são permitidos. Assim, a situação no arranjo `ar` pode ser esquematizada como ("x" representa os caracteres introduzidos pelo usuário):



Como pode-se verificar na ilustração, quando lido, o caractere `'\n'` ocupa a posição do array dada por `strlen(ar) - 1`. Assim, o trecho de programa a seguir é capaz de removê-lo:

```
int posicao = strlen(ar) - 1;

if (ar[posicao] == '\n')
   ar[posicao] = '\0';
```

A função **fputs()** tem o seguinte protótipo:

```
int fputs(const  char  *s, FILE  *stream)
```

onde:

- `s` é um ponteiro para o primeiro elemento de um ***string***
- `stream` representa o ***stream*** onde será feita a escrita.

A função **fputs()** escreve todos os caracteres do ***string*** `s` no ***stream*** recebido como argumento até que o caractere nulo seja encontrado (este último não é escrito no ***stream***). A função **fputs()** retorna um valor positivo quando a escrita é bem-sucedida; caso contrário, ela retorna **EOF**. Esta função não insere um caractere de quebra de linha após a escrita do último caractere do ***string***, como faz a função **puts()** que escreve no meio de saída padrão (v. **Seção 8.4.2**).

Apesar de as funções **fgets()** e **fputs()** serem mais apropriadas para processamento de arquivos de texto, para definir uma função para copiar arquivos linha por linha semelhante àquela do exemplo da **Seção 12.8.1**, os arquivos devem ser novamente abertos em modo binário. As razões para tal medida são as mesmas expostas naquela seção. A função `CopiaArquivoPorLinha()` apresentada a seguir copia o conteúdo de um arquivo em outro linha por linha.

```
#define  TAMANHO_DA_LINHA  100

/****
 *
  * Função CopiaArquivoPorLinha(): Copia um arquivo linha a
linha.
 *
 * Parâmetros:
 *       arquivoDeEntrada (entrada): nome do arquivo que será
copiado
 *      arquivoDeSaida (entrada): nome do arquivo que receberá
a cópia
 *
 * Retorno: 1, se não houver erro; zero, em caso contrário.
 *
 ****/

unsigned  CopiaArquivoPorLinha( const char   *arquivoDeEntrada,
                           const char  *arquivoDeSaida )
{
   FILE *ptrEntrada, *ptrSaida; /* Streams de entrada e saída */
   char   linha[TAMANHO_DA_LINHA];  /* Array de caracteres onde
os */
                /* caracteres lidos serão armazenados */

   ptrEntrada = fopen(arquivoDeEntrada, "rb");

   if (ptrEntrada == NULL)
    return 0;  /* Arquivo de entrada não pode ser aberto */

   /* Neste ponto o arquivo de entrada foi aberto com sucesso.*/
   /* Se o arquivo de saída não puder ser aberto, deve-se */
   /* fechar o arquivo de entrada antes de retornar. */

   ptrSaida = fopen(arquivoDeSaida, "wb");

   if (ptrSaida == NULL) {
        fclose(ptrEntrada);   /* Arquivo de saída não pode ser
aberto */
      return 0;
   }
   while ( fgets(linha, TAMANHO_DA_LINHA, ptrEntrada) != NULL )
      fputs(linha, ptrSaida);
           /* Processamento terminado. É necessário fechar os
arquivos */
```

```
    fclose(ptrEntrada);
    fclose(ptrSaida);

    return 1;
}
```

Compare a função `CopiaArquivoPorLinha()` apresentada acima com a função `CopiaArquivoPorCaractere()` apresentada na **Seção 12.8.1** e convença-se de que realmente entendeu as diferenças entre as duas.

Pode-se imaginar à primeira vista que a função `CopiaArquivoPorCaractere()` da seção anterior é menos eficiente do que a função `CopiaArquivoPorLinha()`, uma vez que esta última copia uma quantidade maior de caracteres de uma vez e, conseqüentemente, o número de chamadas de funções para leitura e escrita é bem menor. Entretanto, isto não corresponde à realidade, pois, usualmente, as funções **fgets()** e **fputs()** são implementadas utilizando as funções **fgetc()** e **fputc()** que são menos eficientes do que as macros **getc()** e **putc()**, com as quais a função `CopiaArquivoPorCaractere()` foi implementada.

12.8.3 Processando Blocos: Funções fread() e fwrite()

Pode-se imaginar um bloco como sendo um array unidimensional de bytes, conforme foi descrito na **Seção 11.2**. Estes bytes podem ser agrupados para constituir elementos multibytes de um array. Por exemplo, um array de elementos do tipo **double** pode ser interpretado como tal ou como um array de bytes, pois não apenas os elementos do tipo **double** são contíguos em memória como também o são os bytes que compõem cada variável do tipo **double**. Assim, quando se lê ou escreve um bloco, é necessário especificar o número de elementos do bloco e o tamanho de cada elemento.

As funções do módulo stdio usadas para entrada e saída de blocos são as funções **fread()** e **fwrite()**, respectivamente. A função **fread()** tem o seguinte protótipo:

```
size_t  fread( void  *ptr, size_t tamanho,
               size_t nItens, FILE *stream )
```

onde:

- `ptr` é um ponteiro para um array onde o bloco será armazenado
- `tamanho` é o tamanho de cada elemento do array
- `nItens` é o número de itens que serão lidos e armazenados no array
- `stream` representa o ***stream*** onde será feita a leitura

A função **fread()** retorna o número de itens (não o número de bytes) que foram realmente lidos. Este valor deverá ser o mesmo valor do terceiro argumento da função, a não ser que ocorra um erro ou o final do ***stream*** seja atingido. Quando esta função retorna 0, este valor é ambíguo porque pode indicar que o final do arquivo foi atingido ou que ocorreu algum erro antes da leitura de qualquer elemento. Esta ambigüidade pode ser resolvida utilizando-se a função **ferror()** ou **feof()** (v. adiante).

O protótipo da função **fwrite()** é muito parecido com o protótipo de **fread()**:

```
size_t  fwrite( const void  *ptr, size_t tamanho,
                size_t nItens, FILE *stream )
```

onde:

- `ptr` é um ponteiro para um array no qual os bytes serão lidos
- `tamanho` é o tamanho de cada elemento do array
- `nItens` é o número de itens do array a ser escritos
- `stream` representa o ***stream*** onde será feita a escrita

A função **fwrite()** retorna o número de itens (não o número de bytes!) que foram realmente escritos no ***stream*** especificado.

Apesar das semelhanças nos protótipos, as funções **fread()** e **fwrite()** diferem bastante em termos de funcionamento, pois **fwrite()** faz o contrário de **fread()**. Isto é, **fwrite()** lê um array em memória e o escreve num ***stream***, enquanto **fread()** lê um array num ***stream*** e o escreve em

memória. Em qualquer situação, o programador deve tomar cuidado para não especificar um número de itens (terceiro argumento) que ultrapasse o número de elementos do array.

A função `CopiaArquivoPorBloco()` apresentada a seguir mostra como copiar, bloco a bloco, o conteúdo de um arquivo para outro.

```
#define  TAMANHO_DO_BLOCO  512

typedef  char  tDados;

/****
 *
  *  Função  CopiaArquivoPorBloco():  Copia  um  arquivo  bloco  a
bloco.
 *
 * Parâmetros:
  *    arquivoDeEntrada  (entrada):  nome  do  arquivo  que  será
copiado
 *    arquivoDeSaida (entrada): nome do arquivo que receberá a
cópia
 *
 * Retorno: 1, se não houver erro; zero, em caso contrário.
 *
 ****/

unsigned  CopiaArquivoPorBloco( const char  *arquivoDeEntrada,
                            const char  *arquivoDeSaida )
{
   FILE   *ptrEntrada, *ptrSaida; /* Streams de entrada e saída
*/
  tDados bloco[TAMANHO_DO_BLOCO]; /* Array de caracteres onde
*/
             /* os caracteres lidos serão armazenados */
   unsigned  numeroDeItensLidos;

   ptrEntrada = fopen(arquivoDeEntrada, "rb");

   if (!ptrEntrada)
    return 0;  /* Arquivo de entrada não pode ser aberto */
   ptrSaida = fopen(arquivoDeSaida, "wb");

   /* Neste ponto o arquivo de entrada foi aberto com sucesso.*/
   /* Se o arquivo de saída não puder ser aberto, deve-se */
   /* fechar o arquivo de entrada antes de retornar. */
```

```
   if (!ptrSaida) { /* Arquivo de saída não pode ser aberto */
   fclose(ptrEntrada); /* Fecha arquivo de entrada... */
      return 0;        /* antes de retornar           */
   }

   do {
      numeroDeItensLidos = fread( bloco, sizeof(tDados),
                          TAMANHO_DO_BLOCO, ptrEntrada );
fwrite(bloco, sizeof(tDados), numeroDeItensLidos, ptrSaida);
   } while ( numeroDeItensLidos == TAMANHO_DO_BLOCO );

   fclose(ptrEntrada);
   fclose(ptrSaida);

    /* A função fread() retorna 0 quando não encontra    */
    /* mais  elementos ou quando ocorre um erro antes da */
    /* leitura de qualquer elemento. A função ferror() é */
    /* usada para resolver esta ambigüidade.          */

   if (ferror(ptrEntrada))   /* Verifica se houve erro de leitura
*/
      return 0;

   return 1;
}
```

Na função do último exemplo, o teste de final de arquivo é realizado por meio da comparação do número de elementos que devem ser lidos com o número de elementos realmente lidos pela função **fread()**. Acontece que, conforme visto anteriormente, esta função retorna um número de elementos lidos menor do que o solicitado não apenas quando o final do arquivo é atingido durante a leitura, mas também quando ocorre algum tipo de erro de leitura. Por isso, testa-se o ***stream*** `ptrEntrada` com a função **ferror()** para verificar se o laço **do-while** foi interrompido porque o final do arquivo foi atingido ou porque ocorreu algum erro de leitura.

A função `CopiaArquivoPorBloco()` foi escrita de modo a permitir modificações com um mínimo de esforço. Isto é, se for desejado modificar o tamanho de cada elemento do array, precisa-se apenas alterar a definição do tipo `tDados` (por exemplo, `typedef long double tDados`), enquanto se for desejado modificar o número de elementos lidos de cada vez, é necessário apenas redefinir a macro `TAMANHO_DO_BLOCO`.

Do mesmo modo que **fgets()** e **fputs()**, as funções **fread()** e **fwrite()** são usualmente implementadas em termos de **fgetc()** e **fputc()**, que são menos eficientes do que as macros **getc()** e **putc()**. Portanto, a função `CopiaArquivoPorCaractere()` provavelmente é mais eficiente do que `CopiaArquivoPorBloco()`.

Deve-se ressaltar que quando se fala em array de bytes, não se está necessariamente considerando array num alto nível de abstração conforme visto no **Capítulo 7**. Um array de bytes é um conceito de baixo nível e refere-se a qualquer agrupamento de bytes contíguos em memória. Assim, um simples valor do tipo **int** ou **double**, por exemplo, constitui um array de bytes. Logo, as funções **fread()** e **fwrite()** podem ser utilizadas para processar valores de tipos elementares, tais como **int** ou **double**, ou mais complexos, e não apenas arrays de alto nível de abstração, como parece ser sugerido. Por exemplo, para copiar um único valor do tipo **double** armazenado em memória para um arquivo, poder-se-ia escrever o seguinte trecho de programa:

```
FILE   *stream = open("MeuArquivo.dat", "wb");
double umDouble;
...
fwrite(&umDouble, sizeof(double), 1, stream);
```

As funções **fread()** e **fwrite()** complementam uma à outra e podem ler e escrever valores de quaisquer tipos de dados. Portanto, para ler dados de um arquivo binário, pode-se utilizar, de modo semelhante, a função **fread()**:

```
double umDouble;
FILE *stream = fopen("arq.dat", "rb");
fread(&umDouble, sizeof(double), 1, stream);
```

12.9 Acesso Direto

Nos exemplos de processamento de arquivos apresentados até aqui, os arquivos são acessados **seqüencialmente**. Isto é, todos os bytes de um arquivo são processados um após o outro, do primeiro até o último byte. Este tipo de processamento foi conveniente para os exemplos apresentados na **Seção 12.8**, pois as funções apresentadas naqueles exemplos copiavam o conteúdo de um arquivo para outro e, portanto, todos os bytes precisavam

ser lidos e escritos em seqüência. Existem aplicações, entretanto, em que se deseja acessar um conjunto particular de bytes que se encontram numa dada posição de um arquivo. Este tipo de processamento de arquivo é denominado processamento com **acesso direto** (ou **acesso aleatório**).

Basicamente, uma operação de acesso direto num arquivo consiste em: (1) mover o apontador de posição do arquivo para o local desejado e (2) executar a operação de leitura ou escrita desejada. Em princípio, qualquer função do módulo stdio capaz de ler ou escrever num ***stream*** pode ser utilizada para realizar a segunda parte de uma operação de acesso direto.

Existem cinco funções no módulo stdio que podem ser utilizadas para mover o apontador de posição de um arquivo. Elas são resumidamente descritas na **Tabela 42**.

FUNÇÃO	DESCRIÇÃO SUMÁRIA
fseek()	Move o apontador de posição do arquivo para um local especificado por seus argumentos.
ftell()	Informa onde se encontra correntemente o apontador de posição do arquivo associado ao ***stream*** especificado como argumento.
fsetpos()	Semelhante à função **fseek()**, em termos de funcionamento.
fgetpos()	Semelhante à função **ftell()**, em termos de funcionamento.
rewind()	Move o apontador de posição do arquivo para seu início.

Tabela 42: Funções de posicionamento utilizadas em acesso direto

A função **fseek()** tem o seguinte protótipo:

```
int fseek(FILE  *stream, long  int  distancia, int  deOnde)
```

onde:

- `stream` é um ponteiro de ***stream*** associado a um arquivo que suporta acesso direto
- `distancia` é uma distância (positiva ou negativa), medida a partir do terceiro argumento, para onde o apontador de posição do arquivo será movido

• `deOnde` é a posição a partir de onde a distância (segundo argumento) será medida

Em arquivos binários, o valor do segundo argumento (`distancia`) é medido em bytes, enquanto em arquivos de texto, ele deve ser especificado utilizando um valor retornado pela função **ftell()**. O terceiro argumento (`deOnde`) pode assumir um dos valores (macros definidas em `<stdio.h>`) apresentados na **Tabela 43**.

MACRO	REPRESENTA...
SEEK_SET	o início do arquivo
SEEK_CUR	a posição corrente do apontador de posição do arquivo
SEEK_END	o final do arquivo

Tabela 43: Macros de posição em arquivos

Se a função **fseek()** conseguir deslocar o apontador de posição do arquivo para a posição desejada, ela retornará zero; caso contrário, retornará um valor diferente de zero. Por exemplo, a chamada de **fseek()** no trecho de programa a seguir:

```
int  retorno;
FILE *meuStream = fopen("meuArq.dat", "rb");
...
if (meuStream)
    retorno = fseek(meuStream, 10, SEEK_SET);
```

moveria (se bem-sucedida) o apontador de posição de arquivo associado a `meuStream` para o byte de índice 10 neste ***stream***.

Deve-se ressaltar que os bytes num arquivo são indexados como os elementos de um array (i.e., a partir de zero); portanto, o byte de índice 10 é o 11° byte no ***stream***.

Note que, considerando que o ***stream*** `meuStream` do exemplo anterior foi aberto no modo de leitura apenas, a chamada:

```
retorno = fseek(meuStream, 1, SEEK_END);
```

retornaria um valor diferente de zero, indicando que a solicitação não pode ser atendida, pois quando um ***stream*** é aberto apenas para leitura, não se pode mover além da marca de final de arquivo. Portanto, 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. Da modo análogo, se a macro **SEEK_SET** for utilizada, a distância deve ser sempre positiva; neste último caso, independentemente do modo de abertura do arquivo.

Para ***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. Para ***streams*** de texto, o segundo argumento de **fseek()** deve ser 0 ou um valor retornado por **ftell()** (v. adiante) para o mesmo ***stream***. Mais precisamente, as únicas chamadas portáveis da função **fseek()** para arquivos de texto são:

- `fseek(fp, 0L, SEEK_CUR);`
- `fseek(fp, 0L, SEEK_END);`
- `fseek(fp, 0L, SEEK_SET);`
- `fseek(fp, posicao, SEEK_SET);`

onde o parâmetro `posicao` é dado por:

```
posicao = ftell(fp);
```

A função **ftell()** recebe apenas um argumento, que é um ponteiro de ***stream*** e retorna a posição corrente do apontador de posição do ***stream***. O protótipo de **ftell()** é:

```
long ftell(FILE *arquivo)
```

Esta função é freqüentemente usada para guardar o valor corrente do apontador de posição de modo que se possa, posteriormente, retornar àquela posição após uma operação de entrada ou saída.

A posição retornada por **ftell()** é sempre medida a partir do início do arquivo. Para ***streams*** binários, o valor retornado por **ftell()** representa o número real de bytes a partir do início do arquivo. Para ***streams*** de

texto, o valor retornado por **ftell()** representa um valor dependente de implementação que faz sentido apenas quando utilizado como distância numa chamada subseqüente da função **fseek()**. Por exemplo, considere o programa a seguir:

```
#include <stdio.h>

long TamanhoDeArquivo(FILE *stream)
{
   long posicaoAtual, tamanho;

   /* Guarda a posição atual do apontador de posição do arquivo
*/
   posicaoAtual = ftell(stream);

    fseek(stream, 0L, SEEK_END); /* Vai para o final do arquivo
*/
   tamanho = ftell(stream); /* Número de bytes do início até o fim
*/

   /* Restaura a posição original do apontador de posição */
   fseek(stream, posicaoAtual, SEEK_SET);

   return tamanho;
}

int main(void)
{
   FILE *stream;
   const char *strArquivo = "teste.txt";

   stream = fopen(strArquivo, "w+b");

   if (!stream)
      return 1; /* Arquivo não pode ser aberto */

   printf("O apontador de arquivo esta' na posicao: "
          "%ld\n", ftell(stream));

   fprintf(stream, "Isto e' um teste.");

   printf("Agora, o apontador de arquivo esta' na posicao: "
          "%ld\n", ftell(stream));

   printf("\nTamanho do arquivo %s em bytes: %ld",
```

```
            strArquivo, TamanhoDeArquivo(stream));

   fclose(stream);

   return 0;
}
```

O programa do último exemplo apresenta a posição do apontador de arquivo em dois instantes e o tamanho do arquivo. Este programa não terá portabilidade se o arquivo for aberto em modo texto, pelas mesmas razões já expostas anteriormente.

A função **fgetpos()** é semelhante a **ftell()** e seu protótipo é:

```
int fgetpos(FILE *stream, fpos_t *posicao)
```

Esta função armazena o valor corrente do apontador de posição do arquivo, representado por seu primeiro argumento, no local indicado em seu segundo argumento. Ela função retorna um valor diferente de zero se ocorrer algum erro; caso contrário, ela retorna zero.

O segundo argumento da função **fgetpos()** é um ponteiro para o tipo **fpos_t**, definido em `<stdio.h>`, que representa uma posição num arquivo. O valor exato armazenado no endereço `posicao` não é importante em si; este valor só é importante como parâmetro de uma chamada subseqüente de **fsetpos()**.

A função **fsetpos()** move o apontador de um arquivo para uma nova posição obtida por meio de uma chamada prévia de **fgetpos()**. A função **fsetpos()** retorna zero se obtém êxito e um valor diferente de zero em caso contrário. Seu protótipo é:

```
int fsetpos(FILE *stream, const fpos_t *posicao)
```

onde:

- `stream` é um ponteiro para a estrutura **FILE** de um arquivo aberto
- `posicao` é o endereço de um valor do tipo **fpos_t**, obtido por meio de uma chamada de **fgetpos()**

A função **fsetpos()** zera o indicador de final de arquivo no *stream* e desfaz qualquer efeito da função **ungetc()** (v. **Seção 12.10.9**) para o arquivo dado. Quando falha, esta função atribui um valor diferente de zero à variável global **errno**.

O programa a seguir mostra como as funções **fgetpos()** e **fsetpos()** podem ser utilizadas.

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

      int main(void)
      {
   FILE   *stream;
   fpos_t  pos;

   stream = fopen("Arq.dat", "w+"); /* Abre arquivo para escrita
*/

   fprintf(stream, "Isto e' "); /* Escreve algo no stream */

   fgetpos(stream, &pos); /* Salva posição corrente do arquivo */
    fprintf(stream, "um teste"); /* Escreve algo mais */

   /* Retorna à posição salva e escreve a partir dela */
   if (fsetpos(stream, &pos) == 0)
      fprintf(stream, "um exemplo");
   else {
         fprintf(stderr, "Erro ao tentar mover apontador de
arquivo.\n");
      return 1;
   }

   fclose(stream); /* Fecha o arquivo */
   return 0;
}
```

Quando o programa acima é executado, ele cria um arquivo de texto cujo conteúdo é:

```
      Isto e' um exemplo
```

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 (i.e., posicionamentos) num arquivo, enquanto **fgetpos()** e **fsetpos()** usam o tipo **fpos_t** para representar deslocamentos. Este tipo é convenientemente escolhido de modo a permitir deslocamentos arbitrariamente grandes. Por outro lado, os posicionamentos permitidos pelas funções **ftell()** e **fseek()** são limitados pelo tamanho do tipo **long int**. O padrão ISO de C 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.

- **fgetpos()** e **fsetpos()** levam em conta o estado de um *stream* multibytes (v. **Volume II**), enquanto **ftell()** e **fseek()** não o fazem.

Apesar de serem mais recomendadas do que as funções **ftell()** e **fseek()**, **fgetpos()** e **fsetpos()** são raramente utilizadas na prática. O formato do tipo **fpos_t** é dependente de implementação e variáveis deste tipo devem apenas servir como argumentos para as funções **fgetpos()** e **fsetpos()**.

A função **rewind()** retorna o apontador de posição de um arquivo para o início do arquivo e seu protótipo é:

```
void rewind(FILE *stream)
```

Onde `stream` é um ponteiro para a estrutura **FILE** de um arquivo aberto que permite acesso direto (v. adiante).

Esta função existe mais por conveniência do que por estrita necessidade. Ou seja, a chamada:

```
rewind(stream);
```

é o mesmo que:

```
fseek(arquivo, 0, SEEK_SET);
```

exceto pelo fato de a função **rewind()** zerar os indicadores de erro e de final de arquivo do *stream* recebido como argumento, enquanto **fseek()** zera apenas o indicador de final de arquivo.

É importante salientar que nem todo *stream* permite acesso direto. Por exemplo, *streams* associados a um terminal de computador não permitem acesso direto, enquanto aqueles associados a arquivos armazenados em disco o permitem.

Pode-se testar se um *stream* permite acesso direto utilizando o seguinte trecho de programa:

```
int naoPermiteAcessoDireto = fseek(stream, 0L, SEEK_CUR);

if (naoPermiteAcessoDireto) {
  /* Neste trecho, o arquivo pode ser acessado diretamente */
} else {
  clearerr(stream); /* Elimina a condição de erro provocada */
                    /* pela chamada malsucedida de fseek() */
  /* Neste trecho, o arquivo só pode ser acessado seqüencialmente
*/
}
```

Esta chamada de **fseek()** no trecho de programa acima especifica um deslocamento de zero com relação à posição corrente do apontador de arquivo. Portanto, ela serve apenas para testar o valor retornado pela função **fseek()**. Se o teste falhar, deve-se chamar a função **clearerr()** (v. **Seção 12.10.7**) para eliminar este indicativo de erro antes de continuar a processar o arquivo.

A **Seção 12.11** apresenta um exemplo prático de uso de acesso direto.

12.10 Outras Funções de Entrada e Saída

Cada função que utiliza entrada e saída padrão possui uma função correspondente (mas não necessariamente equivalente) que executa operação semelhante num *stream* especificado como parâmetro. Este *stream* pode, inclusive, representar entrada ou saída padrão. A **Tabela 44** apresenta estas funções correspondentes.

ENTRADA/SAÍDA PADRÃO	STREAM GENÉRICO
getchar()	getc() e fgetc()
putchar()	putc() e fputc()
gets()	fgets()
puts()	fputs()
scanf()	fscanf() (e outras funções da família scanf)
printf()	fprintf() (e outras funções da família printf)

Tabela 44: Funções de entrada/saída padrão e funções de streams genéricos

Considerando o mais recente padrão ISO da linguagem C (C99), o módulo stdio possui exatamente 46 funções. Várias delas já foram apresentadas ao longo do texto. Aquelas que ainda não foram apresentadas serão vistas nesta seção.

12.10.1 Entrada Formatada: A Família scanf

A família de funções scanf consiste em uma coleção de funções que possuem as seguintes características em comum:

- Todas as funções funcionam de modo análogo à função **scanf()**: lêem caracteres, tentam transformá-los em valores de tipos de dados conhecidos de acordo com uma especificação de formato e armazenam os valores convertidos em variáveis. Isto é denominado **leitura formatada**.

- Com exceção de **scanf()**, todas as demais funções têm como segundo argumento um ***string*** de formatação e os especificadores de formato que podem ser utilizados no ***string*** de formatação de cada função são os mesmos. Em **scanf()**, o ***string*** de formatação é o primeiro argumento. O **Apêndice B** descreve como construir strings de formatação da família scanf.

- Todas as funções possuem a mesma especificação de retorno. Ou seja, todas retornam o número de valores lidos, convertidos e armazenados em variáveis, quando bem-sucedidas; quando ocorre algum erro, todas retornam **EOF**.

• Todas as funções utilizam *scanf* na composição de seus nomes.

A família de funções scanf é apresentada de forma resumida na **Tabela 45**[7].

FUNÇÃO	DESCRIÇÃO RESUMIDA
fscanf()	Lê dados formatados num stream de texto especificado.
scanf()	Lê dados formatados em stdin.
sscanf()	Lê dados de entrada formatados num string.
vfscanf() (C99)	Semelhante à função fscanf(), mas possui um argumento do tipo va_list em vez da lista de argumentos variáveis que fscanf() possui.
vscanf() (C99)	Semelhante à função scanf(), mas possui um argumento do tipo va_list em vez da lista de argumentos variáveis que scanf() possui.
vsscanf() (C99)	Semelhante à função sscanf(), mas possui um argumento do tipo va_list em vez da lista de argumentos variáveis que sscanf() possui.

Tabela 45: Família de funções scanf

Note que, pela descrição resumida das funções da família scanf apresentadas na **Tabela 45**, funções que possuem um mesmo prefixo têm alguma afinidade além daquelas enumeradas acima. Por exemplo, todas as funções que possuem o prefixo "v" possuem um argumento do tipo **va_list**. Para facilitar a identificação de funções da família scanf, o significado de cada prefixo (caractere) usado com membros desta família é apresentado na **Tabela 46**.

[7] A bem dizer, a família scanf possui outros membros que não são descritos neste capítulo. Estes membros são funções utilizadas na leitura de caracteres extensos e fazem parte do módulo wchar. O **Volume II** apresenta as funções da família scanf que não são apresentadas aqui.

PREFIXO	SIGNIFICADO
f	A leitura de dados é feita num stream especificado como argumento.
s	A leitura de dados é feita num string, e não num stream.
v	Um dos argumentos da função é do tipo va_list. Funções que não possuem este prefixo possuem uma lista de argumentos variáveis.

Tabela 46: Prefixos utilizados pela família de funções scanf

Note que os prefixos apresentados na **Tabela 46** podem ser usados de modo combinado, como "vf" e "vs".

Os argumentos das funções da família scanf são interpretados como mostrado na **Tabela 47**.

ARGUMENTO	TTIPO	INTERPRETAÇÃO
`stream`	FILE *	Representa o stream onde é feita a leitura
`formato`	const char *	String de formatação
`Args`	va_list	Ponteiro para uma lista de argumentos variáveis
`string`	const char *	String onde é feita a leitura

Tabela 47: Argumentos utilizados pela família de funções scanf

As funções **sscanf()** e **vsscanf()** são utilizadas em formatação em memória e serão apresentadas na **Seção 12.10.3**. As demais funções que compõem a família scanf serão apresentadas em seguida.

▶ A função fscanf()

A única diferença entre a função **fscanf()**, cujo protótipo é apresentado em seguida, e a função **scanf()** é que **fscanf()** permite a especificação de um *stream*, enquanto **scanf()** lê apenas no *stream* padrão **stdin**.

```
int fscanf(FILE *stream, const char *formato, ...)
```

Como exemplo de uso da função **fscanf()**, considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int    umInt, nValoresLidos = 0;
   FILE  *stream;

       /* Tenta abrir arquivo no modo texto */
   stream = fopen("Arq.dat", "rt");

   if (!stream) {
      printf("Nao foi possivel abrir o stream");
      return 1;
   }

       /* Tenta ler um valor do tipo int no arquivo */
   nValoresLidos = fscanf(stream, "%d", &umInt);

   if (nValoresLidos == 1) /* Leitura foi OK */
      printf("O valor lido foi: %d", umInt);
   else /* Não foi possível fazer a leitura */
      printf("Nao foi lido nenhum valor");

   fclose(stream);

   return 0;
}
```

► **A função vfscanf()**

A função **vfscanf()** tem o seguinte protótipo:

```
int vfscanf( FILE *stream, const char *formato,
             va_list args)
```

Compare os protótipos das funções **fscanf()** e **vfscanf()** e observe que a única diferença entre eles é que **fscanf()** possui uma lista de argumentos variáveis, enquanto **vfscanf()** possui exatamente três parâmetros. Entretanto, o terceiro parâmetro da função **vfscanf()** é do tipo **va_list** e, conforme visto na **Seção 3.4**, aponta para uma lista de argumentos variáveis. Esta lista de argumentos variáveis deve conter endereços de

variáveis onde os valores lidos serão armazenados, do que mesmo modo que ocorre com **fscanf()**. Portanto, a diferença entre as funções **fscanf()** e **vfscanf()** é sutil e diz respeito apenas às situações nas quais cada uma delas pode ser usada. Quer dizer, 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.

O exemplo a seguir ilustra o uso da função **vfscanf()**[8]:

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

int Exemplo_vfscanf(FILE *stream, const char *formato, ...)
{
   int     nValoresAtribuidos;
   va_list args;

    /* Acessa a lista de argumentos variáveis representada por
*/
   /* "..." usando uma variável do tipo va_list e chama a */
   /* função vfscanf() passando esta variável como argumento */
   va_start(args, formato);
   nValoresAtribuidos = vfscanf(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 = Exemplo_vfscanf(stdin, "%d %f %c",
                                   &umInt, &umFloat, &umChar);
      printf("Numero de valores lidos e atribuidos: %d",
nValoresLidos);

   return 0;
}
```

[8] Para entender este exemplo, é necessário conhecer o funcionamento das macros **va_start**, **va_end**, **va_arg** descritas na **Seção 3.4**.

No exemplo acima, a função **vfscanf()** é chamada pela função `Exemplo_vfscanf()` recebendo como terceiro parâmetro a lista de argumentos variáveis desta última função. Esta chamada não seria possível com o uso da função **fscanf()**.

► **A função vscanf()**

A função **vscanf()** tem o seguinte protótipo:

```
int vscanf(const char *formato, va_list args)
```

Por um lado, as funções **vscanf()** e **scanf()** são semelhantes, pois ambas lêem dados formatados em **stdin**. Por outro lado, **vscanf()** é semelhante a **vfscanf()** no sentido de que ambas possuem um argumento do tipo **va_list**. Se você entendeu o papel desempenhado pelo argumento `args` da função **vfscanf()**, não terá dificuldades de deduzir o funcionamento da função **vscanf()**. O programa a seguir apresenta um exemplo de uso da função **vscanf()**.

```
#include <stdio.h>
#include <stdarg.h>
int Exemplo_vscanf(const char *formato, ...)
{
   int     nValoresAtribuidos;
   va_list args;

   va_start(args, formato);
   nValoresAtribuidos = vscanf(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 = Exemplo_vscanf("%d %f %c",
                   &umInt, &umFloat, &umChar);
```

```
      printf("Numero de valores lidos e atribuidos: %d",
nValoresLidos);

   return 0;
}
```

Este último exemplo é muito parecido com aquele apresentado antes para a função **vfscanf()** e dispensa maiores comentários.

12.10.2 Saída Formatada: A Família printf

A família de funções printf consiste em uma coleção de funções que possuem as seguintes características em comum:

- Todas as funções funcionam de modo análogo à função **printf()**: escrevem caracteres que representam valores de tipos de dados conhecidos de acordo com uma especificação de formato. Isto é denominado **escrita formatada**.
- Todas as funções têm 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. O **Apêndice B** descreve como construir strings de formatação da família printf.
- Todas as funções possuem a mesma especificação de retorno. Isto é, todas elas retornam o número de caracteres impressos, quando bem-sucedidas; quando ocorre algum erro, elas retornam um valor negativo.
- Todas as funções utilizam ***printf*** na composição de seus nomes.

A família de funções printf[9] é apresentada de forma resumida na **Tabela 48**.

[9] A família printf possui outros membros que não são descritos neste capítulo. Estes membros são funções utilizadas na escrita de caracteres extensos e fazem parte do módulo wchar. O **Volume II** apresenta as funções da família printf que não são apresentadas aqui.

FUNÇÃO	DESCRIÇÃO RESUMIDA
fprintf()	Escreve dados formatados num stream de texto especificado.
printf()	Escreve dados formatados no stream de saída padrão stdout.
snprintf() (C99)	Escreve dados formatados num array de caracteres, acrescentando o caractere terminal de string '\0' ao final do processo de escrita. No máximo são escritos n caracteres (levando em conta o caractere '\0'), onde n é um valor inteiro fornecido como argumento.
sprintf()	Semelhante à função snprintf(), mas não permite limitar o número de caracteres que serão escritos.
vfprintf()	Semelhante à função fprintf(), mas possui um argumento do tipo va_list em vez da lista de argumentos variáveis que fprintf() possui.
vprintf()	Semelhante à função printf(), mas possui um argumento do tipo va_list em vez da lista de argumentos variáveis que printf() possui.
vsnprintf() (C99)	Semelhante à função snprintf(), mas possui um argumento do tipo va_list em vez da lista de argumentos variáveis que snprintf() possui.
vsprintf()	Semelhante à função sprintf(), mas possui um argumento do tipo va_list em vez da lista de argumentos variáveis que sprintf() possui.

Tabela 48: Família de funções printf

Como ocorre com os membros da família scanf apresentados na **Seção 12.10.1**, os membros da família printf que possuem um mesmo prefixo têm alguma afinidade entre si. Além disso, o significado dos prefixos destas duas famílias de funções são semelhantes. O significado de cada prefixo (caractere) usado com funções da família printf é apresentado na **Tabela 49**.

PREFIXO	SIGNIFICADO
f	A escrita formatada é feita num stream especificado como argumento.
s	A escrita formatada é feita num string, e não num stream.
v	Um dos argumentos da função é do tipo va_list. Funções que não possuem este prefixo possuem uma lista de argumentos variáveis.
n	A função é capaz de limitar o número de caracteres impressos utilizando um de seus argumentos.

Tabela 49: Prefixos utilizados pela família de funções printf

Os prefixos apresentados na **Tabela 49** podem ser usados de modo combinado, como "vf" e "vsn".

Os argumentos das funções da família printf são interpretados conforme mostrado na **Tabela 50**.

ARGUMENTO	TIPO	INTERPRETAÇÃO
`stream`	FILE *	Representa o stream onde é feita a escrita
`formato`	const char *	String de formatação
`Args`	va_list	Ponteiro para uma lista de argumentos variáveis
`Array`	char *	Array onde é feita a escrita
`N`	size_t	Número máximo de caracteres que serão escritos

Tabela50: Argumentos utilizados pela família de funções printf

As funções **snprintf()**, **sprintf()**, **vsnprintf()** e **vsprintf()** são utilizadas em formatação em memória e serão apresentadas na **Seção 12.10.3**. As demais funções que compõem a família printf serão apresentadas em seguida.

▶ A função fprintf()

A única diferença entre a função **fprintf()**, cujo protótipo é apresentado em seguida, e a função **printf()** é que **fprintf()** permite a especificação de um *stream*, enquanto **printf()** imprime apenas no *stream* padrão **stdout**.

```
int fprintf(FILE *stream, const char *formato, ...)
```

Considere o seguinte como exemplo de uso da função **fprintf()**:

```
#include <stdio.h>

int main(void)
{
   FILE  *stream;
   int    i = 100;
   char   c = 'C';
   float  f = 1.234f;
   stream = fopen("Arq1.txt", "w+");

   if (!stream) {
      fprintf(stderr, "Arquivo nao pode ser criado");
      return 1;
   }

   fprintf(stream, "%d %c %f", i, c, f);

   fclose(stream);

   return 0;
}
```

Este programa é simples demais e não requer comentários.

▶ A função vfprintf()

O protótipo da função **vfprintf()** é:

```
int vfprintf( FILE *stream, const char *formato,
              va_list args )
```

A diferença entre as funções **vfprintf()** e **fprintf()** é a mesma entre **fscanf()** e **vfscanf()**. Se você entendeu esta diferença, apresentada na **Seção 12.10.1**, não terá dificuldades de entender o exemplo de uso de **vfprintf()** apresentado a seguir.

```
#include <stdio.h>
#include <stdarg.h>
int ExemploDe_vfprintf(FILE *stream, const char *formato, ...)
{
   va_list   argumentos;
   int       retorno;

  /* Acessa a lista de argumentos variáveis representada por */
  /* "..." usando uma variável do tipo va_list e chama a */
  /* função vfprintf() passando esta variável como argumento */
   va_start(argumentos, formato);
   retorno = vfprintf(stream, formato, argumentos);
   va_end(argumentos);

   return retorno;
}

int main(void)
{
   FILE *stream;
   int   umInteiro = 30;
   float umFloat = 7.45f;
   char  umString[20] = "Teste";

   stream = fopen("Teste.txt", "w+");

   if (!stream) {
      perror("Arquivo nao pode ser criado.");
      return 1;
   }

   ExemploDe_vfprintf( stream, "%d %f %s",
                       umInteiro, umFloat, umString );

  fscanf(stream, "%d %f %s", &umInteiro, &umFloat, umString);

   printf("%d %f %s\n", umInteiro, umFloat, umString);

   fclose(stream);

   return 0;
}
```

► **A função vprintf()**

O protótipo de **vprintf()** é:

```
int vprintf(const char *formato, va_list args)
```

Novamente, a diferença entre as funções **vprintf()** e **printf()** é a mesma diferença entre **scanf()** e **vscanf()**, apresentada na **Seção 12.10.1**. Se você a entendeu, não terá dificuldades em acompanhar o exemplo de uso de **vprintf()** apresentado a seguir.

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

int ExemploDe_vprintf(char *formato, ...)
{
   va_list argumentos;
   int     retorno;

   va_start(argumentos, formato);
   retorno = vprintf(formato, argumentos);
   va_end(argumentos);

   return retorno;
}

int main(void)
{
   int    umInteiro = 32;
   float  umFloat = 44.9;
   char  *umString = "Isto e' um teste";

  ExemploDe_vprintf("%d %f %s\n",umInteiro,umFloat,umString);

   return 0;
}
```

12.10.3 Formatação em Memória

O módulo stdio define uma coleção de funções que permitem entrada e saída entre um programa e um array de caracteres. Este processo é denominado de **formatação em memória** e permite que o programa formate texto num ***string*** que será posteriormente enviado para algum meio

de saída. Por exemplo, um uso comum de formatação em memória é a organização de texto antes de apresentá-lo num campo de texto ou numa caixa de diálogo de uma interface gráfica.

A **Tabela 51** apresenta resumidamente as funções da biblioteca padrão de C utilizadas em formatação em memória. Este resumo já foi apresentado nas **Seções 12.10.1** e **12.10.2**, mas é repetido aqui para facilidade de referência.

FUNÇÃO	DESCRIÇÃO RESUMIDA
sscanf()	Lê dados de entrada formatados num string. Os dados lidos são armazenados em variáveis cujos endereços são passados como argumentos.
Vsscanf() (C99)	Semelhante à função sscanf(), mas possui um argumento do tipo va_list em vez da lista de argumentos variáveis que sscanf() possui.
snprintf() (C99)	Escreve dados formatados num array de caracteres, acrescentando o caractere terminal de string '\0' ao final do processo de escrita. No máximo são escritos n caracteres (levando em conta o caractere '\0'), onde n é um valor inteiro fornecido como argumento.
sprintf()	Semelhante à função snprintf(), mas não permite limitar o número de caracteres que serão escritos.
vsnprintf() (C99)	Semelhante à função snprintf(), mas possui um argumento do tipo va_list em vez da lista de argumentos variáveis que snprintf() possui.
vsprintf()	Semelhante à função sprintf(), mas possui um argumento do tipo va_list em vez da lista de argumentos variáveis que sprintf() possui.

Tabela 51: Funções utilizadas em formatação em memória

Conforme pode-se constatar examinando a **Tabela 51**, as funções utilizadas em formatação em memória incluem dois membros da família

scanf e quatro membros da família printf. Todos os membros da família scanf, incluindo aqueles discutidos aqui, retornam os mesmos valores diante de circunstâncias semelhantes. Funções desta família que lêem em streams retornam **EOF** quando tentam ler além do final do stream, enquanto aquelas que lêem em strings retornam **EOF** quando tentam ler além do final do string. Os membros da família printf discutidos aqui não levam em consideração o caractere `'\0'` quando retornam o número de caracteres escritos.

A seguir, as funções utilizadas em formatação em memória serão descritas em detalhes e serão apresentados exemplos ilustrativos de uso de cada uma destas funções.

► A função sscanf()

O protótipo da função **sscanf()** é:

```
int sscanf( const char *string,
            const char *formato, ... )
```

Como exemplo de uso da função **sscanf()**, considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int   umInt, nValoresLidos;
   float umFloat;
   char  umChar;
   char  entrada[] = "12 3.1415 X algo mais";

   nValoresLidos = sscanf(entrada, "%d %f %c",
                          &umInt, &umFloat, &umChar);

 printf("\nString onde e' 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;
}
```

▶ A função vsscanf()

O protótipo de **vsscanf()** é:

```
int vsscanf( const char *string, const char *formato,
             va_list args )
```

A diferença entre as funções **vsscanf()** e **sscanf()** é mesma que existe entre outros pares de funções cujos nomes diferem apenas pelo uso do prefixo "v". Esta diferença foi discutida na **Seção 12.10.1**.

Exemplo de uso de **vsscanf()**:

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

int Exemplo_vsscanf(const char *s, const char *formato, ...)
{
   int     nValoresAtribuidos;
   va_list args;

  /* Acessa a lista de argumentos variáveis representada por */
  /* "..." usando uma variável do tipo va_list e chama a  */
  /* função vsscanf() passando esta variável como argumento */
   va_start(args, formato);
   nValoresAtribuidos = vsscanf(s, formato, args);
   va_end(args);

   return nValoresAtribuidos;
}

int main(void)
{
   int   umInt, nValoresLidos;
   float umFloat;
   char  umChar;
   char  entrada[] = "12 3.1415 X algo mais";

   nValoresLidos = Exemplo_vsscanf( entrada, "%d %f %c",
                           &umInt, &umFloat, &umChar );

   printf("\nString onde e' 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;
}
```

▶ A função sprintf()

A função **sprintf()** escreve num array de caracteres dados formatados conforme especificado por um *string* de formatação e acrescenta o caractere terminal `'\0'` ao final do processo de escrita. O protótipo de **sprintf()** é:

```
int sprintf(char *array, const char *formato, ...)
```

O uso desta função não é recomendado, uma vez que ela pode escrever além do limite do array. Em vez dela, é preferível usar a função **snprintf()**, pois esta função permite limitar o número máximo de caracteres que serão escritos no array.

Como exemplo de uso de **sprintf()**, considere:

```
#include <stdio.h>

#define MAX 80

int main(void)
{
   char  array[MAX];  /* Array onde sprintf() irá escrever */
   int     umInt = 32, nCarEscritos = 0;
   double  umDouble = 6.35;
   char   *umString = "bola";
   char    umChar = 'X';
   char   *formato = "umInt = %d, umDouble = %lf, umString = %s,"
                     "umChar = %c"; /* String de formatação */

   nCarEscritos = sprintf( array, formato, umInt,
                           umDouble, umString, umChar );

   printf( "String:\n\t%s \n\nNumero de caracteres escritos no
array:"
           " %d\n", array, nCarEscritos );

   return 0;
}
```

Quando executado, o programa anterior imprime o seguinte no meio de saída padrão:

```
String:
    umInt = 32, umDouble = 6.350000, umString = bola, umChar = X

Numero de caracteres escritos no array: 60
```

► A função snprintf()

A função **snprintf()**, cujo protótipo é apresentado a seguir, difere de **sprintf()** pelo fato de permitir limitar o número de caracteres que serão escritos no array.

```
int snprintf( char *array, size_t n,
          const char *formato, ... )
```

A função **snprintf()** escreve, no máximo, n – 1 caracteres (sem levar em conta o terminal de *string*), onde n é valor do segundo argumento. Deve-se dar preferência ao uso desta função com relação à **sprintf()**, que não permite a limitação do número de caracteres escritos.

Exemplo de uso de **snprintf()**:

```
#include <stdio.h>

#define MAX 80

int main(void)
{
   char  array[MAX]; /* Array onde snprintf() irá escrever */
   int     umInt = 32, nCarEscritos = 0;
   double  umDouble = 6.35;
   char   *umString = "bola";
   char    umChar = 'X';
   char   *formato = "umInt = %d, umDouble = %lf, umString = %s,"
                     "umChar = %c"; /* String de formatação */

   nCarEscritos = snprintf( array, MAX, formato, umInt,
                            umDouble, umString, umChar );
    printf( "String:\n\t%s \n\nNumero de caracteres escritos no
array:"
            " %d\n", array, nCarEscritos );

   return 0;
}
```

A saída do programa anterior é exatamente a mesma daquela do exemplo de uso da função **sprintf()**.

► **A função vsprintf()**

A função **vsprintf()** tem o seguinte protótipo:

```
int vsprintf( char *array, const char *formato,  va_list args )
```

A diferença entre as funções **vsprintf()** e **sprintf()** é mesma que existe entre outros pares de função cujos nomes diferem apenas pelo prefixo "v". Esta diferença foi discutida na **Seção 12.10.1**. A função **vsprintf()** padece do mesmo defeito da função **sprintf()**: não há como limitar o número de caracteres escritos no array recebido como argumento. Por isso, o programador deve dar preferência ao uso da função **vsnprintf()**, apresentada mais adiante.

Exemplo de uso de **vsprintf()**:

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

int ExemploDe_vsprintf(char *ar, const char *formato, ...)
{
   va_list argumentos;
   int     retorno;

  /* Acessa a lista de argumentos variáveis representada por */
  /* "..." usando uma variável do tipo va_list e chama a */
   /* função vsprintf() passando esta variável como argumento */
   va_start(argumentos, formato);
   retorno = vsprintf(ar, formato, argumentos);
   va_end(argumentos);
   return retorno;
}

int main(void)
{
   int    umInteiro = 33;
   float  umFloat = 44.5;
   char   umString[20] = "Isto e' um teste";
   char   array[80] = "Isto e' um teste";

   ExemploDe_vsprintf( array, "%d %f %s",
```

```
                          umInteiro, umFloat, umString );
   printf("Conteudo do array: %s\n", array);

   return 0;
}
```

Este programa imprime no meio de saída padrão:

```
Conteudo do array: 33 44.500000 Isto e' um teste
```

► A função vsnprintf()

A função **vsnprintf()** tem como protótipo:

```
int vsprintf( char *array, size_t n,
              const char *formato, va_list args )
```

A diferença entre as funções **vsnprintf()** e **snprintf()** é mesma que existe entre outros pares de funções cujos nomes diferem apenas pelo uso do prefixo "v". A função **vsnprintf()** difere da função **vsprintf()** pela capacidade de limitar o número de caracteres escritos no array recebido como argumento. Se você entendeu o funcionamento das funções **snprintf()** e **vsprintf()**, compreenderá com facilidade o exemplo de uso de **vsnprintf()** apresentado a seguir.

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

#define MAX 80

int ExemploDe_vsnprintf( char *s, size_t n,
                         const char *formato, ... )
{
   va_list  argumentos;
   int      retorno;

   va_start(argumentos, formato);
   retorno = vsnprintf(s, n, formato, argumentos);
   va_end(argumentos);

   return retorno;
}
```

```
int main(void)
{
   char    ar[MAX];
   int     umInt = 32, nCarEscritos = 0;
   double  umDouble = 6.35;
   char   *umString = "bola";
   char    umChar = 'X';
   char   *formato;

   formato = "umInt = %d, umDouble = %lf, umString = %s, "
             "umChar = %c";

   nCarEscritos = ExemploDe_vsnprintf( ar, MAX, formato, umInt,
                            umDouble, umString, umChar );

   printf( "String:\n\t%s \n\nNumero de caracteres escritos no
array:"
           " %d\n", ar, nCarEscritos );

   return 0;
}
```

12.10.4 Especificação de Buffering: Funções setbuf() e setvbuf()

A biblioteca padrão de C oferece meios para modificar o tamanho de um buffer, mas deve-se utilizar esta facilidade com cuidado, pois, provavelmente, o tamanho padrão de buffer é escolhido de modo a otimizar operações de entrada e saída levando em consideração o sistema operacional utilizado.

Talvez a única situação em que se precise modificar o tamanho do buffer seja quando se deseja utilizar um ***stream*** sem ***buffering*** (v. **Seção 12.3**). Isto é, quando se deseja que a entrada do usuário ou a saída do programa chegue ao seu destino imediatamente. Normalmente, **stdin** tem ***buffering*** de linha e requer que o usuário digite um caractere de quebra de linha antes que a entrada seja enviada para o programa. Em muitas aplicações interativas (por exemplo, um editor de texto), este comportamento é indesejável[10].

[10] Observe, entretanto, que tornar o stream padrão **stdin** sem ***buffering*** é dependente de implementação. Isto é, apesar de a maioria dos sistemas prover entrada de dados sem buffering (por exemplo, a função getch() encontrada em vários ambientes de programação), não existe nenhuma função na biblioteca padrão de C que ofereça esta facilidade.

Para anular *buffering*, pode-se utilizar a função **setbuf()** ou a função **setvbuf()**. A função **setbuf()** recebe dois argumentos: o primeiro é um ponteiro de *stream* e o segundo é um array de caracteres que servirá como novo buffer após a chamada. A função **setbuf()** não retorna nenhum valor e seu protótipo é:

```
void setbuf(FILE *stream, char *buffer)
```

Se o segundo argumento numa chamada de **setbuf()** for um ponteiro nulo, não haverá *buffering*. Por exemplo,

```
FILE *meuStream;
meuStream = fopen("arquivo1.dat", "r");
...
setbuf(meuStream, NULL);
```

faz com que o *stream* `meuStream` não tenha *buffering*.

A função **setvbuf()** é similar a **setbuf()**, mas é um tanto mais elaborada. Seu protótipo é:

```
int setvbuf( FILE *stream, char *buffer,
             int tipo, size_t tamanho )
```

A função **setvbuf()** recebe dois argumentos a mais do que a função **setbuf()**. Um destes argumentos adicionais permite a especificação do tipo desejado de *buffering* (linha, bloco ou nenhum). O outro argumento refere-se ao tamanho do array a ser utilizado como buffer. O tipo de buffer deve ser especificado com uma das macros (definidas no arquivo `<stdio.h>`) apresentadas na **Tabela 52**.

MACRO	BUFFERING
_IOFBF	Bloco
_IOLBF	Linha
_IONBF	Nenhum

Tabela 52: Macros de especificação de buffering da função setvbuf()

A função **setvbuf()** retorna um valor diferente de zero quando ela obtém êxito ao estabelecer o novo buffer; caso contrário, ela retorna zero. Quando o segundo argumento numa chamada desta função é **NULL**, é alocado um buffer usando **malloc()**, que será automaticamente liberado quando o arquivo for fechado. Como exemplo de uso da função **setvbuf()**, considere o trecho de programa a seguir que estabelece um ***stream*** sem buffer para leitura de um arquivo:

```
int   retorno;
FILE  *meuStream;
...
meuStream = fopen("arquivo1.dat", "r");
...
retorno = setvbuf(meuStream, NULL, _IOLBF, 0);

if (!retorno) {
   ... /* Escreva aqui o código executado quando setvbuf()
falha */
} else {
    ... /* Código a ser executado quando setvbuf() é bem-
sucedida */
}
```

Quando o segundo argumento numa chamada da função **setbuf()** não é nulo, como na chamada:

```
setbuf(meuStream, buffer);
```

ela é equivalente a uma chamada da função **setvbuf()**:

```
(void) setvbuf(meuStream, buffer, _IOFBF, BUFSIZ);
```

onde **BUFSIZ** é uma constante, definida no módulo stdio, que especifica o tamanho do buffer utilizado pela função **setbuf()**.

Uma causa freqüente 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. Por exemplo:

```
void ProcessaArquivo(FILE *umStream)
{
   char meuBuffer[1024]; /* Será um buffer zumbi? */
   ...
```

```
    setvbuf(umStream, meuBuffer, _IOFBF, 1024); /* Problema
à vista */
    ...
}
```

No último exemplo, se, antes de retornar, a função `ProcessaArquivo()` não fechar o ***stream*** recebido como argumento ou não estabelecer um novo buffer, que não seja uma variável automática – como é o caso do array `meuBuffer`, o programa apresentará um comportamento indefinido. Neste caso, o problema será ocasionado pelo fato de o ***stream*** recebido como argumento passar a utilizar um buffer (zumbi) que poderá ser subseqüentemente utilizado por outra variável de duração automática. Para resolver o problema, pode-se declarar a variável `meuBuffer` com **static** ou alocar um bloco dinamicamente para servir como buffer, sendo esta última a melhor opção.

12.10.5 Gerenciamento de Arquivos: Funções remove() e rename()

A função **remove()** apaga o arquivo cujo nome é passado como argumento. Ela retorna zero quando obtêm êxito ou um valor diferente de zero em caso contrário. Seu protótipo é:

```
int remove(const char *nomeDoArquivo)
```

O resultado da operação é dependente de implementação se o arquivo a ser apagado estiver aberto.

O programa a seguir permite que se especifique, em linha de comando, o nome de um arquivo a ser apagado utilizando a função **remove()**.

```
#include <stdio.h>

int main(int argc, char  *argv[])
{
         /* Deveria haver dois nomes na execução */
         /* deste programa: os nomes do programa */
         /* e do arquivo a ser apagado.          */
   if (argc != 2){
    printf("Uso do programa: apaga \"nome do arquivo\"");
      return 1;
   }
```

```
   if (!remove(argv[1])) {
      printf("O arquivo \"%s\" foi removido", argv[1]);

return 0;
   }

      printf("Nao foi possivel remover o arquivo \"%s\"",
argv[1]);

   return 1; /* Operação falhou */
}
```

A função **rename()** recebe dois *strings* como argumentos: o primeiro *string* representa o nome atual de um arquivo e o segundo representa o nome que se deseja que o arquivo passe a ter após a execução desta função. O protótipo da função **rename()** é:

```
int rename( const char *nomeAtual,
            const char *nomeNovo )
```

Esta função retorna zero quando obtém êxito; caso contrário, ela retorna um valor diferente de zero. 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.

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[])
{
         /* Deveria haver três nomes na execução    */
         /* deste programa: o nome do programa,     */
         /* o nome atual do arquivo e seu novo nome */
   if (argc != 3){
     printf("Uso do programa: renomeia \"nome do arquivo\""
            " \"novo nome\"" );
      return 1;
   }

   if (!rename(argv[1], argv[2])) {
```

```
    printf("O arquivo \"%s\" foi renomeado para \"%s\"",
              argv[1], argv[2]);
       return 0;
   }

    printf("Nao foi possivel rebatizar o arquivo \"%s\"",
argv[1]);

   return 1; /* Operação falhou */
}
```

12.10.6 Usando Arquivos Temporários: Funções tmpfile() e tmpnam()

A função **tmpfile()** cria um arquivo temporário e abre-o para atualização no modo "w+b". Esta função não possui nenhum parâmetro e retorna um ponteiro para o ***stream*** associado ao arquivo recém-criado, se não ocorrer nenhum erro, ou **NULL** se o arquivo não puder ser criado. A função **tmpfile()** tem o seguinte protótipo:

```
FILE *tmpfile(void)
```

Se o diretório que contém o arquivo temporário não for modificado após a criação deste arquivo, ele (o arquivo) será removido pelo sistema quando o mesmo for fechado ou quando o programa terminar.

Considere com o exemplo de uso da função **tmpfile()** o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int    umInteiro = 30;
   float  umFloat = 7.45;
   char   umString[80] = "Teste";
   FILE  *stream;

   stream = tmpfile(); /* Cria um arquivo temporário */

   if (!stream) {
  fprintf(stderr, "Arquivo temporario nao pode ser criado.");
      return 1;
   }
```

```
    /* Escreve alguns dados no arquivo temporário */
  fprintf(stream, "%d %f %s", umInteiro, umFloat, umString);

    /* Faz apontador de posição retornar para */
    /* o início do arquivo antes de lê-lo     */
   rewind(stream);

    /* Lê e imprime o que foi escrito no arquivo temporário */
  fscanf(stream, "%d %f %s", &umInteiro, &umFloat, umString);
   printf("%d %f %s\n", umInteiro, umFloat, umString);

   fclose(stream);

   return 0;
}
```

A função **tmpnam()** cria um nome de arquivo que pode ser utilizado como nome de um arquivo temporário. Um nome diferente é gerado a cada chamada. Esta função tem o seguinte protótipo:

```
char *tmpnam(char *nomeDoArquivo)
```

O parâmetro `nomeDoArquivo` é um ponteiro para um array que receberá o nome do arquivo ou **NULL**. Quando este argumento é **NULL**, a função **tmpnam()** retorna um ponteiro para um array de duração fixa local à função. Este array contém o nome de arquivo gerado pela função. Neste caso, uma chamada subseqüente dela pode modificar o conteúdo deste array.

Quando o parâmetro `nomeDoArquivo` não é **NULL**, ele deve ser um array capaz de conter pelo menos **L_tmpnam** caracteres (v. adiante). Neste caso, a função retorna o endereço do array recebido como argumento.

A **Tabela 53** apresenta os limites que devem ser seguidos na criação de arquivos temporários. Estes valores são constantes definidas em `<stdio.h>`.

MACRO	EXPANSÃO
L_tmpnam	Tamanho mínimo que um array deve ter para ser capaz de conter um nome de arquivo criado pela função **tmpnam()**.
TMP_MAX	Número mínimo de nomes de arquivos distintos criados pela função **tmpnam()**.

Tabela 53: Macros que especificam limites para arquivos temporários

É importante ressaltar que a função **tmpnam()** gera apenas nomes de arquivo; i.e., ela não cria, abre ou remove arquivos. No intervalo entre a geração de um nome de arquivo temporário e a criação do próprio arquivo pode ser que algum outro processo em execução simultânea 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 [como ocorre com **tmpfile()**]. Portanto, é responsabilidade do programador apagar arquivos assim criados [usando **remove()**, por exemplo]. Em termos de utilidade prática e segurança, é mais recomendável usar a função **tmpfile()** para a criação de arquivos temporários.

Como exemplo de uso da função **tmpnam()**, considere o seguinte esboço de programa:

```
#include <stdio.h>

int main(void)
{
   char  nomeTmp[L_tmpnam];
   FILE *streamTmp;

   tmpnam(nomeTmp); /* Obtém um nome para um arquivo temporário
*/

           /* Abre o arquivo temporário */
   streamTmp = fopen(nomeTmp, "w+b");

   if(!streamTmp)
    return 1; /* Arquivo temporário não pode ser criado */

   /* ... Operações envolvendo o arquivo temporário */
```

```
    fclose(streamTmp); /* O arquivo temporário não é mais
necessário */
   remove(nomeTmp);   /* Apaga o arquivo temporário */

   /* ... Operações que não envolvem o arquivo temporário */

   return 0;
}
```

12.10.7 Detecção de Erros em Streams: Funções clearerr() e ferror()

A função **clearerr()** remove qualquer condição de erro ou de final de arquivo no *stream* cujo ponteiro é recebido como argumento. A função **clearerr()** não retorna nenhum valor e tem o seguinte protótipo:

```
void clearerr(FILE *stream)
```

Quando o indicador de erro de um *stream* assume um valor diferente de zero, operações executadas sobre ele continuarão retornando um status de erro até que a função **clearerr()** ou alguma função de posicionamento [**fseek()**, **fsetpos()** ou **rewind()**] seja chamada. Quando chamadas, estas funções também zeram o indicador de final de arquivo de um *stream*.

A função **ferror()** retorna um valor diferente de zero se existir algum erro associado ao *stream* representado pelo ponteiro recebido como argumento ou zero em caso contrário. Esta função tem o seguinte protótipo:

```
int ferror(FILE *stream)
```

Tipicamente, a função **clearerr()** é chamada em conjunto com a função **ferror()**, como mostra o fragmento de programa a seguir:

```
if (ferror(stream))
   clearerr(stream);
```

Como exemplo mais completo de uso das funções **clearerr()** e **ferror()**, considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
```

```
   FILE *stream;

    stream = fopen("Arq2.dat", "w");/* Abre arquivo só para
escrita */
       /* Gera propositalmente uma condição de erro */
   (void) getc(stream);
       /* Verifica se há alguma condição de erro para o arquivo
*/
   if (ferror(stream)) {
      fprintf(stderr, "Ocorreu um erro de leitura\n");

     /* Zera indicadores de erro e de final de arquivo */
      clearerr(stream);
   }
       /* A partir deste ponto, o arquivo */
       /* pode ser processado normalmente */

   fclose(stream);

   return 0;
}
```

12.10.8 Funções perror() e ungetc()

As funções **perror()** e **ungetc()** são as duas últimas funções do módulo stdio que restam para ser exploradas. Elas foram colocadas juntas aqui porque têm pouca afinidade com as demais funções apresentadas anteriormente.

A função **perror()** imprime no *stream* **stderr** uma mensagem de erro correspondente ao valor armazenado na variável global **errno**. Esta função tem o seguinte protótipo:

```
void perror(const char *string)
```

Se o argumento `string` não for **NULL**, o *string* que ele representa será anexado ao início da mensagem de erro. Freqüentemente, o nome do programa faz parte do *string* passado como argumento para a função. O conteúdo da mensagem de erro impressa pela função **perror()** é o mesmo do *string* retornado pela função **strerror()** (v. **Seção 8.4.1**).

Considere o seguinte exemplo de uso da função **perror()**:

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

int main(int argc, char *argv[])
{
   FILE  *stream;
   char   str[50] = "Erro no programa ";

   stream = fopen("Inexistente.dat", "r");

   if (!stream)
      perror(strcat(str, argv[0]));

   return 0;
}
```

Supondo que o programa acima seja denominado `TestePerror`, ele imprime o seguinte no ***stream*** padrão **stderr**:

Erro no programa TestePerror: No such file or directory

A função **ungetc()** insere um caractere num ***stream***, que deve estar aberto para leitura. Este caractere será lido na próxima chamada de uma função de leitura neste ***stream***. A função **ungetc()** tem o seguinte protótipo:

```
int ungetc(int c, FILE *stream)
```

onde:

- `c` – é o caractere (convertido em **unsigned char**) a ser inserido no ***stream*** representado pelo segundo argumento
- `stream` – é o ponteiro para a estrutura **FILE** de um arquivo aberto num modo que permita leitura

Esta função retorna o caractere inserido no ***stream***, se obtém êxito; caso ocorra algum erro durante a operação, ela retorna **EOF**. Se o argumento recebido pela função **ungetc()** for **EOF**, a operação falha e o ***stream*** não é alterado. Uma chamada bem-sucedida da função **ungetc()** elimina uma eventual indicação 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* com **ungetc()**. Além disso, 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*. Por exemplo, no trecho de programa a seguir, não há garantia de que o caractere `'C'` será inserido no stream **stdin**:

```
ungetc('A', stdin);
ungetc('C', stdin);
```

Mas, no trecho seguinte, o padrão ISO garante a inserção de ambos os caracteres:

```
ungetc('A', stdin);
getchar();
ungetc('C', stdin);
```

Com o exemplo de uso da função **ungetc()**, considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   char string[10];
   printf("Digite um caractere: ");
   string[0] = getchar();

   ungetc('A', stdin);

        /* O conteúdo do string será o caractere */
        /* digitado pelo usuário seguido de 'A'  */
   string[1] = getchar();
   string[2] = '\0';

   printf("Conteudo do string: \"%s\"\n", string);

   return 0;
}
```

12.10.9 Funções de Entrada e Saída Não-portáveis

Muitas operações que se deseja executar com arquivos dependem do sistema operacional utilizado e não são providas pela biblioteca padrão de C. Uma extensão da biblioteca padrão bastante comum entre implementações diferentes de C é o módulo conio[11]. Ele provê funções que realizam operações de entrada e saída que não são contempladas pela biblioteca padrão de C.

O módulo conio possui três funções que provavelmente são as mais utilizadas:

- `getch()` – esta função é semelhante à função **getchar()** da biblioteca padrão de C, mas, enquanto **getchar()** usa *buffering*, `getch()` não usa; i.e., o caractere é imediatamente recebido pelo programa.

- `getche()` – a diferença entre esta função e a função `getch()` é que `getche()` ecoa; i.e., ela apresenta o caractere digitado na tela, enquanto a função `getch()` não ecoa.

- `clrscr()` – esta função limpa a tela.

A sugestão que se apresenta é que funções que não fazem parte da biblioteca padrão sejam confinadas numa função definida pelo programador. Por exemplo:

```
void LimpaTela(void)
{
   clrscr();
}
```

Em seguida, usa-se a função definida pelo programador [por exemplo, `LimpaTela()`] em vez da função não-portável [por exemplo, `clrscr()`]. A vantagem desta abordagem é que ela facilita a portabilidade de um programa que, em si, não é portável. Assim, se o programador usar outro ambiente de programação, no qual a função equivalente a

[11] NB: Este módulo não é padronizado. Portanto, sequer existe garantia que uma dada implementação o possua. Quando existe um módulo com as características descritas aqui, ele pode ter outra denominação, pode conter funções semelhantes com outros nomes etc.

clrscr() tivesse sido denominada clear(), ele teria apenas que alterar a implementação da função LimpaTela() para:

```
void LimpaTela(void)
{
   clear();
}
```

Outra situação na qual o uso de funções não-portáveis 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 provejam tais informações.

12.11 Classificação Externa de Arquivos por Indexação

Esta seção apresenta um exemplo completo de processamento de arquivo utilizando acesso direto.

Suponha que se tenha um arquivo contendo um grande número de registros do tipo tRegistro definido a seguir:

```
#define  NUMERO_MAXIMO_DE_REGISTROS   200
#define  COMPRIMENTO_DO_NOME           20

typedef  struct  {
            short  dia, mes, ano;
         } tData;

typedef  struct  {
            char  nome[COMPRIMENTO_DO_NOME];
            tData  data;
         } tRegistro;
```

Suponha ainda que se deseje imprimir este arquivo em ordem alfabética crescente do campo nome de cada estrutura do tipo tRegistro. Uma maneira óbvia de se resolver este problema seria ler todo o arquivo, armazenar seu conteúdo num array de estruturas do tipo tRegistro e classificar este array em ordem alfabética (usando, por exemplo, a função **qsort()** apresentada na **Seção 11.9**). Entretanto, esta solução não é satisfatória se o número de registros e o tamanho dos registros forem realmente grandes, porque pode requerer uma grande quantidade de

memória para tal operação. Esta forma de classificação é denominada de **classificação interna**, pois mantém todos os dados a ser classificados em memória. Uma outra forma de classificar um arquivo é por meio de **classificação externa**, na qual parte dos dados a ser classificados não precisa ser copiada para a memória principal. Nesta seção, será apresentado um método de classificação externa chamado **classificação por indexação**.

No método de classificação por indexação lê-se apenas a parte do registro (aqui denominada **chave**) na qual a ordenação é baseada e emparelha-se cada chave com um índice que representa a posição no arquivo do registro que contém a respectiva chave. A vantagem deste método é a economia de recursos; na prática, classificar uma coleção de chaves e índices é mais econômico em termos de memória do que classificar uma coleção de registros inteiros. Quanto maior for o tamanho do registro em relação à chave, maior será a economia de memória.

Suponha, como ilustração, que o arquivo a ser ordenado contenha apenas os quatro registros seguintes[12]:

Jose da Silva	11/10/80
Maria da Silva	24/12/82
Joao da Silva	30/01/81
Manoel da Silva	21/12/84

Esses registros podem ser indexados no arquivo de 0 até o número de registros menos 1, como se formassem um array. Isto é possível porque, normalmente, em acesso direto, um arquivo contém uma coleção de registros do mesmo tamanho e, por isso, o arquivo pode ser considerado um array de registros. A indexação de registros é mostrada na tabela a seguir:

ÍNDICE	REGISTRO	
0	Jose da Silva	11/10/80
1	Maria da Silva	24/12/82
2	Joao da Silva	30/01/81
3	Manoel da Silva	21/12/84

[12] Estes registros não aparecem formatados desta maneira no arquivo.

Então, cada índice é associado à sua respectiva chave (neste caso, o nome da pessoa), conforme ilustrado na seguinte tabela:

ÍNDICE	CHAVE
0	Jose da Silva
1	Maria da Silva
2	Joao da Silva
3	Manoel da Silva

Após classificar esses pares de chaves/índices, eles apresentar-se-iam na seguinte ordem:

ÍNDICE	CHAVE
2	Joao da Silva
0	Jose da Silva
3	Manoel da Silva
1	Maria da Silva

Com estes pares classificados, sabe-se agora que o primeiro registro a ser impresso é o de índice 2, depois o de índice 0, depois o de índice 3 e, finalmente, o registro de índice 1.

O primeiro passo na classificação por indexação, portanto, é a leitura seqüencial das chaves, com o subseqüente emparelhamento de cada chave com seu índice. Para armazenamento de cada par chave/índice, será utilizada uma estrutura do tipo `tPares` definida como:

```
typedef  struct {
            int   indice;
            char  chave[COMPRIMENTO_DO_NOME];
         } tPares;
```

A função `ColetaPares()` a seguir lê um número de caracteres igual a `COMPRIMENTO_DO_NOME`, correspondente ao campo `nome` de cada registro. Ela move o indicador de posição do arquivo para o início de cada registro, exceto o primeiro, usando a função **fseek()**. Deste modo, evita-se a leitura de partes do registro que não interessam. A função `ColetaPares()` recebe como parâmetros um ponteiro de ***stream***, um array de elementos do tipo `tPares` definido acima e o número máximo de registros que devem ser

lidos. Esta função retorna o número de registros lidos. A função também inclui um teste para assegurar que cada chamada de **fseek()** obtém êxito; quando este teste falha, a função retorna zero.

```
/****
 *
 * Função ColetaPares(): Lê registros de um arquivo e armazena a
chave de cada registro juntamente com seu índice.
 *
 * Parâmetros:
 *       stream (entrada): stream onde será feita a leitura; o
arquivo
 *           correspondente deve ter sido aberto em modo
 *                        binário que permita leitura
 *   arrayDePares (saída): array que armazenará os pares
 *                            chave/índice
 *      maximoDeRegistros (entrada): número máximo de registros
que poderão ser lidos
 *
 *
 * Retorno: número de registros lidos, se não houver erro;
 *          zero, em caso contrário.
 *
 ****/

unsigned ColetaPares( FILE *stream, tPares arrayDePares[],
                      unsigned  maximoDeRegistros )
{
   unsigned   i, nItenLidos, distancia = 0;

   for (i = 0; i < maximoDeRegistros; i++) {
          /* Lê o campo nome de cada registro e armazena */
             /* no campo chave do array de pares  */
      nItenLidos = fread( arrayDePares[i].chave, 1,
                          COMPRIMENTO_DO_NOME, stream );

    if (nItenLidos < COMPRIMENTO_DO_NOME) /* O final de arquivo
*/
         break;            /* foi atingido ou ocorreu */
                           /* algum erro de leitura   */

          /* Armazena índice do registro lido*/
      arrayDePares[i].indice = i;

     /* Posiciona apontador de arquivo para próxima leitura */
      distancia += sizeof(tRegistro);
```

```
          /* Se não for possível mover apontador de    */
         /* arquivo antes de o final do arquivo ter   */
         /* sido atingido é porque ocorreu algum erro */
      if (fseek(stream, distancia, SEEK_SET) && !feof(stream))
           return 0; /* Não foi possível mover o apontador de
posição */
   } /* for */

   return  i;
}
```

Na função `ColetaPares()`, os valores da variável `distancia` são computados levando em consideração o tamanho da estrutura `tRegistro` na instrução:

```
distancia += sizeof(tRegistro);
```

A próxima tarefa é classificar o array de estruturas `tPares`. A função `ClassificaPares()`, apresentada a seguir, utiliza a função **qsort()** da biblioteca padrão de C (v. **Seção 11.9**) para classificar o array recebido como argumento.

```
/****
 *
 * Função ClassificaPares(): Classifica um array de elementos do
tipo tPares utilizando a função qsort().
 *
 * Parâmetros:
 *    arrayDePares (entrada/saída): array que armazena pares
chave/índice
 *   numeroDeElementos (entrada): número elementos do array
 * Retorno: Nada.
 *
 ****/

void   ClassificaPares( tPares  arrayDePares[],
                        unsigned  numeroDeElementos)
{
      /* Alusão da função que será utilizada com a função qsort()
*/
   extern int Compara(const void *, const void *);

       qsort(arrayDePares,  numeroDeElementos,  sizeof(tPares),
Compara);
}
```

```
/****
 *
  *  Função  Compara():  Função  de  comparação  utilizada  por
qsort().
 *            Utiliza a função strcmp() para comparar os
 *               campos "nome" dos registros.
 *
 * Parâmetros:
 *  p1 (entrada): ponteiro para um par do tipo tPares
 *  p2 (entrada): ponteiro para outro par do tipo tPares
 *
 * Retorno: 0, se as chaves são iguais
 *     > 0, se a primeira chave é maior do que a segunda
 *     < 0, se a primeira chave é menor do que a segunda
 *
 ****/

int Compara(const void *p1, const void *p2)
{
   return strcmp(((tPares *)p1)->chave, ((tPares *)p2)->chave);
}
```

Entender o funcionamento da função `ClassificaPares()` é fácil, desde que você possua o necessário conhecimento sobre ordenação e saiba como utilizar a função **qsort()**. Se você apresentar dificuldades para entender estes tópicos, consulte a **Seção 11.9**.

O próximo passo do exemplo de classificação por indexação é imprimir os registros do arquivo em ordem crescente. É o que faz a função `ImprimeRegistrosOrdenados()` apresentada a seguir.

```
/****
 *
  *  Função  ImprimeRegistrosOrdenados():  Imprime  registros
ordenados
 *
 * Parâmetros:
 *    stream (entrada): stream onde será feita a leitura dos
registros que serão impressos; o arquivo correspondente deve ter
sido aberto em modo binário que permita leitura
 *      arrayDePares (entrada): array que armazena os pares
 *                    chave/índice ordenados pela chave
 *        numeroDeElementos (entrada): número de elementos do
array
 *
```

```
 * Retorno: número de registros lidos, se não houver erro;
 *          zero, em caso contrário.
 *
 ****/

unsigned         ImprimeRegistrosOrdenados(FILE         *stream,
                      tPares  arrayDePares[],
                            unsigned numeroDeElementos )
{
   tRegistro   umRegistro;
   unsigned    i;

   for (i = 0;  i < numeroDeElementos; i++) {
      if ( fseek(stream, sizeof(tRegistro)*arrayDePares[i].indi
ce, SEEK_SET) )
        return 0; /* Não foi possível mover apontador de arquivo
*/

                  /* Lê o registro especificado  */
      fread(&umRegistro, sizeof(tRegistro), 1, stream);

            /* Imprime o registro recuperado */
      printf("%s \t\t %d/%d/%d\n", umRegistro.nome,
              umRegistro.data.dia,
              umRegistro.data.mes, umRegistro.data.ano);
   }

   return 1;
}
```

Na função `ImprimeRegistrosOrdenados()`, a posição inicial de cada registro é calculada multiplicando-se o índice do registro pelo tamanho de cada estrutura. Isto é, esta posição é dada por:

```
sizeof(tRegistro)*arrayDePares[i].indice
```

Para completar o exemplo, resta apenas escrever uma função **main()** que chame as funções implementadas acima. A função **main()** apresentada a seguir foi escrita de tal modo que o nome do arquivo, cujos registros serão classificados, possa ser passado como argumento (v. **Seção 8.5**).

```
int main(int  argc, char  *argv[])
{
   FILE    *stream;
   tPares   pares[NUMERO_MAXIMO_DE_REGISTROS];
   unsigned  numeroDeRegistrosLidos;

    /* Deveria haver dois nomes na execução deste    */
    /* programa: os nomes do programa e do arquivo.  */
    /* Em vez de abortar o programa, você poderia  */
    /* dar uma nova chance ao usuário, permitindo que */
    /* ele introduza agora o nome do arquivo.        */
   if (argc != 2){
      printf("Erro: nome do arquivo ausente\n");
      return 1;
   }

   stream = fopen(argv[1], "rb");

   /* Se o arquivo especificado na linha de comando não puder */
   /* ser aberto, dê tchau ao usuário e encerre o programa */
   if (!stream) {
      printf("Erro abrindo arquivo %s\n", argv[1]);
      return 1;
   }

   numeroDeRegistrosLidos = ColetaPares(stream, pares,
                            NUMERO_MAXIMO_DE_REGISTROS);

   if (numeroDeRegistrosLidos) {
      ClassificaPares(pares, numeroDeRegistrosLidos);

      if ( !ImprimeRegistrosOrdenados(stream, pares,
                                numeroDeRegistrosLidos) ) {
          printf("Erro processando arquivo.\n"); /* Houve algum
erro */
         fclose(stream);
         return 1;
      }
   }

   fclose(stream);

   return 0;
}
```

12.12 Exercícios de Revisão

1. Qual é a principal vantagem decorrente do uso de arquivos de dados (no sentido usual)?

2. Qual é o significado de ***arquivo*** em C?

3. (a) O que é um buffer? (b) O que é ***buffering***? (c) Qual é a vantagem que se obtém com a utilização de ***buffering*** em operações de entrada/saída? (d) O que significa descarregar um buffer?

4. (a) O que é um ***stream***? (b) Que vantagem este conceito oferece ao processamento de arquivos?

5. Qual é a relação entre um ponteiro de ***stream*** e uma área de buffer associada a ele?

6. (a) Qual é o significado do identificador **FILE** no processamento de ***streams***? (b) Onde o identificador **FILE** é definido?

7. Em linhas gerais, qual é o conteúdo do arquivo de cabeçalho `<stdio.h>`?

8. Qual é a vantagem de se ter operações de entrada/saída reunidas em funções de biblioteca em vez de na própria linguagem C?

9. (a) O que significa abrir um arquivo? (b) Como isto pode ser efetuado em C?

10. Descreva os diferentes modos de abertura de arquivos que podem ser especificados numa chamada da função **fopen()**.

11. Descreva os ***streams*** padrão de C.

12. (a) O que ocorre quando se fecha um arquivo? (b) Que função é utilizada com este propósito? (c) Qual é a importância de se fechar um arquivo após seu uso?

13. (a) O que é um arquivo de texto? (b) O que é um arquivo binário? (c) Que vantagens cada um destes tipos de arquivo oferece com relação ao outro?

14. Como se informa às funções do módulo stdio que elas devem lidar com um arquivo binário e não com um arquivo de texto (ou vice-versa)?

15. Descreva duas formas diferentes de atualizar (i.e., modificar o conteúdo) de um arquivo de texto.

16. (a) Qual é o significado da constante **EOF**? (b) Por que ela não deve ser utilizada para determinar se o final de um arquivo binário foi atingido?

17. Descreva o significado das macros a seguir definidas no módulo stdio:

(a) _IOFBF
(b) _IOLBF
(c) _IONBF
(d) BUFSIZ
(e) FOPEN_MAX
(f) FILENAME_MAX
(g) SEEK_SET
(h) SEEK_CUR
(i) SEEK_END
(j) L_tmpnam
(k) TMP_MAX

18. (a) Qual é o propósito da função de biblioteca **feof()**? (b) Como ela deve ser usada?

19. Quando um arquivo é aberto para escrita no modo texto, os caracteres escritos nele são interpretados?

20. (a) Um arquivo de texto pode ser seguramente aberto no modo binário com o objetivo de criar uma cópia dele? (b) Um arquivo binário pode ser seguramente aberto no modo texto com o objetivo de criar uma cópia dele?

21. Explique a diferença entre acesso seqüencial e acesso direto de arquivos.

22. Como é possível determinar se um arquivo permite acesso direto?

23. Apresente um exemplo de um *stream* que não permite acesso direto.

24. (a) Para que serve a função **fseek()**? (b) Para que serve a função **ftell()**?

25. Qual é a principal diferença entre as funções **fseek()** e **fsetpos()**?

26. Qual é a diferença entre **getc()** e **fgetc()**?

27. Descreva as funções a seguir:
(a) fread()
(b) fwrite()
(c) fgets()
(d) fputs()

28. Quais são as diferenças entre as funções **gets()** e **fgets()**?

29. É possível reescrever o programa apresentado na **Seção 12.11** substituindo as chamadas de **fseek()** por chamadas de **fsetpos()**?

30. Por que a função **rewind()** pode ser considerada redundante?

31. O que as funções da família scanf têm em comum?

32. Descreva os significados dos seguintes prefixos utilizados por membros da família scanf:
(a) f
(b) s
(c) v

33. O que as funções da família printf têm em comum?

34. Descreva os significados dos seguintes prefixos utilizados por membros da família printf:

(a) f

(b) s

(c) v

(d) n

35. Que diferenças você é capaz de apontar entre os especificadores de formato das famílias printf e scanf apresentados no **Apêndice B**?

36. (a) O que é formatação em memória? (b) Que funções da família scanf são usadas em formatação em memória? (c) Que funções da família printf são usadas em formatação em memória?

37. Por que se pode considerar que a função **setbuf()** é redundante?

38. Para que serve a função **rename()**?

39. (a) Diferencie as funções **tmpfile()** e **tmpnam()**? (b) Qual delas é mais útil na prática?

40. Qual é a diferença entre as funções **clearerr()** e **ferror()**?

41. Descreva o funcionamento das funções **perror()** e **ungetc()**.

42. Por que é recomendável confinar funções que não fazem parte da biblioteca padrão de C em funções definidas pelo programador?

43. (a) Descreva classificação externa e classificação interna de arquivos. (b) Que vantagem um método de classificação externa oferece com relação a um método de classificação interna?

44. Descreva o método de classificação externa por indexação.

45. Pense numa maneira fácil de classificar o array de pares utilizado em classificação por indexação em ordem decrescente, ao invés de em ordem crescente, como foi feito acima, na **Seção 12.11**.

46. O programa a seguir não consegue abrir o arquivo introduzido no meio de entrada padrão pelo usuário mesmo que o arquivo exista no diretório corrente. Explique por que isto ocorre e encontre um modo de corrigir o problema.

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

int main()
{
   char   nome[100]; /* Nome do arquivo */
   FILE   *stream;
   printf("Nome do arquivo: ");
   fgets(nome, sizeof(nome), stdin);

   stream = fopen(nome, "r");
   if (!stream) {
fprintf(stderr, "Nao foi possivel abrir o arquivo\n");
      return 1;
   }

   printf("Arquivo aberto\n");

   fclose(stream);

   return 0;
}
```

47. Descubra o que há de errado com o programa apresentado a seguir e encontre uma maneira de corrigi-lo.

```
int main()
{
   FILE  *stream = fopen("teste.txt", "r");
   char  linha[100];

   while(!feof(stream)) {
      fgets(linha, sizeof(linha), stream);
      fputs(linha, stdout);
   }

   fclose(stream);

   return 0;
}
```

48. Descubra o que há de errado com a função `TamanhoDaLinha()` apresentada a seguir e encontre uma maneira de corrigi-la.

```
unsigned TamanhoDaLinha(FILE *stream)
{
   char     c;
   unsigned contador = 0;
   while( (c = fgetc(stream)) != EOF && c != '\n')
      contador++;

   return contador;
}
```

12.13 Exercícios de Programação

EP12.1) Escreva um programa em C que leia e imprima na tela uma linha de cada vez de um arquivo de texto cujo nome é especificado na linha de comando.

EP12.2) Escreva um programa que apresente na tela o número de linhas, palavras e caracteres do arquivo de texto cujo nome é especificado na linha de comando. (**Sugestão**: utilize funções encontradas no módulo ctype da biblioteca padrão de C para determinar se um caractere é um espaço em branco e, assim, um limite entre palavras.)

EP12.3) Escreva um programa em C que abra um arquivo de texto especificado em linha de comando e o imprima na tela do computador em sentindo inverso.

EP12.4) Escreva um programa em C que imprima na tela as 10 últimas linhas de um arquivo de texto cujo nome é especificado na linha de comando.

EP12.5) Um arquivo de texto contém, em cada linha, o nome de um aluno e sua respectiva nota numa dada disciplina, conforme esquematizado a seguir:

```
Catarina de Aragon    8.0
Ana Bolena            6.3
Jane Seymour          5.8
Ana de Cleves         4.5
...                   ...
Catarina Howard       7.0
Catarina Parr         7.5
```

Os nomes dos alunos têm tamanhos variados e o que separa cada nome de sua respectiva nota é um caractere de tabulação (`'\t'`). Supondo que o nome do arquivo que contém as notas seja `Notas.txt`, escreva um programa em C que leia cada registro de aluno e crie um arquivo de texto denominado `NotasAtuais.txt` contendo os mesmo registros, mas com as notas acrescidas de um ponto.

EP12.6) Escreva um programa em C que leia o conteúdo de um arquivo de texto e o apresente no meio de saída padrão um caractere de cada vez. O programa deve solicitar que o usuário introduza o nome do arquivo e, caso este não consiga ser aberto, responder adequadamente.

EP12.7) Escreva um programa em C que compare byte a byte dois arquivos, cujos nomes são especificados em linha de comando, e informe se os arquivos são iguais ou não.

EP12.8) Escreva um programa que mantenha um histórico de suas execuções. Isto é, este programa deve manter um arquivo contendo o número de execuções do programa e a data da última execução. (**Sugestão**: Utilize as funções **time()**, **localtime()** e **strftime()** para obter um string representando a data de cada execução do programa. Detalhes sobre o uso destas funções são encontrados no **Volume II**.)

EP12.9) Escreva um programa em C que leia uma senha introduzida pelo usuário e imprima `'*'` em vez do caractere digitado pelo usuário. (**Sugestão**: Não existe maneira portável de fazer isto em C. Utilize uma

função semelhante à função `getc()`, descrita na **Seção 12.10.9**, que leia caracteres sem utilizar ***buffering*** e sem ecoar o caractere lido.)

EP12.10) (a) Escreva uma função que receba dois nomes de arquivos como argumentos (***strings***) e retorne um ponteiro para um ***stream*** associado ao primeiro arquivo que represente a concatenação dos dois arquivos cujos nomes são passados como argumentos. Se ocorrer algum problema durante a concatenação, esta função deve retornar **NULL**. (b) Escreva uma função **main()** que receba os nomes de dois arquivos que deverão ser concatenados em linha de comando e chame a função especificada no item (a) para realizar a devida operação.

EP12.11) Escreva um programa em C que copie as n primeiras linhas de um arquivo de texto para um novo arquivo. O nome dos arquivos e o número n de linhas que devem ser copiados são dados solicitados pelo programa ao usuário.

EP12.12) Escreva um programa que divida um arquivo, cujo nome é recebido como argumento de linha de comando, em arquivos menores de tamanho máximo igual a `N` bytes, onde `N` é um valor inteiro positivo introduzido pelo usuário. O número de arquivos resultantes da divisão deve ser limitado a 10 pelo programa. Os nomes dos arquivos resultantes desta divisão devem começar com o nome do arquivo dividido seguido de um número que identifique unicamente cada arquivo. Por exemplo, supondo que o nome do arquivo a ser dividido seja `Arq.dat`, então os arquivos resultantes devem ser nomeados `Arq01.dat`, `Arq02.dat` etc.

EP12.13) Suponha que o indivíduo mais idoso num arquivo contendo estruturas do tipo `tRegistro` definido na **Seção 12.11** tenha nascido em 01/01/1960. Reescreva o programa apresentado naquela seção de modo que ele imprima os registros classificados em ordem cronológica de nascimento de cada indivíduo representado por uma estrutura do tipo `tRegistro`.

Sugestão

A melhor maneira de ordenar os registros pela data de nascimento consiste em ler cada data de nascimento no arquivo e transformá-la num inteiro (use **long int**) que represente o número de dias decorridos desde a data de nascimento do indivíduo mais idoso (o que, supostamente, é sabido de antemão). Utilize a função criada para resolver o problema **EP9.3** para calcular este valor. De posse dele, armazene-o juntamente com o respectivo índice num par do tipo `tPares` definido na **Seção 12.11**, cuja definição deve ser alterada para refletir o novo tipo de chave. O restante da resolução do problema não lhe deverá trazer dificuldades se você realmente entendeu a construção do programa apresentado na **Seção 12.11**.

EP12.14) Escreva um programa em C que ofereça três opções ao usuário, denominadas (a) ***Entrada de texto***, (b) ***Apresentação*** e (c) ***Substituição***, assim descritas:

(a) Entrada de texto: lê linhas de texto (***strings***) introduzidas pelo usuário e as escreve num arquivo cujo nome é fornecido pelo usuário. Deve ser permitido ao usuário introduzir tantas linhas quantas ele desejar.

(b) Apresentação: recebe como entrada do usuário o nome de um arquivo de texto e apresenta seu conteúdo na tela.

(c) Substituição: recebe como entrada do usuário três dados: (i) o nome de um arquivo de texto, (ii) um caractere `C1` e (iii) um outro caractere `C2`. Então, o programa deve procurar todas as ocorrências do caractere `C1` no conteúdo do arquivo, substituí-las pelo caractere `C2` e, finalmente, escrever o texto modificado novamente no arquivo.

EP12.15) Parte do trabalho a ser desenvolvido nesta lista de exercícios encontra-se no arquivo comprimido `ArquivosEP12.15.zip` encontrado no site do livro. Neste arquivo você encontrará os seguintes módulos:

- `Alunos`. Este módulo é responsável pela definição dos tipos `tAluno` (representando o registro de um aluno) e `tTurma` (representando uma coleção destes registros como uma lista encadeada). Este módulo contém ainda as funções necessárias para as operações básicas necessárias sobre estes tipos. Este módulo não está implementado e faz parte do problema implementá-lo.

- `Interface`. Este módulo é responsável pelas interações básicas, em mais alto nível entre o usuário e o programa. Este módulo não se encontra implementado e faz parte do problema implementá-lo.

- `Leitura`. Este módulo é responsável por leitura e validação de valores necessários ao programa. Este módulo está completamente implementado e provavelmente não será necessário modificá-lo.

- `Operacoes`. Este módulo é responsável pela execução das operações oferecidas pelo programa. Este módulo não está implementado e faz parte do problema implementá-lo.

- `main`. Este módulo, composto por um único arquivo, corresponde à parte principal do programa. Este módulo está completamente implementado e provavelmente não será necessário modificá-lo.

O exercício consiste em implementar as funções cujas implementações foram omitidas nos módulos descritos acima e construir o programa resultante. Este programa é muito parecido com aquele do exercício **EP10.3** (com algumas complicações removidas e outras adicionadas) e oferece as seguintes opções para o usuário:

- Acrescentar um aluno na turma
- Remover um aluno da turma
- Modificar o registro de um aluno
- Enumerar todos os alunos da turma
- Ler um arquivo contendo dados de uma turma
- Gravar um arquivo contendo dados de uma turma
- Sair do programa

EP12.16) Escreva um programa que leia o conteúdo de um arquivo contendo informações sobre alunos de uma turma gravado conforme descrito no exercício **EP12.15** e utilize classificação externa por indexação para apresentá-lo na tela em ordem alfabética de nomes de alunos, da primeira, segunda ou terceira nota, de acordo com a opção escolhida. Este programa não é interativo, de modo que a opção do usuário é escolhida em linha de comando utilizando a seguinte sintaxe:

COMANDO	OPERAÇÃO
Programa arquivo	Apresenta os registros na ordem em que eles foram armazenados no arquivo
Programa +a *arquivo*	Apresenta registros do arquivo em ordem alfabética crescente de nomes de alunos
Programa -a *arquivo*	Apresenta registros do arquivo em ordem alfabética decrescente de nomes de alunos
Programa +n1 *arquivo*	Apresenta registros do arquivo em ordem crescente da primeira nota
programa -n1 *arquivo*	Apresenta registros do arquivo em ordem decrescente da primeira nota
programa +n2 *arquivo*	Apresenta registros do arquivo em ordem crescente da segunda nota
programa -n2 *arquivo*	Apresenta registros do arquivo em ordem decrescente da segunda nota
programa +n3 *arquivo*	Apresenta os registros do arquivo em ordem crescente da terceira nota
programa -n3 *arquivo*	Apresenta os registros do arquivo em ordem decrescente da terceira nota

Na tabela acima, *programa* e *arquivo* referem-se, respecti-vamente, ao nome do programa e ao nome do arquivo cujo conteúdo será apresentado. Qualquer outra tentativa de execução do programa deve ser considerada inválida e, neste caso, o usuário deve receber uma resposta adequada do programa informando-lhe como o programa deve ser utilizado. (**Sugestão**: Você não poderá utilizar a função **qsort()** da biblioteca padrão de C, pois esta função aplica-se apenas à classificação de arrays e você estará utilizando listas encadeadas. Portanto, adapte a função `BubbleSort()` apresentada nas **Seções 10.8** e **11.9** para classificar listas encadeadas cujos nós são do tipo definido no exercício **EP12.15**.)