Processamento de Arquivos em C

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar a seguinte terminologia referente a arquivos:
  - ❒ Formato de arquivo
  - ❒ Arquivo de texto
  - ❒ Arquivo binário
  - ❒ Stream
  - ❒ Stream de texto
  - ❒ Stream binário
  - ❒ Buffer
  - ❒ Arquivo temporário
  - ❒ Registro
  - ❒ Campo de registro
  - ❒ Bloco de memória
  - ❒ Condição de exceção
  - ❒ Tratamento de exceção
  - ❒ Indicador de posição
  - ❒ Acessos direto e sequencial
- Descrever o que ocorre quando se abre e fecha um arquivo
- Contrastar os diversos modos de abertura de arquivos
- Explicar as diversas categorias de processamento de arquivo
- Implementar um programa que abre e processa um arquivo usando leitura sequencial ou por acesso direto
- Processar arquivos por byte, linha ou bloco
- Lidar com erros em processamento de arquivos
- Implementar operações básicas de leitura e escrita formatadas
- Manipular arquivos temporários
- Identificar dificuldades no processamento de arquivos grandes

objetivos

EM GERAL, PROCESSAMENTO DE ENTRADA E SAÍDA consiste na troca de dados entre a memória principal do computador e seus dispositivos periféricos, como um disco rígido, por exemplo. Isto é, uma **operação de entrada** (ou de **leitura**) copia dados de um dispositivo de entrada para a memória principal e uma **operação de saída** (ou de **escrita**) copia dados da memória principal para um dispositivo de saída. Em C, **arquivo** refere-se a qualquer dispositivo que possa ser utilizado como origem ou destino de dados de um programa. Assim processamento de entrada e saída e **processamento de arquivos** são termos equivalentes em programação em C. Contudo, neste livro, *arquivo* corresponde ao conceito popular e cotidiano de arquivo; ou seja, um conjunto de dados armazenados num meio de armazenamento não volátil (p. ex., um disco rígido).

Este capítulo discute brevemente as facilidades de processamento de arquivos providas pela biblioteca padrão de C. Ele mostra várias maneiras pelas quais um arquivo pode ser processado. Se preferir, o leitor com pouca experiência em processamento de arquivos poderá dividir seu estudo em duas partes: processamento sequencial e processamento por acesso direto.

Processamento sequencial, discutido na **Seção 2.11.1**, é usado em virtualmente todo este volume, a partir do próximo capítulo. Por sua vez, processamento por acesso direto será necessário a partir do **Capítulo 6**, que é dedicado a operações sobre tabelas de busca armazenadas em memória secundária. Em ambos os casos, este livro lida apenas com registros de comprimento fixo que são recuperados da memória secundária com base no valor de um campo desses registros, denominado chave primária. Como será visto no **Capítulo 3**, uma chave primária identifica inequivocamente um registro e é tipicamente usada como chave de busca ou chave de ordenação.

2.1 Arquivos de Texto e Binários

Formato de um arquivo é uma propriedade que se refere ao modo como os bytes que o compõem são tratados. De acordo com essa propriedade, existem dois tipos de arquivos: (1) **arquivo de texto** e (2) **arquivo binário**.

Informalmente, um arquivo de texto é aquele no qual sequências de bytes são interpretadas como caracteres, de modo que, quando exposto adequadamente, ele apresenta informação humanamente legível. Além disso, os caracteres são agrupados em linhas, cada uma delas terminada por um caractere de **quebra de linha**, representado em C por `'\n'`. O formato preciso de um arquivo de texto depende do sistema operacional no qual ele é usado. Por exemplo, em alguns sistemas, um arquivo de texto pode conter apenas caracteres com representação gráfica, tabulação horizontal e quebra de linha.

Novamente, usando uma definição informal, arquivo binário é aquele que, quando seu conteúdo é exposto, não exibe informação humanamente legível. Em outras palavras, um arquivo binário é uma sequência irrestrita de bytes.

2.2 Streams

Processamento de arquivos em C é baseado no conceito de **stream**[1]. Esse tipo de processamento é realizado por meio de um conjunto de funções da biblioteca padrão de C que se encontram definidas no módulo **stdio**.

2.2.1 Conceito

Stream é uma abstração importante em programação em C e em outras linguagens de programação, porque provê uma interface lógica comum a quaisquer dispositivos de entrada e saída. Isto é, do modo como C

[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 ou saída e um programa. Em ambas as situações, o fluido ou o fluxo de dados é continuamente renovado.

considera o conceito de arquivo, ele pode se referir a um arquivo armazenado em disco, um monitor de vídeo, um teclado, uma porta de comunicação etc. Todos esses arquivos funcionam de maneiras diferentes, mas o uso de streams permite tratá-los do mesmo modo. Em outras palavras, streams provêm uma abstração consistente que é independente de dispositivo e do sistema de arquivos utilizado pelo sistema operacional em uso. Por causa disso, pode-se usar uma mesma função que escreve num arquivo armazenado em disco para escrever na tela ou numa impressora. Além disso, pode-se ainda usar uma mesma função para escrever num arquivo armazenado num sistema da família Unix ou da família Windows.

O conceito de stream é importante apenas para deixar claro que se estão processando arquivos sem levar em consideração diferenças inerentes a dispositivos de entrada e saída. Mas na prática, os termos processamento de streams e processamento de arquivos podem ser usados indiferentemente sem que haja ambiguidade. Ou seja, ler num stream é o mesmo que ler num arquivo, visto que um stream é um intermediário nessa operação sobre os dados que, em última instância, provêm de um arquivo. De modo semelhante não faz diferença falar em escrever num stream ou escrever num arquivo.

2.2.2 Estruturas do Tipo FILE

Em C, o conceito de stream é implementado por meio de estruturas do tipo **FILE**, cuja definição encontra-se no cabeçalho `<stdio.h>`.

Antes de processar um arquivo utilizando o conceito de stream, deve-se primeiro definir um ponteiro para estruturas do tipo **FILE**, como por exemplo:

```
FILE *stream;
```

Esse ponteiro é denominado ponteiro de **stream** ou apenas *stream*. Na prática, um ponteiro dessa natureza representa o conceito de stream.

Depois de definir um ponteiro para a estrutura **FILE**, que representa um stream, deve-se associá-lo a um arquivo. Isso é realizado utilizando-se a função **fopen()**, conforme será visto na **Seção 2.3**.

Após ser associada a um arquivo, uma estrutura do tipo **FILE** passa a armazenar informações sobre o arquivo. Dentre essas informações, as seguintes merecem destaque:

- **Indicador de erro** — a esse campo algumas funções da biblioteca padrão atribuem um valor que indica a ocorrência de erro durante uma operação de escrita ou leitura no stream.
- **Indicador de final de arquivo** — a esse campo algumas funções da biblioteca padrão atribuem um valor que indica que houve tentativa de acesso além do final do arquivo durante uma operação de leitura.
- **Indicador de posição** — esse campo determina onde o próximo byte será lido ou escrito no arquivo. Após cada operação de leitura ou escrita, o indicador de posição é atualizado de modo a refletir o número de bytes lidos ou escritos.
- **Endereço e tamanho de uma área de buffer** — esse campo especifica, se for o caso, um buffer utilizado em operações de entrada ou saída (v. **Seção 2.5**).

A implementação dos campos de uma estrutura do tipo **FILE** depende do sistema operacional em uso e eles não devem ser acessados diretamente num programa. Em vez disso, o programa deve utilizar apenas streams (i.e., ponteiros do tipo **FILE** *) em conjunto com funções do módulo stdio.

2.3 Abrindo e Fechando um Arquivo

Abrir um arquivo significa associá-lo a um stream, que é um ponteiro para uma estrutura do tipo **FILE** (v. **Seção 2.2**) que armazena todas as informações necessárias para processamento do arquivo. Após a abertura de um arquivo, qualquer operação de entrada e saída sobre ele passa a ser realizada por intermédio do respectivo stream associado.

2.3.1 Função fopen()

Uma operação de abertura de arquivo cria dinamicamente uma estrutura do tipo **FILE** e a associa a um arquivo. Essa operação é realizada pela função **fopen()**, cujo protótipo é:

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

A função **fopen()** possui dois parâmetros, sendo ambos strings: o primeiro parâmetro é um nome de arquivo, especificado de acordo com as regras do sistema operacional utilizado e o segundo parâmetro é um modo de acesso ou de abertura (v. adiante). Esse último parâmetro determina a natureza das operações permitidas sobre o arquivo.

Um nome de arquivo pode especificar qualquer dispositivo de entrada ou saída (p. ex., uma impressora) para qual o sistema operacional usado provê uma denominação em forma de string. Por exemplo, no sistema operacional DOS/Windows, uma impressora pode ser denominada `"LPT1"`, enquanto, em sistemas da família Unix, essa denominação pode ser `"/dev/lp0"`.

A função **fopen()** aloca dinamicamente uma estrutura do tipo **FILE**, preenche os campos dessa estrutura com informações específicas do arquivo, cujo nome é recebido como parâmetro, e, finalmente, retorna o endereço da referida estrutura. Esse endereço pode, então, ser utilizado para processar o arquivo. Se, por algum motivo, não for possível abrir o arquivo especificado, **fopen()** retorna **NULL**.

Existem diversas razões pelas quais a abertura de um arquivo pode não ser bem-sucedida. Em particular, a abertura de um arquivo depende do modo de abertura (v. adiante) e do sistema operacional utilizados. Por exemplo, quando se tenta abrir um arquivo apenas para leitura e o arquivo não existe ou o programa não tem permissão para acessá-lo, a abertura do arquivo não logra êxito.

Antes de tentar processar um arquivo, é importante testar o valor retornado pela função **fopen()** para constatar se o arquivo foi realmente aberto, como mostra o fragmento de programa a seguir:

```
FILE *stream;

   /* Tenta abrir o arquivo para leitura */
stream = fopen("Inexistente.txt", "r");

   /* Verifica se a abertura foi bem-sucedida */
if (stream == NULL) { /* Abertura falhou */
   printf( "O arquivo nao pode ser aberto.\n" );
} else { /* Abertura foi OK */
   printf("Arquivo aberto com sucesso\n");

      /* Processa o arquivo */
   /* ... */
}
```

A função `AbreArquivo()`, apresentada a seguir, incorpora tratamento de exceção quando um arquivo não é aberto e pode ser usada em substituição a **fopen()**.

```
FILE *AbreArquivo(const char *nome, const char *modo)
{
   FILE *stream;

   stream = fopen(nome, modo);

   if (!stream) { /* Erro de abertura */
      fprintf( stderr, "\n>>> O arquivo %s nao pode ser aberto", nome);
      exit(1); /* Aborta o programa */
   }

   return stream;
}
```

Pode-se ter mais de um arquivo aberto ao mesmo tempo num programa e o número de arquivos que podem estar simultaneamente abertos varia de acordo com o sistema operacional utilizado. A constante simbólica **FOPEN_MAX**, definida em `<stdio.h>`, representa o número máximo de arquivos que a respectiva implementação de C garante que podem estar simultaneamente abertos. Quer dizer, um número bem maior de arquivos pode estar aberto ao mesmo tempo, mas não há garantia de que isso ocorra. De qualquer modo, raramente, um programa simples, como aqueles apresentados neste livro, precisa abrir mais de dois arquivos ao mesmo tempo.

A constante simbólica **FILENAME_MAX**, definida em `<stdio.h>`, representa o número máximo de caracteres (incluindo `'\0'`) que um string que representa um nome de arquivo pode ter numa dada implementação de C. Essa constante é importante porque permite dimensionar seguramente um array usado para armazenar um string que representa o nome de um arquivo.

2.3.2 Streams de Texto e Binários

Um **stream de texto** é aquele associado a um arquivo (usualmente, de texto) aberto em modo de texto; i.e., usando um dos modos de abertura que se apresentam na **Tabela 2–1**. Normalmente, não faz sentido associar um arquivo binário a um stream de texto.

Streams de texto pressupõem a existência de bytes que representam quebras de linha e uma quebra de linha pode ter diferentes interpretações dependendo do sistema operacional usado como hospedeiro. Comumente, uma quebra de linha pode ser representada por um ou dois bytes e um byte que representa quebra de linha num sistema operacional pode não representar quebra de linha em outro sistema.

Quando um stream de texto é processado, as funções de leitura e escrita do módulo **stdio** da biblioteca padrão de C realizam interpretações de quebra de linha de acordo com o sistema hospedeiro que vigora. Em sistemas operacionais da família Unix, não existem tais interpretações. Em outras palavras, em sistemas operacionais dessa família, não há diferença entre streams de texto e binários.

Um **stream binário** é aquele associado a um arquivo aberto em modo binário (v. adiante). Tipicamente, arquivos binários são associados a streams binários, mas também é comum associar arquivos de texto a streams binários. Num stream binário, bytes são processados sem nenhuma interpretação. Ou seja, cada byte lido no arquivo associado ao stream é armazenado em memória exatamente como ele é e cada byte lido em memória é escrito no arquivo associado ao stream exatamente como ele é.

2.3.3 Modos de Abertura

Existem dois conjuntos de modos de abertura de arquivos: um deles é dirigido para streams de texto e o outro se destina a streams binários. O conjunto de modos de acesso para streams de texto é apresentado na **Tabela 2–1**.

Modo de Acesso	Descrição
`"r"`	Abre um arquivo existente apenas para leitura em modo de texto.
`"w"`	Cria um arquivo apenas para escrita em modo de texto. Se o arquivo já existir, seu conteúdo será destruído.
`"a"`	Abre um arquivo existente em modo de texto para acréscimo; i.e., com escrita permitida apenas ao final do arquivo. Se o arquivo com o nome especificado não existir, um arquivo com esse nome será criado.
`"r+"`	Abre um arquivo existente para leitura e escrita em modo de texto.
`"w+"`	Cria um arquivo para leitura e escrita em modo de texto. Se o arquivo já existir, seu conteúdo será destruído.
`"a+"`	Abre um arquivo existente ou cria um arquivo em modo de texto para leitura e acréscimo. Podem-se ler dados em qualquer parte do arquivo, mas eles podem ser escritos apenas ao final do arquivo.

Tabela 2–1: Modos de Acesso para Streams de Texto

Em termos de formato, única diferença entre os especificadores de modo de acesso para streams binários mostrados na **Tabela 2–2** e aqueles apresentados na **Tabela 2–1** para streams de texto é que os especificadores para streams binários têm a letra *b* acrescentada. Streams de texto também podem ter acrescidos a letra *t* em seus modos de abertura, como mostra a **Tabela 2–2**.

Modo de Acesso para Streams Binários	Modo de Acesso Equivalente para Streams 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 2–2: Modos de Acesso para Streams Binários e de Texto

Os modos de abertura `"r+"` (ou `"r+b"`), `"w+"` (ou `"w+b"`) e `"a+"` (ou `"a+b"`), coletivamente denominados modos de atualização, causam certa confusão entre iniciantes, pois todos eles permitem leitura e escrita. A **Tabela 2–3** tenta esclarecer eventuais dúvidas com relação a esses modos de abertura.

	Modo de Abertura		
	`"r+"` ou `"r+b"`	`"w+"` ou `"w+b"`	`"a+"` ou `"a+b"`
Arquivo deve existir?	Sim	Não. Se ele existir, seu conteúdo será destruído	Não. Se ele existir, seu conteúdo será preservado
Onde escrita pode ocorrer?	Em qualquer local	Em qualquer local	Ao final do arquivo

Tabela 2–3: Modos de Acesso Usados em Atualização de Arquivo

	Modo de Abertura		
	"r+" ou "r+b"	"w+" ou "w+b"	"a+" ou "a+b"
Recomendado quando?	Dados precisam ser lidos, atualizados e escritos novamente no arquivo	Um conteúdo para o arquivo deve ser criado e, depois, lido e casualmente modificado	Dados existentes no arquivo precisam ser lidos e novos dados precisam ser acrescentados

Tabela 2–3: Modos de Acesso Usados em Atualização de Arquivo

2.3.4 Fechando um Arquivo

Quando um programa não precisa mais processar um arquivo, deve-se fechá-lo utilizando a função **fclose()**, que tem o seguinte o protótipo:

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

Essa função possui como único parâmetro um ponteiro de stream associado a um arquivo aberto pela função **fopen()** e retorna zero, se o arquivo for fechado com sucesso. Caso ocorra algum erro durante a operação, ela retorna a constante **EOF**.

Ao fechar-se um arquivo, libera-se o espaço ocupado pela estrutura **FILE** associada ao arquivo e alocada pela função **fopen()** quando ele foi aberto. Antes de liberar esse espaço, quando se trata de um **stream de saída** com buffering (v. **Seção 2.5**), a função **fclose()** descarrega o conteúdo da área de buffer para o arquivo. No caso de um arquivo aberto apenas para leitura (**stream de entrada**) que utilize buffering, o conteúdo do buffer é descartado.

Um erro frequente entre os iniciantes em C é utilizar o nome do arquivo como parâmetro, ao invés do ponteiro de stream associado a ele, numa chamada de **fclose()** [p. ex., `fclose("teste.dat")`]. Isso certamente trará um sério problema durante a execução do programa, pois apesar de um string ser interpretado como um ponteiro, esse ponteiro, obviamente, não é compatível com um ponteiro para uma estrutura do tipo **FILE**.

Um engano ainda mais frequente é achar que a função **fclose()** nunca falha numa tentativa de fechamento de arquivo e, assim, o valor retornado por essa função raramente é testado como deveria. De fato, a ocorrência de erro numa operação de fechamento é muito mais incomum do que numa operação de abertura. Entretanto um programa robusto não pode contar com o acaso e deve testar o resultado de qualquer operação de fechamento.

A função `FechaArquivo()`, apresentada a seguir, tenta fechar um arquivo e, quando isso não é possível, aborta o programa e seus parâmetros são:

- `stream` (entrada) — stream associado ao arquivo
- `nomeArq` (entrada) — nome do arquivo ou **NULL**; se esse parâmetro não for **NULL**, o nome do arquivo aparece na mensagem de erro

```
void FechaArquivo(FILE *stream, const char *nomeArq)
{
   /* Se fclose() retornar um valor diferente de zero ocorreu */
   /* algum erro na tentativa de fechamento do arquivo. Nesse */
   /* caso, apresenta mensagem de erro e aborta o programa.   */

   if (fclose(stream)) { /* Erro de fechamento */
      fprintf( stderr, "\a\n>>> Ocorreu erro no fechamento do arquivo %s."
              "\n>>> O programa sera' encerrado.\n", nomeArq ? nomeArq : "");
      exit(1); /* Aborta o programa */
   }
}
```

A função `FechaArquivo()` apresentada acima é tão simples de usar quanto **fclose()** e oferece como vantagem o fato de testar o resultado de uma operação de fechamento de arquivo, de modo que torna-se desnecessário para o programador escrever uma instrução condicional a cada chamada de **fclose()**. Devido às vantagens oferecidas pela função `FechaArquivo()`, ela será doravante usada em detrimento da função **fclose()**, a não ser quando o fechamento do arquivo ocorrer logo antes de o programa encerrar. Nesse último caso, não faz muito sentido usar a função `FechaArquivo()`, já que o programa será encerrado de qualquer modo.

Como boa norma de programação, uma função só deve fechar um arquivo se ela tiver sido responsável pela abertura desse arquivo. Em outras palavras, uma função que recebe um stream aberto como parâmetro (i.e., um parâmetro do tipo **FILE** *), não deve fechar o arquivo:

A função que abre um arquivo é aquela que tem a responsabilidade de fechá-lo.

É importante seguir essa norma porque uma função que abre um arquivo espera tê-lo ainda aberto quando outra função é chamada e conclui sua execução (v. exemplos na **Seção 2.15**).

Qualquer sistema operacional fecha os arquivos abertos por um programa quando o programa termina normalmente e muitos sistemas os fecham mesmo quando o programa é abortado. Mas como boa norma de programação, é sempre recomendado fechar um arquivo quando não é mais necessário processá-lo.

2.4 Ocorrências de Erros

Esta seção apresenta as funções **feof()** e **ferror()**, que se destinam a apontar erros em processamento de arquivos. Dedique bastante atenção ao uso dessas duas funções, pois elas são de importância essencial em processamento de arquivos.

A função **feof()** retorna um valor diferente de zero quando há uma tentativa de leitura além do final do arquivo associado ao respectivo stream que ela recebe como parâmetro e seu protótipo é:

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

Para verificar se ocorreu erro após uma determinada operação de entrada ou saída, pode-se usar a função **ferror()**, que tem como protótipo:

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

A função **ferror()** retorna um valor diferente de zero após ocorrer algum erro de processamento associado ao stream recebido como parâmetro ou zero, em caso contrário.

A constante simbólica **EOF**, definida no cabeçalho `<stdio.h>`, é retornada por diversas funções que lidam com arquivos para indicar ocorrência de erro ou de tentativa de leitura além do final de um arquivo. Tipicamente, essa constante está associada a um valor negativo do tipo **int** e esse valor é dependente de implementação[2]. O uso dessa constante causa muita confusão entre programadores de C, porque, muitas vezes, ela é ambígua. Quer dizer, ora ela indica tentativa de leitura além do final de arquivo, ora ela indica ocorrência de erro. Além disso, muitas vezes, essa constante só pode ser usada de modo confiável com arquivos de texto.

Para evitar confusão, siga sempre as recomendações resumidas nos quadros a seguir:

❒ ***Após qualquer operação de leitura num arquivo, use feof() para verificar se houve tentativa de leitura além do final do arquivo.***

[2] Apesar de **EOF** ter seu nome derivado de *End Of File* (*final de arquivo*, em inglês), essa constante não é retornada apenas quando uma função de leitura tenta ler além do final de um arquivo nem representa um caractere armazenado num arquivo.

- ***Após qualquer operação de leitura num arquivo, use feof() para verificar se houve tentativa de leitura além do final do arquivo.***
- ***Após qualquer tentativa de leitura ou escrita num arquivo, use ferror() para verificar se ocorreu algum erro durante a operação.***
- ***Evite usar EOF em substituição a feof() ou ferror().***

Um detalhe importante com respeito a indicação de erro é que, quando ocorre um erro durante o processamento de um arquivo, o campo da estrutura **FILE** associada ao arquivo que armazena essa informação (v. **Seção 2.2**) permanece com essa indicação de erro até que ela seja removida. Isso significa que qualquer chamada subsequente de **ferror()** que tenha como parâmetro um stream para o qual haja uma indicação de erro continuará a indicar que houve erro na última operação de entrada ou saída no stream, mesmo quando esse não é o caso. Portanto se for necessário processar novamente um stream para o qual há um indicativo de erro, esse indicativo deve ser removido antes de o processamento do stream prosseguir. Essa mesma discussão se aplica ao caso de indicação de final de arquivo.

Em qualquer caso mencionado, normalmente, a função **rewind()** (v. **Seção 2.12**), cuja finalidade precípua é mover o indicador de posição de um arquivo (v. **Seção 2.11.2**) para seu início, remove a condição de erro ou de final de arquivo de um stream. Por outro lado, **fseek()** (v. **Seção 2.11.2**) e **ungetc()** (v. **Seção 2.10**) removem apenas indicativo de final de arquivo num stream. Finalmente, a função **clearerr()** tem como única finalidade remover ambos os indicativos de erro e de final de arquivo, mas, na prática, raramente ela se faz necessária.

2.5 Buffering e a Função fflush()

Buffer é uma área de memória na qual dados provenientes de um arquivo ou que se destinam a um arquivo são armazenados temporariamente. O uso de buffers permite que o acesso a dispositivos de entrada ou 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**. Nesse tipo de buffering, o sistema armazena caracteres até que um caractere de quebra de linha, representado por `'\n'`, seja encontrado ou até que o buffer esteja repleto. Esse tipo de buffering é utilizado, por exemplo, quando dados são lidos via teclado. Nesse caso, os dados são armazenados num buffer até que um caractere de quebra de linha seja introduzido (p. ex., por meio da digitação de `[ENTER]`) e, quando isso acontece, os caracteres digitados são enviados para o programa. Os streams padrão **stdin** e **stdout** (v. **Seção 2.6**) utilizam buffering de linha.
- **Buffering de bloco**. Nesse caso, bytes são armazenados até que um bloco inteiro seja preenchido (independentemente de o caractere `'\n'` ser encontrado). O tamanho padrão de um bloco é tipicamente definido de acordo com o sistema operacional utilizado. Como padrão, streams associados a arquivos armazenados usam buffering de bloco.

Em qualquer caso, pode-se explicitamente descarregar o buffer associado a um stream de saída ou atualização (v. **Seção 2.3**), forçando o envio de seu conteúdo para o respectivo arquivo associado, por meio de uma chamada da função **fflush()**. Por exemplo, a chamada:

```
fflush(stdout);
```

força a descarga da área de buffer associada ao stream **stdout** (v. **Seção 2.6**), enviando o conteúdo desse buffer para o meio de saída padrão.

A função **fflush()** serve para descarregar apenas buffers associados a streams de saída ou atualização. Ou seja, não existe nenhuma função na biblioteca padrão de C que descarregue buffers associados a streams de entrada.

Algumas implementações de C permitem que a função **fflush()** seja utilizada para expurgar caracteres remanescentes em buffers associados a streams de entrada [p. ex., `fflush(stdin)`], mas esse uso da função **fflush()** não é portável, uma vez que o padrão ISO não especifica que essa função possa ser utilizada com streams de entrada.

Quando o parâmetro único de **fflush()** é **NULL**, essa função descarrega todos os buffers associados a streams de escrita ou atualização correntemente em uso num programa.

A função **fflush()** retorna **EOF** (v. **Seção 2.4**), se ocorrer algum erro durante sua execução; caso contrário, ela retorna `0`. Contudo, raramente, o valor retornado por essa função é testado.

2.6 Streams Padrão

Um **stream padrão** é um stream para o qual existem funções que o processam sem necessidade de especificação explícita do stream. Por exemplo, as funções **scanf()** e **printf()**, utilizadas abundantemente neste livro, não requerem especificação de um stream no qual será feita a leitura ou escrita de dados, respectivamente.

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 desses streams é apresentada abaixo.

- **stdin** — representa a entrada padrão de dados e, no caso de computadores pessoais, tipicamente, é associado ao teclado. A função **scanf()** faz leitura nesse stream.
- **stdout** — representa a saída padrão de dados e, no caso de computadores pessoais, tipicamente, é associado a um monitor de vídeo (tela). A função **printf()** escreve nesse stream.
- **stderr** — representa a saída padrão de mensagens de erro e é associado ao mesmo dispositivo que stdout. A função **perror()**, declarada em `<stdio.h>`, exibe mensagens de erro detectados pelo sistema nesse stream, mas essa função tem pouca importância prática e não receberá maiores considerações neste livro.

2.7 Leitura e Escrita Formatadas

2.7.1 Saída Formatada

Uma operação de escrita formatada consiste em:

1. Ler valores de tipos primitivos armazenados em memória.
2. Converter os dados lidos em texto de acordo com uma especificação de formato.
3. Escrever o texto resultante num stream de texto (v. **Seção 2.2**).

O funcionamento de uma função de escrita formatada é ilustrado na **Figura 2–1** (suponha que a largura do tipo **int** é 16 bits).

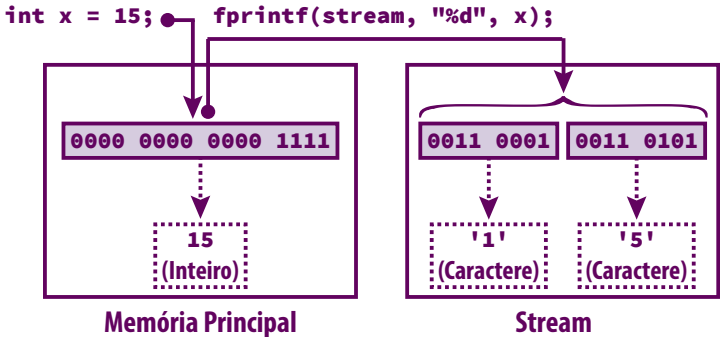

Figura 2–1: Escrita Formatada

A função **fprintf()**, que aparece na **Figura 2–1**, é uma função de escrita formatada semelhante a **printf()** e será discutida adiante.

Leitura formatada é um tipo de processamento de arquivo que consiste em:

1. Ler caracteres num stream de texto.
2. Tentar converter os caracteres lidos em valores de tipos de dados primitivos usando especificadores de formato.
3. Armazenar os valores convertidos em variáveis.

A ação de uma função de leitura formatada é ilustrada na **Figura 2–2** (novamente, suponha que a largura do tipo **int** é 16 bits).

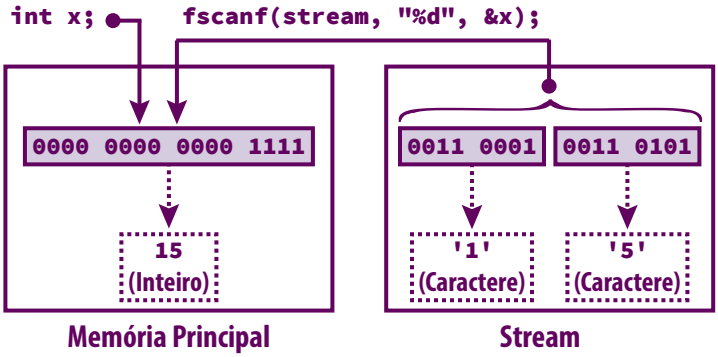

Figura 2–2: Leitura Formatada

A função **fscanf()**, que aparece na **Figura 2–2**, é uma função de leitura formatada semelhante a **scanf()** e será discutida adiante.

2.7.2 Funções fprintf() e sprintf()

A única diferença entre as funções **fprintf()** e **printf()** é que **fprintf()** permite a especificação de um stream, enquanto **printf()** escreve apenas no stream padrão **stdout** (v. **Seção 2.6**). Ambas as funções usam um string de formatação como parâmetro e os especificadores de formato que podem ser utilizados no string de formatação de cada função são os mesmos. No caso de **fprintf()** o stream no qual será efetuada a escrita é o primeiro parâmetro e seu string de formatação é o segundo parâmetro.

As funções **printf()** e **fprintf()** possuem a mesma especificação de retorno. Isto é, elas retornam o número de caracteres escritos, quando bem-sucedidas ou **EOF**, quando ocorre algum erro. Porém muito raramente, o valor retornado por essas funções é usado.

As funções **printf()** e **fprintf()** fazem parte de um conjunto de funções denominadas coletivamente família printf. Outra função notável nessa família de funções, e que será discutida em seguida, é **sprintf()**.

A função **sprintf()** converte valores de tipos primitivos em strings. Isto é, essa função escreve num array de caracteres dados formatados de acordo com um string de formatação, acrescentando o caractere terminal `'\0'` ao final do processo de escrita. O protótipo dessa função é:

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

Nesse protótipo, os parâmetros têm os seguintes significados:

- `ar` é o endereço do array no qual será feita a escrita.
- `formato` é um string de formatação como aqueles usados com **printf()**.
- `...` representa valores que serão escritos no array de acordo com o string de formatação. Esses três pontos fazem realmente parte do protótipo da função, pois **sprintf()** é uma função com parâmetros variantes.

A função **sprintf()** retorna o número de caracteres escritos no array, sem incluir o caractere terminal `'\0'`.

É preciso ser cuidadoso com o uso da função **sprintf()**, pois ela pode escrever além do limite do array recebido como parâmetro, causando corrupção de memória. Além disso, um engano comum no uso de **sprintf()** é achar que a utilização de um especificador de largura pode resolver um eventual problema de corrupção de memória, mas isso não ocorre, porque tal especificador estabelece um valor mínimo, e não um valor máximo, como se poderia supor. Por exemplo, na chamada da função **sprintf()** a seguir, o valor do especificador de largura (i. e, `4`) informa à função que, pelo menos, quatro caracteres devem ser escritos no array apontado por `ar`.

```
sprintf(ar, "\%4d", n);
```

2.7.3 Entrada Formatada e fscanf()

A função **fscanf()** é idêntica à função **scanf()** em quase todos os aspectos. Ambas fazem parte de um conjunto de funções coletivamente denominadas família scanf e a única diferença entre elas é que **scanf()** faz leitura apenas no stream padrão **stdin** (v. **Seção 2.6**), enquanto a função **fscanf()** requer um parâmetro que especifique o stream no qual a leitura será efetuada. Na função **fscanf()**, o string de formatação é o segundo parâmetro e o primeiro parâmetro é do tipo **FILE** *. É exatamente esse primeiro parâmetro que especifica o stream no qual **fscanf()** fará a leitura.

As funções **fscanf()** e **scanf()** possuem a mesma especificação de retorno: quando bem-sucedidas, elas retornam o número de valores lidos, convertidos e armazenados em variáveis; quando ocorre algum erro de leitura, elas retornam **EOF** (v. **Seção 2.4**).

Uma informação importante que o aprendiz de C deve memorizar é que quase todas as funções do módulo stdio cujos nomes começam com *f* requerem um parâmetro do tipo **FILE** * que representa o stream sobre o qual elas atuam. A única exceção a essa regra é a função **fopen()**, cujo nome começando com *f* significa que ela retorna um valor do tipo **FILE** *.

2.8 Trabalhando com Arquivos Temporários

Um **arquivo temporário** é um arquivo utilizado para armazenar dados temporariamente enquanto outro arquivo, que irá armazenar os mesmos dados definitivamente, está sendo processado. Normalmente, arquivos temporários são removidos quando deixam de ser necessários. A propósito, assegurar que um programa sempre remove arquivos temporários por ele utilizados é considerada boa norma de programação.

No cabeçalho `<stdio.h>`, são declaradas duas funções que permitem ao programador o uso de arquivos temporários: **tmpfile()** e **tmpnam()**.

A função **tmpfile()** cria um arquivo temporário e abre-o para leitura e escrita no modo `"w+b"` (v. **Seção 2.3**). Essa função retorna o stream associado ao arquivo recém-criado, se não ocorrer nenhum erro ou **NULL**, se o arquivo não puder ser criado. O protótipo dessa função é:

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

O diretório no qual o arquivo temporário é criado depende do sistema operacional em uso. Além disso, se o caminho que conduz ao arquivo temporário não for modificado após a criação desse arquivo, ele será removido quando for fechado ou quando o programa encerrar.

A função **tmpnam()** cria um string que pode ser utilizado como nome de um arquivo temporário e seu protótipo é:

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

O parâmetro único de **tmpnam()** pode ser **NULL** ou o endereço de um array no qual será armazenado o nome do arquivo. Nesse último caso, o array deve ser capaz de conter um número de caracteres pelo menos igual a **L_tmpnam**, que é uma constante simbólica definida em `<stdio.h>`. A função **tmpnam()** retorna o endereço do array recebido como parâmetro quando esse parâmetro não for **NULL**. Caso contrário, a função retorna o endereço de um array de duração fixa local à função contendo o nome de arquivo gerado.

É importante observar que a função **tmpnam()** gera apenas nomes de arquivo; i.e., ela não cria, abre ou remove arquivos. Por isso arquivos temporários criados por intermédio da função **tmpnam()** não são removidos automaticamente quando o programa termina, de modo que é responsabilidade do programador excluir arquivos assim criados [p. ex., usando **remove()**]. Além disso, quando o parâmetro de **tmpnam()** é **NULL**, uma chamada subsequente da função pode modificar o conteúdo do array cujo endereço é retornado.

Em termos práticos, na maioria das vezes, é mais recomendável usar a função **tmpfile()** para a criação de arquivos temporários.

2.9 Removendo e Rebatizando Arquivos

Remover e dar um novo nome a um arquivo constituem tarefas comuns de gerenciamento de arquivos, e não exatamente de processamento de arquivos. A biblioteca padrão de C oferece duas funções para essas finalidades, a saber, respectivamente, **remove()** e **rename()**.

A função **remove()** exclui o arquivo cujo nome (string) é recebido como parâmetro e seu protótipo é:

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

Essa função retorna zero, quando obtém êxito ou um valor diferente de zero, em caso contrário. Se o arquivo a ser removido estiver aberto, o resultado da operação é imprevisível.

A função **rename()** altera o nome de um arquivo e tem como protótipo:

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

O primeiro parâmetro dessa função é o nome corrente do arquivo a ser rebatizado, enquanto o segundo parâmetro é o nome que o arquivo terá após ter seu nome alterado. Essa função retorna zero quando obtém êxito ou um valor diferente de zero, em caso contrário. Se o arquivo a ser rebatizado estiver aberto ou se já existir um arquivo com o novo nome especificado como parâmetro, o resultado será indefinido.

2.10 A Função ungetc()

A função **ungetc()** é considerada exótica porque permite inserir um caractere num stream de entrada e seu protótipo é:

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

O primeiro parâmetro dessa função é o caractere a ser inserido no stream e o segundo parâmetro representa o próprio stream, que deve estar associado a um arquivo aberto num modo que permite leitura. O retorno de **ungetc()** é o caractere inserido no stream, quando não ocorre erro durante a operação ou **EOF** (v. **Seção 2.4**), em caso contrário.

O caractere inserido no stream por **ungetc()** será lido na próxima operação de leitura nesse stream, o que significa que essa função decrementa o indicador de posição de arquivo e, portanto, não deve ser chamada quando ele estiver apontando para o início do arquivo. Ademais, uma chamada bem-sucedida desta função elimina uma eventual sinalização de final de arquivo no stream.

O caractere inserido num stream por **ungetc()** não precisa necessariamente ter sido lido nesse stream. Contudo, na prática, o caractere inserido num stream por essa função é, normalmente, o último caractere que foi lido no mesmo stream. Além disso, não há garantia de inserção de dois ou mais caracteres num stream com o uso repetido da função **ungetc()** sem chamadas intercaladas de operações de leitura no mesmo stream, como mostram os fragmentos de programa a seguir:

```
ungetc('A', stdin);  /* Inserção garantida */
ungetc('B', stdin);  /* Pode não haver inserção */
...
ungetc('A', stdin);  /* Inserção garantida */
getchar();
ungetc('B', stdin);  /* Inserção garantida */
```

Quando uma chamada de **fseek()** ou **rewind()** (v. **Seção 2.12**) é executada com êxito, ela descarta qualquer caractere inserido no stream por meio de **ungetc()**.

2.11 Tipos de Processamento

Uma vez que um arquivo tenha sido aberto conforme foi descrito na **Seção 2.3**, pode-se usar o ponteiro de stream que o representa para processá-lo. Processar um arquivo significa ler ou escrever dados nele usando o respectivo stream como intermediário. Processamento de arquivos pode ser categorizado conforme exposto adiante:

- **Processamento sequencial**. Quando um stream é processado sequencialmente, suas partições são acessadas uma a uma na ordem em que se encontram no stream. Todo stream permite esse tipo de acesso e, de acordo com as partições nas quais o stream é logicamente dividido, esse tipo de processamento pode ainda ser subdividido em:
  - **Por byte** (ou **por caractere**). Nesse tipo de processamento, as funções utilizadas leem ou escrevem um byte por vez. Esse tipo de processamento é apropriado para qualquer tipo de stream e será apresentado na **Seção 2.11.1**.
  - **Por linha**. As funções utilizadas nesse tipo de processamento leem ou escrevem uma linha de cada vez. Esse tipo de processamento é dirigido para streams de texto e será explorado na **Seção 2.11.1**.
  - **Por bloco**. No contexto de processamento de arquivos, um bloco de memória (ou apenas bloco) é um array de bytes. Mas como será visto adiante, qualquer variável pode ser vista como um array de bytes. Assim as funções utilizadas nesse tipo de processamento leem ou escrevem uma variável ou um array de variáveis de um determinado tipo de cada vez. Esse tipo de processamento é mais apropriado para streams binários associados a arquivos binários e será descrito em detalhes na **Seção 2.11.1**.
  - **Formatado**. As funções que executam processamento dessa natureza convertem caracteres em valores de tipos de dados primitivos durante uma operação de leitura e realizam o inverso durante uma operação de escrita. As funções **scanf()** e **printf()** constituem exemplos de funções usadas em processamento formatado de arquivos. Esse tipo de processamento, que é conveniente apenas para streams de texto, foi discutido na **Seção 2.7**.
- **Processamento por acesso direto**. Num processamento dessa natureza, um conjunto de bytes pode ser acessado num determinado local de um arquivo sem que os bytes que o precedem sejam necessariamente

acessados, porém nem todo arquivo permite esse tipo de acesso. Esse tipo de processamento é conveniente para arquivos binários que podem ser indexados (i.e., divididos em partições de mesmo tamanho) e que, obviamente, admitem acesso direto. Esse tipo de processamento será apresentado na **Seção 2.11.2**.

A **Tabela 2–4** e a **Tabela 2–5** têm o intuito de servirem como rápida referência para ajudar o programador a decidir que tipo de processamento é conveniente numa determinada situação.

Processamento	Funções Tipicamente Usadas	Conveniente para arquivo...
Por byte	❒ **fgetc()** (leitura) ❒ **fputc()** (escrita)	Texto ou binário
Por linha	❒ **fgets()** (leitura) ❒ **fputs()** (escrita)	Texto
Por bloco	❒ **fread()** (leitura) ❒ **fwrite()** (escrita)	Binário
Formatado	❒ **fscanf()** (leitura) ❒ **fprintf()** (escrita)	Texto (apenas)

Tabela 2–4: Processamento Sequencial de Arquivos (Resumo)

Ação	Funções Usadas	O Que Faz
Movimentação	**fseek()**	*Move o indicador de posição do arquivo para um local determinado*
Localização	**ftell()**	*Informa o local onde se encontra o indicador de posição do arquivo*
Processamento	❒ **fread()** (leitura) ❒ **fwrite()** (escrita)	*Lê ou escreve um bloco no local onde se encontra o indicador de posição do arquivo*

Tabela 2–5: Processamento de Arquivos com Acesso Direto (Resumo)

2.11.1 Processamento Sequencial

Esta seção descreve em detalhes as categorias de processamento sequencial.

Por Byte

Existem duas funções para processamento de um stream byte a byte: **fgetc()**, que lê um byte no stream e **fputc()**, que escreve um byte no stream. Antes de retornarem, essas funções movem o indicador de posição do stream para o próximo caractere a ser lido ou escrito. Os protótipos dessas funções são apresentados na **Tabela 2–6**.

Função	Protótipo
fgetc()	`int fgetc(FILE *stream)`
fputc()	`int fputc(int byte, FILE *stream)`

Tabela 2–6: Protótipos de Funções para Processamento de Caracteres (Bytes)

Na **Seção 2.15**, serão apresentados exemplos de uso prático das funções **fgetc()** e **fputc()**.

Por Linha

O processamento de arquivo linha por linha é conveniente apenas para streams de texto. Existem duas funções do módulo stdio que leem e escrevem uma linha num dado stream de texto, respectivamente: **fgets()** e **fputs()**. A função **fgets()** tem como protótipo:

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

O primeiro parâmetro dessa função é o endereço do array que armazenará o string lido e o segundo parâmetro é o tamanho desse array. O terceiro parâmetro de **fgets()** especifica o stream associado ao arquivo no qual a leitura será efetuada. No caso de leitura via teclado, esse parâmetro é, naturalmente, **stdin** (v. **Seção 2.6**).

A função **fgets()** é capaz de ler `n - 1` caracteres, mas a leitura pode encerrar prematuramente se uma quebra de linha (`'\n'`) for encontrada ou o final do arquivo for atingido. Essa função armazena automaticamente um caractere nulo (`'\0'`) após o último caractere armazenado no array `ar[]`. A função **fgets()** retorna o endereço do array recebido como parâmetro quando consegue cumprir sua missão ou **NULL**, quando ocorre erro ou tentativa de leitura além do final do arquivo sem que ela tenha conseguido ler nenhum caractere.

O valor passado como segundo parâmetro pode ser menor do que o tamanho do array (primeiro parâmetro) se o programador não desejar preencher todo o array com o string lido. Mas se esse valor for maior do que o verdadeiro tamanho do array, poderá haver corrupção de memória. Nesse último caso, todavia, o erro é causado por descuido do programador e não é inerente à função **fgets()**.

É interessante ressaltar que, quando a função **fgets()** encontra o caractere `'\n'` durante uma leitura, ela o inclui no string resultante da leitura e, muitas vezes, isso constitui um incômodo para o programador. Se esse for o caso, para remover esse caractere, o programador deve notar que ele deve ser o último caractere do string antes do caractere terminal, porque, ao encontrar o caractere `'\n'`, **fgets()** encerra a leitura e acrescenta o caractere `'\0'` ao final dos caracteres armazenados no array recebido como parâmetro. Por exemplo, suponha que essa função seja chamada como no seguinte trecho de programa:

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

e o usuário introduz apenas cinco caracteres seguidos de `[ENTER]`. Então quando executada, a função **fgets()** encontrará o caractere `'\n'`, antes de ler os nove caracteres que lhe são permitidos. Assim após o retorno dessa função, a situação no array `ar[]` pode ser esquematizada como na **Figura 2–3**, na qual `x` representa cada caractere introduzido pelo usuário antes de ele digitar `[ENTER]`.

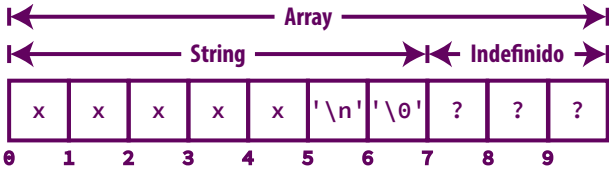

FIGURA 2–3: FUNÇÃO FGETS() LENDO QUEBRA DE LINHA

Como se pode verificar na ilustração, quando lido, o caractere `'\n'` ocupa a posição do array determinada por:

```
strlen(ar) - 1
```

Assim o trecho de programa a seguir é capaz de remover do string o caractere `'\n'`:

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

if (ar[posicao] == '\n') /* '\n' foi encontrado */
   ar[posicao] = '\0'; /* Sobrescreve-o com '\0' */
```

Uma maneira equivalente e mais sucinta de remover o caractere `'\n'` é obtida por meio da função **strchr**(), como é mostrado abaixo:

```
char *p = strchr(ar, '\n');

if (p) /* '\n' foi encontrado */
   *p = '\0'; /* Sobrescreve-o com '\0' */
```

A função **fputs**() é usada para escrita de linhas num stream de texto e tem o seguinte protótipo:

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

Nesse protótipo, os parâmetros são interpretados como:

- `s` é o endereço de um string.
- `stream` representa o stream no qual será feita a escrita.

A função **fputs**() escreve todos os caracteres do string `s` no stream recebido como parâmetro até que o caractere nulo seja encontrado (esse caractere nulo não é escrito no stream). A função **fputs**() retorna um valor não negativo quando a escrita é bem-sucedida; caso contrário, ela retorna **EOF**. Essa função não escreve no stream 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.

A **Seção 2.15** apresentará exemplos de uso prático das funções **fgets**() e **fputs**().

Por Bloco

Conforme foi visto no início deste capítulo, um bloco (de memória) é apenas um array unidimensional de bytes. Esses bytes podem ser agrupados para constituir elementos multibytes de um array. Por exemplo, um array de elementos do tipo **double** pode ser interpretado desse modo ou como um array de bytes, pois não apenas os elementos do tipo **double** são contíguos em memória, como também há contiguidade entre os bytes que compõem cada elemento do tipo **double**. Assim quando se lê ou escreve um bloco, é necessário especificar o número de elementos do bloco e o tamanho (i.e., o número de bytes) de cada elemento.

As funções do módulo stdio usadas para entrada e saída de blocos são **fread**() e **fwrite**(), respectivamente.

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

```
size_t  fread(void  *ar, size_t tamanho, size_t n, FILE *stream)
```

As interpretações dos parâmetros nesse protótipo são as seguintes:

- `ar` é o endereço do array de bytes no qual o bloco lido será armazenado. O tipo **void** * utilizado na declaração desse parâmetro permite que ele seja compatível com ponteiros e endereços de variáveis de quaisquer tipos.
- `tamanho` é o tamanho de cada elemento do array.
- `n` é o número de elementos do tamanho especificado que serão lidos no stream e armazenados no array.
- `stream` é o stream no qual será feita a leitura.

A função **fread**() retorna o número de elementos que foram realmente lidos. Esse valor deverá ser igual ao valor do terceiro parâmetro da função, a não ser que ocorra um erro ou o final do stream seja atingido antes da leitura de todos os elementos especificados nesse parâmetro.

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

```
size_t fwrite(const void *ar, size_t tamanho, size_t n, FILE *stream)
```

Os parâmetros dessa função são interpretados como:

- `ar` é o endereço do array que armazena os bytes que serão escritos no stream. O tipo **void** * utilizado na declaração desse parâmetro permite que ele seja compatível com ponteiros e endereços de variáveis de quaisquer tipos.
- `tamanho` é o tamanho de cada elemento do array.
- `n` é o número de elementos do array que serão escritos no stream.
- `stream` representa o stream no qual será feita a escrita.

A função **fwrite()** retorna o número de itens 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 bytes armazenados em memória e escreve-os num stream, enquanto **fread()** lê bytes num stream e armazena-os em memória. Usando qualquer dessas funções, o programador deve tomar cuidado para não especificar um número de itens (terceiro parâmetro) que ultrapasse o número de elementos do array.

Deve-se ressaltar que quando se fala em *array de bytes*, não se está necessariamente considerando um array de elementos de um tipo específico. 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 primitivos, tais como **int** ou **double**, ou mais complexos, e não apenas arrays convencionais, como parece ser sugerido. Por exemplo, para escrever num arquivo um único valor do tipo **double** armazenado em memória, pode-se usar o seguinte fragmento de programa:

```
double  umDouble = 2.54;
FILE   *stream = fopen("MeuArquivo", "wb");
...
fwrite(&umDouble, sizeof(double), 1, stream);
```

Para ler um valor do tipo **double** num arquivo binário e armazená-lo numa variável, pode-se utilizar, de modo semelhante, a função **fread()**:

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

Algoritmo Geral para Leitura Sequencial de Arquivos

Em geral, leitura sequencial num arquivo pode ser realizada seguindo-se o algoritmo delineado na **Figura 2–4**, que foi escrito com uma mistura de linguagem algorítmica e português.

ALGORITMO LEITURASEQUENCIALDEARQUIVO

1. Repita:
   - **1.1** Leia uma partição do arquivo
   - **1.2** Se houve tentativa de leitura além do final do arquivo, encerre o laço
   - **1.3** Se ocorreu erro de leitura, encerre o laço
   - **1.4** Processe a partição lida
2. Se ocorreu erro de leitura, informe que a leitura foi malsucedida

FIGURA 2–4: ALGORITMO GERAL PARA LEITURA SEQUENCIAL DE ARQUIVOS

Deve-se salientar o seguinte a respeito do algoritmo LeituraSequencialDeArquivo e sua implementação em C:

- Na implementação do laço do algoritmo acima, tipicamente, usa-se um laço **while**, como:

```
while(1) {
   ...
}
```

- Se a natureza da partição considerada for byte, linha ou bloco, a leitura deve ser efetuada usando, respectivamente, **fgetc()**, **fgets()** ou **fread()**.
- Tentativa de leitura além do final do arquivo é checada chamando-se **feof()** (v. **Seção 2.4**).
- Ocorrência erro de leitura deve ser verificada chamando-se **ferror()** (v. **Seção 2.4**).
- *Processar a partição lida* significa efetuar qualquer tipo de operação sobre os dados que, nesse instante, encontram-se armazenados em memória.
- Após o encerramento do laço de repetição, o algoritmo informa, se for o caso, a ocorrência de erro de leitura. Essa comunicação pode ser efetuada, por exemplo, por meio de um valor de retorno da função que implementa o algoritmo.

Existem situações particulares nas quais o algoritmo acima não se aplica. Por exemplo, o último programa apresentado acima escreve e lê todo o conteúdo de um arquivo com uma única chamada de **fwrite()** e uma única chamada de **fread()**, respectivamente. Ou seja, nesse caso, não há necessidade de uso de laço de repetição.

2.11.2 Processamento por Acesso Direto

Nos exemplos de processamento de arquivos apresentados até aqui, os arquivos foram acessados sequencialmente. Isto é, todas as partições de um arquivo (i.e., bytes, linhas ou blocos) foram processadas uma após a outra, do primeiro até o último byte. Existem aplicações, entretanto, em que se deseja processar uma partição particular que se encontra numa determinada posição num arquivo. Esse tipo de processamento de arquivo é denominado **processamento com acesso direto**.

Uma operação de processamento de arquivo com acesso direto envolve duas etapas:

1. Mover o indicador de posição do arquivo para o local desejado. Funções designadas para essa tarefa são denominadas **funções de posicionamento**.
2. Executar a operação de leitura ou escrita desejada.

Nem todo arquivo (no sentido genérico) permite acesso direto. Por exemplo, arquivos armazenados em disco permitem acesso direto, mas arquivos associados a um console não o permitem. Além disso, nem toda configuração de arquivo é conveniente para processamento com acesso direto. Um arquivo é adequado para esse tipo de processamento quando faz sentido dividi-lo em partições do mesmo tamanho, de tal modo que essas divisões possam ser indexadas como um array. Essas partições de um arquivo são comumente denominadas **registros**.

Movimentação do Indicador de Posição

Existem três funções no módulo stdio que podem ser utilizadas para posicionamento do indicador de posição de um arquivo. Elas são resumidamente descritas na **Tabela 2–7**.

Função	Descrição sumária
fseek()	**Move o indicador de posição do arquivo para um local especificado por seus parâmetros**

Tabela 2–7: Funções de Posicionamento Utilizadas em Acesso Direto

Função	Descrição sumária
ftell()	Indica onde se encontra correntemente o indicador de posição do arquivo associado ao stream especificado como parâmetro
rewind()	Move o indicador de posição do arquivo para o início do arquivo associado ao stream recebido como único parâmetro

Tabela 2–7: Funções de Posicionamento Utilizadas em Acesso Direto

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

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

Nesse protótipo, os parâmetros têm os seguintes significados:

- `stream` representa um stream associado a um arquivo que suporta acesso direto.
- `distancia` é um deslocamento (positivo ou negativo), medido a partir do terceiro parâmetro, que indica para onde o indicador de posição do arquivo será movido.
- `deOnde` é o local a partir de onde o deslocamento (segundo parâmetro) será determinado.

Em arquivos binários, o valor do segundo parâmetro (`distancia`) é medido em bytes, enquanto, em arquivos de texto, ele deve ser especificado utilizando um valor retornado pela função **ftell()** (v. adiante). O terceiro parâmetro (`deOnde`) pode assumir um dos valores representados pelas constantes simbólicas definidas em `<stdio.h>` e descritas na **Tabela 2–8**.

Constante	Representa...
SEEK_SET	Início do arquivo
SEEK_CUR	Posição corrente do indicador de posição do arquivo
SEEK_END	Final do arquivo

Tabela 2–8: Constantes Simbólicas de Posicionamento em Arquivos

Quando a função **fseek()** consegue deslocar o indicador de posição do arquivo para a posição desejada, ela retorna zero; caso contrário, ela retorna um valor diferente de zero. Considere, por exemplo, a chamada de **fseek()** no fragmento de programa a seguir:

```
int   retorno;
...
FILE *stream = fopen("arquivo.bin", "rb");

if (stream) { /* Abertura de arquivo bem-sucedida */
   retorno = fseek(stream, 10, SEEK_SET);

   if (!retorno) { /* Movimentação do indicador bem-sucedida */
      ... /* Pode-se ler ou escrever na posição desejada */
   } else { /* Não foi possível mover o indicador de posição */
      ... /* Informa o usuário sobre o problema etc. */
   }
} else {/* Arquivo não pode ser aberto */
   ... /* Informa o usuário sobre o problema etc. */
}
```

Se bem-sucedida, a chamada de **fseek()** nesse exemplo moveria o indicador de posição associado ao stream para o byte de índice `10` nesse stream. Como os bytes de um arquivo são indexados a partir de zero, o byte de índice `10` é o 11º byte no stream.

Considerando que o stream do exemplo acima foi aberto no modo de leitura apenas, a chamada:

```
fseek(stream, 1, SEEK_END);
```

retornaria um valor diferente de zero indicando que a solicitação não pode ser atendida, pois, quando um arquivo é aberto apenas para leitura, não se pode mover o indicador de posição além do final do arquivo. Portanto se a constante simbólica **SEEK_END** for utilizada como valor do terceiro parâmetro de **fseek()** e o arquivo tiver sido aberto apenas para leitura, a distância (segundo parâmetro) deve ser negativa. De modo análogo, se a constante simbólica **SEEK_SET** for utilizada, a distância deve ser sempre positiva; nesse 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 indicador de posição de arquivo ultrapassar os limites do arquivo. Para streams de texto, o segundo parâmetro de **fseek()** deve ser `0` ou um valor retornado por **ftell()** (v. adiante), considerando-se o mesmo stream. Mais precisamente, as únicas chamadas portáveis da função **fseek()** para streams de texto são:

- `fseek(stream, 0, SEEK_CUR);`
- `fseek(stream, 0, SEEK_END);`
- `fseek(stream, 0, SEEK_SET);`
- `fseek(stream, ftell(stream), SEEK_SET);`

A função `MoveApontador()`, definida abaixo, inclui tratamento de exceção e pode ser usada em substituição à função **fseek()**.

```
void MoveApontador(FILE *stream, long bytes, int DeOnde)
{
   int deuErrado;

   deuErrado = fseek(stream, (long) bytes, DeOnde);

      /* Verifica se houve erro */
   if (deuErrado) {
      fprintf( stderr, "\n>>> Erro na tentativa de mover apontador de arquivo" );
      exit(1);
   }
}
```

A função **ftell()** recebe apenas um parâmetro, que é um ponteiro de stream e retorna a posição corrente do indicador de posição do arquivo associado ao stream. Quando ocorre erro, essa função retorna `-1L` e seu protótipo é:

```
long ftell(FILE *stream)
```

Essa função é frequentemente usada para guardar o valor corrente do indicador 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 verdadeiro número de bytes contado a partir do início do arquivo. Para streams de texto, o valor retornado por **ftell()** representa um valor que faz sentido apenas quando utilizado como distância (segundo parâmetro) numa chamada subsequente da função **fseek()**.

A função `ObtemApontador()`, apresentada a seguir, é semelhante à função **ftell()**, mas inclui tratamento de exceção.

```
long ObtemApontador(FILE *stream)
{
   long resultado;
```

```
   resultado = ftell(stream);

      /* Verifica se houve erro */
   if (resultado < 0L) {
      fprintf( stderr, "\n>>> Erro na tentativa de obter posicao "
                       "do apontador de arquivo" );
      exit(1);
   }

   return resultado;
}
```

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. O programa a seguir mostra como determinar se um arquivo permite ou não acesso direto[3].

```
#include <stdio.h>

#define NOME_ARQUIVO "Tudor.txt"

int main(void)
{
   FILE  *stream;

      /* Tenta abrir o arquivo em modo texto apenas para leitura */
   stream = fopen(NOME_ARQUIVO, "r");

      /* Checa se arquivo foi aberto */
   if (!stream) {
      printf("\nImpossivel abrir o arquivo %s\n", NOME_ARQUIVO);
      return 1;
   }

      /* Informa se o arquivo recém aberto suporta acesso direto */
   printf( "\n>>> O stream associado a \"%s\" %spermite acesso direto", NOME_ARQUIVO,
           fseek(stream, 0, SEEK_CUR) ? "NAO " : "" );

      /* Informa se o meio de entrada padrão suporta acesso direto */
   printf("\n>>> O stream stdin %spermite acesso direto\n",
          fseek(stdin, 0, SEEK_CUR) ? "NAO " : "");

   return 0;
}
```

Quando executado, esse programa produz o seguinte resultado:

```
>>> O stream associado a "Tudor.txt" permite acesso direto
>>> O stream stdin NAO permite acesso direto
```

As chamadas de **fseek()** no programa acima especificam um deslocamento de zero em relação à posição corrente do indicador de posição de arquivo. Portanto elas servem apenas para testar o valor retornado pela função **fseek()**. Isto é, quando esse valor é igual a zero, o arquivo permite acesso direto; caso contrário, ele não permite acesso direto.

Funções de posicionamento desempenham um importante papel quando um arquivo é aberto num modo de atualização (v. **Seção 2.3**), que permite leitura e escrita (i.e., um modo de abertura que use o sinal +), pois entre uma operação de leitura e uma operação de escrita (ou vice-versa) deve haver uma chamada de função de posicionamento. Para permitir passagem de escrita para leitura, pode-se ainda usar a função **fflush()**. O seguinte quadro resume esse arrazoado:

[3] O arquivo `Tudor.txt` é descrito no **Apêndice A**.

Quando o modo de abertura de um arquivo é "r+", "r+b", "w+", "w+b", "a+" ou "a+b", entre uma operação de leitura e uma operação de escrita no arquivo ou vice-versa, deve haver uma chamada bem-sucedida de fseek() ou rewind(), que recebe como parâmetro o stream associado ao arquivo.

Leitura e Escrita

Em princípio, qualquer função do módulo stdio que é capaz de ler ou escrever num stream pode ser utilizada para realizar a segunda etapa de uma operação de acesso direto descrita no início desta seção, mas, na prática, tipicamente, usam-se **fread()** e **fwrite()** com o arquivo aberto em modo binário.

Uma situação excepcional na qual pode ser conveniente o uso de acesso direto com arquivos de texto ocorre quando as linhas do arquivo são todas do mesmo tamanho. Nesse caso específico, pode-se abrir o arquivo em modo texto e usar as funções **fgets()** e **fputs()** para leitura e escrita, respectivamente.

2.11.3 Inserção, Remoção e Alteração de Registros

Por simplicidades, os arquivos de interesse neste texto são divididos em registros de **tamanho fixo**. Como esses registros têm o mesmo tamanho e são armazenados contiguamente, eles podem ser indexados e, assim, ser processados sequencialmente como se constituíssem um array. Além disso, cada campo que constitui um registro também tem tamanho fixo. Por exemplo, todos os valores de um campo que representa um nome de rua têm o mesmo tamanho, mesmo que uma rua com nome muito pequeno cause desperdício de memória e uma rua com nome muito grande precise ter seu nome truncado.

Essas simplificações não podem ser subestimadas numa aplicação real, mas neste livro elas serão negligenciadas porque, aqui, organização e processamento de arquivos de dados é apenas um tópico secundário. Tais simplificações permitem acesso direto a cada registro para leitura ou alteração por meio de uma simples operação aritmética combinada com uma chamada de função de posicionamento [p. ex., **fseek()**], como será visto na **Seção 2.14**.

Apesar das semelhanças com arrays, as operações de inserção e remoção não são tão facilmente implementadas com arquivos quanto ocorre com arrays. Inserir um registro entre dois outros que já se encontram num arquivo só faz sentido quando existe algum tipo de ordenação entre esses registros e pode ser uma operação excessivamente demorada se ela for efetuada com afastamento de registros (como ocorre com arrays). Neste livro, inserção (ou melhor, acréscimo) de um registro num arquivo de dados é sempre efetuada ao final do arquivo.

Existem diversas abordagens para remoção de registros de arquivo, mas a maioria delas referem-se à remoção lógica; i.e., um registro removido não é considerado integrante de um arquivo mesmo que, fisicamente, ele ainda esteja armazenado no arquivo. A abordagem a ser utilizada nos programas que ilustram aplicações das estruturas de dados discutidas nos capítulos seguintes consiste no seguinte:

1. Quando um registro é removido, sua chave é removida da estrutura de dados usada para acessá-lo. Logo, embora o registro continue fazendo parte do arquivo, ele não poderá mais ser acessado por meio dessa estrutura de dados. Esse é um exemplo de remoção lógica provisória.
2. A chave do registro que foi removida fisicamente da estrutura de dados mencionada é armazenada em outra estrutura de dados, denominada `removidos` nos referidos programas.
3. Ao encerramento do programa, o arquivo de dados é reconstruído, deixando-se fora dele todos os registros cujas chaves se encontrem na estrutura de dados `removidos`. Mais precisamente, todos os registros do arquivo de dados original são copiados para um novo arquivo com exceção daqueles cujas chaves façam parte da estrutura de dados `removidos`. Neste passo, ocorre remoção física de cada registro removido logicamente durante a execução do programa.

Devido às simplificações citadas no início desta seção, atualizar ou alterar o conteúdo de um registro é uma tarefa trivial: basta obter os novos dados do registro e utilizá-los para sobrescrever o antigo conteúdo do registro.

2.12 rewind() ou fseek()?

A função **rewind()** é uma função de posicionamento, mas ela merece destaque especial por ser mais usada em processamento sequencial do que em processamento com acesso direto. Diferentemente da função **fseek()**, essa função move o indicador de posição de arquivo para uma posição específica apenas, a saber, o início do arquivo [mas **fseek()** também faz isso — v. adiante].

Imediatamente após a abertura de um arquivo, o indicador de posição do arquivo aponta para seu início. Portanto se uma função que abre um arquivo deseja processá-lo sequencialmente do início até certo ponto, logo após sua abertura, ela não precisa chamar **rewind()**. Entretanto se uma função recebe como parâmetro um stream já aberto e deseja garantir que o processamento do arquivo inicia-se no primeiro byte do arquivo, ela deve chamar a função **rewind()** ou **fseek()** (v. adiante) antes de iniciar o processamento.

A função **rewind()** é comumente usada em operações de leitura de arquivos e raramente usada em operações de escrita. Contudo, apesar de essa função ser utilizada com muita frequência, ela não é recomendada quando se deseja ter um programa 100% robusto, pois ela não permite testar se foi bem-sucedida. Assim se robustez completa for desejável, deve-se dar preferência ao uso de **fseek()** em substituição a **rewind()**, como mostrado esquematicamente a seguir:

```
if (fseek(stream, 0, SEEK_SET)) {
   /* Indicador de posição não foi    */
   /* movido para o início do arquivo */
   ...
}
```

2.13 Condições de Exceção e a Lei de Murphy

Uma **condição de exceção** é uma situação que impede o funcionamento considerado normal de uma função. Por exemplo, quando se chama a função **fgetc()**, espera-se que, sob condições normais, ela seja capaz de ler um byte num arquivo. Nesse caso, entretanto, existem inúmeros fatores que podem impedir essa função de cumprir sua missão (p. ex., o modo de abertura de arquivo não permite leitura, falha de dispositivo, tentativa de leitura além do final do arquivo etc.). Na maioria das vezes, quando encontra uma condição de exceção, uma função não é capaz de sinalizar exatamente qual foi a causa de seu insucesso, mas, a maior parte delas informa, por meio de um valor de retorno, quando fracassa. O arrazoado apresentado neste parágrafo não se refere exclusivamente a funções de entrada e saída ou à linguagem C. Quer dizer, exceção, condição de exceção e tratamento de exceção são conceitos genéricos em programação.

A **Lei de Murphy** é um adágio popular que afirma que o que pode dar errado, certamente, dará. Aplicada a processamento de arquivos pode-se reformular essa afirmação como:

> ***Se uma operação sobre um arquivo que pode ser malsucedida não for testada, certamente, ela será malsucedida.***

Mas existe o **Corolário 1** que serve de consolo para o programador:

> ***Se uma operação sobre um arquivo for testada, um erro nunca se manifestará nessa operação.***

Ou o **Corolário 2**, que é ainda mais específico:

> ***Se uma função de processamento de arquivos que pode ser malsucedida for testada logo após ser chamada, ela nunca será malsucedida.***

Em processamento de arquivos, quase todas as chamadas de função do módulo stdio podem ser malsucedidas (i.e., resultar em erro). Portanto para evitar que seu programa seja mais uma vítima da Lei de Murphy de processamento de arquivos, siga as recomendações preconizadas na **Tabela 2–9**.

Operação	Função	Como testar se ocorreu erro ou precaver-se
Abertura de arquivo	**fopen**()	Teste se o retorno da função é **NULL**
Fechamento de arquivo	**fclose**()	Use `FechaArquivo()` (v. **Seção 2.3**), em vez de **fopen**()
Leitura	Qualquer função de leitura	Use **ferror**() após cada operação
Escrita	Qualquer função de escrita	Use **ferror**() após cada operação
Posicionamento	**fseek**()	Verifique se o retorno da função é diferente de zero (ou **EOF**)
Posicionamento	**rewind**()	Use **fseek**(), em vez de **rewind**()
Posicionamento	**ftell**()	Teste se o retorno da função é negativo
Descarga de buffer	**fflush**()	Verifique se o retorno da função é diferente de zero (ou **EOF**)

Tabela 2–9: Proteção contra a Lei de Murphy de Processamento de Arquivos

Em qualquer operação enumerada na **Tabela 2–9**, o que pode dar errado é, obviamente, o fato de a respectiva operação não ser bem-sucedida. Mas qualquer que seja a causa do erro (o que nem sempre é óbvio), do ponto de vista pragmático, o importante é verificar se ele ocorreu e, se for o caso, adotar a medida que a situação requer.

2.14 Lidando com Arquivos Grandes em C

Por um longo tempo, muitos sistemas operacionais e suas implementações de sistemas de arquivos usaram valores inteiros de 32 bits para representar tamanhos de arquivos e posições de bytes em arquivos. Consequentemente, nenhum arquivo poderia ser maior do que $2^{31} - 1$ bytes (i.e., aproximadamente *2 GiB*). Assim no contexto de processamento de arquivos, um arquivo grande demais para ser manipulado por um **sistema operacional de 32 bits** passou a ser conhecido como **arquivo grande**. Os modernos sistemas operacionais de 64 bits surgiram com o intuito de suprir essas deficiências.

2.14.1 Portabilidade

Como foi na **Seção 2.11.2**, as funções **fseek**() e **ftell**() da biblioteca padrão de C são essenciais em processamento por acesso direto, que, por sua vez, é fundamental na implementação de estruturas de dados em memória secundária, como será visto no **Capítulo 6**. Ocorre, porém, que essas funções especificam posições (índices) de bytes em arquivos usando inteiros do tipo **long int**, que não é portável. Quer dizer, assim como ocorre com a maioria dos tipos inteiros de C, o tipo **long int** não tem largura especificada por nenhum padrão da linguagem C, de modo que um compilador que segue um padrão de C pode usar qualquer largura desde que ela não seja menor do que a largura do tipo **int** usada pelo mesmo compilador. E, infelizmente, isso de fato ocorre. Por exemplo, na versão do compilador GCC para Windows, a largura do tipo **long int** é 32 bits que não permite que as funções **fseek**() e **ftell**() lidem com arquivos grandes[4]. As funções de posicionamento **fsetpos**() e **fsetpos**(), que fazem parte da biblioteca padrão de C, não resolvem esse problema. Portanto tais compiladores não possuem meios para oferecer uma solução portável para acesso direto para arquivos grandes. Por outro

[4] É possível que quando você estiver lendo este livro, essa deficiência já tenha sido suprida e, então, você poderá ignorar a discussão que segue.

lado, se o programador usar um compilador para o qual a largura do tipo **long int** é 64 bits (p. ex., Clang ou GCC para Linux/Unix), ele não precisa se preocupar.

2.14.2 Índice de Registro e Índice de Byte

Em qualquer arquivo que permite acesso direto, os bytes que o constituem podem ser indexados de zero até o tamanho do arquivo (i.e., seu número de bytes) menos um. Mas quando um arquivo é composto de registros do mesmo tamanho, esses registros também podem ser indexados de zero até o número de registros menos um, como ocorre com arrays armazenados em memória principal. Portanto, nesse caso, um registro pode ser indexado por seu primeiro byte ou por seu próprio índice.

Quando os registros de um arquivo são representados pelas chaves desses registros, como será visto no **Capítulo 6**, a posição de cada registro no arquivo deve ser armazenada juntamente com sua chave. Para arquivos muito grandes, é mais vantajoso armazenar o índice de um registro do que o índice do primeiro byte desse registro. Essa vantagem é decorrente do fato de, normalmente, o número de bytes de um arquivo ser muito maior do que seu número de registros. Logo, para arquivos muito grandes, o índice armazenado junto com a chave do registro poderia requerer o uso de um tipo inteiro com largura de 64 bits (p. ex., o tipo **long long** em C).

Para entender melhor o argumento apresentado no último parágrafo, considere um arquivo bem grande, como aquele que será utilizado para testar árvores multidirecionais no **Capítulo 6**. Esse arquivo possui cerca de 4 GiB e, portanto, seus bytes só podem ser indexados utilizando-se um inteiro com largura de 64 bits. Entretanto esse mesmo arquivo possui *apenas* cerca de 9,5 milhões de registros, que podem ser indexados utilizando-se um tipo inteiro de 32 bits. A única desvantagem que essa indexação de registros apresenta é que ela requer uma operação de multiplicação adicional cada vez que se precisa mover o apontador de posição de um arquivo para o local em que se encontra um determinado registro (com exceção do primeiro registro). Mais precisamente, para mover o apontador de arquivo para um determinado registro por meio da função **fseek()**, é necessário calcular o índice do byte para o qual essa função deve mover o apontador de arquivo. Por exemplo, suponha que o tamanho de um registro é *80* bytes. Então, usando indexação de registros, o registro de número *655* do arquivo poderia ser localizado como:

```
fseek(stream, 655*80, SEEK_SET);
```

A desvantagem mencionada acima é desprezível porque, quando se processam dados armazenados em memória secundária, o principal interesse é minimizar o número de acessos ao meio de armazenamento. No caso de estruturas de dados armazenadas em arquivos, essa minimização de acesso é obtida maximizando-se o tamanho de cada nó da estrutura. Mas quanto menor for o tamanho de cada par chave/índice armazenado nos nós de uma tal estrutura, maior poderá ser o tamanho do nó. Assim está justificada a opção de indexação de registros descrita acima.

2.15 Exemplos de Programação

Nesta seção serão apresentados exemplos que usam o arquivo `Tudor.txt` descrito no **Apêndice A**.

2.15.1 Saltando Linhas de um Arquivo de Texto

Preâmbulo: Linha vazia de um arquivo de texto é aquela contendo apenas quebra de linha (i.e., um caractere `'\n'`).

Problema: Escreva uma função que salta o restante da linha corrente e linhas vazias subsequentes de um arquivo de texto a partir da posição corrente do indicador de posição do arquivo.

Solução: A função `SaltaLinhas()` apresentada abaixo salta o restante da linha corrente e linhas vazias subsequentes de um arquivo de texto. O único parâmetro dessa função é stream, que representa o stream

associado ao arquivo. Essa função retorna o número de linhas saltadas, se não ocorrer erro ou um valor negativo, em caso contrário. Para usar essa função adequadamente, stream que ela recebe como parâmetro deve ter sido aberto em modo texto que permite leitura.

```
int SaltaLinhas(FILE *stream)
{
   int c, /* Armazena um caractere */
       linhasSaltadas = 0; /* Conta o número de linhas saltadas */

      /* O laço encerra quando o final do arquivo for atingido, */
      /* ocorrer erro ou for encontrado um caractere após a     */
      /* linha corrente que não seja quebra de linha            */
   while (1) {
      c = fgetc(stream); /* Lê o próximo caractere */

         /* Se ocorreu erro de leitura, retorna um valor que indica esse fato */
      if (ferror(stream))
         return -1; /* Ocorreu erro */

         /* Se o final do arquivo foi atingido antes que fosse encontrada */
         /* uma quebra de linha,  o número de linhas saltadas é zero      */
      if (feof(stream)) /* O arquivo acabou e não foi encontrado '\n'*/
         return 0;

         /* Se o final da linha corrente foi atingido, */
         /* saltam-se as linhas vazias subsequentes    */
      if(c == '\n') { /* Fim da linha corrente */

            /* O laço encerra quando o final do arquivo for atingido, ocorrer */
            /* erro ou for encontrado um caractere que  não é '\n'            */
         while(c == '\n') { /* Salta linhas */
            linhasSaltadas++; /* Mais uma quebra de linha encontrada */

            c = fgetc(stream); /* Lê o próximo caractere */

               /* Se ocorreu erro retorna um valor que indica esse fato */
            if (ferror(stream))
               return -1;

               /* Se o final do arquivo foi atingido, não */
               /* há mais nada a fazer a não ser retornar */
            if (feof(stream))
               return linhasSaltadas; /* Retorna número de linhas saltadas */
         }

            /* Neste ponto, sabe-se que o último caractere lido é o primeiro */
            /* caractere da primeira  linha que não é vazia. Esse caractere  */
            /* deve ser o primeiro a ser lido na próxima leitura do arquivo. */
            /* Portanto é preciso devolver  esse caractere para o stream. É  */
            /* para isso que serve a função ungetc().                        */

            /* Devolve o último caractere lido para o stream */
         if (ungetc(c, stream) == EOF)
            return -1; /* Não foi possível devolver caractere */

         break; /* Encerra o laço externo */
      }
   }

      /* Retorna o número de caracteres '\n' que foram lidos */
   return linhasSaltadas;
}
```

O uso de **ungetc()** (v. **Seção 2.10**) em `SaltaLinhas()` é apropriado porque essa última função precisa ler e descartar caracteres até encontrar um caractere que não seja quebra de linha. Quando tal caractere é lido, ele precisa ser devolvido ao stream, pois ele não deve ser saltado.

2.15.2 Copiando Arquivos

Problema: (a) Escreva uma função que copia, byte a byte, o conteúdo de um arquivo para outro. (b) Escreva um programa que recebe como argumentos de linha de comando o nome do arquivo que será copiado e o nome do arquivo que receberá a cópia e efetua a devida cópia.

Solução de (a): A função `CopiaArquivo()` faz aquilo que é solicitado e seus parâmetros são:

- `streamEntrada` (entrada) — stream associado ao arquivo que será copiado
- `streamSaida` (entrada) — stream associado ao arquivo que receberá a cópia

A função `CopiaArquivo()` retorna `0`, se não ocorrer erro, ou 1, em caso contrário. Note que o stream que será lido deve estar aberto num modo que permita leitura e o stream que será escrito deve estar aberto num modo que permite escrita.

```
int CopiaArquivo(FILE *streamEntrada, FILE *streamSaida)
{
   int c; /* Armazenará cada byte lido e escrito  */

      /* Garante que a leitura começa no início do arquivo */
   rewind(streamEntrada);

      /* O laço encerra quando houver tentativa de leitura   */
      /* além do final do arquivo de entrada ou ocorrer erro */
      /* de leitura ou escrita em qualquer dos arquivos      */
   while (1) {
         /* Lê um byte no arquivo de entrada */
      c = fgetc(streamEntrada);

         /* Testa se final do arquivo de entrada foi */
         /* atingido ou ocorreu erro de leitura      */
      if (feof(streamEntrada) || ferror(streamEntrada))
         break; /* Processamento encerrado */

         /* Escreve o byte lido no arquivo de saída */
      fputc(c, streamSaida);

         /* Verifica se ocorreu erro de escrita */
      if (ferror(streamSaida))
         break;
   }

   /* O processamento está terminado, mas a função não deve fechar os */
   /* arquivos, já que ela não foi responsável por suas aberturas     */

      /* Se ocorreu erro de escrita ou leitura, o   */
      /* retorno será 1. Caso contrário, será zero. */
   return ferror(streamEntrada) || ferror(streamSaida);
}
```

A função `CopiaArquivo()` funciona quando ambos os streams são abertos em modo binário ou ambos são abertos em modo de texto, uma vez que, dessa maneira, o que é lido no arquivo de entrada corresponde exatamente àquilo que é escrito no arquivo de saída. Entretanto se os arquivos forem abertos em modo texto, a operação tende a ser menos eficiente em consequência da necessária interpretação de quebra de linha tanto na leitura quanto na escrita. É importante notar que essa função não fecha os arquivos ao encerrar sua tarefa, seguindo a norma preconizada na **Seção 2.3**.

Solução de (b): A função **main()** apresentada abaixo realiza o que foi solicitado.

```
int main(int argc, char *argv[])
{
   FILE *streamEntrada, /* Stream de entrada */
        *streamSaida;   /* Stream de saída  */
   int   resultado;

      /* Verifica se o usuário informou quais serão os arquivos envolvidos na cópia */
   if (argc != 3) {
      printf("\n\t>>> Este programa deve ser usado assim: %s arquivo-a-ser-copiado "
             "\n\t> arquivo-que-recebe-a-copia\n", argv[0]);
      return 1;
   }

      /* Verifica se os arquivos de entrada e de saída são os mesmos */
   if (!strcmp(argv[1], argv[2])) {
      printf("\nOs nomes dos arquivos nao podem ser iguais\n");
      return 1;
   }

   streamEntrada = fopen(argv[1], "rb"); /* Tenta abrir o arquivo de entrada */

      /* Verifica se o arquivo de entrada foi aberto */
   if (!streamEntrada) {
      printf("\nO arquivo \"%s\" nao pode ser aberto\n", argv[1]);
      return 1;
   }

      /* Aqui, o arquivo de entrada foi aberto com sucesso. Se o arquivo de saída */
      /* não for aberto, deve-se fechar o arquivo de entrada antes de retornar.   */
   streamSaida = fopen(argv[2], "wb"); /* Tenta abrir arquivo de saída */

      /* Verifica se o arquivo de saída foi aberto */
   if (!streamSaida) {
      printf("\nO arquivo \"%s\" nao pode ser aberto\n", argv[2]);
      fclose(streamEntrada);
      return 1;
   }

   resultado = CopiaArquivo(streamEntrada, streamSaida); /* Efetua a cópia */

   fclose(streamEntrada); /* Os arquivos não precisam mais estar abertos */
   fclose(streamSaida);

      /* Comunica ao usuário o resultado da operação */
   if (!resultado) {
      printf( "\n\t>>> O arquivo %s foi copiado em %s\n", argv[1], argv[2] );
   } else {
      printf("\n\t>>> Impossivel copiar o arquivo %s\n", argv[1]);
      return 1; /* Operação falhou */
   }

   return 0;
}
```

2.15.3 Atualizando Registros de um Arquivo de Texto

Problema: Considerando o arquivo de texto `Tudor.txt` descrito no **Apêndice A**, escreva um programa que lê esse arquivo, acrescenta um ponto à segunda nota de cada aluno e, finalmente, cria um arquivo de texto contendo os dados atualizados.

Esboço de solução:

Para resolver o problema proposto é preciso seguir a seguinte sequência de passos:

1. Ler cada linha do arquivo.
2. Converter cada linha lida numa estrutura do seguinte tipo:

```
typedef  struct {
            char    nome[MAX_NOME + 1];
            char    matr[TAM_MATR + 1];
            double  n1, n2;
         } tAluno;
```

   Nessa definição de tipo, `MAX_NOME` e `TAM_MATR` são constantes simbólicas previamente definidas.
3. Acrescentar um ponto à segunda nota (campo `n2`) de cada estrutura obtida no **Passo 2**.
4. Escrever a estrutura modificada num arquivo de texto especificado usando o mesmo formato de linha do arquivo original.

Solução: A função **main()** apresentada a seguir implementa o esboço de solução exposto acima.

```
int main(void)
{
   FILE *streamE, /* Associado ao arquivo de entrada */
        *streamS; /* Associado ao arquivo de saída   */

      /* Tenta abrir arquivo de entrada para leitura em modo texto */
   streamE = fopen(NOME_ARQUIVO, "r");
      /* Se o arquivo de entrada não pode     */
      /* ser aberto, nada mais pode ser feito */
   if (!streamE) {
      printf( "\nArquivo \"%s\" nao pode ser aberto\n", NOME_ARQUIVO );
      return 1; /* Arquivo de entrada não foi aberto */
   }

      /* Tenta abrir arquivo de saída para escrita em modo texto */
   streamS = fopen(NOME_ARQUIVO_ATUAL, "w");

      /* Se o arquivo de saída não pode ser aberto, nada mais pode ser feito */
   if (!streamS) {
      fclose(streamE); /* Fecha arquivo de entrada antes de partir */

      printf( "\nArquivo \"%s\" nao pode ser aberto\n", NOME_ARQUIVO_ATUAL );

      return 1; /* Arquivo de saída não foi aberto */
   }

      /* Cria o arquivo atualizado */
   if (AtualizaArquivo(streamS, streamE)) {
      printf( "\n\t>>> Ocorreu um erro na atualizacao do"
              "\n\t>>> arquivo \"%s\"\n", NOME_ARQUIVO_ATUAL );
      return 1; /* Ocorreu algum erro durante atualização */
   }
   printf( "\n\t>>> Atualizacao do arquivo \"%s\" foi \n\t>>> escrita no "
           "arquivo \"%s\"\n", NOME_ARQUIVO, NOME_ARQUIVO_ATUAL );

   return 0; /* Tudo ocorreu bem */
}
```

Essa função **main()** abre os arquivos de entrada e saída em questão e, em seguida, chama a função `AtualizaArquivo()` para completar a tarefa de criação do arquivo que conterá a atualização. Essa última função lê cada linha do arquivo de entrada, converte-a numa estrutura do tipo `tAluno`, atualiza a estrutura e

escreve-a modificada no arquivo de saída por meio de **fprintf()** usando o mesmo formato do arquivo de entrada. A implementação da função `AtualizaArquivo()`, que será apresentada a seguir, tem como parâmetros:

- `streamSaida` (entrada) — stream associado ao arquivo que conterá a atualização. Esse stream deve estar aberto em modo de texto que permite leitura.
- `streamEntrada` (entrada) — stream associado ao arquivo que será lido. Esse stream deve estar aberto em modo `"w"`.

A função `AtualizaArquivo()` retorna zero, se não ocorrer nenhum erro, ou um valor diferente de zero em caso contrário.

```
int AtualizaArquivo(FILE *streamSaida, FILE *streamEntrada)
{
   tAluno umAluno; /* Armazenará dados de uma estrutura */
   char   linha[MAX_LINHA + 1]; /* Armazena uma linha do arquivo de entrada */

      /* Garante que a leitura começa no primeiro byte */
   rewind(streamEntrada);

      /* Lê cada linha do arquivo de entrada, armazena os dados numa estrutura,   */
      /* atualiza a estrutura e escreve-a no arquivo de saída no formato de linha */
      /* do arquivo original. O laço encerra quando houver tentativa de leitura   */
      /* além do final do arquivo de entrada, ou ocorrer erro de leitura/escrita. */
   while (1) {
         /* Lê uma linha no arquivo de entrada */
      fgets(linha, MAX_LINHA + 1, streamEntrada);

         /* Verifica se ocorreu erro ou tentativa */
         /* de leitura além do final do arquivo   */
      if ( feof(streamEntrada) || ferror(streamEntrada) )
         break;

         /* Converte a linha lida numa estrutura do tipo tAluno */
      LinhaEmRegistro(&umAluno, linha);

         /* Atualiza o campo n2 da estrutura */
      umAluno.n2 = umAluno.n2 + 1;

         /* Escreve a estrutura no arquivo de saída usando fprintf() */
      fprintf(streamSaida, "%s\t%s\t%3.1f\t%3.1f\n", umAluno.nome,
               umAluno.matr, umAluno.n1, umAluno.n2);

         /* Encerra o laço se ocorreu algum erro de escrita */
      if (ferror(streamSaida))
         break;
   }

      /* Se ocorreu algum erro de processamento, o valor retornado será 1 */
   return ferror(streamEntrada) || ferror(streamSaida);
}
```

A função `AtualizaArquivo()` usa um laço de repetição para ler cada linha do arquivo de entrada, convertê-la numa estrutura, atualizar o campo `n2` da estrutura e, finalmente, escrever a estrutura atualizada no arquivo de saída no formato de linha do arquivo original.

A função `LinhaEmRegistro()` que converte o conteúdo de uma linha do arquivo numa estrutura do tipo `tAluno` é definida como se mostra a seguir. Essa função usa os seguintes parâmetros:

- `pAluno` (saída) — ponteiro para a estrutura resultante da conversão

- `linha` (entrada/saída) — linha que será convertida. Para simplificar, a função assume que esse parâmetro realmente é um string no formato das linhas do arquivo. Portanto os valores retornados por **strtok()** não são testados como deveriam. Esse parâmetro é alterado por **strtok()**.

A função `LinhaEmRegistro()` retorna o endereço da estrutura que armazena o resultado.

```
tAluno *LinhaEmRegistro(tAluno *pAluno, char *linha)
{
   char *str; /* Apontará para cada token da linha */

      /* Obtém o nome e acrescenta-o ao respectivo campo da estrutura */
   str = strtok(linha, "\t\n");
   strcpy(pAluno->nome, str);

      /* Obtém a matrícula e acrescenta-a ao respectivo campo da estrutura */
   str = strtok(NULL, "\t\n");
   strcpy(pAluno->matr, str);

      /* Obtém a 1a. nota, converte-a em double e acrescenta o */
      /* valor convertido ao campo respectivo da estrutura     */
   str = strtok(NULL, "\t\n"); /* Obtém a 1a. nota e... */
   pAluno->n1 = strtod(str, NULL); /* converte em double */

      /* Idem para a 2a. nota */
   str = strtok(NULL, "\t\n"); /* Obtém a 2a. nota e... */
   pAluno->n2 = strtod(str, NULL); /* converte em double */

   return pAluno;
}
```

A função `LinhaEmRegistro()` extrai cada token da linha recebida como parâmetro usando a função **strtok()** (v. **Seção 9.10.1**) e usa-os para preencher os campos da estrutura cujo endereço é recebido como parâmetro. Os campos `nome` e `matr` da referida estrutura são preenchidos copiando-se os respectivos tokens com **strcpy()**. Os preenchimentos dos campos `n1` e `n2` usam a função **strtod()**, que converte um string num valor do tipo **double**. As funções **strtok()**, **strcpy()**, e **strtod()** fazem parte da biblioteca padrão de C.

O complemento do programa poderá ser encontrado no site dedicado a este livro na internet.

2.15.4 Convertendo um Arquivo de Registros de Texto para Binário

Problema: Escreva uma função que lê cada linha de um arquivo com o mesmo formato do arquivo `Tudor.txt`, descrito no **Apêndice A** e converte-a numa estrutura do tipo `tAluno` definido na **Seção 2.15.3**.

A referida função deve armazenar cada estrutura num arquivo binário especificado como parâmetro. Se não ocorrer nenhum erro, essa função deve retornar o número de registros escritos no arquivo binário; caso contrário, a função deve retornar um valor negativo.

Solução: A função `CriaArquivoBin()` apresentada abaixo lê cada linha de um arquivo no formato especificado, converte-a numa estrutura do tipo `tAluno` e armazena a estrutura num arquivo binário. Os parâmetros dessa função são:

- `streamTexto` (entrada) — stream associado ao arquivo que será lido. Esse stream deve estar aberto em modo de texto que permite leitura
- `streamBin` (entrada) — stream associado ao arquivo que será escrito. Esse stream deve estar aberto em modo `"wb"`.

A função `CriaArquivoBin()` retorna o número de estruturas escritas no arquivo binário ou um valor negativo, se ocorrer erro.

```
int CriaArquivoBin(FILE *streamTexto, FILE  *streamBin)
{
   char    linha[MAX_LINHA + 1]; /* Armazenará cada linha lida */
   tAluno  umAluno; /* Dados de uma linha convertida em estrutura */
   int     nRegistros = 0; /* Número de estruturas escritas no arquivo binário */

      /* Garante leitura a partir do início do arquivo */
   rewind(streamTexto);

      /* Lê cada linha do arquivo, cria um registro do tipo     */
      /* tAluno e armazena-o no arquivo binário. O laço encerra */
      /* quando há tentativa de leitura além do final do        */
      /* arquivo de entrada, ou ocorre erro de leitura/escrita  */
   while (1) {
         /* Lê uma linha no arquivo de entrada */
      fgets(linha, MAX_LINHA + 1, streamTexto);

         /* Verifica se o final do arquivo foi atingido */
      if (feof(streamTexto))
         break;

         /* Verifica se ocorreu erro de leitura */
      if (ferror(streamTexto))
         return -1; /* Operação mal sucedida */

         /* Converte a linha lida em estrutura */
      LinhaEmRegistro(&umAluno, linha);

         /* Escreve a estrutura resultante da conversão no arquivo binário */
      fwrite(&umAluno, sizeof(umAluno), 1, streamBin);

         /* Verifica se ocorreu erro de escrita */
      if (ferror(streamBin))  /* Ocorreu */
         return -1; /* Operação mal sucedida */

      ++nRegistros; /* Mais um registro lido */
   }

   return nRegistros; /* Não ocorreu nenhum erro */
}
```

A função `CriaArquivoBin()` usa o array `linha[]` definido como:

```
char linha[MAX_LINHA + 1];
```

para armazenar cada linha lida no arquivo de texto. Nessa definição, `MAX_LINHA` é uma constante simbólica, previamente definida, que representa o tamanho máximo estimado de uma linha do arquivo de texto. Isso constitui uma limitação dessa função, pois nem sempre é possível, do ponto de vista prático, estimar seguramente qual é o tamanho da maior linha de um arquivo. Por exemplo, como o programador estimará o tamanho da maior linha num arquivo com milhões de linhas? Esse problema é facilmente resolvido com alocação dinâmica de memória, como foi visto no **Volume 1**.

O funcionamento normal (i.e., sem ocorrência de erros de processamento de arquivos) da função `CriaArquivoBin()` é simples e resume-se ao seu laço while: uma linha é lida pela função **fgets()**, essa linha é convertida numa estrutura do tipo `tAluno` e essa estrutura é escrita no arquivo binário.

A função `LinhaEmRegistro()` chamada por `CriaArquivoBin()` é aquela definida na **Seção 2.15.3**.

2.16 Exercícios de Revisão

Arquivos de Texto e Binários (Seção 2.1)

1. O que é um arquivo armazenado?
2. Qual é o significado de arquivo em C (no sentido genérico)?
3. Os dispositivos teclado e monitor de vídeo são considerados arquivos? Explique.
4. Em linhas gerais, qual é o conteúdo do arquivo de cabeçalho `<stdio.h>`?
5. O que é formato de arquivo?
6. (a) O que é um arquivo de texto? (b) O que é um arquivo binário?
7. Que restrições um sistema operacional pode impor a arquivos de texto?

Streams (Seção 2.2)

8. (a) O que é um stream? (b) Que vantagem esse conceito oferece ao processamento de arquivos?
9. Qual é a diferença entre stream e arquivo?
10. Como o conceito de stream é implementado em C?
11. Do ponto de vista prático, faz diferença confundir stream com arquivo?
12. (a) Qual é o significado do identificador **FILE**? (b) Onde o identificador **FILE** é definido?
13. Cite algumas informações que devem estar armazenadas numa estrutura do tipo **FILE**.

Abrindo e Fechando um Arquivo (Seção 2.3)

14. (a) O que significa abrir um arquivo? (b) Como isso é realizado em C?
15. (a) Quais são os parâmetros da função **fopen()**? (a) O que essa função retorna?
16. Para que serve a constante simbólica **FILENAME_MAX**?
17. Cite três razões pelas quais um arquivo não pode ser aberto.
18. (a) Como se testa se um arquivo foi realmente aberto? (b) O que pode ocorrer se esse teste não for efetuado?
19. (a) O que é um stream de texto e (b) O que é um stream binário?
20. (a) O que ocorre quando um arquivo de texto é associado a um stream binário? (b) O que ocorre quando um arquivo de texto é associado a um stream de texto?
21. Um arquivo binário pode estar associado a um stream de texto?
22. O que é modo de abertura de arquivo?
23. O que é um modo de atualização?
24. Descreva os seguintes modos de abertura de arquivo:
    (a) `"r"`
    (b) `"w"`
    (c) `"a"`
    (d) `"r+"`
    (e) `"w+"`
    (f) `"a+"`
25. Qual formato de arquivo é recomendado para cada modo de abertura do exercício anterior?
26. Qual é a diferença entre modos de abertura para streams de texto e modos de abertura correspondentes para streams binários em termos de sintaxe?
27. Quais são as diferenças entre os seguintes modos de abertura de arquivo:
    (a) `"r"` e `"rb"`

(b) `"r"` e `"rt"`

(c) `"rt"` e `"rb"`

28. Quais são as diferenças entre os modos de abertura `"r+"`, `"w+"` e `"a+"`?

29. Que precaução deve ser tomada quando se usam os modos de abertura `"w"`, `"w+"`, `"wb"` e `"w+b"`?

30. (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 mesmo objetivo?

31. O que deve ser imediatamente feito após chamar **fopen()** para abrir um arquivo e antes de processá-lo?

32. Como se pode determinar o número máximo de arquivos que podem ser abertos simultaneamente?

33. Como se pode determinar o tamanho máximo de um nome de arquivo no sistema operacional utilizado?

34. Como a constante simbólica **FILENAME_MAX** deve ser usada na prática?

35. O que representa a constante simbólica **FOPEN_MAX**?

36. O programa a seguir não consegue abrir o arquivo introduzido no meio de entrada padrão pelo usuário, mesmo que o arquivo exista no diretório (pasta) corrente. Explique por que isso ocorre e encontre uma maneira de corrigir o problema.

```
#include <stdio.h>

int main(void)
{
   char   nome[FILENAME_MAX]; /* 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;
}
```

37. (a) O que ocorre quando um arquivo é fechado? (b) Que função é utilizada com esse propósito? (c) Qual é a importância de fechar um arquivo após seu processamento?

38. O que a função **fclose()** retorna?

39. Qual é a facilidade que a função `FechaArquivo()` apresentada na **Seção 2.3** oferece?

40. É importante fechar um arquivo mesmo quando ele é aberto apenas para leitura?

41. O que está errado no seguinte fragmento de programa?

```
FILE *p = fopen("teste.bin", "r+b");

... /* Processamento do arquivo */

fclose("teste.bin")
```

42. (a) Descreva a função `AbreArquivo()`. (a) Qual é a função da biblioteca padrão de C que corresponde à função `AbreArquivo()`? (b) Qual vantagem essa função oferece com relação à função correspondente da biblioteca padrão de C?

Ocorrências de Erros (Seção 2.4)

43. O que representa a constante simbólica **EOF**?

44. É verdade que **EOF** representa um caractere encontrado num arquivo? Explique.

45. Como a constante **EOF** deve ser usada?

46. Por que, muitas vezes, quando uma função do módulo stdio retorna **EOF**, esse valor é ambíguo?

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

48. Por que o uso de **feof()** é mais recomendável do que o uso de **EOF**?

49. Qual é a importância da função **ferror()**?

50. Que vantagem a função **ferror()** oferece em comparação a **EOF** para testar ocorrência de erro?

51. O que pode acontecer quando um indicativo de erro ou de final de arquivo não é removido?

52. Como um indicativo de erro de processamento de arquivo pode ser removido (a) implicitamente e (b) explicitamente?

53. Como um indicativo de final de arquivo pode ser removido (a) implicitamente e (b) explicitamente?

54. (a) Para que serve a função **clearerr()**? (b) Por que raramente essa função se faz necessária?

Buffering e a Função fflush() (Seção 2.5)

55. (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 ou saída?

56. (a) O que é buffering de linha? (b) O que é buffering de bloco?

57. O que significa descarregar um buffer?

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

59. Para que serve a função **fflush()**?

60. O que há de errado com a seguinte chamada da função **fflush()**?

```
fflush(stdin);
```

61. Qual é o efeito da seguinte chamada da função **fflush()**?

```
fflush(NULL);
```

Streams Padrão (Seção 2.6)

62. Descreva os streams padrão de C.

63. Por que os streams padrão são qualificados como padrão?

Leitura e Escrita Formatadas (Seção 2.7)

64. (a) O que é leitura formatada? (b) O que é escrita formatada?

65. Qual é a principal diferença entre as funções **fscanf()** e **scanf()**?

66. (a) Uma chamada da função **fscanf()** pode ser usada em substituição a uma chamada de **scanf()**? (b) Se a resposta for afirmativa, como deve ser efetuada a chamada de **fscanf()**? (c) Uma chamada da função **fscanf()** pode ser substituída por uma chamada de **scanf()**?

67. Qual é a principal diferença entre as funções **printf()** e **fprintf()**?

68. (a) Uma chamada da função **fprintf()** pode ser usada em substituição a uma chamada de **printf()**? (b) Se a resposta for afirmativa, como deve ser efetuada a chamada de **fprintf()**? (c) Uma chamada da função **fprintf()** pode ser substituída por uma chamada de **printf()**?

69. (a) Descreva a função **sprintf()**. (b) Em que situações essa função é útil?

70. Que precaução deve ser tomada quando se usa **sprintf()**?

Trabalhando com Arquivos Temporários (Seção 2.8)

71. Em que situações tipicamente se faz necessário o uso de arquivos temporários?

72. Como funciona a função **tmpfile**()?

73. Arquivos temporários criados pela função **tmpfile**() precisam ser fechados explicitamente pelo programador?

74. É possível fechar um arquivo temporário criado por meio de **tmpfile**() usando a função `FechaArquivo()` definida na **Seção 2.3**?

75. Qual é a finalidade da função **tmpnam**()?

76. Quais são os significados das constantes simbólicas: (a) **L_tmpnam** e (b) **TMP_MAX**?

77. O que ocorre quando a função **tmpnam**() é chamada recebendo **NULL** como parâmetro?

78. É possível fechar um arquivo temporário criado por meio de **tmpnam**() usando a função `FechaArquivo()` definida na **Seção 2.3**?

79. (a) Arquivos temporários criados por **tmpfile**() são removidos automaticamente? (b) Arquivos temporários criados com auxílio de **tmpnam**() são excluídos automaticamente?

80. Que vantagens a função **tmpfile**() oferece em relação à função **tmpnam**()?

81. Suponha que um arquivo temporário precisa ser criado para leitura e escrita em modo de texto. Que função você usaria para tal finalidade: **tmpfile**() ou **tmpnam**()?

Removendo e Rebatizando Arquivos (Seção 2.9)

82. O que faz a função **remove**()?

83. O que a função **remove**() retorna?

84. O que ocorre quando o arquivo cujo nome é passado como parâmetro para a função **remove**() está aberto?

85. (a) Qual é a finalidade da função **rename**()? (b) Quais são os parâmetros dessa função? (c) O que essa função retorna?

A Função ungetc() (Seção 2.10)

86. Descreva o funcionamento da função **ungetc**().

87. O que há de esquisito com a função **ungetc**()?

88. Em que situações a função **ungetc**() é normalmente usada?

89. Quando não há garantia de inserção de um caractere num stream por meio de uma chamada de **ungetc**()?

90. Por que não se deve chamar **ungetc**() quando o indicador de posição do arquivo associado ao stream usado na chamada aponta para o início do arquivo?

Tipos de Processamento (Seção 2.11)

91. Descreva as categorias de processamento sequencial de arquivos.

92. Defina os seguintes conceitos:

(a) Acesso sequencial a arquivo
(b) Acesso direto a arquivo
(c) Entrada formatada
(d) Saída formatada

93. Para que tipo de stream cada uma das seguintes categorias de processamento é conveniente?

(a) Por byte
(b) Por linha
(c) Por bloco

(d) Formatado

94. Que funções são tipicamente usadas para:

(a) Processamento por byte

(b) Processamento por linha

(c) Processamento por bloco

(d) Processamento formatado

95. (a) No contexto de processamento de arquivos, o que são registros? (b) O que é um campo de registro?

96. (a) Descreva as funções **fgetc()** e **fputc()**. (b) Por que essas funções podem funcionar de modo diferente dependendo do modo de abertura do arquivo (i.e., se o arquivo é aberto em modo de texto ou binário)?

97. O funcionamento das funções **fgetc()** e **fputc()** depende do formato do arquivo representado no modo de abertura do arquivo se o sistema utilizado for da família Unix?

98. Qual é o problema como o seguinte laço **while**?

```
while (c = fgetc(stream) != EOF) {
   ...
}
```

99. Suponha que `streamEntrada` e `streamSaida` sejam streams abertos respectivamente nos modos `"rb"` e `"wb"`. O que há de errado com o seguinte laço **while**?

```
while ( !feof(streamEntrada) )
    fputc(fgetc(streamEntrada), streamSaida);
```

100. Suponha que `streamA` e `streamB` são dois streams binários, sendo que o primeiro stream é aberto para leitura e o segundo é aberto para escrita. Escreva um trecho de programa que demonstre como copiar o conteúdo do primeiro stream para o segundo.

101. Descreva o funcionamento das funções: (a) **fgets()** e (b) **fputs()**.

102. (a) Descreva o funcionamento de cada uma das seguintes funções. (b) Em que tipo de processamento cada uma delas é mais adequada?

(a) **fread()**

(b) **fwrite()**

103. O que é um bloco de memória em processamento de arquivos?

104. Qual é a diferença entre as funções **fgetc()** e **getchar()**?

105. Em que situações o uso das funções **fscanf()** e **fread()** é mais conveniente?

106. A função `TamanhoDaLinha()` apresentada adiante se propõe a calcular o tamanho da linha corrente do stream de texto recebido como parâmetro. Descubra o que há de errado com essa função e encontre uma maneira de corrigi-la.

```
int TamanhoDaLinha(FILE *stream)
{
   char c;
   int  contador = 0;

   while((c = fgetc(stream)) != EOF && c != '\n')
      contador++;

   return contador;
}
```

107. Suponha que **teste** seja o nome de um arquivo de texto residente no mesmo diretório em que se encontra o programa executável correspondente ao programa-fonte a seguir. Descubra o que há de errado com esse programa e encontre uma maneira de corrigi-lo.

```
#include <stdio.h>

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

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

   fclose(stream);

   return 0;
}
```

108. O programa apresentado a seguir é semelhante àquele do exercício anterior. No entanto, quando executado, esse programa é abortado, enquanto aquele, apesar de não estar correto, não é abortado. Assumindo as mesmas suposições referentes ao arquivo **teste** do exercício anterior, explique por que o programa a seguir é abortado.

```
#include <stdio.h>

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

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

   fclose(stream);

   return 0;
}
```

109. Um famoso livro de programação ilustra por meio do seguinte fragmento de programa como arquivos devem ser lidos em C:

```
do {
   ch = fgetc(fp);
   /* ... */
} while (!feof(fp));
```

Nesse fragmento de programa, **ch** é uma variável do tipo **char**, **fp** é um stream associado a um arquivo aberto para leitura e o comentário representa a operação a ser efetuada com cada caractere lido no stream. O que há de errado com essa recomendação para leitura de arquivo?

110. Explique a diferença entre acesso sequencial e acesso direto a arquivos.

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

112. Apresente um exemplo de stream que não permite acesso direto.

113. O seguinte programa foi escrito com o objetivo de copiar o conteúdo de um arquivo para outro. O que há de errado com esse programa?

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   FILE *entrada, *saida;
   int   c;

   if (argc != 3) {
      printf( "Uso do programa: %s arquivo-entrada "
              "arquivo-saida\n", argv[0] );
      return 1;
   }

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

   if (!entrada) {
      printf("Impossivel abrir %s\n", argv[1]);
      return 1;
   }

   saida = fopen(argv[2], "wb");

   if (!saida) {
      printf("Impossivel abrir %s\n", argv[2]);
      fclose(entrada);
      return 1;
   }

   while ((c = getc(entrada)) != EOF) {
      putc(c, saida);
   }

   fclose(entrada);
   fclose(saida);

   return 0;
}
```

114. O que é uma função de posicionamento?

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

116. Como é interpretado o valor retornado pela função **fseek()**?

117. Por que o valor retornado por **ftell()** depende do modo de abertura do arquivo associado ao stream que essa função recebe como parâmetro?

118. Apresente o significado de cada constante simbólica a seguir:
(a) **SEEK_SET**
(b) **SEEK_CUR**
(c) **SEEK_END**

119. (a) O que se deve fazer entre uma operação de leitura e uma operação de escrita subsequente num arquivo aberto para atualização? (b) O que se deve fazer entre uma operação de escrita e uma operação de leitura subsequente num arquivo aberto para atualização?

120. (a) O que é um registro de tamanho fixo? (b) De que maneira um arquivo que contém apenas registros de tamanho fixo facilita seu processamento?

121. Uma abordagem para inserção de registros num arquivo de dados consiste em inseri-los em substituição a registros que foram logicamente removidos. (a) Explique como essa abordagem pode ser implementada à luz do que foi exposto na **Seção 2.11.3**. (b) Qual é a vantagem dessa abordagem?

122. (a) Descreva a função `MoveApontador()`. (a) Qual é a função da biblioteca padrão de C que corresponde à função `MoveApontador()`? (b) Qual vantagem essa função oferece com relação à função correspondente da biblioteca padrão de C?

123. (a) Descreva a função `ObtemApontador()`. (a) Qual é a função da biblioteca padrão de C que corresponde à função `ObtemApontador()`? (b) Qual vantagem essa função oferece com relação à função correspondente da biblioteca padrão de C?

rewind() ou fseek()? (Seção 2.12)

124. Qual é a finalidade da função **rewind()**?

125. Por que, apesar de ser uma função de posicionamento, a função **rewind()** é mais usada em processamento sequencial?

126. Em que situação o uso de **rewind()** é recomendado?

127. (a) Por que se recomenda usar **rewind()** [ou **fseek()**] quando uma função que efetua leitura sequencial recebe como parâmetro um stream? (b) Por que uma função que abre um arquivo com o mesmo propósito não precisa usar **rewind()**?

128. Como **fseek()** pode ser usada em substituição a **rewind()**?

129. Por que o uso de **fseek()** é mais recomendado do que o uso de **rewind()**?

Condições de Exceção e a Lei de Murphy (Seção 2.13)

130. O que é uma condição de exceção?

131. O que afirma a Lei de Murphy específica para processamento de arquivos?

132. Enuncie os corolários 1 e 2 da Lei de Murphy para processamento de arquivos.

133. Como precaver-se ou testar a ocorrência de erro em chamadas das seguintes funções:

(a) **fopen()**
(b) **fclose()**
(c) Qualquer função de leitura
(d) Qualquer função de escrita
(e) **fseek()**
(f) **rewind()**
(g) **ftell()**
(h) **fflush()**

Lidando com Arquivos Grandes em C (Seção 2.14)

134. O que é um arquivo grande no contexto de processamento de arquivos?

135. Qual é o problema com o qual um programador pode se deparar quando lida com arquivos grandes em C?

136. Por que o processamento de arquivos grandes em C não é portável?

137. (a) Qual é a diferença entre índice de registro e índice de byte num arquivo binário contendo registros de mesmo tamanho? (b) Quais são as vantagens e desvantagens do uso de índice de registro em vez de índice de byte no processamento de arquivos?

138. Qual é o problema com o qual o programador pode se deparar quando usa **fseek()** ou **ftell()** para lidar com arquivos grandes?

139. (a) O que é índice de registro? (b) O que é índice de byte. (c) Quando é mais vantajoso usar índices de registro em contraste com índices de bytes?

140. Qual é o pressuposto necessário para que registros possam ser indexados?

Exemplos de Programação (Seção 2.15)

141. (a) Descreva o funcionamento da função `SaltaLinhas()` apresentada na **Seção 2.15.1**. (b) Qual é a utilidade prática dessa função? (c) Descreva o uso de **ungetc()** na função `SaltaLinhas()`.

142. A função `CopiaArquivo()`, apresentada na **Seção 2.15.2**, funciona quando ambos os streams são abertos em modo de texto? Explique.

143. Por que a função `CopiaArquivo()` não fecha arquivos?

144. (a) Descreva a função `LinhaEmRegistro()` apresentada na **Seção 2.15.3**. (b) Qual é o papel desempenhado por essa função na atualização do arquivo de texto dessa seção?

2.17 Exercícios de Programação

EP2.1 Sabendo que a maior linha de um arquivo, denominado `Tudor.txt` (v. **Apêndice A**), contém `29` caracteres, incluindo o caractere de quebra de linha, escreva um programa que cria uma cópia desse arquivo num arquivo denominado `BK.txt` usando as funções **fgets()** e **fprintf()**.

EP2.2 Reescreva a função `CopiaArquivo()`, apresentada na **Seção 2.15.2**, de tal modo que ela conte o número de bytes copiados. A nova versão dessa função deve retornar o número de caracteres copiados, se ela for bem-sucedida ou um valor negativo, em caso contrário.

EP2.3 Reescreva o programa apresentado na **Seção 2.15.3**, de maneira que o resultado da atualização seja escrito no próprio arquivo de entrada (que, então, passará a ser de entrada e saída).

EP2.4 Escreva um programa em C que lê e apresenta na tela o conteúdo de um arquivo de texto cujo nome é especificado em linha de comando.

EP2.5 Implemente um programa que escreve na tela os conteúdos de todos os arquivos especificados em linha de comando.

EP2.6 Escreva um programa em C que lê e exibe na tela, de dez em dez linhas, o conteúdo de um arquivo de texto cujo nome é especificado na linha de comando. Isto é, o programa apresenta na tela as 10 primeiras linhas e solicita que o usuário digite uma tecla qualquer para que as próximas 10 linhas sejam escritas e assim por diante até que todo arquivo tenha sido exibido.

EP2.7 Escreva um programa em C que copia, no máximo, as `n` primeiras linhas de um arquivo de texto para um novo arquivo. Os nomes dos arquivos e o número de linhas que serão copiadas devem ser solicitados pelo programa ao usuário.

EP2.8 Escreva um programa que converte todas as letras de um arquivo em maiúsculas. O nome do arquivo deve ser um argumento de linha de comando.

EP2.9 Escreva um programa que converte o arquivo `Tudor.bin`, criado na **Seção 2.15.4**, num arquivo de texto no formato descrito no início da **Seção 2.15**. O resultado deve ser armazenado num arquivo denominado `Tudor2.txt` e seu conteúdo deverá ser exatamente igual ao arquivo `Tudor.txt` usado nos exemplos apresentados na **Seção 2.15**.

EP2.10 Escreva um programa em C que escreve na tela as 10 últimas linhas de um arquivo de texto cujo nome é especificado na linha de comando.

EP2.11 Escreva um programa que lê o arquivo `Tudor.txt` (v. **Apêndice A**) e apresenta o nome e a média de cada aluno e, ao final, a média da turma.

EP2.12 Escreva um programa que lê o arquivo `Tudor.bin`, criado na **Seção 2.15.4**, e apresenta o nome e a média de cada aluno e, ao final, a média da turma.

EP2.13 Escreva uma função que determina o tamanho da maior linha de um stream de texto. (b) Escreva um programa que apresenta o tamanho da maior linha de um arquivo de texto cujo nome é introduzido via linha de comando.

EP2.14 Escreva um programa que conta os números de linhas, palavras e caracteres de um arquivo de texto cujo nome é recebido como argumento de linha de comando.

EP2.15 Escreva um programa que determina qual é a maior linha de um arquivo de texto cujo nome é introduzido via linha de comando.

EP2.16 Escreva um programa em C que lê e apresenta na tela, de 10 em 10 linhas, o conteúdo de um arquivo de texto cujo nome é especificado na linha de comando. Isto é, o programa deve escrever na tela as 10 primeiras linhas e solicitar que o usuário digite uma tecla qualquer para que as próximas 10 linhas sejam escritas e assim por diante, até que todo arquivo tenha sido apresentado. Suponha que o número máximo de caracteres numa linha do arquivo é `50`.

EP2.17 Escreva um programa em C que copia as `n` primeiras linhas de um arquivo de texto para um novo arquivo de texto. O nome dos arquivos e o número `n` de linhas que devem ser copiadas constituem dados solicitados pelo programa ao usuário. Assuma que o número máximo de caracteres numa linha do arquivo é `50`.

EP2.18 Escreva um programa que conta o número de ocorrências de cada letra encontrada num arquivo de texto. O nome do arquivo deve ser um argumento de linha de comando.

EP2.19 Considerando o arquivo binário `Tudor.bin` criado na **Seção 2.15.4**, escreva um programa que realize o seguinte:

(i) Leia o conteúdo desse arquivo binário e armazene num array apenas as matrículas de cada registro lido.
(ii) Obtenha uma matrícula válida do usuário.
(iii) Verifique se a matrícula faz parte do array que contém as matrículas.
(iv) Se a matrícula não for encontrada no array, informe o usuário e encerre o programa.
(v) Se a matrícula for encontrada no array, leia o registro que a possui no arquivo.
(vi) Altere para um valor especificado pelo usuário o valor do campo `n2` do registro lido no arquivo.
(vii) Atualize o arquivo escrevendo o registro no devido local.

EP2.20 Escreva um programa que conta os números de linhas, palavras e caracteres de um arquivo de texto cujo nome é recebido como argumento de linha de comando. O que deve separar palavras são espaços em branco, de acordo com **isspace()** e símbolos de pontuação, de acordo com **ispunct()**. Esse programa não considera palavra no sentido usual; i.e., palavra aqui é apenas uma sequência de caracteres sem espaços em branco ou caracteres de pontuação em seu interior.

EP2.21 Escreva um programa que remove comentários de um programa-fonte escrito em C. Suponha que o único tipo de comentário seja aquele delimitado por `/*` e `*/` e que o nome do programa-fonte seja recebido pelo programa via linha de comando.

EP2.22 Escreva uma função que substitui todas as ocorrências de um determinado caractere num arquivo de texto por outro caractere e retorna o número de substituições efetuadas.