Capítulo 11

Processamento de Arquivos

Após estudar este capítulo, você deverá ser capaz de:

OBJETIVOS

- Definir e usar a seguinte terminologia referente a arquivos:
  - ❒ Formato de arquivo
  - ❒ Arquivo de texto
  - ❒ Arquivo binário
  - ❒ Arquivo (genérico)
  - ❒ Arquivo armazenado
  - ❒ Stream
  - ❒ Stream de texto
  - ❒ Stream binário
  - ❒ Buffer
  - ❒ Buffering
  - ❒ Buffer de entrada
  - ❒ Registro
  - ❒ Campo de registro
  - ❒ Bloco de memória
  - ❒ Condição de exceção
  - ❒ Tratamento de exceção
  - ❒ Indicador de posição
  - ❒ Acessos direto e sequencial
  - ❒ Modo de abertura de arquivo
  - ❒ Modo de atualização de arquivo
- Expor como são utilizados os seguintes componentes da biblioteca padrão de C:
  - ❒ **FOPEN_MAX**
  - ❒ **FILENAME_MAX**
  - ❒ **SEEK_SET**
  - ❒ **SEEK_CUR**
  - ❒ **SEEK_END**
  - ❒ **FILE**
  - ❒ **clearerr()**
  - ❒ **getchar()**
  - ❒ **fopen()**
  - ❒ **fclose()**
  - ❒ **feof()**
  - ❒ **ferror()**
  - ❒ **scanf()**
  - ❒ **fgetc()**
  - ❒ **fputc()**
  - ❒ **fflush()**
  - ❒ **fgets()**
  - ❒ **fputs()**
  - ❒ **fread())**
  - ❒ **fwrite()**
  - ❒ **fscanf()**
  - ❒ **fseek()**
  - ❒ **ftell()**
  - ❒ **rewind()**
- Descrever o que ocorre quando se abre ou fecha um arquivo
- Explicar o funcionamento de leitura de dados via teclado
- 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 acesso sequencial ou acesso direto
- Processar arquivos por byte, linha ou bloco
- Lidar com erros em processamento de arquivos
- Implementar uma função para entrada de dados robusta
- Explicar a razão pela qual a função **gets()** nunca dever ser usada

11.1 Introdução

EM GERAL, PROCESSAMENTO DE ENTRADA E SAÍDA consiste em 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 e uma operação de **saída** (ou de **escrita**) copia dados da memória principal para um dispositivo de saída.

Em C, assim como em sistemas operacionais da família Unix, **arquivo** refere-se a qualquer dispositivo que possa ser utilizado como origem (p. ex., teclado) ou destino (p. ex., impressora) de dados de um programa. Assim, *processamento de entrada e saída* e *processamento de arquivos* são termos equivalentes em programação em C.

Processamento de arquivos em C é baseado no conceito de stream. 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. Qualquer arquivo (no sentido genérico descrito acima) pode ser associado a um stream e esse conceito permite que os programas processem entrada e saída de maneira uniforme e com independência de dispositivo.

Neste livro, **arquivo armazenado** 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). Além disso, aqui, na maioria das vezes, o termo *arquivo* é usado com o significado de *arquivo armazenado*.

11.2 Arquivos de Texto e Arquivos 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, ele não exibe informação humanamente legível. Em outras palavras, um arquivo binário é uma sequência irrestrita de bytes.

11.3 Streams

11.3.1 O Conceito de Stream

Stream[1] 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 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.

[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.

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*.

11.3.2 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 11.4.

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 11.7).

A implementação dos campos de uma estrutura **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.

11.4 Abrindo um Arquivo

Abrir um arquivo significa associá-lo a um stream, que é um ponteiro para uma estrutura do tipo **FILE** (v. Seção 11.3.2) 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 ao arquivo.

11.4.1 A Função fopen()

Uma operação de **abertura de arquivo** cria dinamicamente (v. Capítulo 12) 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. **Seção 11.4.3**). 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. **Seção 11.4.3**) 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 programa a seguir:

```
#include <stdio.h>

int main(void)
{
   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( "\nO arquivo nao pode ser aberto. O programa sera' encerrado.\n" );
      return 1;
   } else { /* Abertura foi OK */
      printf("\nArquivo aberto com sucesso\n");

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

   fclose(stream); /* Fecha o arquivo antes de encerrar */

   return 0;
}
```

Esse programa tenta abrir um arquivo de texto denominado `"Inexistente.txt"` apenas para leitura (v. **Seção 11.4.3**) por meio da instrução:

```
stream = fopen("Inexistente.txt", "r");
```

Em seguida, o programa verifica se a abertura foi bem sucedida ou se falhou usando:

```
if (stream == NULL)
```

Quando a expressão entre parênteses resulta em `1` (i.e., quando `stream` é **NULL**), o arquivo não pode ser aberto. Mas, como **NULL** vale zero (v. **Seção 5.2.4**), a linha inicial dessa instrução **if** pode ser reescrita como:

```
if (!stream)
```

A instrução **if-else** do programa em questão tem duas ramificações: uma seguida quando o arquivo não pode ser aberto (parte **if** da instrução) e a outra quando a abertura do arquivo é bem sucedida (parte **else** da instrução). Nesse caso específico, quando o arquivo não pode ser aberto, o programa apresenta uma mensagem e encerra sua execução. Mas, poderia ser diferente. Por exemplo, o programa poderia solicitar ao usuário o nome de um outro arquivo.

Quando a abertura do arquivo é bem sucedida, o último programa apenas apresenta a informação correspondente na tela. Mas, na prática, em vez disso, o programa deveria efetuar algum tipo de processamento com o arquivo.

Finalmente, após a escrita promovida pela última instrução **printf()** do programa acima, a função **fclose()** é chamada para fechar o arquivo antes que o programa seja encerrado (v. **Seção 11.5**).

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 não ser como exercício didático.

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, como ilustra o programa a seguir.

```
#include <stdio.h>
#include "leitura.h"

int main(void)
{
   char nomeArquivo[FILENAME_MAX];

   printf( "\n>>> Digite o nome do arquivo (maximo: %d "
           "caracteres)\n\t> ", FILENAME_MAX - 1);
   LeString(nomeArquivo, FILENAME_MAX);

   printf("\n>>> Nome do arquivo aceito: %s\n",nomeArquivo);

   return 0;
}
```

Na implementação de C usada para testar esse programa, ele apresenta o seguinte resultado:

```
>>> Digite o nome do arquivo (maximo: 259 caracteres)
        > MeuArquivo.txt

>>> Nome do arquivo aceito: MeuArquivo.txt
```

11.4.2 Streams de Texto e Streams 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 11–1** da **Seção 11.4.3**. 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**; i.e., usando um dos modos de abertura que se apresentam na **Tabela 11–2** da **Seção 11.4.3**. 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.

Resumindo o que foi exposto nesta seção, num stream de texto podem ocorrer traduções de acordo com o sistema operacional utilizado. Num stream binário, não ocorre nenhuma tradução, mesmo que ele esteja associado a um arquivo de texto.

11.4.3 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 11–1**.

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 11–1: 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 11–2** e aqueles apresentados na **Tabela 11–1** 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 11–2**.

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 11–2: 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 11–3 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
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 11–3: Modos de Acesso Usados em Atualização de Arquivo

11.5 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 11.7), 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, como mostra o seguinte exemplo:

```
#include <stdio.h>  /* Entrada e saída */
#include <stdlib.h> /* Função exit()   */

#define NOME_ARQ "Teste.txt" /* Nome do arquivo usado para testar o programa */

/****
 * FechaArquivo(): Tenta fechar um arquivo e, quando isso
 *                 não é possível, aborta o programa
 *
 * Parâmetros:
 *      stream (entrada) - stream associado ao arquivo
 *      nomeArq (entrada) - nome do arquivo ou NULL
 *
 * Retorno: Nada
 *
 * Observação: Se o segundo 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 */
   }
}

/****
 *
 * main(): Testa a função FechaArquivo()
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero, se não ocorrer nenhum erro.
 *          Um valor diferente de zero em caso contrário.
 *
 ****/
int main(void)
{
   FILE *stream;
   char  str[] = "Este e' um arquivo de texto";
```

```
   stream = fopen(NOME_ARQ, "w");

      /* Testa se arquivo foi aberto */
   if (!stream) {
      fprintf(stderr, "\a\nArquivo nao pode ser aberto\n");
      return 1;
   }

      /* Escreve conteúdo do string armazenado em str[] no arquivo */
   fprintf(stream, "%s", str);

      /* Força a descarga do buffer associado ao arquivo */
   fflush(stream);

      /* Solicita que o usuário remova o arquivo sendo processado pelo   */
      /* programa. Mas, nem todo usuário será capaz de realizar essa     */
      /* tarefa porque o arquivo provavelmente estará bloqueado pelo SO. */
   printf( "\n\t>>> Remova o arquivo \"%s\" e digite uma"
           "\n\t>>> tecla depois de fazer isso", NOME_ARQ );
   getchar(); /* Lê um caractere seguido de [ENTER] */

      /* Tenta fechar o arquivo e, se não conseguir, o programa será abortado */
   FechaArquivo(stream, NOME_ARQ);

      /* Esta instrução só será executada se o   */
      /* usuário não conseguir remover o arquivo */
   return 0;
}
```

Enquanto um arquivo está aberto por um programa, normalmente, o sistema operacional vigente bloqueia qualquer tentativa comum de remoção do arquivo ou alteração de seu nome. Mas existem maneiras de ludibriar o sistema. Por exemplo, em sistemas da família Windows, existem programas (p. ex., *Unlocker Assistant*) que remove o bloqueio do sistema operacional. O comando `rm`, encontrado na família de sistemas Unix, também possui opções que o permitem remover um arquivo mesmo que ele esteja bloqueado. Assim, quando o último programa é executado e o usuário segue devidamente as instruções, ele apresenta o seguinte resultado:

```
        >>> Remova o arquivo "Teste.txt" e digite uma
        >>> tecla depois de fazer isso

>>> Ocorreu erro no fechamento do arquivo Teste.txt.
>>> O programa sera' encerrado.
```

A função `FechaArquivo()` apresentada 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()**, como, por exemplo:

```
if (fclose(stream)) {
   printf("Erro no fechamento do arquivo");
   exit(1);
}
```

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 do 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:

Recomendação *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 11.15**).

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.

11.6 Ocorrências de Erros em Processamento de Arquivos

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. O seguinte programa ilustra essa argumentação.

```
#include <stdio.h>

int main(void)
{
   FILE *stream;
   char  str[] = "\nEste e' um arquivo de texto\n";
   int   c;

      /* Tenta criar um  arquivo para escrita e leitura */
   stream = fopen("Teste.txt", "w+");

      /* Verifica se arquivo foi aberto */
   if (!stream) {
      printf("\nArquivo nao pode ser aberto\n");
      return 1; /* Abertura de arquivo falhou */
   }
```

[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.

```
   /* Escreve o conteúdo do string armazenado em str[] no arquivo */
fprintf(stream, "%s", str);

   /* Verifica se ocorreu erro de escrita */
if (ferror(stream)) {
   printf("\n>>> Ocorreu um erro de escrita no arquivo"
          "\n>>> indicado por ferror()\n");
}

   /* Volta ao início do arquivo para lê-lo em seguida. Se ocorreu */
   /* erro de escrita, rewind() remove o indicativo de erro        */
rewind(stream);

printf("\n\t>>> Conteudo do arquivo <<<\n");

   /* Lê conteúdo do arquivo. O laço encerra quando ocorrer */
   /* erro ou tentativa de leitura além do final do arquivo */
while (1) {
   c = fgetc(stream); /* Lê um caractere no stream */

      /* Se ocorreu erro ou tentativa de leitura além */
      /* do final do arquivo, encerra o laço          */
   if (feof(stream) || ferror(stream)) {
      break;
   }

     /* Se não ocorreu erro nem tentativa de leitura além do */
     /* final do arquivo, exibe o caractere lido na tela     */
   putchar(c);
}

   /* Verifica se a função feof() indica que houve  */
   /* tentativa de leitura além do final do arquivo */
if (feof(stream)) {
   printf("\n>>> Tentativa de leitura alem do final"
          "\n>>> do arquivo indicada por feof()\n");
}

   /* Verifica se o último caractere retornado por fgetc() foi EOF */
if (c == EOF) {
   printf("\n>>> A funcao fgetc() retornou EOF\n");
}

   /* Verifica se ferror() indica ocorrência de erro */
if (ferror(stream)) {
   printf("\n>>> Ocorreu um erro de leitura no arquivo"
          "\n>>> indicado por ferror()\n");
}

fclose(stream); /* Fecha o arquivo */

return 0;
}
```

Apesar de o último programa conter algumas funções que serão discutidas apenas no próximo capítulo, seu funcionamento é simples e fácil de entender. Ou seja, o que esse programa faz é o seguinte:

1. Abre o arquivo `"Teste.txt"` no modo `"w+"`.
2. Escreve o string `"\nEste e' um arquivo de texto\n"` usando a função **fprintf**(). Essa função é semelhante à função **printf**(), mas diferentemente dessa última função que escreve apenas no meio de entrada padrão, **fprintf**() permite que se especifique o stream no qual a escrita será efetuada.

3. Faz o indicador de posição voltar ao início do arquivo por meio de uma chamada de **rewind()**. Essa função será discutida na Seção 11.13, mas pode-se adiantar que seu papel num programa é exatamente garantir que um arquivo seja processado a partir de seu primeiro byte.
4. Lê, caractere a caractere, o arquivo recém-criado usando a função **fgetc()**, que será escrutinada na Seção 11.11.1, e escreve cada caractere lido na tela usando **putchar()** (v. Seção 3.14.1).

O laço de repetição **while**, no corpo do qual os caracteres do arquivo são lidos e escritos, encerra quando há tentativa de leitura além do final do arquivo ou ocorre erro de leitura. Em qualquer dos casos, a função **fgetc()** retorna **EOF**. Portanto, para discriminar o que realmente causa o encerramento do aludido laço, é necessário usar **feof()** ou **ferror()**.

Quando o programa acima é executado, ele apresenta como resultado:

```
       >>> Conteudo do arquivo <<<
Este e' um arquivo de texto
>>> Tentativa de leitura alem do final
>>> do arquivo indicada por feof()
>>> A funcao fgetc() retornou EOF
```

Portanto, nesse caso, o significado de **EOF** é tentativa de leitura além do final do arquivo. Mas, se você trocar o modo de abertura do arquivo do programa acima de `"w+"` para `"w"` (i.e., de leitura e escrita para apenas escrita), essa singela alteração fará com que o resultado do programa passe a ser:

```
        >>> Conteudo do arquivo <<<
>>> A funcao fgetc() retornou EOF
>>> Ocorreu um erro de leitura no arquivo
>>> indicado por ferror()
```

Ou seja, agora, o significado de **EOF** é *ocorrência de erro de leitura*.

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

Recomendações

- ***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 11.3.2) 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 11.13), cuja finalidade precípua é mover o indicador de posição de um arquivo (v. Seção 11.3.2) para seu início, remove condição de erro ou de

final de arquivo de um stream. Por outro lado, **fseek()** (v. Seção 11.12.1) remove 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.

11.7 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 11.8) 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 11.4.3), 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 11.8), 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 arquivos 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 11.6), se ocorrer algum erro durante sua execução; caso contrário, ela retorna `0`. Contudo, raramente, o valor retornado por essa função é testado.

11.8 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.

11.9 Leitura de Dados via Teclado

Os programas apresentados até aqui usam funções da biblioteca LeituraFacil para leitura, por meio do teclado, de valores dos tipos de dados primitivos usados neste livro. Essa biblioteca foi projetada com o objetivo de tornar entrada de dados em C palatável ao aprendiz de programação. Isto é, sem o uso dessa biblioteca, para escrever programas capazes de responder adequadamente a qualquer espécie de informação introduzida pelo usuário, o aprendiz teria que lidar com um tema complexo em programação em C. A complexidade deste tópico é tal que poucos programadores com relativa experiência em linguagem C são capazes de dominá-lo completamente. Mas, com a assimilação do material exposto até este ponto, espera-se que o leitor já tenha cabedal suficiente para acompanhar a discussão que será apresentada adiante.

Todas as funções da biblioteca padrão de C que serão discutidas a seguir são declaradas no cabeçalho `<stdio.h>`.

11.9.1 A Função getchar()

A função **getchar()** lê um caractere no meio de entrada padrão (v. Seção 11.8) e seu protótipo é:

```
int getchar(void)
```

Quando a função **getchar()** é chamada e o buffer associado a **stdin** está vazio (v. Seção 11.9.3), a execução do programa que a chama é interrompida até que o usuário digite um caractere seguido de `[ENTER]`. Então, a função **getchar()** lê o caractere digitado e retorna o valor inteiro correspondente a ele. Por exemplo, se um programa contém o seguinte fragmento:

```
int meuCaractere;

meuCaractere = getchar();
```

e, quando a chamada de **getchar()** for executada, o usuário pressionar a tecla `[A]` e, então, `[ENTER]`, à variável `meuCaractere` será atribuído o valor inteiro correspondente ao caractere `'A'` no código de caracteres vigente.

Note, no exemplo acima, que o tipo da variável `meuCaractere` é **int**, e não **char**, como se poderia esperar. Isso ocorre porque a função **getchar()** retorna um valor inteiro correspondente a um caractere *apenas* quando ela consegue realmente ler um caractere. Quando isso não é possível, essa função retorna o valor da constante **EOF** (v. Seção 11.6), que não pode ser contido numa variável do tipo **char**. Por outro lado, a largura do tipo **int** é duas, quatro ou até oito vezes maior do que a largura do tipo **char**. Portanto uma variável do tipo **int** é capaz de acomodar com folga um valor do tipo **char**, que sempre ocupa um byte.

Como exemplo de uso de **getchar()**, considere a função `LeLinha()` que faz parte do programa a seguir. Essa função lê uma linha de texto introduzida pelo usuário via teclado desprezando espaços em branco iniciais.

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

   /* Número máximo de caracteres permitidos, sem */
   /* considerar caracteres em branco iniciais    */
```

```
#define TAMANHO_MAX_LINHA 30

/****
 *
 * LeLinha(): Lê uma linha de texto introduzida pelo usuário no
 *            meio de entrada padrão desprezando espaços em branco iniciais
 *
 * Parâmetros:
 *     str[] (saída) - array no qual os caracteres lidos mais '\0' serão armazenados
 *     n (entrada) - tamanho do array str[]
 *
 * Retorno: O número de espaços em branco iniciais saltados
 *
 ****/
int LeLinha(char str[], int n)
{
   int c, /* Armazena cada caractere lido */
       i = 0, /* Indexador de caracteres armazenados */
       espacosSaltados = 0; /* Número de espaços saltados */

      /* Salta espaços no início da linha. O último caractere  */
      /* atribuído a c não é espaço em branco e encerra o laço */
   while ( isspace(c = getchar()) ) {
      ++espacosSaltados;
   }

      /* Lê os demais caracteres da linha */
   while (c != '\n' && c != EOF) {
         /* Verifica se o número máximo de caracteres permitido foi lido */
      if (i >= n - 1) { /* Limite atingido */
         break; /* Encerra o laço */
      }
      str[i] = c; /* Armazena o caractere lido */

      i++; /* Obtém o índice do próximo caractere */

      c = getchar(); /* Lê o próximo caractere */
   }

   str[i] = '\0'; /* Termina o string */

   return espacosSaltados;
}

/****
 *
 * main(): Testa a função LeLinha()
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   char str[TAMANHO_MAX_LINHA + 1];
   int  nEspacosSaltados;

   printf( "\n>>> Digite uma linha de texto com espacos em"
           "\n>>> branco no inicio (Maximo = %d caracteres)"
           ":\n\t> ", TAMANHO_MAX_LINHA );
```

```
   nEspacosSaltados = LeLinha(str, TAMANHO_MAX_LINHA);
   printf("\n>>> String sem espacos iniciais: \"%s\"", str);
   printf( "\n>>> Numero de espacos em branco saltados: %d\n", nEspacosSaltados );
   return 0;
}
```

A seguir, um exemplo de execução do último programa:

```
>>> Digite uma linha de texto com espacos em
>>> branco no inicio (Maximo = 30 caracteres):
        >       Bom dia

>>> String sem espacos iniciais: "Bom dia"
>>> Numero de espacos em branco saltados: 6
```

O programa acima usa a função `LeLinha()` para ler uma linha de texto introduzida pelo usuário no meio do teclado desprezando espaços em branco iniciais (incluindo `[ENTER]` e `[TAB]`) que porventura o usuário digitar. Essa função salta e conta espaços iniciais digitados usando o laço:

```
while ( isspace(c = getchar()) ) {
   ++espacosSaltados;
}
```

A expressão que controla esse laço é o valor retornado pela função **isspace()** (v. Seção 9.7.1), que recebe como parâmetro o valor retornado e atribuído à variável `c` pela função **getchar()**. Assim, quando essa última função lê um caractere considerado espaço em branco pela função **isspace()**, o corpo do laço é adentrado e conta-se mais um espaço em branco lido. O referido laço encerra quando **getchar()** lê um caractere que não é considerado espaço em branco. Então, a função `LeLinha()` passa a ler os demais caracteres digitados pelo usuário utilizando o laço:

```
while (c != '\n' && c != EOF) {
   if (i >= n - 1) {
      break;
   }
   str[i] = c;
   i++;
   c = getchar();
}
```

Esse último laço possui três condições de parada (apesar de apenas duas serem evidentes):

- O último caractere lido é quebra de linha (`'\n'`).
- A função **getchar()** retorna **EOF**, que sinaliza erro de leitura ou tentativa de leitura além do final do arquivo (v. Seção 11.6). Em condições normais, não há final de arquivo em leitura via teclado, mas essa condição pode ser emulada por meio da combinação de teclas `[CTRL]+[Z]` (em sistemas da família Windows/DOS) ou `[CTRL]+[D]` (em sistemas da família Unix).
- O número de caracteres lidos excede o tamanho do array no qual esses caracteres serão armazenados menos um (porque é necessário deixar espaço sobressalente para armazenar o caractere `'\0'`). Essa condição de parada é representada pela instrução **if** no corpo do laço sob discussão.

Enquanto em execução, o corpo do referido laço **while** armazena o último caractere lido e, então, lê o próximo caractere. Logo após a saída do segundo laço, a instrução:

```
str[i] = '\0';
```

acrescenta o caractere `'\0'` depois do último caractere armazenado no array no interior do laço, de forma que, ao final, o array armazenará um string.

11.9.2 A Função scanf()

A função **scanf()** permite a leitura de um número arbitrário de valores de vários tipos ao mesmo tempo no meio de entrada padrão. O primeiro parâmetro de **scanf()**, que é obrigatório, é um **string de formatação** semelhante àquele usado pela função **printf()**. No entanto, normalmente, no caso da função **scanf()**, constam no string de formatação apenas especificadores de formato e espaços em branco. Quer dizer, quando caracteres que não são especificadores de formato nem espaços em branco são incluídos no string de formatação, **scanf()** espera que o usuário digite esses caracteres exatamente como eles são. Por exemplo, na chamada:

```
scanf("Digite um inteiro: %d", &x);
```

a função **scanf()** espera que o usuário digite exatamente os caracteres contidos em *Digite um inteiro:* antes de digitar um número inteiro em base decimal. Nesse exemplo, muito provavelmente, a intenção do programador seria apresentar um prompt para o usuário, o que não ocorrerá, visto que **scanf()** é uma função de entrada (e não de saída). Assim, com **scanf()**, não é recomendado o uso de caracteres que não fazem parte de especificadores de formato nem são considerados espaços em branco.

Os parâmetros que seguem o string de formatação de **scanf()** são endereços de variáveis nas quais os dados lidos serão armazenados. Esses últimos parâmetros são todos de saída, o que justifica o uso de endereços de variáveis (v. **Seção 5.5**). Tenha cuidado, pois um erro muito comum de programação em C é esquecer de preceder com o símbolo **&** cada variável usada como parâmetro de **scanf()**. Alguns compiladores, como GCC, apresentam mensagens de advertência alertando o programador para esse fato.

O protótipo da função **scanf()** é:

```
int scanf(const char *string, ...)
```

Os três pontos que aparecem no protótipo da função **scanf()** têm sentido conotativo, e não simbólico. Isto é, eles realmente fazem parte do protótipo da função. Funções dessa natureza ainda não foram formalmente discutidas neste livro. A função **printf()** tem um protótipo semelhante ao da função **scanf()**, que ainda não foi apresentado formalmente para não acrescentar embaraços desnecessários num texto introdutório. Essas funções são denominadas **funções com parâmetros variantes** e permitem o uso de parâmetros em quantidade e tipos indeterminados (representados por três pontos). Uma completa discussão sobre como essas funções são definidas está bem além do escopo de um livro introdutório, mas o uso delas é relativamente fácil [quantas vezes você já usou **printf()** até aqui sem ter ciência dessa informação?].

O valor retornado por **scanf()** representa o número de variáveis que tiveram seus valores alterados em virtude de uma chamada dessa função, a não ser que ocorra erro de leitura ou sinalização de final de arquivo. Nesse último caso, a função **scanf()** retorna **EOF** (v. **Seção 11.6**). O valor retornado por **scanf()** é usado para determinar se a chamada foi bem sucedida ou não, como será visto na **Seção 11.9.5**.

O string de formatação de **scanf()** especifica qual é o formato esperado dos dados que serão lidos e atribuídos às variáveis cujos endereços constituem os parâmetros que seguem o referido string. A **Tabela 11–4** enumera os especificadores de formato mais comuns utilizados pela função **scanf()**.

Especificador de Formato	O que scanf() espera ler?
%c	Um caractere
%s	Uma cadeia de caracteres
%d	Um número inteiro em base decimal do tipo **int**
%lf	Um número real do tipo **double**

Tabela 11–4: Especificadores de Formato Comuns Utilizados por scanf()

A despeito do que alguns programadores de C imaginam, os especificadores **%d** e **%i** (que não aparece na Tabela 11–4) são equivalentes quando usados com **printf()**, mas esse não é o caso quando eles são usados com **scanf()**. Isto é, quando **%d** é usado com **scanf()**, essa função espera ler um número inteiro na base decimal. Por outro lado, **%i** permite que **scanf()** seja capaz de ler valores em outras bases numéricas. Como este livro usa apenas inteiros na base decimal, apenas **%d** é utilizado e recomendado para leitura de números inteiros.

O uso correto da função **scanf()** requer que haja um endereço de variável para cada especificador de formato no string de formatação e que o tipo de cada variável seja compatível com a especificação de formato correspondente. Por exemplo, a chamada de **scanf()** no fragmento de programa a seguir espera ler dois valores no meio de entrada padrão:

```
int    umInt;
double umReal;

printf("Digite um valor inteiro e outro real: ");
scanf("%d %lf", &umInt, &umReal);
```

Nesse trecho de programa, a função **scanf()** espera, primeiro, ler caracteres que possam ser convertidos num valor do tipo **int** a ser atribuído à variável `umInt`. Em seguida, ela presume ler caracteres que possam ser convertidos num valor do tipo **double** a ser atribuído à variável `umReal`.

Apesar de **scanf()** permitir a leitura de vários valores a cada chamada, é recomendável fazer a leitura de apenas um dado de cada vez, pois, quando vários valores são lidos simultaneamente e detecta-se que um deles foi introduzido incorretamente, talvez o usuário tenha que reintroduzir todos os valores novamente, mesmo aqueles que ele digitou corretamente. Adotando essa recomendação, a chamada da função **scanf()** do último exemplo deveria ser substituída por:

```
printf("Digite um numero inteiro: ");
scanf("%d", &umInt);

printf("Digite um numero real: ");
scanf("%lf", &umReal);
```

O último trecho de programa representa um melhoramento em relação à chamada única da função **scanf()** do exemplo anterior, mas ainda padece de um grave problema que afeta quase todas as leituras de dados: a dependência do usuário. Isto é, diferentemente de saída de dados, que depende exclusivamente do programador, entrada de dados depende do comportamento do usuário do programa ou até mesmo do bom funcionamento do meio de entrada (e. g, um teclado defeituoso pode acarretar uma leitura de dados indevida). Para sentir melhor esse problema, digite e compile o programa abaixo.

```
#include <stdio.h>

int main(void)
{
   int x;

   printf("\nDigite um inteiro: ");
   scanf("%d", &x);

   printf("\nO numero inteiro introduzido foi: %d", x);
   printf("\nO dobro desse numero e': %d", 2*x);
   printf("\nO quadrado desse numero e': %d\n", x*x);

   return 0;
}
```

Depois de compilar o programa, exerça o papel de um usuário desatento (ou mal intencionado) e, quando instado a digitar um número inteiro, digite algumas letras. O resultado que você obtém na tela depende de compilador e sistema utilizados, mas, qualquer que seja o resultado, ele não faz absolutamente nenhum sentido porque você não digitou um número com o qual o programa pudesse calcular seu dobro ou quadrado.

A solução do último problema exposto não é muito trivial e será apresentada nas próximas seções.

11.9.3 Entendendo Leitura de Dados via Teclado

Na prática, usar corretamente uma função da biblioteca padrão que faz leitura via teclado não é tão trivial quanto sugerem as funções da biblioteca LeituraFacil [p. ex., `LeInteiro()`]. As dificuldades, nesse caso, residem no fato de a leitura de dados depender do comportamento do usuário do programa. Isto é, diferentemente de saída de dados que, repetindo, depende apenas do programador, leitura de dados depende daquilo que o usuário introduz como dados para o programa, o que nem sempre corresponde àquilo que o programa espera. Foi levando isso em consideração que as funções da biblioteca LeituraFacil foram criadas.

Para ser capaz de construir programas robustos (i.e., resistentes a usuários), é preciso entender o funcionamento da leitura de dados via teclado. Depois, é necessário aprender a construir trechos de programa dedicados a leitura e análise de dados introduzidos pelo usuário.

Leitura de dados para programas interativos baseados em console é normalmente efetuada por meio do teclado. Além disso, quaisquer que sejam as teclas pressionadas pelo usuário durante uma leitura de dados, apenas caracteres são enviados para um programa. Por exemplo, mesmo que o usuário tecle apenas dígitos, o programa não recebe números diretamente do teclado. Nesse caso, o programa só receberá um número se houver uma função que leia os caracteres digitados e converta-os num número. A função **scanf()**, por exemplo, é capaz de realizar essa tarefa.

Outro ponto importante que deve ser ressaltado é que, quando uma função que efetua leitura via teclado [p. ex., **getchar()**] é executada, ela não tenta ler caracteres diretamente nesse dispositivo de entrada. Ao invés disso, a leitura é feita num buffer (v. **Seção 11.7**) para o qual os caracteres introduzidos pelo usuário são enviados e enfileirados após o usuário pressionar a tecla `[ENTER]`. Além disso, uma função que efetua leitura no meio de entrada padrão só causa interrupção do programa à espera da introdução de caracteres se ela não encontrar nenhum deles no buffer associado a esse meio de entrada.

Compreender bem o funcionamento do buffer associada ao teclado é fundamental para lidar com possíveis problemas com leitura de dados. Para começar a entender o funcionamento da leitura de dados via teclado, considere, por exemplo, o seguinte programa:

Programa 1

```
#include <stdio.h>

int main(void)
{
   int umChar;

   printf("\nDigite um caractere: ");
   umChar = getchar();

   printf("\nDigite outro caractere: ");
   umChar = getchar();

   return 0;
}
```

Esse programa é bem simples, mas causa enorme frustração ao programador iniciante que não entende o funcionamento da leitura de dados via teclado. Tudo que esse programa faz é solicitar que o usuário introduza dois caracteres e ler esses caracteres usando a função **getchar**(). O que o programador certamente deseja que aconteça é que o usuário digite cada caractere quando solicitado pelo programa, mas não é isso que ocorre.

Para sentir melhor o que se está afirmando, execute o último programa e tente introduzir o caractere `A` quando for escrito na tela Digite um caractere: e o caractere `B` quando for escrito Digite outro caractere:. Se você seguir essas recomendações, verá que, quando tentar introduzir o caractere `B`, o programa já terá encerrado. Por que o programa age dessa maneira? Para perceber o que realmente acontece com esse programa, acompanhe, a seguir, sua execução passo a passo.

A primeira instrução do programa é:

```
printf("Digite um caractere:");
```

que escreve na tela do computador:

```
Digite um caractere:
```

Em seguida, a função **getchar**() começa a ser executada. Como essa função não encontra nenhum caractere armazenado no buffer, ela causa a interrupção do programa e espera que um caractere seja depositado no buffer. Quando você pressiona a tecla `[A]` para introduzir o caractere `'A'`, também precisa pressionar `[ENTER]` para encerrar a leitura de dados. Acontece que `[ENTER]` representa igualmente um caractere, que é o caractere de quebra de linha, representado em C por `'\n'`. Portanto, logo após a digitação desses caracteres, o conteúdo do buffer é aquele mostrado na Figura 11–1.

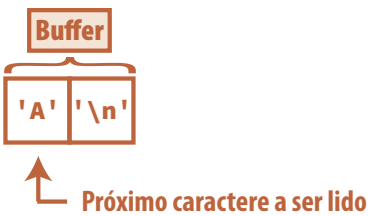

Figura 11–1: Conteúdo do Buffer de Teclado 1

Observe nessa figura que os caracteres são enfileirados no buffer; i.e., eles são armazenados na ordem na qual o usuário os introduz. As operações de leitura no buffer também obedecem a essa mesma ordem.

Prosseguindo com a execução do programa, a função **getchar**(), que estava à espera que algum caractere fosse armazenado no buffer, conclui sua execução lendo o primeiro caractere que lá se encontra. Então, esse caractere é removido do buffer, que fica com o conteúdo mostrado na Figura 11–2.

FIGURA 11–2: CONTEÚDO DO BUFFER DE TECLADO 2

Em seguida, é executada a próxima instrução do programa:

```
printf("Digite outro caractere:");
```

causando a escrita na tela de:

```
Digite outro caractere:
```

Depois, a segunda chamada de **getchar()** é executada. Agora, diferentemente do que ocorre com a primeira chamada dessa função, ela encontra um caractere (i.e., `'\n'`) no buffer de entrada (v. última figura) e, portanto, não interrompe a execução do programa à espera de outro caractere. Nesse caso, a função **getchar()** lê o caractere `'\n'` que já se encontrava no buffer. Assim, o usuário não tem chance de digitar outro caractere, pois, logo em seguida, o programa é encerrado.

Antes de apresentar uma solução para o problema suscitado no programa anterior, é importante ressaltar que, além de **getchar()**, outras funções de leitura podem deixar caracteres remanescentes no buffer que podem causar o mesmo problema em tentativas subsequentes de leitura. Para apreciar o que foi afirmado, considere como exemplo o seguinte programa:

PROGRAMA 2

```
#include <stdio.h>

int main(void)
{
   int  umChar, umInt, nValoresLidos;

   printf("\nDigite um valor inteiro:");
   nValoresLidos = scanf("%d", &umInt);

   printf( "\n\t>>> Numero de valores lidos: %d"
           "\n\t>>> Valor lido: %d\n", nValoresLidos, umInt );

   printf("\nDigite um caractere: ");
   umChar = getchar();

   return 0;
}
```

Um exemplo de interação com o último programa é apresentado abaixo:

```
Digite um valor inteiro: 345

        >>> Numero de valores lidos: 1
        >>> Valor lido: 345

Digite um caractere:
```

Novamente, o programa não permite que o usuário digite o último caractere solicitado. Para compreender melhor a razão desse resultado inesperado, observe inicialmente na Figura 11–3 o conteúdo do buffer de entrada logo após o usuário pressionar em sequência as teclas [3], [4], [5] e [ENTER].

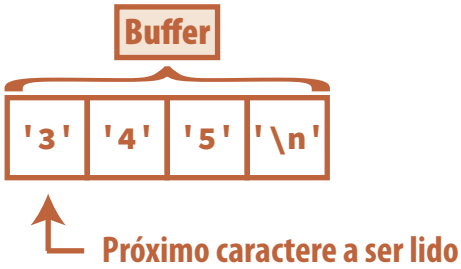

FIGURA 11–3: CONTEÚDO DO BUFFER DE TECLADO 3

É importante notar que o comportamento da função **scanf()** é ditado pelos especificadores de formato que se encontram no string de formatação (primeiro parâmetro da função). No caso em questão, há apenas um especificador de formato, que é **%d**. Esse especificador informa à função **scanf()** que ela deve ler caracteres com os quais ela possa constituir um número inteiro na base decimal. Evidentemente, os únicos caracteres que satisfazem essa especificação são os dígitos de `0` a `9` e os símbolos de sinal `+` e `-`. Portanto a tarefa que a função **scanf()** espera realizar nesse caso é ler dígitos da base decimal e, eventualmente, um símbolo de sinal até encontrar um caractere que não satisfaça essa especificação. Quando a função **scanf()** lê um caractere que não está de acordo com essa especificação, ela devolve-o ao buffer, converte apenas aqueles caracteres que ela acumulou durante a leitura num número inteiro e, finalmente, armazena o valor obtido na variável correspondente ao especificador **%d** (nesse caso, a variável `umInt`).

No caso específico do exemplo em questão, a atuação da função **scanf()** obedece à seguinte sequência de eventos:

1. A função lê e guarda os caracteres `'3'`, `'4'` e `'5'`, pois eles podem constituir um número inteiro na base decimal.
2. Ela lê e devolve ao buffer o caractere `'\n'`, pois esse caractere não pode fazer parte de um número.
3. Ela encerra a leitura.
4. A função converte os caracteres `'3'`, `'4'` e `'5'` no número inteiro `345`.
5. O número inteiro obtido (i.e., `345`) é armazenado na variável `umInt`.
6. Finalmente, a função retorna `1`, porque uma variável foi modificada (i.e., lhe foi atribuído um valor).

Ao final da ação da função **scanf()**, a situação no buffer de entrada pode ser ilustrada como na **Figura 11–4**.

FIGURA 11–4: CONTEÚDO DO BUFFER DE TECLADO 4

Como a função **scanf()** deixa no buffer o caractere `'\n'` e, adiante, o último programa tenta ler um caractere usando a função **getchar()**, o resto da história de execução desse programa é a mesma do penúltimo exemplo.

Antes de conhecer a solução para os problemas detectados nos últimos programas, considere mais um exemplo (**Programa 3**) que ilustra outra característica da função **scanf()**.

Programa 3

```
#include <stdio.h>

int main(void)
{
   int  umInt, outroInt;

   printf("\nDigite um valor inteiro: ");
   scanf("%d", &umInt);

   printf("Valor lido: %d\n", umInt);

   printf("\nDigite outro valor inteiro: ");
   scanf("%d", &outroInt);

   printf("Valor lido: %d\n", outroInt);

   return 0;
}
```

Se você compilar e executar esse último programa verá que, se você agir como um usuário *bem comportado*, ele funcionará corretamente como mostra o exemplo de interação a seguir:

```
Digite um valor inteiro: 345
Valor lido: 345

Digite outro valor inteiro: 543
Valor lido: 543
```

Agora, a diferença mais relevante entre esse último programa e aquele que o antecede é que a segunda leitura nesse último programa é feita com a função **scanf**(), enquanto, naquele programa, a segunda tentativa de leitura foi realizada com **getchar**(). Novamente, a execução desse último programa será esmiuçada adiante para que você possa compreender por que isso acontece.

Até logo antes da execução da segunda chamada de **printf**(), o último programa comporta-se como o programa anterior a ele. Ou seja, a função **scanf**() lê os caracteres `'3'`, `'4'` e `'5'`, deixa o caractere `'\n'` no buffer, forma um inteiro e o atribui à variável `umInt`. Então, o programa executa uma chamada de **printf**() que apresenta o valor lido. Nesse instante, a configuração do buffer de entrada pode ser ilustrada como na Figura 11–5.

Figura 11–5: Conteúdo do Buffer de Teclado 5

Essa configuração de buffer é exatamente igual àquela no instante em que o programa anterior tenta ler um caractere usando **getchar**() e fracassa. Acontece que, agora, tenta-se ler um número inteiro e consegue-se, conforme mostra o último exemplo de interação apresentado. Então, qual é a diferença entre os dois casos?

No primeiro caso, conforme já foi explicado, o problema ocorreu porque a função **getchar**() encontrou o caractere `'\n'` no buffer, leu esse caractere e, portanto, não aguardou que o usuário digitasse outro caractere. No caso corrente, a função **scanf**() também encontrou o mesmo caractere no buffer, mas, mesmo assim, ela permitiu que o usuário introduzisse um número inteiro. Quer dizer, a diferença entre os dois casos é que, quando

a função **scanf()** é solicitada a ler um valor numérico, como é o caso quando o especificador de formato é **%d**, ela lê e despreza espaços em branco encontrados no início do buffer e ela considera o caractere `'\n'` como espaço em branco (outros espaços em branco comuns são tabulação e o caractere de espaço obtido com a barra de espaço de um teclado). Assim, como a função encontra apenas `'\n'` no buffer de entrada, ela salta esse caractere e espera que o usuário digite algum caractere que não seja considerado espaço em branco.

Agora, o último programa funciona adequadamente apenas porque o usuário foi bem comportado. Isto é, o usuário seguiu exatamente as recomendações do programa. Se o usuário for mal comportado, o programa deixará de funcionar corretamente. Para constatar o que está sendo afirmado, execute novamente o último programa e, quando instado, digite um número inteiro seguido de um caractere (p. ex., `5x`) e, então, digite `[ENTER]`. Se você fizer isso, verá que o programa não lhe dá chance para digitar o segundo número inteiro. Pior ainda, se, após o primeiro prompt, você digitar uma letra seguida de `[ENTER]`, observará o encerramento do programa sem a leitura de nenhum valor. Com base no que foi discutido nesta seção, tente explicar o comportamento do último programa nessas duas situações. Se não conseguir encontrar as justificativas procuradas, releia esta seção antes de prosseguir.

11.9.4 Esvaziamento do Buffer de Entrada

A conclusão que se obtém dos exemplos apresentados na **Seção 11.9.3** é que, antes de fazer uma leitura via teclado, é necessário garantir que buffer associado a esse meio de entrada esteja vazio. Uma maneira de realizar isso é por meio de uma chamada da função `LimpaBuffer()` definida como:

```
void LimpaBuffer(void)
{
   int c; /* Caractere lido */

   do {
      c = getchar();
   } while (c != '\n' && c != EOF);
}
```

A função `LimpaBuffer()` lê e descarta repetidamente todos os caracteres encontrados no buffer de entrada até encontrar o caractere `'\n'` ou **getchar()** retornar **EOF** (v. **Seção 11.6**).

Normalmente, o caractere `'\n'` encerra entrada de dados via teclado, mas existem programas, tipicamente, denominados **filtros** (v. **Seção 11.15.2**), que usam o meio de entrada padrão como se ele fosse um editor de texto, de modo que a entrada de dados não encerra quando o usuário digita `[ENTER]` (representado por `'\n'` em C). Nesses casos, a entrada de dados encerra quando o usuário sinaliza final de arquivo por meio de uma combinação de teclas (v. **Seção 11.9.1**).

Apesar da simplicidade da função `LimpaBuffer()`, não existe nenhuma função na biblioteca padrão de C que execute a operação que ela realiza (v. **Seção 11.7**).

Para resolver os problemas apresentados nos programas da **Seção 11.9.3**, deve-se acrescentar a função `LimpaBuffer()` a esses programas e inserir uma chamada dela entre duas operações seguidas de leitura, como mostram os seguintes programas, que corrigem os dois primeiros programas da **Seção 11.9.3**. (O terceiro programa dessa seção pode ser corrigido de maneira semelhante.)

Programa 1 Corrigido

```
#include <stdio.h>

void LimpaBuffer(void)
{
   int c; /* Caractere lido */

   do {
      c = getchar();
   } while (c != ‘\n’ && c != EOF);
}

int main(void)
{
   int umChar;

   printf("\nDigite um caractere:");
   umChar = getchar();

      /* Se foi lido algum cara- */
      /* ctere, limpa o buffer   */
   if (umChar !=  EOF) {
      LimpaBuffer();
   }

   printf("\nDigite outro caractere:");
   umChar = getchar();

   return 0;
}
```

Programa 2 Corrigido

```
#include <stdio.h>

void LimpaBuffer(void)
{
   int c; /* Caractere lido */

   do {
      c = getchar();
   } while (c != '\n' && c != EOF);
}

int main(void)
{
   int  umChar, umInt, nValoresLidos;

   printf("\nDigite um valor inteiro: ");
   nValoresLidos = scanf("%d", &umInt);

      /* Se scanf() não retornou    */
      /* EOF, apresenta o resultado */
      /* e limpa o buffer           */
   if (nValoresLidos !=  EOF) {
      printf("\n\t>>> Numero de valores"
             " lidos: %d \n\t>>> Valor",
             " lido: % d\n",
             nValoresLidos, umInt );

      LimpaBuffer();
   } else { /* scanf() retornou EOF */
      printf("\n\t>>> Ocorreu erro de "
             "leitura\n");
   }
   printf("\nDigite um caractere: ");
   umChar = getchar();

   return 0;
}
```

Agora, os programas corrigidos exibidos acima funcionam mesmo quando o usuário digita caracteres além do esperado. Por exemplo, suponha que ocorra a seguinte execução do **Programa 2 Corrigido**:

```
Digite um valor inteiro: 345ab

        >>> Numero de valores lidos: 1
        >>> Valor lido: 345

Digite um caractere: U
```

Nessa execução do **Programa 2 Corrigido**, o usuário digitou `345ab`, ao invés do número inteiro solicitado. Portanto, logo após a introdução desses caracteres seguidos de `[ENTER]`, o conteúdo do buffer de entrada é aquele mostrado na **Figura 11–6**.

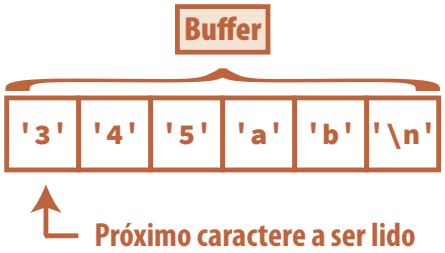

FIGURA 11–6: CONTEÚDO DO BUFFER DE TECLADO 6

Como antes, a função **scanf()** lê caracteres que possam compor um número inteiro e encerra a leitura quando encontra o primeiro caractere que não pode fazer parte do número inteiro. No exemplo de interação do **Programa 2** da **Seção 11.9.3**, essa função deixava no buffer apenas o caractere `'\n'`, mas agora ela deixa os caracteres `'a'`, `'b'` e `'\n'`. Entretanto, isso não é mais obstáculo, pois a função `LimpaBuffer()` remove todos os caracteres do buffer de entrada, independentemente de quantos ou quais eles sejam.

Agora, suponha que, ao ser solicitado a digitar um número inteiro, o usuário digite `ab`. Nesse caso, o programa não obteria o número inteiro solicitado. Mas, e se essa fosse uma situação real na qual o programa necessitasse desse valor para prosseguir com sua execução? A subseção seguinte lida com essa categoria de problemas.

11.9.5 Uso de Laços de Repetição em Leitura de Dados

Laços de repetição são utilizados em leitura de dados para oferecer novas chances ao usuário após ele ter introduzido dados considerados inusitados pelo programa. Para entender como isso funciona, considere o programa a seguir:

```
#include <stdio.h>

int main(void)
{
   int  umInt, outroInt;

   printf("\nDigite um inteiro: ");
   scanf("%d", &umInt);

   printf("\nDigite outro inteiro: ");
   scanf("%d", &outroInt);

   printf( "\n\n\t>>> %d + %d = %d\n",
           umInt, outroInt, umInt + outroInt );

   return 0;
}
```

Esse programa funciona perfeitamente bem, desde que o usuário seja *bem comportado*; i.e., se ele introduzir corretamente dois números inteiros. Mas, este é justamente o problema desse programa: ele depende do usuário para funcionar bem. Um bom programador não deve jamais fazer suposições de bom comportamento por parte do usuário. O que aconteceria, então, se o usuário não agisse conforme o esperado? Algumas possibilidades serão analisadas logo abaixo.

Suponha, inicialmente, que, quando solicitado a introduzir o primeiro inteiro, o usuário digite `345ab`. Conforme foi visto na **Seção 11.9.3**, nesse caso, a função **scanf()**, lê e transforma em número inteiro os caracteres `'3'`, `'4'` e `'5'` e deixa no buffer os caracteres `'a'`, `'b'` e `'\n'`. A solução para o problema de caracteres remanescentes foi apresentada na **Seção 11.9.4**. Ou seja, basta inserir, após essa chamada de **scanf()**, uma chamada da função `LimpaBuffer()`, descrita na **Seção 11.9.4**, e esse problema estará resolvido.

Se o usuário digitar apenas um número inteiro, a função **scanf()** deixa o caractere `'\n'` no buffer, mas isso não constitui problema para a próxima chamada dessa função porque, nesse caso, ela salta caracteres em branco encontrados no início do buffer. A função **scanf()** só não age desse modo quando se utiliza o especificador **%c** para leitura de um único caractere.

Agora, suponha que o usuário digita `ab25` quando solicitado a introduzir o primeiro inteiro. Nessa situação, a função **scanf()** não será capaz de transformar a entrada do usuário num número inteiro, pois o primeiro caractere encontrado é `'a'` e esse caractere não pode fazer parte de um número inteiro na base decimal. Consequentemente, nenhum valor será atribuído à variável `umInt`.

Como primeira tentativa de solução desse último problema, deve-se verificar se a função **scanf()** lê realmente um número inteiro como deveria. Isso é efetuado testando-se o valor retornado por essa função, como mostrado no fragmento de programa a seguir:

```
printf("\nDigite um inteiro: ");
nValLidos = scanf("%d", &umInt);

if (nValLidos != 1) { /* Nenhum inteiro foi lido */
      /* Se scanf() retornou EOF não há caracteres a serem removidos */
   if (nValLidos != EOF) {
      LimpaBuffer();
   }

   printf("\a\nValor incorreto.\nDigite um inteiro: ");
   nValLidos = scanf("%d", &umInt);
}
```

Esse trecho de programa é fácil de ser entendido. A função **scanf()** retorna o número de variáveis que tiveram seus valores alterados por ela ou **EOF** (v. **Seção 11.9.2**). No caso em questão, a função **scanf()** recebe apenas um endereço de variável (i.e., `&umInt`) como parâmetro e, portanto, o valor retornado só poderá ser:

- `0`, quando a variável `umInt` não for alterada porque os caracteres digitados pelo usuário não puderem ser convertido num valor inteiro na base decimal.
- `1`, quando houver conversão num valor inteiro na base decimal e o valor convertido for atribuído à variável.
- **EOF**, quando ocorrer erro ou tentativa de leitura além do final do arquivo (v. **Seção 11.9.2**).

Portanto a instrução **if** do último trecho de programa testa se a variável foi alterada e, se não for esse o caso, solicita ao usuário para introduzir um outro valor inteiro. Antes da nova leitura, porém, se **scanf()** não retornou **EOF**, chama-se a função `LimpaBuffer()` para remover os caracteres que ficaram armazenados no buffer de entrada em consequência dos dados incorretamente digitados pelo usuário. Quando **scanf()** retorna **EOF**, a função `LimpaBuffer()` não é chamada porque não há caracteres a serem removidos do buffer.

A correção apresentada no trecho de programa acima quando introduzida no programa em questão melhora sua qualidade, porque permite que o usuário tenha uma nova chance para corrigir um eventual erro. Mas, essa ainda não é a solução mais adequada, pois não leva em consideração que o usuário pode errar mais de uma vez. Logo, a solução ideal é construir um laço de repetição que só deve encerrar quando o usuário digitar corretamente o dado esperado pelo programa. Desse modo, ele terá incontáveis chances de errar antes de introduzir um dado corretamente. Uma pequena alteração no trecho de programa anterior produz o resultado desejado:

```
printf("\nDigite um inteiro: ");
nValLidos = scanf("%d", &umInt);

while (nValLidos != 1) { /* Nenhum inteiro foi lido */
   if (nValLidos != EOF) {
      LimpaBuffer();
   }
   printf("\a\nValor incorreto.\nDigite um inteiro: ");
   nValLidos = scanf("%d", &umInt);
}
```

Fazendo as devidas alterações, o programa apresentado no início desta seção ficaria assim:

```
#include <stdio.h>

void LimpaBuffer(void)
{
   int c; /* Caractere lido */

   do {
      c = getchar();
   } while (c != '\n' && c != EOF);
}

int main(void)
{
   int umInt, outroInt, nValLidos;

      /* Tenta ler o primeiro valor */
   printf("\nDigite um inteiro: ");
   nValLidos = scanf("%d", &umInt);

      /* Enquanto o usuário não digitar um valor */
      /* considerado correto, o laço não encerra */
   while (nValLidos != 1) { /* Nenhum inteiro foi lido */
         /* Se scanf() retornou EOF não há caracteres a serem removidos */
      if (nValLidos != EOF) {
         LimpaBuffer();
      }

         /* Oferece nova chance para o usuário se redimir */
      printf("\a\nValor incorreto.\nDigite um inteiro: ");
      nValLidos = scanf("%d", &umInt);
   }

      /* Mesmo que scanf() tenha lido o primeiro valor (o que é */
      /* verdade neste ponto), pode haver caracteres restantes  */
      /* no buffer além de '\n', e eles precisam ser removidos  */
   LimpaBuffer();

      /* Tenta ler o segundo valor */
   printf("\nDigite outro inteiro: ");
   nValLidos = scanf("%d", &outroInt);

      /* Enquanto o usuário não digitar um valor */
      /* considerado correto, o laço não encerra */
   while (nValLidos != 1) { /* Nenhum inteiro foi lido */
         /* Se scanf() retornou EOF não há caracteres a serem removidos */
      if (nValLidos != EOF) {
         LimpaBuffer();
      }
```

```
         /* Oferece nova chance para o usuário se redimir */
      printf("\a\nValor incorreto.\nDigite um inteiro: ");
      nValLidos = scanf("%d", &outroInt);
   }

   printf( "\n\n\t>>> %d + %d = %d\n", umInt, outroInt, umInt + outroInt );

   return 0;
}
```

Agora, se você examinar atentamente os dois trechos responsáveis pelas leituras dos dois números inteiros do último programa, notará que eles diferem, substancialmente, apenas pelo fato de o primeiro trecho usar a variável **umInt**, enquanto o outro usa a variável **outroInt**. Portanto essa é uma situação apropriada para o uso de uma função para leitura de inteiros que seja incorporada no programa, como mostrado a seguir.

```
#include <stdio.h>

void LimpaBuffer(void)
{
   int c; /* Caractere lido */

   do {
      c = getchar();
   } while (c != '\n' && c != EOF);
}

int LeInteiro2(void)
{
   int oInteiro, nValoresLidos;

      /* Tenta ler um valor inteiro */
   nValoresLidos = scanf("%d", &oInteiro);

      /* Enquanto o usuário não digitar um  */
      /* valor esperado, o laço não encerra */
   while (nValoresLidos != 1) {/* Nenhum inteiro foi lido */
         /* Se scanf() retornou EOF não há caracteres  */
         /* a serem removidos. Caso contrário, haverá. */
      if (nValoresLidos != EOF) {
         LimpaBuffer();
      }

         /* Faz mais uma tentativa de leitura */
      printf("\a\nValor incorreto.\nDigite um valor inteiro: ");
      nValoresLidos = scanf("%d", &oInteiro);
   }

      /* Deixa o buffer limpo antes de retornar */
   LimpaBuffer();

   return oInteiro;
}

int main(void)
{
   int  umInt, outroInt;

      /* Lê o primeiro valor */
   printf("\nDigite um valor inteiro: ");
   umInt = LeInteiro2();

      /* Lê o segundo valor */
```

```
   printf("\nDigite outro valor inteiro: ");
   outroInt = LeInteiro2();

   printf( "\n\n\t>>> %d + %d = %d\n",
           umInt, outroInt, umInt + outroInt );

   return 0;
}
```

Veja como o novo programa ficou muito mais conciso e fácil de ser entendido com a introdução da função `LeInteiro2()`. Note ainda que essa função foi denominada *LeInteiro2*, em vez de apenas *LeInteiro*, para não ocorrer conflito de identificadores com o nome da função `LeInteiro()` da biblioteca LeituraFacil. Aliás, as funções `LeInteiro2()` e `LeInteiro()` são bem parecidas, mas não são funcionalmente equivalentes (v. Apêndice B).

Agora, observe o seguinte exemplo de execução do último programa apresentado:

```
Digite um valor inteiro: XY123

Valor incorreto.
Digite um valor inteiro: 123XY

Digite outro valor inteiro: 321

        >>> 123 + 321 = 444
```

Nesse exemplo de execução, na primeira tentativa de leitura, o usuário digitou um valor que não pode ser convertido em inteiro na base decimal e o programa reagiu adequadamente, conforme esperado. Na segunda tentativa de leitura, o usuário digitou `123XY`, que também não corresponde a um número inteiro na base decimal. No entanto, nesse último caso, o valor digitado foi aceito pelo programa. Se você entendeu o funcionamento da função **scanf()**, concluirá que esse comportamento do programa também era esperado. Mas, neste ponto, surgem duas questões:

[1] O valor `123XY` deveria ter sido aceito como válido?

[2] Se a resposta à primeira questão for negativa, é possível detectar esse problema e reagir adequadamente?

A resposta para a primeira questão depende do fato de o programador desejar ser rigoroso ou condescendente com o usuário. A função `LeInteiro()` da biblioteca LeituraFacil é simpática ao usuário e aceita esse tipo de entrada como válida. Enfim, a resposta para a primeira questão é uma decisão de projeto de programação.

A resposta para a segunda questão é um pouco mais complicada, pois usando apenas **scanf()** não é possível saber se o usuário digitou caracteres (sem incluir `'\n'`) além daqueles que essa função conseguiu converter. Então, uma solução é reimplementar a função `LimpaBuffer()` de tal modo que ela retorne o número de caracteres que ela lê e descarta, sem levar em consideração o onipresente caractere `'\n'`. Em seguida, a função `LeInteiro2()` também precisa ser reimplementada para incluir essa verificação adicional, como mostra o seguinte programa:

```
#include <stdio.h>

int LimpaBuffer2(void)
{
   int carLido, /* Armazena cada caractere lido */
       nCarLidos = 0; /* Conta o número de caracteres lidos */

      /* Lê e descarta cada caractere lido até    */
      /* encontrar '\n' ou getchar() retornar EOF */
   do {
      carLido = getchar(); /* Lê um caractere */
      ++nCarLidos; /* Mais um caractere foi lido */
   } while ((carLido != '\n') && (carLido != EOF));
```

```
      /* O último caractere lido foi '\n' ou EOF e não deve ser considerado sobra */
   return nCarLidos - 1;
}

int LeInteiro3(void)
{
   int oInteiro, nValoresLidos, resto;

inicio: /* Desvia para cá para fazer uma nova tentativa */
      /* Tenta ler um valor inteiro */
   nValoresLidos = scanf("%d", &oInteiro);

      /* Se scanf() retornou EOF não há caracteres  */
      /* a serem removidos. Caso contrário, haverá. */
   if (nValoresLidos != EOF) {
      resto = LimpaBuffer2();
   }

      /* Enquanto o usuário não digitar um valor esperado, o laço não encerra */
   while (nValoresLidos != 1 || resto) {
         /* Faz mais uma tentativa de leitura */
      printf("\a\nValor incorreto.\nDigite um valor inteiro: ");
      goto inicio;
   }

   return oInteiro;
}

int main(void)
{
   int  umInt, outroInt;

      /* Lê o primeiro valor */
   printf("\nDigite um valor inteiro: ");
   umInt = LeInteiro3();

      /* Lê o segundo valor */
   printf("\nDigite outro valor inteiro: ");
   outroInt = LeInteiro3();

   printf( "\n\t>>> %d + %d = %d\n", umInt, outroInt, umInt + outroInt );

   return 0;
}
```

Se algum purista lhe disser que não se deve nunca usar **goto**, desafie-o a implementar a função `LeInteiro3()` de modo mais elegante e eficiente.

A seguir, um exemplo de execução do último programa:

```
Digite um valor inteiro: XY123

Valor incorreto.
Digite um valor inteiro: 123XY

Valor incorreto.
Digite um valor inteiro: 123

Digite outro valor inteiro: 321

        >>> 123 + 321 = 444
```

Compare o exemplo de execução acima com aquele apresentado no penúltimo exemplo e constate a diferença.

11.9.6 Leitura de Strings via Teclado: scanf(), gets() e fgets()

Rigorosamente falando, nenhuma função é capaz de ler strings introduzidos por meio de teclado, porque é impossível introduzir por essa via o caractere terminal que faz parte de qualquer string. Assim, do mesmo modo que *ler um número* é uma simplificação linguística para *ler caracteres e convertê-los num número*, *ler um string* é uma simplificação para *ler caracteres e convertê-los num string*.

Podem-se ler strings via teclado utilizando a função **scanf()** em conjunto com o especificador de formato **%s**. O parâmetro correspondente a esse especificador deve ser o endereço de um array de caracteres com espaço suficiente para conter os caracteres introduzidos pelo usuário mais o caractere `'\0'`, pois, após a leitura dos caracteres, **scanf()** acrescenta automaticamente o caractere nulo ao array, de modo que o resultado constitua um string.

Considere o seguinte programa como exemplo de leitura de string:

```
#include <stdio.h>

int main(void)
{
   char nome[30];

   printf("\nDigite seu nome (maximo de 29 letras)\n\t> ");
   scanf("%s", nome);

   printf("\nSeu nome e': %s\n", nome);

   return 0;
}
```

A chamada de **scanf()** nesse programa seria capaz de ler, no máximo, 29 caracteres introduzidos pelo usuário do programa. Isto é, o número máximo de caracteres que podem ser lidos é no máximo o tamanho do array menos um, já que se deve reservar um espaço para o caractere terminal de string acrescentado por **scanf()**.

Note no programa anterior que não é necessário o uso do operador **&** precedendo o parâmetro `nome` na chamada de **scanf()**, pois `nome` já é um endereço (v. Seção 8.7).

Um sério problema que acomete a função **scanf()** quando ela é usada com o especificador **%s** é que, por maior que seja o tamanho do array que se usa como parâmetro, não se pode garantir que ele terá espaço suficiente para conter os caracteres introduzidos pelo usuário. Quer dizer, considerando novamente o exemplo anterior, quem garante que o usuário não introduzirá mais de 29 caracteres? Portanto o programador jamais deve usar apenas **%s** como especificador de formato de strings com a função **scanf()**, pois o uso desse especificador pode causar o mau funcionamento do programa por causa de corrupção de memória (v. Seção 8.9.3). Nesse caso, o problema reside no fato de a função **scanf()**, quando utilizada com o especificador **%s**, poder alterar porções de memória que não estão alocadas para o array utilizado como parâmetro.

Os possíveis problemas causados pelo uso do especificador **%s** com **scanf()** podem ser facilmente resolvidos utilizando-se o especificador de formato:

%*ns*

Nesse especificador de formato, ***n*** determina o número máximo de caracteres a serem lidos pela função **scanf()**. Assim, o último programa apresentado pode se tornar seguro substituindo-se sua chamada de **scanf()** por:

```
scanf("%29s", nome);
```

Usando-se o especificador **%*ns***, resolve-se o problema de corrupção de memória que poderia ocorrer numa chamada de **scanf()**, mas essa função ainda apresenta uma desvantagem para leitura de strings, que é o fato

de ela encerrar a leitura quando encontra um espaço em branco no interior de um string. Isso significa que um string contendo caracteres em branco em seu interior não pode ser introduzido usando essa função, como mostra um exemplo de interação com o último programa:

```
Digite seu nome (maximo de 29 letras)
        > Dom Pedro I

Seu nome e': Dom
```

Uma característica interessante de **scanf()** é que ela permite limitar os caracteres que podem fazer parte de um string lido por meio do uso do especificador **%[***caracteres***]**. Esse especificador solicita à função **scanf()** que leia no meio de entrada padrão todos os caracteres entre colchetes e os armazene no endereço representado pelo parâmetro (array) correspondente. A leitura é encerrada quando a função encontra o primeiro caractere que não faz parte do conjunto de caracteres entre colchetes. Esse especificador pode ainda ser usado em conjunto com um inteiro positivo que limita o número de caracteres lidos, como mostra o exemplo a seguir:

```
#include <stdio.h>

int main(void)
{
   char str[5]; /* 5 é um número tirado da cartola... */

   printf("\nDigite uma palavra contendo vogais: ");

      /* São aceitas apenas vogais minúsculas */
   scanf("%4[aeiou]", str);

   printf("Palavra aceita: \"%s\"\n", str);

   return 0;
}
```

Os seguintes exemplos de execução ilustram o funcionamento desse programa:

Exemplo 1:

```
Digite uma palavra contendo vogais: bao
Palavra aceita: ""
```

Análise: Nesse exemplo de execução, o primeiro caractere encontrado por **scanf()** foi `'b'` e, como ele não é vogal minúscula, nenhum caractere foi lido e o string resultante da leitura é vazio.

Exemplo 2:

```
Digite uma palavra contendo vogais: aob
Palavra aceita: "ao"
```

Análise: Nesse caso, os dois primeiros caracteres encontrados foram `'a'` e `'o'`, que, por serem vogais minúsculas, são aceitos. A leitura encerra quando o caractere `'b'` é encontrado.

Exemplo 3:

```
Digite uma palavra contendo vogais: auiaaaiiiaauuooee
Palavra aceita: "auia"
```

Análise: Nesse último caso, todos os caracteres digitados são vogais minúsculas e, em princípio aceitáveis. Mas, a leitura encerra quando o número máximo de caracteres, especificado por meio de **%4[aeiou]**, é lido.

Quando os caracteres entre colchetes são iniciados com o caractere `'^'`, a função **scanf()** considera todos os caracteres que não se encontram entre colchetes. Nesse caso, a leitura é encerrada quando **scanf()** encontra o primeiro caractere que faz parte do conjunto de caracteres entre colchetes.

Uma forma ilusória (ou melhor, delirante) de superar a incapacidade demonstrada por **scanf()** para ler strings com espaços em branco é utilizar a função **gets()**, que é específica para leitura de strings via teclado. Essa função recebe um único parâmetro, que é o endereço de um array de caracteres, e, quando executada, lê e armazena no array os caracteres introduzidos até encontrar o caractere `'\n'` (que representa `[ENTER]`). Então, essa função acrescenta o caractere nulo ao final dos caracteres lidos e armazenados no array citado. O caractere `'\n'` não é armazenado nesse array.

Como exemplo de uso de **gets()**, considere o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   char  nome[30];

   printf("\nDigite seu nome (maximo de 29 letras)\n\t> ");
   gets(nome);

   printf("\nSeu nome e': %s\n", nome);

   return 0;
}
```

A seguir, um exemplo de execução desse último programa:

```
Digite seu nome (maximo de 29 letras)
        > Dom Pedro I

Seu nome e': Dom Pedro I
```

Como no caso da função **scanf()**, o array passado como parâmetro para **gets()** deve ter capacidade suficiente para conter os caracteres digitados pelo usuário mais o caractere `'\0'`, que será acrescentado. Entretanto, diferentemente do que ocorre com a função **scanf()**, a função **gets()** não possui nenhum meio para limitar o número de caracteres lidos. Por exemplo, no seguinte caso de execução:

```
Digite seu nome (maximo de 29 letras)
        > Pedro de Alcantara Francisco Antonio Joao Carlos
Xavier de Paula Miguel Rafael Joaquim Jose Gonzaga Pascoal
Cipriano Serafim de Braganca e Bourbon

Seu nome e': Pedro de Alcantara Francisco Antonio Joao
Carlos Xavier de Paula Miguel Rafael Joaquim Jose Gonzaga
Pascoal Cipriano Serafim de Braganca e Bourbon
```

o último programa é abortado, mesmo que ele consiga escrever o nome digitado pelo usuário.

Em resumo, como não é possível corrigir o grave defeito incorporado na função **gets()**, ela não deve *jamais* ser utilizada.

A função **gets()** é tão maligna que, finalmente, o mais recente padrão de C (ISO C11) decidiu expurgá-la da linguagem. No entanto, infelizmente, ela ainda faz parte da maioria das bibliotecas que acompanham compiladores de C e, pior, tem seu uso recomendado por alguns textos sobre programação em C.

A função **fgets()** não padece do grave problema que acomete **gets()** e 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 11.8).

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 (v. **Seção 8.9.3**). Todavia, nesse último caso, o erro é causado por descuido do programador e não é inerente à função **fgets()** [como ocorre com **gets()**].

As funções **gets()** e **fgets()** compartilham semelhanças e diferenças que são resumidas na **Tabela 11–5**.

fgets()	gets()
Lê caracteres incluindo espaços em branco até encontrar um caractere `'\n'`.	Idem.
Acrescenta o caractere nulo ao final dos caracteres lidos.	Idem.
Inclui o caractere `'\n'` no string, se ele for lido.	Não inclui o caractere `'\n'` no string.
Permite limitar o número de caracteres lidos.	Não permite limitar o número de caracteres lidos.
Permite especificar o meio de entrada.	Permite leitura apenas via teclado.
Seu uso é seguro (dependendo, é claro, do programador).	Até a homologação do padrão C11, era o grande satã da biblioteca padrão de C, pois não há uso seguro para essa função.
Faz parte de todos os padrões ISO da linguagem C.	Foi excluída da linguagem C pelo padrão C11 (felizmente, ufa!).

TABELA 11–5: COMPARAÇÃO ENTRE fgets() E gets()

A principal diferença entre **gets()** e **fgets()** é de natureza pragmática:

Recomendação *Definitivamente, a função gets() nunca deve ser usada!*

Usando-se **fgets()**, pode-se obter uma versão mais adequada dos programas apresentados anteriormente nesta seção:

```
#include <stdio.h>

int main(void)
{
   char nome[30], *p;

   printf("\nDigite seu nome (maximo de 29 letras)\n\t> ");
   p = fgets(nome, 30, stdin);

   if (!p) {
      printf("\nErro de leitura\n");
      return 1;
   }

   printf("\nSeu nome e': %s\n", nome);

   return 0;
}
```

Exemplo de execução do último programa:

```
Digite seu nome (maximo de 29 letras)
        > Pedro de Alcantara Francisco Antonio Joao Carlos
Xavier de Paula Miguel Rafael Joaquim Jose Gonzaga Pascoal
Cipriano Serafim de Braganca e Bourbon
Seu nome e': Pedro de Alcantara Francisco
```

Apesar de o último programa não estar preparado para ler sequer a metade do nome completo de Dom Pedro I, ele não foi abortado como o penúltimo programa. Ademais, o ponteiro **p** foi usado para testar se o valor retornado por **fgets()** indica a ocorrência de erro de leitura ou tentativa de leitura além do final do arquivo (v. Seção 11.6).

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 11–7, na qual `x` representa cada caractere introduzido pelo usuário antes de ele digitar `[ENTER]`.

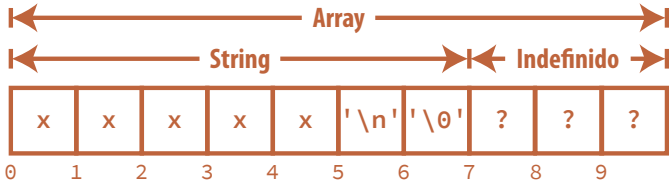

Figura 11–7: Função fgets() Lendo Quebra de Linha

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()** (v. Seção 9.5.8), como é mostrado abaixo:

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

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

11.10 Acessos Sequencial e Direto

Uma vez que um arquivo tenha sido aberto conforme foi descrito na **Seção 11.4**, 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 11.11.1**.
  - 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 11.11.2**.
  - 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 11.11.3**.
  - **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, não é discutido em detalhes neste livro.
- **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 11.12**.

A **Tabela 11–6** e a **Tabela 11–7** têm o intuito de servirem como rápida referência para ajudarem 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

Tabela 11–6: Processamento Sequencial (Resumo)

Processamento	Funções Tipicamente Usadas	Conveniente para arquivo...
Formatado	❒ **fscanf()** (leitura) ❒ **fprintf()** (escrita)	Texto (apenas)

Tabela 11–6: Processamento Sequencial (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 11–7: Processamento com Acesso Direto (Resumo)

11.11 Processamento Sequencial de Arquivos

Esta seção descreve em detalhes as categorias de processamento sequencial.

11.11.1 Processamento Sequencial por Bytes

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 11–8.

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

Tabela 11–8: Protótipos de Funções para Processamento de Caracteres (Bytes)

Na Seção 11.15, serão apresentados alguns exemplos de uso prático das funções **fgetc()** e **fputc()**.

11.11.2 Processamento Sequencial por Linhas

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()**, sendo que a função **fgets()** já foi explorada na Seção 11.9.6. Assim, apenas **fputs()** precisará ser discutida aqui.

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 (v. Seção 9.5.2).

A Seção 11.15 apresentará exemplos de uso prático das funções **fgets**() e **fputs**().

11.11.3 Processamento Sequencial por Blocos

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 (v. Seção 12.3).
- `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 (v. Seção 12.3).
- `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, como aqueles descritos no **Capítulo 8**. 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);
```

A seguir, será apresentado um exemplo simples de uso das funções **fread()** e **fwrite()**.

```
#include <stdio.h>

int main(void)
{
   FILE *stream;
   int arInt[] = {1, 2, 3, 4, 5},
       arIntLido[sizeof(arInt)/sizeof(arInt[0])],
       i;

         /* Abre arquivo para escrita em modo binário */
   if ( !(stream = fopen("Teste", "wb")) ) {
      fprintf(stderr, "\nArquivo nao pode ser aberto\n");
      return 1;
   }

         /* Escreve o array arInt[] no arquivo */
   fwrite( arInt, sizeof(arInt[0]), sizeof(arInt)/sizeof(arInt[0]), stream );

      /* Verifica se ocorreu erro de escrita */
   if (ferror(stream)) {
      printf("\nOcorreu erro de escrita de arquivo\n");
      return 1;
   }

      /* Tenta fechar arquivo */
   if (fclose(stream)) {
      fprintf(stderr, "\nArquivo nao pode ser fechado\n");
      return 1;
   }
      /* Abre o arquivo para leitura em modo binário */
   if ( !(stream = fopen("Teste", "rb")) ) {
      fprintf(stderr, "\nArquivo nao pode ser reaberto\n");
      return 1;
   }
```

```
   /* Lê array no arquivo e armazena-o em arIntLido[] */
fread( arIntLido, sizeof(arIntLido[0]),
       sizeof(arIntLido)/sizeof(arIntLido[0]), stream );

   /* Verifica se ocorreu erro de leitura */
if (ferror(stream)) {
   printf("\nOcorreu erro de leitura de arquivo\n");
   return 1;
}
printf("\nArray lido no arquivo: { ");

   /* Exibe na tela os elementos do array lido */
for(i = 0; i < sizeof(arIntLido)/sizeof(arIntLido[0]); ++i) {
   printf("%d, ", arIntLido[i]);
}

printf("%d }\n", arIntLido[i]); /* Escreve o último elemento */

fclose(stream);

return 0;
}
```

O que esse programa faz é escrever um array de elementos do tipo **int** num arquivo binário e, em seguida, ler o conteúdo desse arquivo e apresentá-lo na tela.

11.11.4 Algoritmo Geral para Leitura Sequencial de Arquivos

Em geral, leitura sequencial num arquivo pode ser realizada seguindo-se o algoritmo delineado na **Figura 11–8**, que foi escrito com uma mistura de linguagem algorítmica (v. **Capítulo 2**) e português.

ALGORITMO LEITURASEQUENCIALDEARQUIVO

1. Enquanto (verdadeiro) faça
   1.1 Leia uma partição do arquivo
2. Se houve tentativa de leitura além do final do arquivo
   2.1 Pare
3. Se ocorreu erro de leitura
   3.1 Pare
4. Processe a partição lida
5. Se ocorreu erro de leitura
   5.1 Informe que a leitura foi mal sucedida

FIGURA 11–8: ALGORITMO DE LEITURA SEQUENCIAL DE ARQUIVOS

Comentários sobre o algoritmo e sua implementação em C:

- Na implementação do laço enquanto-faça 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()** (v. **Seção 11.11.1**), **fgets()** (v. **Seção 11.11.2**) ou **fread()** (v. **Seção 11.11.3**).
- Tentativa de leitura além do final do arquivo é checada chamando-se **feof()** (v. **Seção 11.6**).

- Ocorrência erro de leitura deve ser verificada chamando-se **ferror()** (v. Seção 11.6).
- *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 na Seção 11.11.3 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.

11.12 Acesso Direto a Arquivos

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** e serão discutidas na Seção 11.12.1.
2. Executar a operação de leitura ou escrita desejada. Leitura e escrita com acesso direto serão discutidas na Seção 11.12.2.

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**.

11.12.1 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 11–9.

Função	Descrição sumária
fseek()	Move o indicador de posição do arquivo para um local especificado por seus parâmetros
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 11–9: Funções de Posicionamento Utilizadas em Acesso Direto

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

```
int fseek(FILE  *stream, int  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 11–10**.

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

Tabela 11–10: 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 do 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 **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 ou `-1`, se ocorrer algum erro. O protótipo de **ftell()** é:

```
int ftell(FILE *arquivo)
```

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 (v. exemplo na Seção 11.15.7).

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()**.

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.

```
#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 11.4.3**), 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:

Recomendação

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.

11.12.2 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.

11.13 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 do arquivo, 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 um desiderato, 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 */
   ...
}
```

11.14 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. Entretanto, nesse caso, 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:

Recomendação ***Se uma operação sobre um arquivo que pode ser mal sucedida não for testada, certamente, ela será mal sucedida.***

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

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

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

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

Em processamento de arquivos, quase todas as chamadas de função do módulo stdio podem ser mal sucedidas (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 11–11**.

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 11.5**), 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()**	Cheque 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()**	Cheque se o retorno da função é diferente de zero (ou **EOF**)

Tabela 11–11: Proteção contra a Lei de Murphy de Processamento de Arquivos

Em qualquer operação enumerada na **Tabela 11–11**, 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.

11.15 Exemplos de Programação

Nesta seção e no próximo capítulo, serão apresentados vários exemplos que irão processar os mesmos dados de maneiras diversas. Os dados constituem uma turma escolar surreal formada pelo rei Henrique VIII e suas seis esposas. O minúsculo banco de dados formado por membros da dinastia Tudor é armazenado num arquivo de texto, denominado `Tudor.txt` e apresenta o seguinte conteúdo exibido na Tabela 11–12.

Henrique VIII	1029	9.5	9.0
Catarina Aragon	1014	5.5	6.5
Ana Bolena	1012	7.8	8.0
Joana Seymour	1017	7.7	8.7
Ana de Cleves	1022	4.5	6.0
Catarina Howard	1340	6.0	7.7
Catarina Parr	1440	4.0	6.0

Tabela 11–12: Conteúdo do Arquivo Tudor.txt

Para tornar os exemplos mais palpáveis, supõe-se que o conteúdo do arquivo representa informações referentes a uma turma fictícia de uma disciplina imaginária. Cada linha desse arquivo (neste contexto, usualmente, denominada **registro**) é dividida em quatro partes, tipicamente, denominadas **campos**, separadas por caracteres de tabulação. Outros tipos de separadores, como vírgula ou ponto e vírgula, podem ser usados. Tabulação foi escolhida como separador de campos porque facilita a visualização do conteúdo do arquivo. Por outro lado, o uso de tabulação pode causar a impressão de que as linhas são do mesmo tamanho. Mas, observando-se atentamente, verifica-se que elas têm tamanhos variados.

Os campos de cada registro são interpretados como mostra a Tabela 11–13.

Nome do Campo	Interpretação	Tipo de Dado
`nome`	Nome do aluno	`char[21]`
`matr`	Matrícula do aluno	`char[5]`
`n1`	Primeira nota	`double`
`n2`	Segunda nota	`double`

Tabela 11–13: Campos do Arquivo Tudor.bin

Considerando as interpretações de campos apresentadas na Tabela 11–13, na primeira linha da Tabela 11–12, por exemplo, o nome do aluno é `Henrique VIII`, sua matrícula é `1029`, sua primeira nota é `9.5` e sua segunda nota é `9.0`. O arquivo `Tudor.txt` é facilmente encontrado no site dedicado ao livro na internet (*www.ulysseso.com/ip*).

11.15.1 Tamanho de um Texto Digitado via Teclado

Problema: Escreva um programa que permite que o usuário digite tantos caracteres quanto ele deseje e, ao final, informe-o sobre quantos caracteres foram digitados. (**NB**: Os caracteres em si não são armazenados pelo programa; eles são apenas contados.)

Solução:

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

```
/****
 *
 * TamanhoTexto(): Lê e conta caracteres introduzidos via teclado
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Número de caracteres digitados
 *
 * Observação: O caractere '\n' não é levado em consideração na contagem, pois um
 *             usuário comum teria dificuldade em entender que [ENTER] é um caractere
 *
 ****/
int TamanhoTexto(void)
{
   int c,        /* Armazena cada caractere lido */
       nCar = 0; /* Armazena o número de caracteres lidos */

   c = getchar(); /* Lê o primeiro caractere */

      /* Enquanto o caractere '\n' não tiver sido lido e   */
      /* getchar() não retornar EOF, lê e conta caracteres */
   while (c != '\n' && c != EOF) {
      ++nCar;
      c = getchar();
   }

         /* Se desejar incluir '\n' na contagem, basta incrementar */
         /* a variável nCar neste ponto antes de retornar          */

   return nCar;
}
/****
 * main(): Determina quantos caracteres o usuário digita
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
      /* Apresenta o programa */
   printf( "\n\t>>> Este programa permite que o usuario digite"
           "\n\t>>> quantos caracteres desejar e informa o"
           "\n\t>>> numero de caracteres digitados.\n" );

  printf("\nDigite um texto:\n> ");

     /* Chama a função TamanhoTexto() para ler o texto */
     /* e informa quantos caracteres foram digitados   */
  printf( "\n\t>>> O texto digitado possui %d caracteres\n", TamanhoTexto() );

  return 0;
}
```

Análise: Esse programa pode não ter muita utilidade prática, mas é fácil de entender. Leia atentamente os comentários que acompanham o programa e, se não conseguir entendê-lo, releia a Seção 11.9.

Exemplo de execução do programa:

```
        >>> Este programa permite que o usuario digite
        >>> quantos caracteres desejar e informa o
        >>> numero de caracteres digitados.

Digite um texto:
> Pedro de Alcantara Francisco Antonio Joao Carlos Xavier de Paula Miguel Rafael
Joaquim Jose Gonzaga Pascoal Cipriano Serafim de Braganca e Bourbon

        >>> O texto digitado possui 146 caracteres
```

11.15.2 Filtro de Numeração de Linhas

Preâmbulo: No contexto de processamento de arquivos, **filtro** é um programa que executa alguma operação sobre os dados de um arquivo após eles serem lidos ou antes de eles serem escritos no arquivo.

Problema: Escreva um filtro que numera as linhas de um arquivo criado via teclado e apresenta-as na tela.

Solução:

```
#include <stdio.h>

/****
 * main(): Implementa um filtro que acrescenta numeração de
 *         linhas para arquivos de texto
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int linha = 0, /* Contadora de linhas */
       c, /* Armazena cada caractere lido */
       novaLinha = 1; /* Indica se o próximo caractere é o início de uma linha nova */

      /* Enquanto o final do arquivo associado a stdin não for atingido, */
      /* lê caracteres e numera cada nova linha encontrada               */
   while ( (c = getchar()) != EOF ) {
         /* Uma linha nova é a primeira ou aquela que começa logo após '\n' */
      if (novaLinha) { /* Início de nova linha */
         printf("%.4d: ", ++linha); /* Escreve numeração */
         novaLinha = 0; /* Desliga indicação de linha nova */
      }

      putchar(c); /* Escreve o caractere lido em stdout */

         /* Se o último caractere lido foi '\n', o  */
         /* próximo caractere inicia uma nova linha */
      if (c == '\n') {
         novaLinha = 1; /* Indica início de nova linha */
      }
   }
   return 0;
}
```

Análise: O programa acima requer conhecimento de execução de programas de linha de comando mais profundo do que aquele que usuários comuns costumam possuir. Em particular, para tirar bom proveito desse programa é necessário o conhecimento de canalização (*pipe*) e redirecionamento de entrada e saída. Isto é, supondo que o nome do programa executável seja `numera` e que o nome do arquivo a ser filtrado seja `Arq.txt`, ele deve ser executado como descrito a seguir.

Em sistemas da família DOS/Windows:

```
type Arq.txt | numera | more
```

Em sistemas da família Unix:

```
cat Arq.txt | numera | more
```

Em sistemas de qualquer das famílias mencionadas:

```
numera < Arq.txt > ArqNumerado.txt
```

Nesse último caso, a saída do programa será escrita no arquivo **ArqNumerado.txt**. Mas, se desejar que o resultado seja exibido na tela, basta remover a parte do comando que segue o nome do arquivo de entrada, como mostrado no exemplo de execução a seguir:

```
C:\Programas> numera < AnedotaBulgara.txt
0001: Anedota Bulgara
0002: Carlos Drummond de Andrade
0003:
0004: Era uma vez um czar naturalista
0005: que cacava homens.
0006: Quando lhe disseram que tambem se
0007: cacam borboletas e andorinhas,
0008: ficou muito espantado
0009: e achou uma barbaridade
```

No exemplo de execução, o nome do arquivo usado como entrada, via redirecionamento, é denominado **AnedotaBulgara.txt**. Esse arquivo contém uma poesia irônica do ilustre poeta mineiro Carlos Drummond de Andrade, cuja estátua os cariocas adoram depredar...

11.15.3 Alinhando na Tela Valores Inteiros Lidos via Teclado

Problema: Escreva um programa que alinha na tela valores inteiros lidos via teclado.

Solução:

```
#include <stdio.h>   /* Entrada e saída          */
#include <string.h>  /* Processamento de strings */
#include "leitura.h" /* LeituraFacil             */

   /* Tamanho do array que armazenará os strings */
#define TAM_ARRAY  50

typedef enum {ESQUERDA, CENTRO, DIREITA} tAlinhamento; /* Tipos de alinhamento */

/****
 * main(): Alinha na tela um valor inteiro lido via teclado
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   char         str[TAM_ARRAY], /* Armazenará o string a ser escrito na tela */
                ar[TAM_ARRAY]; /* Armazenará o string antes de ser alinhado */
   int          x, /* Inteiro que será lido e apresentado */
                op, /* Opção escolhida pelo usuário */
                preenche; /* Caractere de preenchimento */
```

```
   tAlinhamento alinhamento; /* Opção de alinhamento */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa alinha o numero inteiro"
           "\n\t>>> introduzido de acordo com a sua preferencia.\n" );

      /* Lê o valor inteiro */
   printf("\n\t>>> Digite um valor inteiro: " );
   x = LeInteiro();

      /* Lê a opção de alinhamento do usuário */
   printf( "\n\t>>> Opcoes de alinhamento: Esquerda (E),"
           "\n\t>>> Direita (D) e Centro (C). "
           "\n\t>>> Qual e' a sua opcao? ");
   op = LeOpcao("eEdDcC");

   if (op == 'e' || op == 'E') {
      alinhamento = ESQUERDA;
   } else if (op == 'd' || op == 'D') {
      alinhamento = DIREITA;
   } else {
      alinhamento = CENTRO;
   }
      /* Lê o caractere de preenchimento de espaços em branco */
   printf("\n\t>>> Caractere de preenchimento: ");
   preenche = LeCaractere();

      /* Escreve no array o valor inteiro lido */
   sprintf(ar, "%d", x);

      /* Alinha o string antes de apresentá-lo na tela */
   AlinhaString(str, TAM_ARRAY, preenche, alinhamento, ar);
      /* Apresenta o string alinhado na tela */
   printf( "\n%s\n", str );

   return 0;
}
```

Análise:

- O programa usa a função **`AlinhaString()`**, apresentada na **Seção 10.10.3**. Essa função alinha o string recebido como parâmetro no array que o armazena, de acordo com o tamanho desse array.
- O tipo de operação efetuada por esse programa é denominada **formatação em memória**, porque os dados que serão apresentados na tela são previamente formatados em memória.

Exemplo de execução do programa:

```
        >>> Este programa alinha o numero inteiro
        >>> de acordo com a sua preferencia.

        >>> Digite um valor inteiro: 12345

        >>> Opcoes de alinhamento: Esquerda (E),
        >>> Direita (D) e Centro (C).
        >>> Qual e' a sua opcao? c

        >>> Caractere de preenchimento: +

++++++++++++++++++++12345++++++++++++++++++++
```

11.15.4 Abrindo Arquivos Compulsoriamente

Problema: (a) Escreva uma função que impele o usuário a introduzir um nome de arquivo que possa ser aberto. (b) Escreva um programa que teste a função descrita no item (a).

Solução de (a):

```
/****
 * AbreArquivo(): Lê o nome de um arquivo e abre-o
 *
 * Parâmetros:
 *      prompt (entrada) - prompt que será apresentado ao usuário
 *      modo (entrada) - modo de abertura do arquivo
 *
 * Retorno: Stream associado ao arquivo aberto
 * ****/
FILE *AbreArquivo(const char *prompt, const char *modo)
{
      /* Array que armazenará o nome do arquivo */
   char  nomeArq[FILENAME_MAX];
   FILE *stream; /* Stream associado ao arquivo aberto */

      /* O laço só encerra quando o arquivo indicado pelo usuário for aberto */
   while (1) {
         /* Apresenta o prompt e lê o nome do arquivo */
      printf("%s", prompt);
      LeString(nomeArq, FILENAME_MAX);

         /* Tenta abrir o arquivo */
      stream = fopen(nomeArq, modo);

         /* Se o arquivo foi aberto, o serviço está completo */
      if (stream) {
         break; /* Encerra o laço */
      }


        /* Se o arquivo não foi aberto, apresenta */
         /* mensagem correspondente ao usuário */
      printf( "\a\n\t>>> Nao foi possivel abrir o arquivo \"%s\"\n", nomeArq );
   }

   return stream;
}
```

Análise: A função acima usa a constante **FILENAME_MAX**, discutida na Seção 11.4.1, para dimensionar o tamanho do array que armazenará o nome de arquivo introduzido pelo usuário.

Solução de (b):

```
/****
 * main(): Obriga o usuário a introduzir um nome de arquivo que possa ser aberto
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   FILE *stream;
```

```
   printf( "\n\t>>> Este programa requer que voce introduza"
           "\n\t>>> um nome de arquivo que possa ser aberto\n" );

      /* Abre o arquivo indicado pelo usuário */
   stream = AbreArquivo( "\n\t>>> Digite o nome do arquivo: ", "rb" );

      /* Informa o usuário que, agora, ele está liberado */
   printf( "\n\t>>> Abertura do arquivo foi bem sucedida\n" );

      /* Fecha o arquivo, já que não há mais nada a fazer com ele */
   fclose(stream);

   return 0;
}
```

Complemento do programa:

```
#include <stdio.h>    /* Entrada e saída */
#include "leitura.h"  /* LeituraFacil    */
```

Exemplo de execução do programa:

```
        >>> Este programa requer que voce introduza
        >>> um nome de arquivo que possa ser aberto

        >>> Digite o nome do arquivo: Inexistente

        >>> Nao foi possivel abrir o arquivo "Inexistente"

        >>> Digite o nome do arquivo: Tudor.txt

        >>> Abertura do arquivo foi bem sucedida
```

11.15.5 Número de Linhas de um Arquivo de Texto

Problema: Escreva um programa que recebe um nome de arquivo de texto como entrada via linha de comando e informa quantas linhas o arquivo possui.

Solução:

```
#include <stdio.h> /* Entrada e saída */

/****
 * NumeroDeLinhas(): Conta o número de linhas de um arquivo de texto
 *
 * Parâmetros: stream (entrada): stream associado ao arquivo
 *
 * Retorno: O número de linhas do arquivo ou  um valor negativo se ocorrer erro
 *
 * Observação: O arquivo deve ter sido aberto em modo de texto que permite leitura
 ****/
int NumeroDeLinhas(FILE *stream)
{
   int    c, /* Armazena temporariamente um caractere */
          linhas = 0; /* Armazena o total de linhas */

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

      /* Lê cada caractere do stream e conta quantos caracteres */
      /* '\n' são encontrados. O laço while encerra quando o    */
      /* final do arquivo é atingido ou ocorre erro de leitura. */
   while (1) {
      c = fgetc(stream); /* Lê um caractere no stream */
```

```
         /* Se o final do arquivo foi atingido    */
         /* ou ocorreu algum erro, encerra o laço */
      if ( feof(stream) || ferror(stream) )
         break;

      if (c == '\n') {
         linhas++; /* Mais uma linha encontrada */
      }
   }

      /* Se ocorreu erro retorna -1. Caso contrário, retorna o número de linhas */
   return ferror(stream) ? -1 : linhas;
}

/****
 * main(): Conta o número de linhas de um arquivo de texto
 *
 * Parâmetros:
 *     argc (entrada) - Número de argumentos de linha de comando
 *     argv (entrada) - Array de strings presentes na linha de
 *                      comando quando o programa é executado
 *
 * Retorno: Zero, se não ocorrer nenhum erro.
 *          Um valor diferente de zero em caso contrário.
 ****/
int main(int argc, char* argv[])
{
   int   nLinhas;
   FILE *stream;

      /* Verifica se o programa foi invocado corretamente */
   if (argc != 2) { /* Não foi */
         /* Faltou o usuário informar o nome do arquivo */
      printf( "\n\t>>> Este programa deve ser usado assim:"
              "\n\t\t%s nome-do-arquivo\n", argv[0] );
      return 1;
   }

      /* Tenta abrir o arquivo para leitura em modo texto */
   stream = fopen(argv[1], "r");

      /* Se o arquivo não foi aberto, */
      /* não é possível continuar     */
   if (!stream) {
      printf( "\nNao foi possivel abrir o arquivo %s\n", argv[1] );
      return 1;
   }

      /* Determina o número de linhas do arquivo */
   nLinhas = NumeroDeLinhas(stream);

      /* Verifica se ocorreu erro durante a operação. Se */
      /* for o caso, informa o usuário e aborta programa */
   if (nLinhas < 0) {
      printf( "\n\t>>> Ocorreu erro ao tentar calcular o numero "
              "\n\t>>> de linhas do arquivo \"%s\"\n", argv[1] );
      fclose(stream); /* Fecha arquivo antes de partir */
      return 1;
   }

      /* Apresenta o número e linhas do arquivo */
```

```
   printf( "\n\t>>> Numero de linhas do arquivo \"%s\": %d\n", argv[1], nLinhas );

   fclose(stream); /* Fecha arquivo */

   return 0;
}
```

Exemplo de execução do programa: O nome do programa executável é linhas:

```
C:\Programas>linhas Tudor.txt
        >>> Numero de linhas do arquivo "Tudor.txt": 7
```

11.15.6 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):

```
/****
 *
 * CopiaArquivo(): Copia conteúdo de um arquivo byte a byte
 *
 * Parâmetros:
 *      streamEntrada (entrada) - stream associado ao arquivo que será copiado
 *      streamSaida (entrada) - stream associado ao arquivo que receberá a cópia
 *
 * Retorno: 0, se não ocorrer erro; 1, em caso contrário
 *
 * Observação: O stream que será lido deve estar aberto em modo que permita leitura e
 *             o stream que será escrito deve estar aberto em 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 */
      }

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

         /* 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 pelas aberturas. */
   /*************************************************************/

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

Análise:

- 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 11.5.

Solução de (b):

```
/****
 * main(): Copia o conteúdo de um arquivo para outro byte a byte
 *
 * Parâmetros:
 *     argc (entrada) - Número de argumentos de linha de comando
 *     argv (entrada) - Array de strings presentes na linha de
 *                      comando quando o programa é executado
 *
 * Retorno: Zero, se não ocorrer nenhum erro.
 *          Um valor diferente de zero em caso contrário.
 *
 * Observações:
 *     1. Os nomes dos arquivos devem acompanhar o nome na linha de comando
 *     2. O nome do arquivo de entrada é o primeiro e o nome do
 *        arquivo de saída é o segundo argumento de linha de comando
 ****/
int main(int argc, char *argv[])
{
   FILE *streamEntrada, /* Stream de entrada */
        *streamSaida;   /* Streams 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:"
              "\n\t> %s arquivo-a-ser-copiado "
              "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;
   }
```

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

   /* 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. Tenta abrir arquivo de saída.                            */
streamSaida = fopen(argv[2], "wb");

   /* 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;
}

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

   /* Os arquivos não precisam mais estar abertos */
fclose(streamEntrada);
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>>> Nao foi possivel copiar o arquivo %s\n", argv[1] );
   return 1; /* Operação falhou */
}
return 0;
}
```

Os seguintes cabeçalhos precisam ser incluídos para completar o programa:

```
#include <stdio.h>  /* Entrada e saída */
#include <string.h> /* Função strcmp() */
```

Exemplos de execução do programa (o nome do programa executável é `copia`):

Exemplo 1:

```
C:\Programas>copia Tudor.txt Tudor.txt
Os nomes dos arquivos nao podem ser iguais
```

Exemplo 2:

```
C:\Programas>copia Tudor.txt TudorBK.txt
        >>> O arquivo Tudor.txt foi copiado em TudorBK.txt
```

11.15.7 Calculando o Tamanho de um Arquivo

Problema: (a) Escreva uma função que recebe um stream como parâmetro e calcula o tamanho em bytes do arquivo associado ao stream. A posição corrente do indicador de posição do arquivo deve ser preservada. (b) Escreva um programa que recebe um nome de arquivo como argumento de linha de comando e calcula seu tamanho usando a função solicitada no item (a). O tamanho do arquivo

deve ser calculado em duas situações: (1) com o arquivo aberto em modo binário e (2) com o arquivo aberto em modo de texto.

Solução de (a):

```
/****
 * TamanhoDeArquivo(): Calcula o tamanho de um arquivo lendo cada
 *                     byte que o constitui e contando quantos bytes são lidos
 *
 * Parâmetros:
 *     stream (entrada) - stream binário associado ao arquivo cujo
 *                        tamanho será calculado
 *
 * Retorno: -1, se ocorrer algum erro. O tamanho do arquivo, em caso contrário.
 *
 * Observações:
 *     1. O arquivo deve estar aberto em modo binário que permite leitura
 *     2. A posição atual do indicador de posição do arquivo é preservada
 *     3. O arquivo deve permitir acesso direto
 ****/
int TamanhoDeArquivo(FILE *stream)
{
   int posicaoAtual, tamanho = 0;

      /* Guarda a posição atual do indicador de posição */
      /* do arquivo para que ele possa ser restaurado   */
   posicaoAtual = ftell(stream);

      /* Verifica se ftell() foi bem sucedida */
   if (posicaoAtual < 0) {
      return -1; /* A chamada de ftell() falhou */
   }

      /* Assegura leitura a partir do início do arquivo */
   if ( fseek(stream, 0, SEEK_SET) ) {
         /* O movimento do indicador de posição não foi   */
         /* possível. Nesse caso, não precisa restaurá-lo */
      return -1;
   }

      /* O laço encerra quando o final do arquivo for */
      /* atingido ou se ocorrer erro de leitura       */
   while (1){
      fgetc(stream); /* Lê e descarta o próximo byte */

         /* Checa se o arquivo já foi inteiramente lido */
      if (feof(stream)) {
         break; /* Encerra laço */
      }

         /* Verifica se ocorreu erro de leitura */
      if (ferror(stream)) {
            /* Restaura a posição original do indicador */
            /* de posição do arquivo  antes de retornar */
         fseek(stream, posicaoAtual, SEEK_SET);
         return -1; /* Ocorreu erro de leitura */
      }

      ++tamanho; /* Mais um byte foi lido */
   }
```

```
      /* Restaura a posição original do indicador de posição */
   if ( fseek(stream, posicaoAtual, SEEK_SET) ) {
      return -1; /* O movimento não foi possível */
   }

   return tamanho;
}
```

Análise:

- A função `TamanhoDeArquivo()` calcula o tamanho de um arquivo lendo cada byte do arquivo e contando quantos bytes são lidos.
- Além de calcular o tamanho de um arquivo, essa função preserva o valor do indicador de posição do arquivo no instante em que ela é chamada. Ela realiza essa tarefa guardando a posição atual do indicador de posição chamando a função **ftell()** antes de movê-lo para o início do arquivo. Então, antes de retornar, ela restaura a o valor do indicador de posição por meio de uma chamada de **fseek()**.
- O problema com a função `TamanhoDeArquivo()` é que ela só funciona se receber como parâmetro um stream binário (v. exemplos de execução apresentados adiante), mas, como a função não tem como determinar se isso realmente ocorre, não há como garantir que o resultado retornado pela função é confiável. Isso ocorre porque uma função que recebe como parâmetro um stream (i.e., uma parâmetro do tipo **FILE** *) não tem como saber, de modo portável, qual é o modo de abertura do arquivo associado ao stream.

Solução de (b):

```
/****
 * main(): Determina o tamanho em bytes do arquivo cujo nome é
 *         introduzido via linha de comando
 *
 * Parâmetros:
 *     argc (entrada) - Número de argumentos de linha de comando
 *     argv (entrada) - Array de strings presentes na linha de
 *                      comando quando o programa é executado
 *
 * Retorno: Zero, se não ocorrer nenhum erro.
 *          Um valor diferente de zero em caso contrário.
 ****/
int main(int argc, char *argv[])
{
   FILE  *stream;
   int    tamArquivo;

      /* Verifica se o programa foi invocado corretamente */
   if (argc != 2) { /* Não foi */
      printf( "\n\t>>> Este programa deve ser usado assim:"
              "\n\t>>> %s nome-do-arquivo\n", argv[0] );
      return 1; /* Usuário não sabe usar o programa */
   }

      /* Tenta abrir o arquivo para leitura em modo */
      /* binário e checa se o arquivo foi aberto    */
   if (!(stream = fopen(argv[1], "rb"))) {
      printf("\n\t> Impossivel abrir o arquivo %s\n", argv[1]);
      return 1; /* Erro de abertura de arquivo */
   }

   tamArquivo = TamanhoDeArquivo(stream); /* Calcula o tamanho do arquivo */
```

```
   /* Apresenta o resultado */
   if (tamArquivo < 0) {
      printf( "\n\t> Nao foi possivel determinar o tamanho "
              "de \"%s\" em modo binario", argv[1] );
   } else {
      printf( "\n\t> Tamanho de \"%s\" em modo binario: "
              "%d bytes\n", argv[1], tamArquivo );
   }

   FechaArquivo(stream, argv[1]); /* Fecha o arquivo */

      /* Tenta abrir o arquivo para leitura em modo */
      /* de texto e checa se o arquivo foi aberto   */
   if (!(stream = fopen(argv[1], "r"))) {
      printf("\n\t> Impossivel abrir o arquivo %s\n", argv[1]);
      return 1; /* Erro de abertura de arquivo */
   }

      /* Calcula o tamanho do arquivo */
   tamArquivo = TamanhoDeArquivo(stream);

      /* Apresenta o resultado */
   if (tamArquivo < 0) {
      printf( "\t> Nao foi possivel determinar o tamanho de"
              " \"%s\" em modo de texto", argv[1] );
   } else {
      printf( "\t> Tamanho de \"%s\" em modo de texto: "
              "%d bytes\n", argv[1], tamArquivo );
   }
      /* Aqui, não é preciso usar FechaArquivo(),  */
      /* pois o programa será encerrado em seguida */
   fclose(stream);

   return 0;
}
```

Análise:

- A função **main()** chama a função `FechaArquivo()`, definida na Seção 11.5, para fechar o arquivo antes de abri-lo novamente em modo de texto. No segundo fechamento desse arquivo, essa função não precisa ser chamada para testar se a operação foi bem sucedida, porque, logo em seguida, o programa será encerrado de qualquer maneira.
- O problema com essa função **main()** é que ela não deveria nunca chamar uma função que se propõe a calcular o tamanho de um arquivo abrindo-o em modo texto, como mostram os exemplos de execução abaixo.

Complemento do programa:

```
#include <stdio.h>  /* Entrada e saída */
#include <stdlib.h> /* Função exit()   */
```

Exemplos de execução do programa (o nome do programa executável é `tamanho`):

Exemplo 1:

```
C:\Programas> tamanho Tudor.txt

        > Tamanho de "Tudor.txt" em modo binario: 197 bytes
        > Tamanho de "Tudor.txt" em modo de texto: 190 bytes
```

Exemplo 2:

```
C:\Programas> tamanho Tudor.bin
        > Tamanho de "Tudor.bin" em modo binario: 336 bytes
        > Tamanho de "Tudor.bin" em modo de texto: 94 bytes
```

Análise: Como mostram os exemplos de execução, a função `TamanhoDeArquivo()` funciona adequadamente apenas quando é chamada tendo como parâmetro um stream binário, quer o arquivo associado ao stream seja binário, quer ele seja de texto.

11.15.8 Comparando Dois Arquivos

Problema: Escreva um programa que recebe dois nomes de arquivos como argumentos de linha de comando e verifica se eles são iguais ou não. Além disso, se os arquivos forem diferentes, o programa deve informar o índice do byte em que ocorre a primeira diferença entre os arquivos. A indexação dos bytes dos arquivos deve iniciar em `1`.

Solução:

```
#include <stdio.h>   /* Entrada e saída */
#include <stdlib.h>  /* Função exit()   */

/****
 * main(): Compara dois arquivos byte a byte. Os nomes dos
 *         arquivos são argumentos de linha de comando.
 *
 * Parâmetros:
 *     argc (entrada) - Número de argumentos de linha de comando
 *     argv (entrada) - Array de strings presentes na linha de
 *                      comando quando o programa é executado
 *
 * Retorno: Zero, se não ocorrer nenhum erro.
 *          Um valor diferente de zero em caso contrário.
 ****/
int main(int argc, char *argv[])
{
   int  contaBytes = 0, /* Conta os byte comparados em cada arquivo */
        diferentes = 0, /* Assumirá 1, se forem encontrados dois bytes  diferentes */
        erro1 = 0, /* Indicador de erro no arquivo 1 */
        erro2 = 0, /* Indicador de erro no arquivo 2 */
        byte1, /* Byte lido no arquivo 1 */
        byte2; /* Byte lido no arquivo 2 */
   FILE *stream1, /* Stream associado ao arquivo 1 */
        *stream2; /* Stream associado ao arquivo 2 */

      /* O usuário deve informar os nomes dos arquivos */
      /* que serão comparados na linha de comando      */
   if (argc != 3) {
      printf( "\n\t>>> Este programa deve ser usado assim:"
              "\n\t\t> %s nome-de-arquivo1 nome-de-arquivo2\n", argv[0] );
      return 1; /* Usuário não sabe usar o programa */
   }

      /* Tenta abrir o primeiro arquivo em modo binário para leitura */
   stream1 = fopen(argv[1], "rb");

      /* Se o arquivo não pode ser aberto nada mais pode ser feito */
   if (!stream1) {
      printf( "\n\t>>> Nao foi possivel abrir o arquivo \"%s\"\n", argv[1] );
      return 1; /* Primeiro arquivo não foi aberto */
   }
```

```
   /* Tenta abrir o segundo arquivo em modo binário para leitura */
stream2 = fopen(argv[2], "rb");

   /* Se o arquivo não pode ser aberto nada mais pode ser feito */
if (!stream2) {
   printf( "\n\t>>> Nao foi possivel abrir o arquivo \"%s\"\n", argv[2] );

   fclose(stream1); /* Fecha primeiro arquivo antes de retornar */
   return 1; /* Segundo arquivo não foi aberto */
}

   /*                                                    */
   /* Condições de parada do laço while a seguir:        */
   /*  - A leitura atinge o final de um dos arquivos     */
   /*  - Ocorre erro de leitura em um dos arquivos       */
   /*  - São lidos dois bytes correspondentes que diferem */
   /*                                                    */
while ( !feof(stream1) && !feof(stream2) ) {
   byte1 = fgetc(stream1); /* Lê um byte no arquivo 1 */
   byte2 = fgetc(stream2); /* Lê um byte no arquivo 2 */

   contaBytes++; /* Mais um byte lido em cada arquivo */

      /* Se ocorreu erro de leitura no arquivo 1, encerra o laço */
   if (ferror(stream1)) {
      erro1 = 1; /* Se não desejar usar goto nem duplicar código, */
                 /* essa variável é necessária (v. abaixo)       */
      break;
   }

      /* Se ocorreu erro de leitura no arquivo 2, encerra o laço */
   if (ferror(stream2)) {
      erro2 = 1; /* Se não desejar usar goto nem duplicar */
                 /* código, essa variável é necessária    */
      break;
   }

      /* Se a leitura chega ao final num arquivo antes de chegar ao */
      /* final no outro, os  arquivo são obviamente diferentes      */
   diferentes = ( feof(stream1) && !feof(stream2) ) ||
                ( !feof(stream1) && feof(stream2) );

      /* Se já foi constatado que os arquivos são */
      /* diferentes, encerra-se a comparação      */

   if (diferentes) {
      break;
   }

      /* Se os últimos bytes lidos forem diferentes, os arquivos são diferentes */
   if (byte1 != byte2) {
      diferentes = 1;
      break;
   }
}
   /* Fecha os arquivos */
FechaArquivo(stream1, argv[1]);
FechaArquivo(stream2, argv[2]);

   /* Se ocorreu erro no arquivo 1, informa o usuário. Não é mais */
   /* possível usar ferror() pois os arquivos já foram fechados.  */
   /* Isso justifica o uso da variável erro1.                     */
```

```
   if (erro1) {
      printf( "\n\t>>> Erro de leitura no arquivo \"%s\"\n", argv[1] );
      return 1;
   }
      /* Se ocorreu erro no arquivo 2, informa o usuário. Não é mais */
      /* possível usar ferror() pois os arquivos já foram fechados.  */
      /* Isso justifica o uso da variável erro2.                     */
   if (erro2) {
      printf( "\n\t>>> Erro de leitura no arquivo \"%s\"\n", argv[2] );
      return 1;
   }

      /* Se os arquivos forem diferentes, informa o índice dos respectivos */
      /* bytes em que foi detectada a primeira diferença. Aqui, os bytes   */
      /* são indexados a partir de 1.                                      */
   if (diferentes) {
      printf( "\n\t>>> Os arquivos diferem no byte numero %d\n", contaBytes );
   } else { /* Não foi detectada diferença */
      printf("\n\t>>> Os arquivos sao identicos\n");
   }

   return 0;
}
```

Análise:

- Apesar de ser relativamente longo, esse programa é fácil de entender acompanhando atentamente os comentários distribuídos no mesmo.
- A indexação de bytes, utilizada pela variável `contaBytes`, começa em `1` (e não em zero).
- A função `FechaArquivo()` usada pelo programa foi apresentada na Seção 11.5 e sua definição não aparece na listagem acima.

Exemplos de execução do programa (o nome do programa executável é `Compara`):

```
C:\Programas>Compara Tudor.txt TudorBK.txt

        >>> Os arquivos diferem no byte numero 38

C:\Programas>Compara Tudor.txt Tudor.txt

        >>> Os arquivos sao identicos
```

11.15.9 Atualizando Registros de um Arquivo de Texto

Problema: Considere o arquivo de texto `Tudor.txt` descrito na página 601. Escreva um programa que lê o arquivo `Tudor.txt`, 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.

```
/****
 * main():
 *   1. Lê cada linha do arquivo cujo nome é especificado pela constante NOME_ARQUIVO
 *   2. Converte cada a linha lida numa estrutura do tipo tAluno.
 *   3. Atualiza a estrutura acrescentando 1.0 ao campo n2.
 *   4. Escreve a estrutura modificada no arquivo especificado pela constante
 *      NOME_ARQUIVO_ATUAL usando o mesmo formato de linha do arquivo original.
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero, se não ocorrer nenhum erro.
 *          Um valor diferente de zero em caso contrário.
 ****/
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) {
         /* Fecha arquivo de entrada antes de partir */
      fclose(streamE);

      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 */
}
```

Análise: 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()` é apresentada abaixo.

```
/****
 * AtualizaArquivo(): Lê cada linha do arquivo de entrada, converte-a numa
 *                    estrutura do tipo tAluno, atualiza a estrutura e escreve
 *                    a estrutura modificada no arquivo de saída usando o
 *                    mesmo formato do arquivo de entrada
 *
 * Parâmetros:
 *     streamSaida (entrada) - stream associado ao arquivo que conterá a atualização
 *     streamEntrada (entrada) - stream associado ao arquivo que será lido
 *
 * Retorno: Zero, se não ocorrer nenhum erro.
 *          Um valor diferente de zero em caso contrário.
 *
 * Observações:
 *     1. O stream de entrada deve estar aberto em modo de texto que permite leitura
 *     2. O stream de saída deve estar aberto em modo "w"
 ****/
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 ou 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);

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

         /* 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);
}
```

Análise:

- 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.

```
/****
 * LinhaEmRegistro(): Converte uma linha do arquivo numa estrutura do tipo tAluno
 *
 * Parâmetros:
 *     pAluno (saída) - ponteiro para a estrutura resultante da conversão
 *     linha (entrada/saída) - linha que será convertida
 *
 * Retorno: Endereço da estrutura que armazena o resultado
 *
 * Observações:
 *     1. Para simplificar, esta função assume que o parâmetro 'linha' realmente é
 *        um string no formato das linhas do arquivo. Portanto os valores
 *        retornados por strtok() não são testados como deveriam.
 *     2. O parâmetro 'linha' é alterado por strtok().
 ****/
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;
}
```

Análise:

- A função `LinhaEmRegistro()` extrai cada token da linha recebida como parâmetro usando a função **strtok()** (v. Seção 9.5.9) 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()** (v. Seção 9.5.4).

- Os preenchimentos dos campos `n1` e `n2` usam a função **strtod()** (v. Seção 9.9.2), que converte um string num valor do tipo **double**.

Complemento do programa:

```
********************** Includes ************************/

#include <stdio.h>  /* Entrada e saída          */
#include <string.h> /* Processamento de strings */
#include <stdlib.h> /* Função strtod()          */

/***************** Constantes Simbólicas ******************/

#define MAX_NOME  20 /* Número máximo de caracteres num nome */
#define TAM_MATR   4 /* Número de dígitos numa matrícula */
#define MAX_LINHA 50 /* Tamanho máximo de uma linha no arquivo */
#define NOME_ARQUIVO "Tudor.txt" /* Arquivo original */
   /* Arquivo atualizado */
#define NOME_ARQUIVO_ATUAL "TudorAtual.txt"

/***************** Definições de Tipos ******************/

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

11.15.10 Convertendo um Arquivo de Registros de Texto para Binário

Problema: (a) Escreva uma função que lê cada linha de um arquivo com o mesmo formato do arquivo `Tudor.txt`, descrito na Seção 11.15.9 e converte-a numa estrutura do tipo `tAluno` definido como:

```
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 definidas como:

```
#define MAX_NOME 20 /* Número máximo de caracteres num nome */
#define TAM_MATR  4 /* Número de dígitos numa matrícula */
```

Então, a 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. (b) Escreva um programa que testa a função solicitada no item (a) com o arquivo `Tudor.txt`.

Solução de (a):

```
/****
 * CriaArquivoBin(): Lê cada linha de um arquivo no formato especificado,
 *                   numa converte-a estrutura do tipo tAluno e armazena
 *                   a estrutura num arquivo binário.
 *
 * Parâmetros:
 *     streamTexto (entrada) - stream associado ao arquivo que será lido
 *     streamBin (entrada) - stream associado ao arquivo que será escrito
 *
 * Retorno: Número de estruturas escritas no arquivo
 *          binário ou um valor negativo, se ocorrer erro
```

```
 *
 * Observações:
 *     1. O stream de entrada deve estar aberto em modo de texto que permite leitura
 *     2. O stream de saída deve estar aberto em modo "wb"
 ****/
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 */
      }

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

         /* 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 */
}
```

Análise:

- 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? E se novas linhas puderem ser acrescentadas ao mesmo arquivo depois que o programa estiver pronto? Com o conhecimento acumulado até aqui estudando este livro, o leitor ainda não possui ferramentas para lidar com essa categoria de problemas, uma vez que ela requer noções de alocação dinâmica de memória, que só será discutida no próximo capítulo.

- 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 11.15.9 e não será apresentada novamente.

Solução de (b):

```
/****
 * main(): Converte um arquivo de texto em arquivo binário
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero, se não ocorrer nenhum erro.
 *          Um valor diferente de zero em caso contrário.
 ****/
int main(void)
{
   int   nRegistros; /* Número de registros nos arquivos */
   FILE *streamTexto, /* Associado ao arquivo de texto */
        *streamBin;   /* Associado ao arquivo binário */

      /* Tenta abrir arquivo para leitura em modo texto */
   streamTexto = fopen(NOME_ARQUIVO_TEXTO, "r");

      /* Se o arquivo não pode ser aberto, nada mais pode ser feito */
   if (!streamTexto) {
      printf( "O arquivo %s nao pode ser aberto\n", NOME_ARQUIVO_TEXTO );
      return 1; /* Operação mal sucedida */
   }

      /* Tenta criar arquivo binário. Se um arquivo com esse   */
      /* nome existir no diretório corrente ele será destruído */
   streamBin = fopen(NOME_ARQUIVO_BIN, "wb");

      /* Se o arquivo não pode ser aberto, nada mais pode ser feito. */
   if (!streamBin) {
      printf("Arquivo %s nao pode ser aberto\n",NOME_ARQUIVO_BIN);

      fclose(streamTexto); /* Antes de retornar, fecha arquivo aberto */

      return 1; /* Operação mal sucedida */
   }

      /* Lê cada linha do arquivo de texto, converte-a em estrutura */
      /* do tipo tAluno e armazena-a no arquivo binário.            */
   nRegistros = CriaArquivoBin(streamTexto, streamBin);

      /* Verifica se ocorreu erro na criação do arquivo binário. Se for o caso, */
      /* informa o usuário, fecha os arquivos e encerra o programa.              */
   if (nRegistros < 0) {
      printf( "\n\t>>> Ocorreu erro na criacao do arquivo"
              "\n\t>>> \"%s\"\n", NOME_ARQUIVO_BIN );

         /* Fecha os arquivos antes de encerrar */
      fclose(streamTexto);
      fclose(streamBin);

      return 1; /* Ocorreu erro */
   }
```

```
      /* Apresenta o resultado da conversão */
   printf( "\n\t>>> Foram armazenados %d registros no arquivo"
           "\n\t>>> \"%s\"\n", nRegistros, NOME_ARQUIVO_BIN );

      /* Fecha os arquivos */
   fclose(streamTexto);
   fclose(streamBin);

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

Análise: Espera-se que a função **main()** apresentada acima seja suficientemente simples e não requeira comentários além daqueles presentes nela.

Complemento do programa:

```
/********************** Includes *************************/

#include <stdio.h>  /* Funções de Entrada e Saída */
#include <string.h> /* Processamento de strings   */
#include <stdlib.h> /* Função strtod()            */

/***************** Constantes Simbólicas *****************/

#define MAX_NOME  20 /* Número máximo de caracteres num nome */
#define TAM_MATR   4 /* Número de dígitos numa matrícula */
#define MAX_LINHA 50 /* Tamanho máximo de uma linha no arquivo */
#define NOME_ARQUIVO_TEXTO "Tudor.txt" /* Arquivo original */
#define NOME_ARQUIVO_BIN "Tudor.bin" /* Arquivo atualizado */

/****************** Definições de Tipos ******************/

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

11.15.11 Exibindo Registros de um Arquivo Binário na Tela

Problema: (a) Escreva uma função que lê o conteúdo de um arquivo binário que armazena estruturas do tipo `tAluno`, definido nos dois últimos exemplos, e apresenta-o na tela em forma de tabela. (b) Escreva um programa que testa a função solicitada no item (a) com o arquivo `Tudor.bin` criado na Seção 11.15.10.

Solução de (a):

```
/****
 * ExibeArquivoBin(): Exibe na tela os registros de um arquivo binário contendo
 *                    estruturas do tipo tAluno acessando-os sequencialmente
 *
 * Parâmetros: stream (entrada) - stream associado ao arquivo que será exibido
 *
 * Retorno: Zero, se não houver erro. Um valor diferente de zero, em caso contrário.
 *
 * Observação: O arquivo associado ao stream recebido como
 *             parâmetro deve ter sido aberto em modo binário que permite leitura
 ****/
int ExibeArquivoBin(FILE *stream)
{
   tAluno  registro; /* Armazena cada registro lido */
```

```
      /* Move indicador de posição para o início do arquivo.   */
      /* Se ocorrer erro, retorna um valor indicativo do fato. */
      /* A instrução a seguir é mais segura do que rewind().   */
   if (fseek(stream, 0, SEEK_SET)) {
      return 1; /* Indicador de posição não foi movido */
   }

   /* Antes de apresentar o cabeçalho, deve-se tentar ler o primeiro registro, */
   /* pois se isso não for feito e o arquivo estiver vazio ou ocorrer erro de  */
   /* leitura, o cabeçalho aparecerá sem dados abaixo dele.                    */

      /* Tenta ler o primeiro registro */
   fread(&registro, sizeof(tAluno), 1, stream);

      /* Se o final do arquivo foi atingido ou */
      /* ocorreu erro de leitura, retorna 1    */
   if (feof(stream) || ferror(stream)) {
      return 1; /* Nada foi apresentado */
   }

      /* Escreve o cabeçalho da tabela */
   printf( "\n\tNome\t\tMatricula\tNota 1\tNota 2"
           "\n\t====\t\t=========\t======\t======\n\n" );

      /* Escreve o primeiro registro lido */
   printf( "    %-15s\t%6s\t\t %3.1f\t %3.1f\n",
           registro.nome, registro.matr, registro.n1, registro.n2 );

      /* Acessa cada registro do arquivo na ordem em que ele se encontra. O  */
      /* laço encerra quando o final do arquivo for atingido ou ocorrer erro */
   while (1) {
         /* Lê um registro */
      fread(&registro, sizeof(tAluno), 1, stream);

         /* Se o final do arquivo foi atingido ou   */
         /* ocorreu erro de leitura, encerra o laço */
      if (feof(stream) || ferror(stream)) {
         break;
      }

         /* Escreve o registro lido. Não tente entender o embelezamento */
         /* usado com printf(). Ele foi obtido por tentativa e erro, e  */
         /* não se aplica a dados com outros formatos.                  */
      printf( "    %-15s\t%6s\t\t %3.1f\t %3.1f\n",
              registro.nome, registro.matr, registro.n1, registro.n2 );
   }

      /* Se não ocorreu erro de leitura, o retorno é 0; */
      /* caso contrário, o retorno é diferente de 0     */
   return ferror(stream);
}
```

Análise:

- Conforme recomendado na Seção 11.13, a função `ExibeArquivoBin()` move o indicador de posição do arquivo para seu início por meio de uma chamada de **fseek()**.
- Como os dados serão apresentados em forma de tabela, por uma questão de bom senso e estética, antes de apresentar o cabeçalho dessa tabela, a função sob discussão verifica se existe pelo menos um registro no arquivo. Isso previne que o cabeçalho da referida tabela seja exibido sem dados abaixo dele.

- Os especificadores de formato usados nas chamadas de **printf()** que apresentam os dados do arquivo tornam a saída do programa esteticamente agradável ao usuário, mas o programador não deve desperdiçar tempo tentando entendê-los a fundo. Esses especificadores foram ajustados por tentativa e erro; i.e., escrevendo-os, reescrevendo-os e verificando o resultado exibido na tela. Conhecer bem todos os especificadores de formato de **printf()** não irá torná-lo um grande programador. Utilize seu tempo em atividades mais produtivas recomendas neste livro.

Solução de (b):

```
/****
 *
 * main(): Lê o conteúdo do arquivo binário especificado pela
 *         constante NOME_ARQUIVO_BIN e apresenta-o na tela
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero, se não ocorrer nenhum erro.
 *          Um valor diferente de zero em caso contrário.
 *
 ****/
int main(void)
{
   FILE   *stream; /* Stream associado ao arquivo */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa exibe o conteudo do"
           "\n\t>>> arquivo binario \"%s\" na tela.\n", NOME_ARQUIVO_BIN );

      /* Tenta abrir o arquivo em modo binário para leitura apenas */
   stream = fopen(NOME_ARQUIVO_BIN, "rb");

      /* Se o arquivo não foi aberto, não há mais nada que se possa fazer  */
   if (!stream) {
      printf( "\n\t>>> O arquivo %s nao pode ser aberto\n", NOME_ARQUIVO_BIN );
      return 1; /* Arquivo não foi aberto */
   }

      /* Apresenta o conteúdo do arquivo na tela checando ocorrência de erro */
   if (ExibeArquivoBin(stream)) {
      printf( "\n\t>>> Ocorreu erro de processamento do"
              "\n\t>>> arquivo \"%s\"\n", NOME_ARQUIVO_BIN );

      fclose(stream); /* Fecha arquivo antes de encerrar */

      return 1; /* Erro de processamento */
   }

   fclose(stream); /* Fecha o arquivo */

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

Análise: A função **main()** é suficientemente simples e não requer comentários adicionais.

Complemento do programa:

```
/********************** Includes *************************/

#include <stdio.h>   /* Entrada e saída */
#include <stdlib.h>  /* Função exit()   */

/***************** Constantes Simbólicas ******************/
```

```
#define MAX_NOME  20 /* Número máximo de caracteres num nome */
#define TAM_MATR   4 /* Número de dígitos numa matrícula */
#define NOME_ARQUIVO_BIN "Tudor.bin" /* Nome do arquivo binário */

/****************** Definições de Tipos ******************/

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

Exemplo de execução do programa:

```
        >>> Este programa exibe o conteudo do
        >>> arquivo binario "Tudor.bin" na tela.

        Nome            Matricula       Nota 1  Nota 2
        ====            =========       ======  ======

    Henrique VIII         1029           9.5     9.0
    Catarina Aragon       1014           5.5     6.5
    Ana Bolena            1012           7.8     8.0
    Joana Seymour         1017           7.7     8.7
    Ana de Cleves         1022           4.5     6.0
    Catarina Howard       1340           6.0     7.7
    Catarina Parr         1440           4.0     6.0
```

Análise: A escrita na tela apresentada por esse programa é visualmente agradável e fácil de interpretar e a melhor maneira de obter um resultado assim é por meio de tentativa e erro, como já foi afirmado. Quer dizer, para obter o embelezamento desejado na apresentação de dados, acrescente ou remova espaços em branco ou outros caracteres, recompile o programa e teste até obter um resultado que satisfaça você e, principalmente, o usuário do seu programa.

11.15.12 Número de Registros de um Arquivo Binário

Problema: (a) Escreva uma função que recebe como parâmetros um stream binário contendo registros de mesmo tamanho e o tamanho de cada registro e retorna o número de registros do arquivo associado ao stream. (b) Escreva um programa que teste a função proposta no item (a) com o arquivo `Tudor.bin` criado na Seção 11.15.10.

Solução de (a):

```
/****
 *
 * NumeroDeRegistros(): Calcula o número de registros (i.e.,
 *                      partições de tamanho especificado) de um arquivo binário
 *
 * Parâmetros:
 *    stream (entrada) - stream binário associado ao arquivo cujo
 *                       tamanho será calculado
 *    tamRegistro (entrada) - tamanho de cada partição do arquivo
 *
 * Retorno: O número de partições do arquivo, se não ocorrer
 *          nenhum erro, ou -1, em caso contrário.
 *
 * Observação: O arquivo deve estar aberto em modo binário que permite leitura
 *
 ****/
```

```
int NumeroDeRegistros( FILE *stream, int tamRegistro )
{
   int nBytes = 0; /* Número de bytes do arquivo */

      /* Calcula o número de bytes do arquivo */
   nBytes = TamanhoDeArquivo(stream);

      /* Verifica se ocorreu erro na contagem de bytes */
   if (nBytes < 0) {
      return -1; /* Erro na contagem de bytes */
   }

      /* O número de registros é o número de bytes do arquivo */
      /* dividido pelo número de bytes de cada registro       */
   return nBytes/tamRegistro;
}
```

Análise: A função `NumeroDeRegistros()` espera receber como parâmetro um stream binário associado ao arquivo cujo número de registros será calculado. Essa função chama a função `TamanhoDeArquivo()`, definida na Seção 11.15.7, para calcular o número de bytes do arquivo. Então, a função `NumeroDeRegistros()` divide o número de bytes do arquivo pelo número de bytes de uma de suas partições e obtém o resultado desejado.

Solução de (b):

```
/****
 *
 * main(): Assume que um arquivo binário é constituído de registros
 *         de um mesmo tipo e calcula o número de registros que compõem o arquivo
 * Parâmetros: Nenhum
 * Retorno: Zero, se não ocorrer nenhum erro.
 *          Um valor diferente de zero em caso contrário.
 *
 ****/
int main(void)
{
   FILE  *stream; /* Stream associado ao arquivo */
   int    nRegistros; /* Número de registros do arquivo */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa apresenta o numero de registros"
           "\n\t>>> do arquivo \"%s\"\n", NOME_ARQUIVO_BIN );

      /* Tenta abrir arquivo binário */
   stream = fopen(NOME_ARQUIVO_BIN, "rb");

      /* Se o arquivo não pode ser aberto, nada mais pode ser feito */
   if (!stream) {
      printf( "\n\t>>> Arquivo \"%s\" nao pode ser aberto", NOME_ARQUIVO_BIN );
      return 1; /* Operação mal sucedida */
   }

      /* Calcula o número de registros do arquivo */
   nRegistros = NumeroDeRegistros( stream, sizeof(tAluno) );

      /* Verifica se ocorreu erro na operação */
   if (nRegistros < 0) {
      printf( "\n\t>>> Ocorreu um erro no processamento do "
              "\n\t>>> arquivo \"%s\"", NOME_ARQUIVO_BIN );
      return 1;
   }
```

```
      /* Apresenta o resultado */
   printf( "\n\t>>> Numero de registros do arquivo \"%s\":"
           " %d\n", NOME_ARQUIVO_BIN, nRegistros );

   fclose(stream); /* Fecha o arquivo */

   return 0;
}
```

Complemento do programa:

```
/********************** Includes *************************/

#include <stdio.h> /* Entrada e saída */

/***************** Constantes Simbólicas *****************/

#define MAX_NOME 20 /* Número máximo de caracteres num nome */
#define TAM_MATR  4 /* Número de dígitos numa matrícula */
#define NOME_ARQUIVO_BIN "Tudor.bin" /* Arquivo binário */

/****************** Definições de Tipos ******************/

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

Exemplo de execução do programa:

```
>>> Este programa apresenta o numero de registros
>>> do arquivo "Tudor.bin"

>>> Numero de registros do arquivo "Tudor.bin": 7
```

11.16 Exercícios de Revisão

Introdução (Seção 11.1)

1. O que é um arquivo armazenado?
2. Qual é o significado de *arquivo* em C (no sentido genérico)?
3. Descreva processamento de arquivos baseado em streams.
4. Os dispositivos teclado e monitor de vídeo são considerados arquivos? Explique.
5. Em linhas gerais, qual é o conteúdo do arquivo de cabeçalho `<stdio.h>`?

Arquivos de Texto e Arquivos Binários (Seção 11.2)

6. O que é formato de arquivo?
7. O que é um arquivo de texto?
8. O que é um arquivo binário?
9. Que restrições um sistema operacional pode impor a arquivos de texto?

Streams (Seção 11.3)

10. (a) O que é um stream? (b) Que vantagem esse conceito oferece ao processamento de arquivos?
11. Qual é a diferença entre stream e arquivo?
12. Como o conceito de stream é implementado em C?

13. Do ponto de vista prático, faz diferença confundir stream com arquivo?
14. (a) Qual é o significado do identificador **FILE**? (b) Onde o identificador **FILE** é definido?
15. Cite algumas informações que devem estar armazenadas numa estrutura do tipo **FILE**.

Abrindo um Arquivo (Seção 11.4)

16. (a) O que significa abrir um arquivo? (b) Como isso é realizado em C?
17. (a) Quais são os parâmetros da função **fopen()**? (a) O que essa função retorna?
18. Para que serve a constante simbólica **FILENAME_MAX**?
19. Cite três razões pelas quais um arquivo não pode ser aberto.
20. (a) Como se testa se um arquivo foi realmente aberto? (b) O que pode ocorrer se esse teste não for efetuado?
21. (a) O que é um stream de texto e (b) O que é um stream binário?
22. (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?
23. Um arquivo binário pode estar associado a um stream de texto?
24. O que é modo de abertura de arquivo?
25. O que é um modo de atualização?
26. Descreva os seguintes modos de abertura de arquivo:
    (a) `"r"`
    (b) `"w"`
    (c) `"a"`
    (d) `"r+"`
    (e) `"w+"`
    (f) `"a+"`
27. Que formato de arquivo é recomendado para cada modo de abertura do exercício anterior?
28. Qual é a diferença entre modos de abertura para streams de texto e modos de abertura correspondentes para streams binários em termos de sintaxe?
29. Quais são as diferenças entre os seguintes modos de abertura de arquivo:
    (a) `"r"` e `"rb"`
    (b) `"r"` e `"rt"`
    (c) `"rt"` e `"rb"`
30. Quais são as diferenças entre os modos de abertura `"r+"`, `"w+"` e `"a+"`?
31. Que precaução deve ser tomada quando se usam os modos de abertura `"w"`, `"w+"`, `"wb"` e `"w+b"`?
32. (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?
33. O que deve ser imediatamente feito após chamar **fopen()** para abrir um arquivo e antes de processá-lo?
34. Como se pode determinar o número máximo de arquivos que podem ser abertos simultaneamente?
35. Como se pode determinar o tamanho máximo de um nome de arquivo no sistema operacional utilizado?
36. Como a constante simbólica **FILENAME_MAX** deve ser usada na prática?
37. O que representa a constante simbólica **FOPEN_MAX**?
38. 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;
}
```

Fechando um Arquivo (Seção 11.5)

39. (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?

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

41. Qual é a facilidade que a função `FechaArquivo()` apresentada na Seção 11.5 oferece?

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

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

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

... /* Processamento do arquivo */

fclose("teste.bin")
```

Ocorrências de Erros em Processamento de Arquivos (Seção 11.6)

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

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

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

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

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

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

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

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

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

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

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

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

Buffering e a Função fflush() (Seção 11.7)

56. (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?

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

58. O que significa descarregar um buffer?

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

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

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

```
fflush(stdin);
```

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

```
fflush(NULL);
```

Streams Padrão (Seção 11.8)

63. Descreva os streams padrão de C.

64. Por que os streams padrão são qualificados como *padrão*?

Leitura de Dados via Teclado (Seção 11.9)

65. Descreva o funcionamento da função **getchar**().

66. Se a função **getchar**() lê apenas caracteres, por que o tipo do valor que ela retorna é **int**, e não **char**?

67. Por que nem sempre a função **getchar**() interrompe a execução do programa e espera que o usuário digite um caractere?

68. Por que leitura de dados via teclado é sempre mais complicada do que escrita de dados na tela?

69. Sabe-se que, via teclado, é possível introduzir apenas caracteres. Então, como é possível um programa receber um número como entrada?

70. Qual deve ser o conteúdo de um string de formatação da função **scanf**()?

71. O que significam os três pontos no protótipo de **scanf**()?

72. Descreva detalhadamente o funcionamento da função **scanf**().

73. (a) O que representa o valor retornado pela função **scanf**()? (b) Como esse valor deve ser utilizado?

74. Como age a função **scanf**() quando lê um número inteiro corretamente digitado pelo usuário?

75. O seguinte programa foi escrito com o propósito de ler e somar dois números inteiros e apresentar o resultado na tela. O que há de errado com esse programa?

```
#include <stdio.h>

int main(void)
{
   int x, y, soma;

   soma = x + y;

   printf("Digite dois numeros inteiros: ");
   scanf("%d %d", &x, &y);
   printf("Resultado: %d + %d = %d", x, y, soma);

   return 0;
}
```

76. O seguinte programa tem o mesmo propósito do programa anterior, mas também é defeituoso. Qual é seu defeito?

```
#include <stdio.h>

int main(void)
{
   int x, y;

   printf("Digite dois numeros inteiros: ");
   scanf("%d %d", x, y);

   printf("Resultado: %d + %d = %d", x, y, x + y);

   return 0;
}
```

77. (a) O que se espera que o usuário introduza com a chamada de **scanf()** no seguinte programa? (b) O que esse mesmo programa exibe na tela quando o usuário digita `5`?

```
#include <stdio.h>

int main(void)
{
   int x;

   scanf("Digite um inteiro: %d", &x);
   printf("x = %d", x);

   return 0;
}
```

78. O que os strings de formatação das funções **printf()** e **scanf()** têm em comum?

79. (a) Existe diferença entre os especificadores **%d** e **%i** quando eles são utilizados com **scanf()**? (b) Existe diferença entre esses mesmos especificadores quando eles são utilizados com **printf()**?

80. O que ocorre quando o número de especificadores de formato numa chamada de **scanf()** é (a) *maior* ou (b) *menor* do que o número de endereços de variáveis que seguem o string de formatação?

81. Suponha que `x` e `y` sejam variáveis do tipo **int**. Quantos valores cada uma das chamadas de **scanf()** a seguir espera ler?

(a) `scanf("%d", &x, &y);`

(b) `scanf("%d %d", &x);`

82. Suponha que a função **scanf()** seja chamada como no seguinte trecho de programa:

```
int x, retorno;

retorno = scanf("%d", &x);
```

O que significa quando o valor atribuído à variável `retorno` é:

(a) `0`?

(b) `1`?

(c) **EOF**?

83. Suponha que a função **scanf()** seja chamada como no seguinte fragmento de programa:

```
int x, y, retorno;

retorno = scanf("%d %d", &x, &y);
```

Quais sãos os possíveis valores que podem ser atribuídos à variável retorno?

84. Por que o seguinte programa será provavelmente abortado quando for executado?

```
#include <stdio.h>

int main(void)
{
   char *p = "otorrinolaringologista";

   printf("\nDigite no maximo 5 caracteres: ");
   scanf("%6s", p);

   printf("String lido: %s", p);

   return 0;
}
```

85. Por que, quando o programa abaixo for executado, ele poderá ser abortado?

```
#include <stdio.h>

int main(void)
{
   int x;

   printf("\nDigite um numero inteiro: ");
   scanf("%d", &x);

   printf("Numero lido: %s", x);

   return 0;
}
```

86. Suponha que `x` seja uma variável do tipo **int**. (a) O que há de errado com cada uma das seguintes chamadas de **scanf()** a seguir? (b) Supondo que essas instruções fazem parte de um programa, esses erros serão detectados durante a compilação ou execução do programa?

(i) `scanf("%d", x);`

(ii) `scanf("%lf", &x);`

87. Como a função **scanf()** pode ser usada de modo seguro na leitura de strings?

88. Por que às vezes a função **scanf()** é inconveniente para leitura de strings?

89. (a) Qual é o significado do especificador **%[0123456789]** usado com **scanf()**? (b) Qual é o significado do especificador **%[^0123456789]** usado com **scanf()**?

90. Por que a função **gets()** foi removida da biblioteca padrão de C pelo padrão C11?

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

92. Qual deve ser o terceiro parâmetro da função **fgets()** quando essa função é invocada para ler strings via teclado?

93. (a) Considerando o trecho de programa a seguir, o que a função **scanf()** lê se o usuário digitar `123z`? (b) Que consequência danosa ao programa essa entrada de dados pode infligir?

```
int x;
...
printf("\nDigite um valor inteiro: ");
scanf("%d", &x);
```

94. (a) Por que o seguinte programa sempre informa que o nome do usuário tem um caractere a mais do que seria esperado? (b) Como esse problema pode ser sanado?

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

int main(void)
{
   char linha[200];

   printf("\nDigite seu nome: ");
   fgets(linha, sizeof(linha), stdin);

   printf( "Seu nome tem %d caracteres\n", strlen(linha) );

   return 0;
}
```

95. Considerando o programa abaixo, por que o usuário não consegue digitar o caractere `'B'` solicitado?

```
#include <stdio.h>

int main(void)
{
   int c;

   printf("\nDigite o caractere A: ");
   c = getchar();

   printf("\nDigite o caractere B: ");
   c = getchar();

   return 0;
}
```

96. (a) Considerando o mesmo trecho de programa da questão anterior, o que a função **scanf()** lerá se o usuário digitar `z123`? (b) Qual será o conteúdo do buffer associado ao teclado logo após o usuário ter concluído sua digitação?

97. Considerando o mesmo trecho de programa das duas últimas questões, qual será o conteúdo do buffer associado ao teclado após a atuação de **scanf()** quando o usuário digitar `123`?

98. (a) Em que situação a função **scanf()** lê o caractere `'\n'` encontrado no buffer associado ao teclado? (b) Quando ela o remove do buffer? (c) Quando ela não o remove do buffer?

99. (a) Que caracteres remanescentes no buffer associado ao teclado não afetam uma leitura de valor numérico efetuada por **scanf()**? (b) Em que situação qualquer caractere remanescente no buffer afeta uma leitura efetuada por **scanf()**?

100. Descreva o funcionamento do buffer associado à entrada de dados padrão.

101. Considere o fragmento de programa a seguir. (a) O que ocorreria se a chamada da função `LimpaBuffer()` fosse removida do corpo do laço **while**? (b) Por que, nesse caso, o uso de **goto** é aceitável?

```
   int  umInt, nValoresLidos;

leitura:
   printf("Digite um valor inteiro: ");
   nValoresLidos = scanf("%d", &umInt);

   while (nValoresLidos == 0) { /* Nenhum inteiro foi lido */
      LimpaBuffer();
      printf("Valor incorreto. ");
      goto leitura;
   }
```

102. Reescreva o trecho de programa do exercício **101** sem usar **goto**.

103. Para que serve a função `LimpaBuffer()` apresentada na Seção 11.9.4?

104. (a) Por que uma função como `LimpaBuffer()` se faz necessária? (b) Por que não se deve usar a função **fflush()** da biblioteca padrão de C em substituição a `LimpaBuffer()`?

105. Como laços de repetição são usados em leitura de dados via teclado?

Acessos Sequencial e Direto (Seção 11.10)

106. Descreva as categorias de processamento sequencial de arquivos.

107. Defina os seguintes conceitos:
(a) Acesso sequencial a arquivo
(b) Acesso direto a arquivo
(c) Processamento formatado

108. 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

109. Que funções são tipicamente usadas para:
(a) Processamento por byte
(b) Processamento por linha
(c) Processamento por bloco

Processamento Sequencial de Arquivos (Seção 11.11)

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

111. Descreva as funções **fgetc()** e **fputc()**.

112. 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?

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

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

114. 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);
}
```

115. Suponha que `streamA` e `streamB` sejam 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.

116. Descreva as funções a seguir:
(a) **fgets()**
(b) **fputs()**

117. (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()**

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

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

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

121. Suponha que `teste` seja o nome de um arquivo de texto residente no mesmo diretório do 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;
}
```

122. 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;
}
```

123. 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;
}
```

124. 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?

Acesso Direto a Arquivos (Seção 11.12)

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

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

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

128. O que é uma função de posicionamento?

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

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

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

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

133. (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?

134. 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? [**Dica**: O que acontece quando os dois nomes de arquivo recebidos como argumentos pelo programa são os mesmos?]

```
#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");
```

CONTINUA

```
    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;
}
```

rewind() ou fseek()? (Seção 11.13)

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

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

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

138. (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**()?

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

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

Condições de Exceção e a Lei de Murphy (Seção 11.14)

141. O que é uma condição de exceção?

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

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

144. 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**()

11.17 Exercícios de Programação

11.17.1 Fácil

EP11.1 Escreva um programa que lê, via teclado, um valor inteiro, um valor real e um string. Então, o programa deve armazenar os valores lidos em linhas separadas de um novo arquivo de texto denominado `Teste.txt`. Em seguida, o programa recupera esses valores do arquivo e os apresenta na tela. [**Sugestões**: Abra o arquivo no modo `"w+"` e use as funções **fprintf**() e **fscanf**() para escrever e ler no arquivo, respectivamente. Não esqueça de usar **rewind**() antes de efetuar a leitura.]

EP11.2 Escreva um programa semelhante àquele proposto no exercício **EP11.1** que armazene os dados um narquivo temporário criado usando **tmpfile()**. [**Sugestões**: Use as mesmas sugestões do exercício **EP11.1**.]

EP11.3 Escreva um programa semelhante àquele proposto no exercício **EP11.1** que armazene os dados num arquivo temporário criado por meio de **tmpnam()**. O programa deve ainda rebatizar esse arquivo para `Teste.txt`. [**Sugestões**: Use as mesmas sugestões do exercício **EP11.1** e a função **rename()**.]

EP11.4 Sabendo que a maior linha de um arquivo, denominado `Tudor.txt`, 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()**. [**Sugestões**: (1) Abra ambos os arquivos em modo texto e use um laço de repetição no corpo do qual se lê uma linha no arquivo `Tudor.txt`, por meio de **fgets()** e escreve-se a mesma linha no arquivo `BK.txt` usando **fprintf()** em conjunto com o especificador **%s**. (2) Use as funções **feof()** e **ferror()** para encerrar o laço quando houver tentativa de leitura além do final do arquivo ou ocorrer erro de leitura ou escrita.]

EP11.5 Escreva um programa que lê via teclado um valor inteiro, um valor real e um string. Então, o programa armazena os valores lidos num arquivo binário. Em seguida, o programa recupera esses valores do arquivo e os apresenta na tela. [**Sugestão**: Use as funções **fread()** e **fwrite()** para ler e escrever no arquivo, que deve ser aberto no modo `"w+b"`.]

EP11.6 Reescreva o programa apresentado na **Seção 11.15.6**, de modo que ele possa receber via linha de comando nomes de um número arbitrário de arquivos de texto e informar qual é o número de linhas de cada um deles. Se algum arquivo não puder ser aberto o programa apenas informa o usuário sobre o fato, mas não deve ser abortado. [**Sugestões**: (1) Escreva uma função **main()** com dois parâmetros, conforme descrito na **Seção 9.6**. (2) Use um laço **for** para acessar cada nome de arquivo, abra o arquivo correspondente, chame a função `NumeroDeLinhas()` definida **Seção 11.15.5** e informe o resultado. (3) Não esqueça de fechar cada arquivo após apresentar seu número de linhas.]

EP11.7 Reescreva a função `CopiaArquivo()`, apresentada na **Seção 11.15.6**, 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. [**Sugestão**: Use uma variável inteira local à função que é iniciada com zero e incrementada cada vez que um byte é escrito no arquivo que recebe a cópia.]

EP11.8 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. [**Sugestões**: (1) Abra o arquivo no modo `"r"`. (2) Use as funções **fgetc()** para ler cada caractere do arquivo e **putchar()** para escrevê-lo na tela.]

EP11.9 Implemente um programa que escreve na tela os conteúdos de todos os arquivos especificados em linha de comando. [**Sugestões**: (1) Use as sugestões do exercício **EP11.8** para implementar uma função que escreve o conteúdo de cada arquivo na tela. (2) Na função **main()** desse programa, escreva um laço de repetição **for** que abre cada arquivo especificado, chama a função proposta na sugestão (1) e feche o arquivo.]

EP11.10 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. [**Sugestões**: (1) Utilize uma variável inteira como contadora e inicie-a com zero. (2) Abra o arquivo com o modo `"r"`. (3) Leia cada caractere do arquivo com **fgetc()** e escreva-o na tela com **putchar()**. (4) Incremente a variável de contagem a cada caractere de quebra de linha encontrado. (5) Quando a variável de contagem atingir

o valor `10`, reinicie-a com zero, apresente um prompt apropriado para o usuário e use **getchar()** para interromper a execução do programa à espera da resposta do usuário.]

EP11.11 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. [**Sugestões**: (1) Crie uma função baseada na função `CopiaArquivo()` apresentada na **Seção 11.15.6**. (2) Utilize nessa nova função um parâmetro adicional que representa o número máximo de linhas que serão copiadas. (3) Utilize uma variável local à função para contar quantas linhas são copiadas (v. sugestões do exercício **EP11.10**). (4) A função deve encerrar quando o final do arquivo de leitura for atingido ou quando o número de linhas atingir o máximo especificado pelo parâmetro descrito na sugestão (2).]

EP11.12 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. [**Sugestões**: (1) Abra o arquivo original no modo `"r"`. (2) Crie um arquivo temporário e copie para esse arquivo os caracteres do arquivo original convertidos em maiúsculas usando **toupper()**. (3) Feche o arquivo original e reabra-o no modo `"w"`. (4) Use a função `CopiaArquivo()` apresentada na **Seção 11.15.6**, para copiar o conteúdo do arquivo temporário para o arquivo original.]

EP11.13 Escreva um programa que converte o arquivo `Tudor.bin`, criado na **Seção 11.15.10**, num arquivo de texto no formato descrito no início da **Seção 11.15**. 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 11.15**. [**Sugestão**: Leia cada estrutura encontrada no arquivo binário `Tudor.bin` e use **fprintf()** para escrevê-la no arquivo de texto no formato apresentado no início da **Seção 11.15**.]

EP11.14 Escreva um programa que lê o arquivo `Tudor.bin` e apresenta o nome e a média de cada aluno e, ao final, a média da turma. [**Sugestão**: Este exercício é bem mais fácil do que o exercício **EP11.13**.]

EP11.15 Escreva uma função que determina o tamanho da maior linha de um stream de texto. [**Sugestões**: (1) Utilize dois laços de repetição **while**. O primeiro laço é executado enquanto houver linhas a serem lidas e o segundo laço é executado enquanto houver caracteres a serem lidos na linha corrente. Os dois laços podem ser encerrados prematuramente se ocorrer erro durante a leitura do arquivo. (2) Defina as variáveis `tamLinhaCorrente`, para armazenar a linha corrente, e `tamMaiorLinha`, para guardar o tamanho da maior linha até então encontrada. O valor de `tamLinhaCorrente` deve ser atualizado no laço **while** interno a cada novo caractere (com exceção de `'\n'`) encontrado na linha ora escrutinada. O valor de `tamMaiorLinha` deve ser atualizado no laço **while** externo cada vez que o tamanho da linha corrente é maior do que o valor corrente dessa variável.] (b) Escreva um programa que apresenta o tamanho da maior linha de um arquivo de texto cujo nome é introduzido via linha de comando.

EP11.16 Reescreva o programa apresentado na **Seção 11.15.9**, de maneira que o resultado da atualização seja escrito no próprio arquivo de entrada (que, então, passará a ser de entrada e saída). [**Sugestão**: Use um arquivo temporário e a função `CopiaArquivo()` apresentada na **Seção 11.15.6**.]

EP11.17 Escreva um programa que lê o arquivo `Tudor.txt` e apresenta o nome e a média de cada aluno e, ao final, a média da turma. [**Sugestão**: Use a função `LinhaEmRegistro()` definida na **Seção 11.15.9**.]

11.17.2 Moderado

EP11.18 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`. [**Sugestões**: (1) Use um laço de repetição que encerra quando ocorre erro de leitura ou tentativa de leitura além do final do arquivo. (2) No corpo desse laço, use **fgets()** para ler as linhas do arquivo e **printf()** para escrevê-las na tela. (3) Use uma variável que conte o número de linhas lidas e escritas. (4) Quando o resto da divisão dessa variável por `10` for igual a zero, dez linhas terão sido escritas. Nesse caso, solicite que o usuário digite um caractere e leia-o usando a função `LeCaractere()` da biblioteca LeituraFacil. (5) Depois disso, prossiga com a execução do laço.]

EP11.19 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`. [**Sugestões**: (1) Use a função `AbreArquivo()`, definida na **Seção 11.15.4**, para abrir os dois arquivos. (2) Crie um laço infinito com quatro condições de parada em seu corpo: erro de leitura ou escrita, tentativa de leitura além do final do arquivo ou número de linhas escritas igual a `n`. (3) Use **fgets()** (v. **Seção 11.9.6**) para ler as linhas do arquivo de entrada e **fputs()** (v. **Seção 11.11.2**) para escrevê-las no arquivo de saída.]

EP11.20 Implemente uma função, denominada `LeReal2()`, funcionalmente equivalente à função `LeReal()` da biblioteca LeituraFacil, e um programa que teste a função implementada. [**Sugestão**: Use como modelo a função `LeInteiro2()` apresentada na **Seção 11.9.5**.]

EP11.21 Implemente uma função, denominada `LeCaractere2()`, funcionalmente equivalente à função `LeCaractere()` da biblioteca LeituraFacil, e um programa que teste a função implementada. [**Sugestão**: Como não há restrição sobre que caracteres são permitidos, só há duas coisas que podem dar errado na leitura de um caractere: tentativa de leitura além do final do arquivo e erro de processamento. Em ambos os casos, **getchar()** retorna **EOF**. Portanto use um laço de repetição que encerra quando **getchar()** não retorna **EOF**. Antes de retornar o caractere lido, não esqueça de deixar o buffer limpo.]

EP11.22 Implemente uma função, denominada `LeOpcao2()`, funcionalmente equivalente à função `LeOpcao()` da biblioteca LeituraFacil, e um programa que teste a função implementada. [**Sugestão**: Use a função `LeCaractere2()`, implementada no exercício anterior, para ler um caractere que corresponde à opção do usuário e, então, chame a função **strchr()** para checar se a opção é válida.]

EP11.23 Implemente uma função, denominada `LeString2()`, funcionalmente equivalente à função `LeString()` da biblioteca LeituraFacil, e um programa que teste a função implementada. [**Sugestões**: (1) Use a função **fgets()** para ler os caracteres em **stdin**, conforme foi discutido na **Seção 11.9.6**. (2) Use a função `LimpaBuffer2()`, implementada na **Seção 11.9.5**, para limpar o buffer e obter o número de caracteres remanescentes. (3) Use a função **strchr()** para remover qualquer eventual quebra de linha lida, conforme foi visto na **Seção 11.9.6**.]

EP11.24 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. [**Sugestões**: (1) Defina um array de elementos do tipo **int** com tamanho igual a `26`, que é o número de letras do alfabeto. Cada elemento desse array será responsável pela contagem de ocorrências de uma letra, de forma que o primeiro elemento conta as ocorrências de `'A'` ou `'a'`, o segundo elemento conta as ocorrências de `'B'` ou `'b'` e assim por diante. (2) Abra o arquivo e leia-o sequencialmente, verificando quais, dentre os caracteres lidos, são letras usando **isalpha()**. (3) Para cada letra encontrada no arquivo, incremente o elemento do array mencionado que é responsável pela contagem da respectiva letra.]

EP11.25 Considerando o arquivo binário `Tudor.bin` criado pelo programa da **Seção 11.15.10**, 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. [**Sugestão**: Defina o tipo `tMatricula` como: `typedef char tMatricula[TAM_MATR + 1]` e um array de elementos desse tipo para armazenar as matrículas.]

(ii) Obtenha uma matrícula válida do usuário. [**Sugestão**: Use a função `LeIdentidade()` definida na **Seção 9.10.2**.]

(iii) Verifique se a matrícula faz parte do array que contém as matrículas. [**Sugestão**: Faça uma busca sequencial no array usando **strcmp()** para comparar 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.

[**Sugestão**: Para implementação dos últimos três passos, escreva uma função que atualiza o arquivo binário seguindo os passos a seguir. (1) Use um laço de repetição para ler cada registro do arquivo até que o registro desejado seja encontrado. (2) Quando isso acontecer, use **fseek()** para fazer o apontador de posição de arquivo apontar novamente para o referido registro. (3) Use **fwrite()** para escrever o registro alterado na devida posição.]

EP11.26 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()** (v. **Seção 9.7.1**). 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. [**Sugestões**: (1) Abra o arquivo em modo de texto para leitura. (2) Use três variáveis para contar o número de linhas, palavras e caracteres. Essas variáveis devem ser iniciadas com zero. (3) Use um laço de repetição infinito que encerra quando o final do arquivo for atingido ou se ocorrer algum erro de leitura. (4) Para cada caractere lido, incremente a contagem de caracteres e, para cada caractere `'\n'` lido, incremente o número de linhas. (5) Como entre duas palavras pode haver um ou mais separadores, para obter o número correto de palavras, deve-se usar uma variável (p. ex., chamada `separador`) que indica se o último caractere lido foi um separador. Ou seja, quando um caractere separador for encontrado, a variável `separador` recebe `0`, de modo que, se o próximo caractere não for separador, o número de palavras é incrementado e a variável `separador` recebe `1`. (6) Para complicar um pouco mais, o caractere `'-'`, que une palavras compostas, deve ser testado separadamente pois **ispunct()** o considera símbolo de pontuação.]

EP11.27 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. [**Sugestões**: (1) Abra o arquivo-fonte em modo `"r"`. (2) Copie para um arquivo temporário todos os caracteres do arquivo-fonte que não se encontram entre o par de caracteres `'/'` e `'*'` e o par de caracteres `'*'` e `'/'`. (3) Feche o arquivo-fonte e reabra-o para escrita no modo `"w"`. (4) Use a função `CopiaArquivo()` (v. **Seção 11.15.6**) para copiar o conteúdo do arquivo temporário para o arquivo-fonte.]

11.18 Projetos de Programação

Os problemas propostos nesta seção incorporam complexidades que demandam um tempo razoável para serem resolvidos. Também, não são oferecidas sugestões, de sorte que eles são apropriados como projetos de programação que verificam a aprendizagem de boa parte do material apresentado neste livro.

PP11.1 Removendo Registros de um Arquivo com Desfaz e Refaz

Problema: (a) Escreva uma função que remove de um arquivo todos os registros que possuem um campo que casa com um determinado valor (**chave**). Essa função deve permitir que a operação seja desfeita. (b) Escreva uma função que desfaz e refaz, indefinidas vezes, a operação descrita em (a). (c) Escreva um programa contendo uma função **main()** que ofereça o menu de opções a seguir e executa a respectiva operação escolhida pelo usuário:

```
1. Remove registro
2. Exibe arquivo
3. Desfaz
4. Refaz
5. Encerra o programa
```

O arquivo usado para testar o programa deve ser **`Tudor.bin`**, criado na **Seção 11.15.10**.

Exemplo de execução do programa que resolve o problema proposto:

```
        >>> Este programa permite remover registros de um arquivo
        >>> e desfazer e refazer essa operacao.
        >>> Este programa requer que o arquivo "Tudor.bin"
        >>> esteja presente no mesmo diretorio do programa.

       ********** Opcoes *********

        1. Remove registro
        2. Exibe arquivo
        3. Desfaz
        4. Refaz
        5. Encerra o programa

Escolha uma das opcoes disponiveis: 2

        Nome            Matricula       Nota 1  Nota 2
        ====            =========       ======  ======

    Henrique VIII         1029           9.5     9.0
    Catarina Aragon       1014           5.5     6.5
    Ana Bolena            1012           7.8     8.0
    Joana Seymour         1017           7.7     8.7
    Ana de Cleves         1022           4.5     6.0
    Catarina Howard       1340           6.0     7.7
    Catarina Parr         1440           4.0     6.0

        ********** Opcoes *********

        1. Remove registro
        2. Exibe arquivo
        3. Desfaz
        4. Refaz
        5. Encerra o programa

Escolha uma das opcoes disponiveis: 1
Digite uma matricula com exatamente 4 digitos:
        > 1012
```

```
       >>> Foi efetuada 1 remocao

        ********** Opcoes *********

        1. Remove registro
        2. Exibe arquivo
        3. Desfaz
        4. Refaz
        5. Encerra o programa

Escolha uma das opcoes disponiveis: 2

        Nome            Matricula       Nota 1  Nota 2
        ====            =========       ======  ======

    Henrique VIII         1029            9.5     9.0
    Catarina Aragon       1014            5.5     6.5
    Joana Seymour         1017            7.7     8.7
    Ana de Cleves         1022            4.5     6.0
    Catarina Howard       1340            6.0     7.7
    Catarina Parr         1440            4.0     6.0

        ********** Opcoes *********

        1. Remove registro
        2. Exibe arquivo
        3. Desfaz
        4. Refaz
        5. Encerra o programa

Escolha uma das opcoes disponiveis: 3

        >>> Operacao bem sucedida

        ********** Opcoes *********

        1. Remove registro
        2. Exibe arquivo
        3. Desfaz
        4. Refaz
        5. Encerra o programa

Escolha uma das opcoes disponiveis: 2

        Nome            Matricula       Nota 1  Nota 2
        ====            =========       ======  ======

    Henrique VIII         1029            9.5     9.0
    Catarina Aragon       1014            5.5     6.5
    Ana Bolena            1012            7.8     8.0
    Joana Seymour         1017            7.7     8.7
    Ana de Cleves         1022            4.5     6.0
    Catarina Howard       1340            6.0     7.7
    Catarina Parr         1440            4.0     6.0

        ********** Opcoes *********

        1. Remove registro
        2. Exibe arquivo
        3. Desfaz
        4. Refaz
        5. Encerra o programa
```

```
Escolha uma das opcoes disponiveis: 4

       >>> Operacao bem sucedida

        ********** Opcoes *********

        1. Remove registro
        2. Exibe arquivo
        3. Desfaz
        4. Refaz
        5. Encerra o programa

Escolha uma das opcoes disponiveis: 2

        Nome            Matricula       Nota 1  Nota 2
        ====            =========       ======  ======

    Henrique VIII         1029           9.5     9.0
    Catarina Aragon       1014           5.5     6.5
    Joana Seymour         1017           7.7     8.7
    Ana de Cleves         1022           4.5     6.0
    Catarina Howard       1340           6.0     7.7
    Catarina Parr         1440           4.0     6.0

        ********** Opcoes *********

        1. Remove registro
        2. Exibe arquivo
        3. Desfaz
        4. Refaz
        5. Encerra o programa

Escolha uma das opcoes disponiveis: 5

        >>> Obrigado por usar este programa. Bye.
```

PP11.2 Identificando e Convertendo Arquivos de Texto entre Famílias

Problema: Escreva um programa que classifica um arquivo de texto de acordo com as convenções de quebra de linha utilizadas pelas famílias de sistemas Windows/DOS, Unix e Macintosh (até Mac OS 9). O nome do arquivo deve ser um argumento de linha de comando. Após classificar o citado arquivo, o programa deve oferecer opções de conversão para uma das duas famílias às quais o arquivo não pertence. Por exemplo, se o arquivo pertencer à família Windows, o programa apresenta sua classificação e oferece o menu de opções:

```
>>> O arquivo é da família Windows.
>>> Deseja convertê-lo para:

1. Unix
2. Macintosh
3. Deixe-o em paz

>>> Escolha sua opcao:
```

PP11.3 Gerenciamento da Turma Tudor usando Array Estático

Problema: Considerando que o arquivo binário **`Tudor.bin`**, criado pelo programa da Seção 11.15.10, armazena dados de uma turma escolar. Escreva um programa que faça o seguinte:

(a) Lê estruturas do tipo **`tAluno`** do arquivo **`Tudor.bin`** e armazena-as num array estático ordenado por um campo de estrutura especificado pelo usuário por:

```
1. Nome
2. Matrícula
3. Primeira nota
4. Segunda nota
5. Média
6. Sem ordenação
```

(d) Apresenta um menu com as seguintes opções para o usuário:

```
A. Acrescenta um aluno
R. Remove um aluno
E. Exibe turma na tela
C. Consulta dados de aluno
L. Altera dados de aluno
T. Altera a ordenação da turma
I. Inverte a ordenação da turma
N. Encerra o programa
```

(e) Enquanto o usuário não escolher a opção de saída do programa, lê a opção escolhida pelo usuário e executa a operação correspondente.

(f) Logo antes de encerrar, se houve alteração de dados durante a execução do programa, o arquivo utilizado deve ser atualizado.