Capítulo 5

PROCESSAMENTO DE ARQUIVOS EM C

5.1 Introdução

Muitos programas precisam transferir informação da memória principal para um meio de armazenamento externo e vice-versa. Esta informação é armazenada em forma de **arquivos de dados** do programa. Estes arquivos de dados, que serão doravante referidos aqui apenas como **arquivos**, permitem que os programas acessem e alterem as informações contidas nos mesmos sempre que se fizer necessário. Programas também podem trocar informações com dispositivos de entrada e saída, como, por exemplo, modem e impressora. Como será visto mais adiante, a linguagem C não faz distinção entre qualquer fonte ou origem de dados. Portanto, no presente contexto, o termo *arquivo* refere-se não apenas a arquivos de dados propriamente ditos, como também a qualquer dispositivo que possa ser utilizado como fonte ou origem de dados de um programa.

As facilidades oferecidas para processamento de arquivos constituem uma das características mais importantes de qualquer linguagem de programação. Em C, processamento de arquivos é feita por meio de um conjunto de funções de biblioteca. A biblioteca padrão de C contém aproximadamente quarenta funções para executar operações de entrada/saída através da processamento de arquivos. Algumas das funções mais importantes para tratamento de arquivos serão vistas neste capítulo.

Antes de prosseguir, é interessante apontar que a linguagem C faz distinção entre dois tipos de arquivos: (1) **arquivos baseados em *streams*** e (2) **arquivos baseados em sistemas**. Arquivos baseados em *stream*s são mais fáceis de ser utilizados e programas que utilizam esta abordagem são mais portáteis do que aqueles que utilizam arquivos baseados em sistemas. Por outro lado, arquivos baseados em sistemas são mais intimamente relacionados com o sistema operacional em uso e, portanto, podem ser mais eficientes do que arquivos baseados em *stream*s. Aqui, apenas arquivos baseados em *stream*s serão estudados.

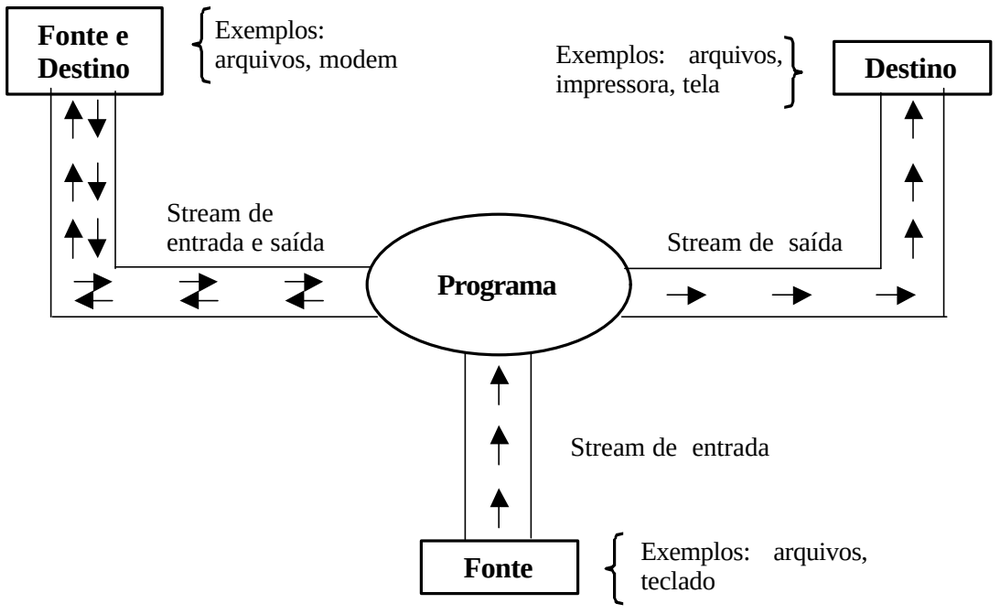
5.2 Streams e Buffers

5.2.1 Streams

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

Um *stream*[1] consiste de uma seqüência de bytes, similar a um arranjo unidimensional de bytes. Ler dados de um arquivo ou escrever dados em um arquivo significa ler de um *stream* ou escrever em um *stream*, respectivamente. A figura a seguir ilustra o conceito de stream e sua relação com um programa escrito em C, bem como com origens e fontes de dados:

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



Antes de processar um arquivo, deve-se declarar um *stream* e associá-lo ao arquivo. A declaração de um *stream* em C é feita por meio da declaração de um ponteiro para uma estrutura predefinida, chamada `FILE`. Por exemplo, a declaração:

```
FILE *meuStream;
```

estabelece um **ponteiro para *stream*** (ou, simplesmente, um ***stream***) que será utilizado subseqüentemente nas operações de processamento do arquivo ao qual o *stream* será associado (v. adiante).

A estrutura `FILE` encontra-se no módulo de biblioteca responsável por entrada/saída padrão. Para utilizá-la, deve-se incluir o arquivo `stdio.h`. `FILE` é uma estrutura complexa, cujos campos contêm informações sobre o arquivo associado. A implementação destes campos é muito dependente do sistema operacional no qual o programa é executado. Portanto, estes campos não devem ser manipulados diretamente pelo programa. Ao invés disso, o programa deve utilizar apenas ponteiros para *streams* em conjunto com funções de biblioteca para processamento de arquivos. Por exemplo, um dos campos da estrutura `FILE` é o **indicador de posição do arquivo** que indica a posição do próximo byte a ser lido do *stream* ou escrito para o *stream*. Quando desejar mover este indicador de posição, o programador não deve fazê-lo diretamente através da alteração do valor do campo que contém esta informação. Ao invés disto, ele deve utilizar uma função de biblioteca apropriada para este objetivo.

Antes de efetivamente utilizar um *stream* declarado conforme visto acima, deve-se abri-lo com o uso da função **fopen**(), que associa o *stream* com um arquivo real. Por exemplo, considerando a declaração do *stream* `meuStream` do último exemplo, poder-se-ía abrir este *stream* através da chamada:

```
meuStream = fopen("arquivo1.dat", "w+");
```

A execução desta chamada preencheria os campos da estrutura `FILE` apontada por `meuStream` com informações específicas do arquivo `arquivo1.dat`. (O significado do segundo argumento da função **fopen**() será descrito mais adiante.)

Pode-se ter mais de um *stream* aberto num mesmo programa, mas o número de *stream*s que podem estar simultaneamente abertos varia de acordo com o sistema operacional utilizado.

5.2.2 Streams Padronizados

Existem três *stream*s-padrão em C que são automaticamente abertos para qualquer programa que usa o módulo stdio. Estes *stream*s são denominados:

- **sdtin** representando entrada padrão;
- **stdout** representando saída padrão; e
- **stderr** representando saída padrão de mensagens de erro.

Usualmente, estes *stream*s estão associados com o terminal do computador (tipicamente, **sdtin** está associado com o teclado, enquanto que **stdout** e **stderr** estão associados com a tela do computador). Entretanto, muitos sistemas operacionais permitem que estes *stream*s sejam redirecionados. Por exemplo, o programador pode desejar que as mensagens de erro sejam escritas num arquivo, ao invés de na tela do computador.

As funções de entrada e saída vistas anteriormente [por exemplo, **scanf**() e **printf**()] usam os *stream*s-padrão **sdtin** e **stdout**. Estas funções podem ler de ou escrever em arquivos através do redirecionamento destes *stream*s-padrão [usando a função **freopen**()]. Entretanto, uma forma mais fácil de ler de e escrever num arquivo é, respectivamente, com o uso das funções **fscanf**() e **fprintf**(), que permitem a especificação de *streams* para leitura e escrita, respectivamente.

5.2.3 Buffers

Um ***buffer*** é uma área de memória aonde dados de um arquivo são armazenados temporariamente. O uso de *buffers* permite que o acesso aos dispositivos de entrada/saída, 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** e **buffering de bloco**. Em *buffering* de linha, o sistema armazena caracteres até que um caractere de nova linha (representado pela seqüência de escape '`\n`') seja encontrado, ou até que o *buffer* esteja cheio. Este tipo de *buffering* é utilizado, por exemplo, quando dados são lidos do teclado. Estes dados são armazenados num *buffer* até que um caractere de nova linha seja introduzido (por exemplo, por meio da digitação de `<ENTER>` ou `<RETURN>`). Quando isto acontece, os dados (caracteres) digitados na linha antes do caractere de nova linha são enviados para o programa.

Em *buffering* de bloco, o sistema armazena caracteres até que um bloco inteiro seja preenchido (independentemente de o sistema encontrar o caractere '`\n`' ou não). O tamanho padrão de um bloco é definido de acordo com o sistema operacional utilizado (tipicamente, 512 ou 1024 bytes). Por padrão, todos os *stream*s associados a arquivos de dados têm *buffering* de bloco. *Stream*s associados a um terminal de computador (por exemplo, teclado e tela) têm *buffering* de linha, ou não têm *buffering* (v. adiante), dependendo da implementação.

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

```
fflush(stdout);
```

força o sistema a descarregar a área de buffer associada com o stream de saída padrão `stdout`, enviando seu conteúdo para o meio de saída correspondente. Note, entretanto, que a função **fflush**() serve para descarregar apenas buffers associados com stream de saída.

Mais precisamente, não existe nenhuma função na biblioteca padrão de C que descarregue streams de entrada[2].

5.2.4 Streams sem Buffering

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

5.2.5 Formatos de Streams

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

Num *stream* de texto, os bytes que constituem o *stream* são interpretados como caracteres, e o *stream* como um todo consiste de um conjunto de linhas, cada uma delas terminada pelo caractere de nova linha (`'\n'`). Os *stream*s-padrão **stdin**, **stdout** e **stderr** são *stream*s de texto. Em termos de portabilidade de programa, deve-se tomar cuidado ao utilizar *stream*s de texto, pois um programa que manipula *stream*s de texto pode funcionar bem sob um sistema operacional mas não funcionar tão bem em outro. Por exemplo, se o *stream* contém caracteres especiais (por exemplo, caracteres acentuados de Português), estes caracteres são interpretados de um modo dependente de plataforma.

Em *stream*s binários, não existe interpretação de conteúdo. Em outras palavras, num *stream* binário cada bit é lido exatamente como ele aparece no *stream*. *Stream*s binários são indicados para dados não-textuais, aonde é importante preservar o conteúdo exato do arquivo sendo processado.

Um arquivo de texto pode ser criado de duas maneiras: (1) por meio do uso de um editor de texto, e (2) através de um programa que escreve para o arquivo. Arquivos binários podem ser criados apenas através desta segunda maneira.

5.3 Abrindo e Fechando um Arquivo

5.3.1 Abrindo um Arquivo

Após declarar um *stream*, conforme foi visto na seção precedente, deve-se abri-lo, antes que se possa ler ou escrever no mesmo. Isto deve ser feito por meio da função de biblioteca **fopen**(). Esta função recebe dois argumentos: o primeiro argumento é um **nome de arquivo**, especificado de acordo com as regras do sistema operacional utilizado, e o segundo argumento é um **modo de acesso**. Ambos os argumentos devem ser *strings*.

Existem dois conjuntos de modos de acesso: um conjunto de modos de acesso para *stream*s de texto e outro para *stream*s binários. O conjunto de modos de acesso para *stream*s de texto é apresentado na **Tabela 1**.

---

[2] Alguns sistemas permitem que a função **fflush**() seja utilizada para descarga (ou melhor, limpeza) de buffers de entrada (por exemplo, `fflush(stdin)`), mas este uso da função **fflush**() não é portável.

ESPECIFICAÇÃO	DESCRIÇÃO
`r`	Abre um arquivo de texto existente apenas para leitura. A leitura começa no início do arquivo.
`w`	Cria um novo arquivo de texto apenas para escrita. Se o arquivo já existir, seu conteúdo será destruído. O indicador de posição do arquivo aponta inicialmente para o início do arquivo.
`a`	Abre um arquivo existente para acréscimo. É permitido escrever no arquivo apenas no final dele, mesmo que o indicador de posição do arquivo seja movido para outra posição antes do final do arquivo. Se o arquivo cujo nome foi especificado não existir, um novo arquivo com este nome será criado.
`r+`	Abre um arquivo existente para leitura e escrita. O indicador de posição do arquivo aponta inicialmente para o início do arquivo.
`w+`	Cria um novo arquivo de texto para leitura e escrita. Se o arquivo já existir, seu conteúdo será destruído. O indicador de posição do arquivo aponta inicialmente para o início do arquivo.
`a+`	Abre um arquivo existente ou cria um novo arquivo para acréscimo. Pode-se ler dados em qualquer parte do arquivo, mas dados podem ser escritos apenas no final do arquivo.

Tabela 1: Modos de Acesso para Arquivos

A única diferença entre os especificadores de modo de acesso para arquivos binários e aqueles apresentados na **Tabela 1** para arquivos de texto é que, os especificadores para arquivos binários têm acrescentados a letra `b` após as letras `r`, `w` e `a`. Por exemplo, para a criação de um novo arquivo binário para leitura e escrita, deve-se escrever o modo do arquivo como sendo `"wb+"`.

A função **fopen**() retorna um ponteiro para a estrutura `FILE` que pode ser posteriormente utilizado no programa para processar o arquivo. Se não for possível, por algum motivo, abrir o arquivo especificado, **fopen**() retorna um ponteiro nulo (`NULL`). É sempre importante, portanto, antes de tentar processar o arquivo, testar o valor retornado pela função com `NULL`. Por exemplo, o trecho de programa a seguir abre um arquivo chamado `teste.dat` para leitura apenas:

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

       ptrStream = fopen("teste.dat", "r");
       if (ptrStream == NULL)
          printf("O arquivo teste.dat nao pode ser aberto.\n")
       else {
          /* Aqui o arquivo pode ser processado normalmente */
          ...
       }
```

O teste para verificar se o ponteiro retornado por **fopen**() é nulo no trecho de programa acima pode ser escrito em conjunto com a própria chamada da função como:

```
if ((ptrStream = fopen("teste.dat", "r")) == NULL)
```

É importante observar que, nesta última instrução, os parênteses em torno da chamada da função são essencialmente necessários, pois o operador `==` tem precedência sobre o operador `=`. Portanto, sem os parênteses, `ptrStream` teria sempre atribuído o valor zero ou um, dependendo do fato de **fopen**() retornar um ponteiro válido ou nulo. Este é um erro muito freqüente em programação em C e não ocorre apenas com a função **fopen**(). Por questões de segurança, é melhor evitar unir uma chamada de função com o teste de seu valor retornado, como foi feito na última instrução **if** acima.

5.3.2 Fechando um Arquivo

Quando um programa não precisa mais processar um arquivo, deve-se fechá-lo utilizando a função de biblioteca **fclose**(). Esta função recebe como único argumento um ponteiro de *stream* associado a um arquivo aberto pela função **fopen**(). Por exemplo, considerando o trecho de programa do último exemplo pode-se fechar o arquivo associado a `ptrStream` utilizando a chamada:

```
fclose(ptrStream)
```

Um erro freqüente entre os iniciantes em C é utilizar o nome do arquivo como argumento, ao invés do ponteiro de *stream* [por exemplo, `fclose("teste.dat")`]. Isto produz um erro de compilação, pois apesar de um string ser interpretado como um ponteiro, este ponteiro, obviamente, não é compatível com um ponteiro para a estrutura `FILE`.

Ao fechar-se um arquivo, libera-se o espaço ocupado pela estrutura `FILE` para a qual o ponteiro de *stream* associado ao arquivo aponta. Também, se for o caso, a função **fclose**() *descarrega* o conteúdo da área de *buffer* para o arquivo armazenado no meio externo antes de liberar a área de buffer[3].

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

5.4 Processamento de Arquivos

Uma vez que o arquivo desejado tenha sido aberto conforme foi descrito na seção precedente, pode-se então utilizar o ponteiro de *stream* associado ao arquivo para processá-lo (i.e., ler ou escrever).

A **granulosidade** de uma operação de entrada/saída refere-se ao tamanho dos objetos manipulados de cada vez. Em C, existem três graus de granulosidade para processamento de arquivos:

- **Um caractere por vez**;
- **Uma linha por vez**; e
- **Um bloco por vez**.

---

[3] *Descarregar* uma área de *buffer* não é o mesmo que liberar o espaço de memória ocupado pelo *buffer*. *Descarregar o buffer* significa liberar seu conteúdo de modo que o *buffer* possa ser reutilizado. No caso de um arquivo aberto para escrita ou leitura/escrita, isto envolve a gravação do conteúdo do *buffer* no meio externo de armazenamento, e o subseqüente movimento do apontador de arquivo para o início do *buffer*. No caso de um arquivo aberto apenas para leitura, o conteúdo do *buffer* é simplesmente descartado.

A seguir serão apresentados exemplos de processamentos de arquivos em cada um destes níveis de granulosidade.

5.4.1 Processando um Caractere por Vez

Existem duas macros e duas funções para processamento de um caractere por vez num *stream*. Estas macros e funções são apresentadas na **Tabela 2**.

MACRO/FUNÇÃO	DESCRIÇÃO
getc()	Macro que lê um caractere do *stream*
fgetc()	Função que lê um caractere do *stream*
putc()	Macro que escreve um caractere no *stream*
fputc()	Função que escreve um caractere no *stream*

Tabela 2: Macros e Funções para Processamento de Caracteres

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

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

      if ((ptrEntrada = fopen(arquivoDeEntrada, "rb")) == NULL)
          return 0; /* Arquivo de entrada não pode ser aberto */

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

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

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

      return 1;
}
```

Há vários pontos interessantes a serem observados no exemplo acima. Primeiro, note que ambos os arquivos foram abertos em modo binário, uma vez que se está lendo e escrevendo sem preocupação com a estrutura de linhas que os arquivos possam porventura apresentar[4].

Outros pontos interessantes são os funcionamentos das macros **getc**() e **putc**(), que recebem como argumento um *stream*, e lêem ou escrevem, respectivamente, no *stream* passado como argumento. Antes de retornarem, estas funções movem o apontador de posição do *stream* para o próximo caractere (byte).

A função **feof**() utilizada como teste para saída do **while** da função do último exemplo retorna um valor diferente de zero logo após o final do *stream* que ela recebe como

[4] Se o arquivo de entrada for realmente um arquivo de texto, o arquivo de saída também o será, pois o caractere '\n' será naturalmente lido e copiado.

argumento ser atingido. Aliás, esta função é uma fonte de grande confusão e causa de erro em programas escritos por iniciantes em C e até mesmo por autores de livros sobre esta linguagem. Esta confusão é provavelmente derivada da expectativa de esta função funcionar do mesmo modo que uma função similar (também denominada `EOF()`) existente na linguagem Pascal. Ocorre, porém, que a função `EOF()` de Pascal *antecipa* o encontro do final do arquivo, enquanto que a função `eof()` de C sinaliza o final do arquivo apenas quando há uma tentativa de cruzá-lo. Ou seja, a função `EOF()` de Pascal retorna ***true*** quando a *próxima* operação de leitura irá ultrapassar o final do arquivo, mas a função `eof()` de C retorna um valor diferente de zero apenas quando se tenta ler além do final do arquivo. Com o raciocínio utilizado na linguagem Pascal em mente, um programador inexperiente em C seria tentado a escrever o laço **while** da função do último exemplo simplesmente como[5]:

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

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

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

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

provavelmente não produzirá o resultado desejado. O problema aqui é que o laço **while** será terminado quando o primeiro caractere que tenha um valor inteiro igual ao da constante `EOF` for encontrado, o que pode não necessariamente representar o final do arquivo. Por exemplo, freqüentemente, a constante `EOF` é definida como -1, que não corresponde ao valor numérico de nenhum caractere; logo, `EOF` pode ser identificada inequivocamente num arquivo que contém apenas caracteres (i.e., num arquivo-texto). No entanto, este valor pode ser encontrado (talvez várias vezes) num arquivo contendo dados arbitrários (i.e., bytes que não correspondem necessariamente a caracteres). Entretanto, se o *stream* utilizado fosse um *stream* de texto, ambas as construções seriam válidas.

5.4.2 Processando uma Linha por Vez

O processamento de um arquivo linha por linha é conveniente apenas para arquivos de texto. Existem duas funções de biblioteca que lêem e escrevem uma linha num dado *stream* de texto. Estas funções são, respectivamente, **fgets**() e **fputs**(). A função **fgets**() tem o seguinte protótipo:

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

onde:

- `s` é um ponteiro para o primeiro elemento de um arranjo de caracteres para onde os caracteres da linha lida serão armazenados.

---

[5] De fato, Darnell e Margolis fazem essa bobagem em seu livro (v. Bibliografia).

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

- `n` representa o número máximo de caracteres a serem lidos.
- `stream` representa o *stream* de onde será feita a leitura.

A função **fgets**() lê caracteres até atingir um caractere de nova linha ('`/n`'), o final do arquivo ou o número máximo de caracteres especificado. Esta função automaticamente escreve um caractere nulo ('`\0`') após o último caractere armazenado no arranjo `s`[7]. Quando o final do arquivo é atingido durante uma chamada de **fgets**(), esta função retorna `NULL`.

É importante notar que existem diferenças entre **fgets**() e a **gets**(), a função que lê *strings* do teclado. Ambas as funções acrescentam um caractere nulo após o último caractere armazenado no arranjo. Entretanto, **gets**() não armazena nenhum caractere de nova linha no arranjo, enquanto que **fgets**() escreve este caractere no arranjo se o mesmo for encontrado. Também, **fgets**() permite que se especifique o número máximo de caracteres a serem lidos, enquanto a função **gets**() não permite isto.

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

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

onde:

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

A função **fputs**() escreve todos os caracteres do arranjo `s` no *stream* até que o caractere nulo seja encontrado (este último não é escrito no *stream*). É crucial, portanto, que o arranjo de caracteres contenha o caractere nulo. A função **fputs**() retorna zero quando a escrita é bem sucedida; caso contrário, ela retorna um valor diferente de zero. Note que **fputs**() não insere um caractere de nova linha após a escrita do último caractere do string, como faz a função **puts**() que escreve no meio de saída padrão.

Para que a função do exemplo da seção anterior possa ser rescrita para processar os arquivos uma linha por vez, os arquivos devem ser abertos em modo texto. Os arquivos em si não precisam ser do tipo texto, mas o requerimento de abrir os *stream*s em modo texto deve-se ao fato de a função **fgets**() não funcionar apropriadamente no modo binário. A função `CopiaArquivoPorLinha()` apresentada a seguir copia o conteúdo de um arquivo em outro linha por linha:

```
#define  TAMANHO_DA_LINHA   100

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

    if ((ptrEntrada = fopen(arquivoDeEntrada, "r")) == NULL)
       return 0;    /* Arquivo de entrada não pode ser aberto */

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

    while ( fgets(linha, TAMANHO_DA_LINHA - 1, ptrEntrada) != NULL )
       fputs(linha, ptrSaida);

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

    return 1;
```

[7] É por causa disso que, na função a ser apresentada nesta seção, o número máximo de caracteres foi especificado como sendo o tamanho do arranjo menos 1.

```
}
```

Compare a função `CopiaArquivoPorLinha()` apresentada acima com a função `CopiaArquivoPorCaractere()` e convença-se de entendeu as diferenças entre as duas.

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

5.4.3 Processando um Bloco por Vez

Em termos de granulosidade, dados podem ser acessados (i.e., lidos ou escritos) em porções denominadas **blocos**. Pode-se imaginar um bloco como sendo simplesmente um arranjo unidimensional, de modo que, quando se lê ou escreve um bloco, é necessário especificar o número de elementos no bloco e o tamanho de cada elemento. As funções de biblioteca para entrada e saída utilizando blocos são as funções **fread**() e **fwrite**(), respectivamente.

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

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

onde,

- `ptr` é um ponteiro para um arranjo aonde o bloco será armazenado.
- `tamanho` é o tamanho de cada elemento do arranjo.
- `nItens` é o número de itens a serem lidos (pode ser menor que o número de elementos do arranjo).
- `stream` representa o *stream* aonde será feita a leitura.

A função **fread**() retorna o número de itens que foram realmente lidos. Este valor deverá ser o mesmo que o valor do terceiro argumento da função, a não ser que ocorra um erro ou o final do *stream* seja atingido.

O protótipo da função **fwrite**() é muito parecido com o protótipo de **fread**(). Entretanto, estas funções diferem bastante em termos de funcionamento, pois **fwrite**() faz o contrário de **fread**(). Isto é, **fwrite**() lê de um arranjo e escreve em um *stream*. Em qualquer situação, o programador deve tomar cuidado para não especificar um número de elementos (terceiro argumento) que ultrapasse o número de elementos do arranjo.

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

```
#define  TAMANHO_DO_BLOCO   512

typedef  char  tDados;

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

   if ((ptrEntrada = fopen(arquivoDeEntrada, "rb")) == NULL)
      return 0;     /* Arquivo de entrada não pode ser aberto */

   if ((ptrSaida = fopen(arquivoDeSaida, "wb")) == NULL) {
      fclose(ptrEntrada);    /* Arquivo de saída não pode ser aberto */
      return 0;
```

```
   }

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

   fclose(ptrEntrada);
   fclose(ptrSaida);

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

   return 1;
}
```

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

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

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

Finalmente, é interessante notar que o tamanho do bloco utilizado em operações de entrada/saída de blocos é independente do tamanho utilizado para a área de *buffer*. Por exemplo, considerando o tamanho de bloco definido no último exemplo como 512 e o tamanho do *buffer* como sendo 1024 bytes, o sistema continuará a obter 1024 bytes no meio de armazenamento e armazená-los em memória. Entretanto, apenas os primeiros 512 bytes estarão disponíveis para a função **fread**() na primeira chamada desta função; na próxima chamada, o sistema buscará os seguintes 512 em memória (e não em disco) e colocará em disponibilidade para **fread**(); e assim por diante. Portanto, o tamanho do bloco escolhido no último exemplo não afeta o número de operações de entrada/saída do dispositivo utilizado.

5.5 Entrada/Saída sem Buffering

A biblioteca de C oferece meios para modificar o tamanho do *buffer*, mas deve-se utilizar esta facilidade com cuidado, pois, muitas vezes, o tamanho padrão de *buffer* foi previamente escolhido para otimizar operações de entrada e saída levando em consideração o sistema operacional utilizado. Provavelmente, a única vez que se precisa modificar o tamanho do *buffer* é quando se deseja utilizar *stream*s sem *buffering* (v. **Seção 5.2.4**). Isto é, quando se deseja que a entrada do usuário ou a saída do programa seja processada imediatamente. Normalmente, **stdin** tem *buffering* de linha, e requer que o usuário digite um caractere de nova linha antes que a entrada seja enviada para o programa. Em muitas aplicações interativas (por exemplo, um editor de texto), este comportamento é indesejável[8].

[8] Observe, entretanto, que tornar **stdin** sem buffering é dependente de implementação. Isto é, apesar de a maioria dos sistemas prover entrada de dados sem buffering (por exemplo, a função `getch()` encontrada no ambiente Borland C++), não existe nenhuma função na biblioteca padrão de C que ofereça esta facilidade.

Para anular o *buffering*, pode-se utilizar a função **setbuf**() ou a função **setvbuf**(). A função **setbuf**() recebe dois argumentos: o primeiro é um ponteiro de *stream*, e o segundo é um arranjo de caracteres que servirá como novo *buffer* após a chamada. Se o segundo argumento for um ponteiro nulo, não haverá *buffering*. Por exemplo,

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

transforma o stream apontado por `meuStream` num *stream* sem *buffering*. A função **setbuf**() não retorna nenhum valor.

A função **setvbuf**() é similar a **setbuf**(), mas é um tanto mais elaborada. Esta função recebe mais dois argumentos adicionais. Um destes argumentos adicionais permite a especificação do tipo desejado de *buffering* (linha, bloco, ou nenhum). O outro argumento refere-se ao tamanho do arranjo a ser utilizado como *buffer*. O tipo de *buffer* deve ser especificado com uma das macros (definidas no arquivo `stdio.h`) apresentadas na **Tabela 3**.

MACRO	BUFFERING
_IOFBF	Bloco
_IOLBF	Linha
_IONBF	Nenhum

Tabela 3: Macros de Buffering

Por exemplo, para obter um *stream* sem *buffering* para a entrada padrão, poder-se-ía escrever:

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

A função **setvbuf**() retorna um valor diferente de zero se ela obtiver êxito ao estabelecer o novo *buffer*; caso contrário, ela retorna zero.

5.6 Acesso Aleatório

Nos exemplos de processamento de arquivos apresentados até aqui, os arquivos são acessados **seqüencialmente**; i.e., todos os bytes de um arquivo são processados um após o outro, do primeiro até o último byte. Este tipo de processamento é conveniente para os exemplos apresentados acima, pois estes copiavam o conteúdo de um arquivo para outro e, portanto, todos os bytes precisavam ser lidos e escritos. Existem aplicações, entretanto, em que se deseja acessar um conjunto particular de bytes no interior do arquivo. Este tipo de processamento de arquivo é denominado processamento com **acesso aleatório** (ou **acesso direto**).

Existem duas funções na biblioteca padrão de C que podem ser utilizadas para o acesso aleatório de arquivos: **fseek**() e **ftell**(). O protótipo de **fseek**() é:

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

onde:

- `stream` é um ponteiro de *stream*;

• `distancia` é uma distância (positiva ou negativa), medida em bytes, a partir do terceiro argumento, para onde o apontador de posição do arquivo será movido;
• `deOnde` é a posição inicial a partir de onde a distância (segundo argumento) será medida. Este argumento pode assumir um dos valores (macros definidas em `stdio.h`) apresentados na **Tabela 4**:

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

Tabela 4: Macros de Posição em Arquivos

Se a função **fseek**() conseguir deslocar o apontador de posição do arquivo para a posição solicitada, ela retorna zero; caso contrário, ela retorna um valor diferente de zero. Por exemplo, a chamada de **fseek**() apresentada a seguir:

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

if (meuStream)
    retorno = fseek(meuStream, 10, SEEK_SET);
```

moveria o apontador de posição de arquivo associado a `meuStream` para o byte de índice 10 neste *stream*. É importante observar que os bytes num arquivo são numerados como os elementos de um arranjo (i.e., a partir de zero); portanto, o byte de índice 10 é, na realidade, o 11º byte no *stream*.

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

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

retornaria um valor diferente de zero indicando que a solicitação não pode ser atendida, pois quando um *stream* é aberto apenas para leitura não se pode mover além da marca de final de arquivo. Portanto, se a macro `SEEK_END` for utilizada como valor do terceiro argumento de **fseek**(), a distância (segundo argumento) deve ser negativa. Da mesma forma, se a macro `SEEK_SET` for utilizada, a distância deve ser sempre positiva.

Para *stream*s binários, a distância utilizada com **fseek**() pode ser qualquer valor inteiro que não faça o apontador de posição do arquivo ultrapassar os limites do arquivo. Para *streams* de texto, esta distância deve ser nula ou um valor retornado pela função **ftell**().

A função **ftell**() recebe apenas um argumento, que é um ponteiro de *stream*, e retorna a posição corrente do apontador de posição do *stream*. Esta função é usada principalmente para fazer o apontador de posição retornar para uma determinada posição após uma ou mais operações de entrada/saída. Por exemplo, na maioria dos editores de texto, existe um comando que permite ao usuário procurar por uma determinada palavra. Quando a procura falha, o cursor (e o apontador de posição no arquivo) deve retornar à posição aonde a busca foi iniciada. Isto poderia ser implementado como esquematizado no trecho de programa a seguir, que assume a existência de uma função `Busca()` que retorna `1` se a procura teve êxito (i.e., se a palavra foi encontrada), e `0` em caso contrário:

```
long posicaoCorrente = ftell(meuStream);

if (!Busca(palavra))
   fseek(meuStream, posicaoCorrente, SEEK_SET);
```

Note que a posição retornada por **ftell**() é sempre medida a partir do início do arquivo. Para *stream*s binários, o valor retornado por **ftell**() representa o número real de bytes a partir do início do arquivo. Para *stream*s de texto, o valor retornado por **ftell**() representa um valor dependente de implementação que faz sentido apenas quando utilizado como distância para uma chamada da função **fseek**(). Por exemplo, considere o programa a seguir:

```
#include <stdio.h>

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

   posicaoAtual = ftell(stream);
   fseek(stream, 0L, SEEK_END);
   tamanho = ftell(stream);
   fseek(stream, posicaoAtual, SEEK_SET);

   return tamanho;
}

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

   stream = fopen(strArquivo, "w+");
   printf("O apontador de arquivo esta' na posicao:
            %ld\n", ftell(stream));
   fprintf(stream, "Isto e' um teste.");
   printf("Agora, o apontador de arquivo esta' na posicao:
            %ld\n", ftell(stream));
   printf("\nTamanho do arquivo %s em bytes: %ld",
            strArquivo, TamanhoDeArquivo(stream));
   fclose(stream);

   return 0;
}
```

O programa do último exemplo apresenta a posição do apontador de arquivo em dois instantes e o tamanho do arquivo. Este programa pode funcionar em alguns sistemas, mas ***não*** tem portabilidade pelas razões expostas acima.

5.7 Classificação Externa de Arquivos por Indexação

Esta seção apresenta um exemplo mais completo de processamento de arquivo. Suponha que se tenha um arquivo contendo um grande número de registros do tipo `tRegistro` definido a seguir:

```
#define  NUMERO_MAXIMO_DE_REGISTROS   200
#define  COMPRIMENTO_DO_NOME          20

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

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

Suponha ainda que se deseje imprimir este arquivo em ordem alfabética de nomes. Uma maneira óbvia de se resolver este problema seria ler todo o arquivo num arranjo de estruturas do tipo `tRegistro` e classificar este arranjo em ordem alfabética (usando, por exemplo, uma função similar à função `BubbleSort()` apresentada na **Seção 3.12**).

Entretanto, esta solução não é satisfatória se o número de registros e o tamanho dos registros forem realmente grandes, porque pode requer uma grande quantidade de memória para a operação. Esta forma de classificação é denominada de **classificação interna**, pois mantém todos os dados a serem classificados em memória. Uma outra forma de classificar um arquivo é por meio de **classificação externa**, na qual parte dos dados a serem classificados não precisam ser carregados para a memória principal. Nesta seção, será apresentado um método de classificação externa chamado **classificação por indexação** (i.e., por meio de índices).

O método de classificação por indexação lê apenas a parte do registro (denominada **chave**) que se deseja classificar, e emparelha cada chave com um **índice** que aponta para o registro que contém a respectiva chave no arquivo. A vantagem deste método é que classificar uma coleção de chaves e índices é mais econômico em termos de memória do que classificar uma coleção de registros inteiros (quanto maior for o tamanho do registro em relação à chave, maior será a economia de memória). Suponha, como ilustração, que o arquivo contenha apenas os quatros registros seguintes (obviamente, estes registro não aparecem formatados desta maneira no arquivo):

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

Estes registros seriam indexados a partir de 0 até o número de registros menos 1, e cada índice seria associado à sua respectiva chave (neste caso, o nome da pessoa), conforme ilustrado nas seguintes tabelas:

Índice	Chave
0	Jose da Silva
1	Maria da Silva
2	Joao da Silva
3	Manoel da Silva

Após classificar estes pares de chaves/índices, ter-se–ía a seguinte ordem:

Índice	Chave
2	Joao da Silva
0	Jose da Silva
3	Manoel da Silva
1	Maria da Silva

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

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

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

A função `ColetaPares()` a seguir lê um número de caracteres igual a `COMPRIMENTO_DO_NOME`, correspondente ao campo nome (a chave) de cada registro. Então, esta função move o indicador de posição do arquivo para o início do próximo registro com a função **fseek**(). Deste modo, evita-se a leitura de partes do registro que não

interessam[9]. A função `ColetaPares()` recebe como parâmetros um ponteiro de *stream*, um arranjo de elementos do tipo `tPares` definido acima, e o número máximo de registros que devem ser lidos. Esta função retorna o número de registros lidos. A função também inclui um teste para assegurar que cada chamada de **fseek**() obtém êxito; quando este teste falha, a função retorna zero.

```
/****
 *
 * ColetaPares()
 *
 * Lê um número máximo de registros dado por maximoDeRegistros do arquivo
 * associado ao ponteiro stream, e armazena a chave de cada registro no
 * arranjo arranjoDePares. Retorna o número de registros lidos, se não
 * houver erro; caso contrário, retorna zero.
 *
 ****/

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

    for (i = 0; i < maximoDeRegistros; i++) {
                   /* Lê campo nome de cada registro e armazena   */
                   /* no campo chave do arranjo de pares          */
        nItenLidos = fread( arranjoDePares[i].chave, 1, COMPRIMENTO_DO_NOME,
                            stream );
        if (nItenLidos < COMPRIMENTO_DO_NOME) /* O final de arquivo foi */
            break;          /* atingido ou ocorreu algum erro de leitura */

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

        distancia += sizeof(tRegistro); /* Posiciona apontador de arquivo */
                                        /* para próxima leitura            */
        if (fseek(stream, distancia, SEEK_SET) && !feof(stream))
            return 0; /* Não foi possível mover o apontador de posição */
    } /* for */

    return  i;
}
```

Note que, na função `ColetaPares()` acima, os valores da variável `distancia` são computados levando em consideração o tamanho da estrutura `tRegistro`.

A próxima tarefa é classificar o arranjo de estruturas `tPares`. A função a seguir utiliza a função de biblioteca **qsort**() (incluir `stdlib.h`) para classificar o arranjo dado e retorna um ponteiro para o arranjo classificado.

```
/****
 *
 * ClassificaPares()
 *
 * Classifica um arranjo de elementos do tipo tPares  com um número de
 * elementos dado por numeroDeElementos. Esta função simplesmente chama
 * a função de biblioteca qsort() para classificar o arranjo dado.
 *
 ****/
void   ClassificaPares(tPares  arranjoDePares[], unsigned  numeroDeElementos)
{
       /* Alusão da função que será utilizada com a função qsort() */
   int FuncaoDeComparacao(const void *p1, const void *p2);

   qsort(arranjoDePares, numeroDeElementos,sizeof(tPares), FuncaoDeComparacao);
}


/****
 *
 * FuncaoDeComparacao()
 *
 * Função de comparação utilizada pela função qsort(). Utiliza a função
 * de comparação de strings strcmp() (incluir string.h) para comparar os
```

[9] Na realidade, o mecanismo de entrada/saída com *buffering* faz com que sejam lidos 512 ou 1024 bytes, de modo que os registros inteiros são lidos de qualquer modo. Dentro do *buffer*, entretanto, é necessário acessar apenas o primeiro campo de cada registro.

```
 * campos nome dos registros.
 *
 ****/
int  FuncaoDeComparacao(const void *p1, const void *p2)
{
    return   strcmp(((tPares *)p1)->chave, ((tPares *)p2)->chave);
}
```

Antes de prosseguir com o exemplo corrente, a função de biblioteca **qsort**(), utilizada por `ClassificaPares()` para classificar o arranjo de pares nome/índice em ordem crescente, merece alguns comentários . A função **qsort**() tem o seguinte protótipo:

```
void   qsort( void *arranjo, size_t  nItens, size_t  tamanhoDoElemento,
              int  (*FComparacao)(const  void*, const  void*) );
```

O primeiro argumento de **qsort**() é o arranjo a ser classificado, o segundo argumento é o número de elementos deste arranjo e o terceiro argumento é o tamanho de cada elemento do arranjo. A função **qsort**() classifica o arranjo de acordo com uma função de comparação cujo endereço (v. Seção 3.10) é fornecido como seu quarto argumento. Esta função de comparação, que deve ser provida pelo programador, recebe dois ponteiros genéricos como argumentos e retorna um número inteiro cujo valor deve ser: menor que zero, se o primeiro argumento é menor do que o segundo; maior que zero, se o primeiro argumento é maior do que o segundo; e `0` se os dois argumentos são iguais. A função `FuncaoDeComparacao()` definida acima satisfaz este critério pois ela utiliza a função **strcmp**() cujos valores de retorno estão de acordo com esta especificação.

Exercício: Pense numa maneira fácil de classificar o arranjo dado em ordem decrescente, ao invés de em ordem crescente como foi feito acima.

Voltando ao exemplo de classificação por indexação, o próximo passo é imprimir os registros do arquivo em ordem crescente. A função `ImprimeRegistrosClassificados()` apresentada a seguir faz exatamente isto:

```
/****
 *
 * ImprimeRegistrosClassificados()
 *
 * Imprime os registros ordenados. Retorna 0 se ocorrer algum erro.
 *
 ****/
unsigned ImprimeRegistrosClassificados( FILE  *stream,
                                        tPares  arranjoDePares[],
                                        unsigned  numeroDeElementos )
{
    tRegistro   umRegistro;
    unsigned    i;

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

                   /* Lê o registro especificado  */
        fread(&umRegistro, sizeof(tRegistro), 1, stream);
                   /* Imprime o registro recuperado */
        printf("%s, \t %d/%d/%d\n", umRegistro.nome, umRegistro.data.dia,
              umRegistro.data.mes, umRegistro.data.ano);
    }

    return 1;
}
```

Note que, na função `ImprimeRegistrosClassificados()` acima, a posição inicial de cada registro é calculada multiplicando-se o índice do registro pelo tamanho de cada estrutura. Isto é, a posição é dada por:

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

Para completar o exemplo, resta apenas escrever a função **main**() que chama as funções apresentadas acima. A função **main**() apresentada a seguir foi escrita de tal modo que o nome do arquivo, cujos registros serão classificados, possa ser passado como argumento (v. **Seção 3.13**).

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

   if (argc != 2){ /* Deveria haver dois nomes na chamada deste programa: */
       printf("Erro: nome do arquivo ausente\n");  /* os nomes do programa */
                                                   /* e do arquivo         */
       return 1;
   }

   if (!(stream = fopen(argv[1], "rb"))) { /* Arquivo especificado na linha */
       printf("Erro abrindo arquivo %s\n", argv[1]); /*  de comando não  */
       return 1;                                    /* pode ser aberto  */
   }

   if ( numeroDeRegistrosLidos =
               ColetaPares(stream, pares, NUMERO_MAXIMO_DE_REGISTROS) ) {
      ClassificaPares(pares, numeroDeRegistrosLidos);
      if ( !ImprimeRegistrosClassificados(stream, pares,
                                          numeroDeRegistrosLidos) ) {
         printf("Erro processando arquivo.\n");  /* Ocorreu algum erro */;
         fclose(stream);
         return 1;
      }
   }

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

5.8 Exercícios de Revisão

1. Qual é a principal vantagem decorrente do uso de arquivos de dados?

2. Descreva as diferentes maneiras nas quais arquivos de dados podem ser organizados em C.

3. O que é ***buffer***?

4. Qual é o propósito de uma área de *buffer* quando se trabalha com um arquivo de dados?

5. O que é um *stream*?

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

7. Qual é o significado do identificador `FILE` no processamento de *streams*? Aonde `FILE` é definido?

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

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

10. Descreva os diferentes tipos de arquivos que podem ser especificados pela função **fopen**().

11. (a) Qual é o propósito da função **fclose**()? (b) É sempre necessário o uso desta função num programa que manipula arquivos?

12. (a) Descreva duas formas diferentes de criar um arquivo de dados num programa em C. (b) É possível utilizar ambas as formas com arquivos não-formatados (i.e., binários)?

13. (a) Descreva duas formas diferentes de atualizar (i.e., modificar o conteúdo) de um arquivo de dados. (b) Que abordagem é melhor e por que?

14. Para que tipos de aplicações arquivos de dados formatados são convenientes?

15. Para que tipos de aplicações arquivos de dados não-formatados são convenientes?

16. Qual é o propósito da função de biblioteca **feof**()?

5.9 Exercícios de Programação

EP5.1) Escreva um programa em C que conte o número de caracteres, palavras e linhas de um arquivo-texto.

EP5.2) Escreva um programa em C que abre um arquivo-texto especificado em linha de comando e o imprime na tela do computador em sentindo inverso (i.e., de trás para a frente). (**Sugestão**: Estude a descrição da função **main**() na **Seção 3.13**.)