9

E/S com Arquivo

Como você provavelmente sabe, a linguagem C não contém nenhum comando de E/S. Ao contrário, todas as operações de E/S ocorrem mediante chamadas a funções da biblioteca C padrão. Essa abordagem faz o sistema de arquivos de C extremamente poderoso e flexível. O sistema de E/S de C é único, porque dados podem ser transferidos na sua representação binária interna ou em um formato de texto legível por humanos. Isso torna fácil criar arquivos que satisfaçam qualquer necessidade.

E/S ANSI *Versus* E/S UNIX

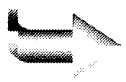
O padrão C ANSI define um conjunto completo de funções de E/S que pode ser utilizado para ler e escrever qualquer tipo de dado. Em contraste, o antigo padrão C UNIX contém dois sistemas distintos de rotinas que realizam operações de E/S. O primeiro método assemelha-se vagamente ao definido pelo padrão C ANSI e é denominado de *sistema de arquivo com buffer* (algumas vezes os termos *formatado* ou *alto nível* são utilizados para referenciá-lo). O segundo é o sistema de arquivo tipo UNIX (algumas vezes chamado de *não formatado* ou *sem buffer*) definido apenas sob o antigo padrão UNIX. O padrão ANSI não define o sistema sem buffer porque, entre outras coisas, os dois sistemas são amplamente redundantes e o sistema de arquivo tipo UNIX pode não ser relevante a certos ambientes que poderiam, de outro modo, suportar C. Este capítulo dá ênfase ao sistema de arquivo C ANSI. O fato de o ANSI não ter definido o sistema de E/S tipo UNIX sugere que seu uso irá declinar. De fato, seria muito difícil justificar

seu uso em qualquer projeto atual. Porém, como as rotinas tipo UNIX foram utilizadas em milhares de programas em C existentes, elas são discutidas brevemente no final deste capítulo.

E/S em C *Versus* E/S em C++

Como C forma o embasamento de C++ (C melhorada através da orientação a objetos), às vezes surge a pergunta sobre como se relaciona o sistema de E/S de C com C++. A breve digressão a seguir esclarece este ponto.

C++ suporta todo o sistema de arquivos definido pelo C ANSI. Assim, se você portar algum código para C++ em algum momento no futuro, não precisará modificar todas as suas rotinas de E/S. No entanto, C++ também define seu próprio sistema de E/S, orientado a objetos, que inclui tanto funções quanto operadores de E/S. O sistema de E/S C++ duplica por completo a funcionalidade do sistema de E/S C ANSI. Em geral, se usar C++ para escrever programas orientados a objetos, você vai querer usar o sistema de E/S orientado a objetos. Em outros casos, você é livre para escolher usar o sistema de arquivos orientado a objetos ou o sistema de arquivos C ANSI. Uma vantagem de usar este último é que atualmente ele está padronizado e é reconhecido por todos os compiladores C e C++ atuais.



NOTA: Para uma análise completa de C++, incluindo seu sistema de E/S orientado a objetos, consulte C++ — The Complete Reference, de Herbert Schildt (Berkeley, CA: Osborne McGraw-Hill).

Streams e Arquivos

Antes de começar nossa discussão do sistema de arquivo C ANSI, é importante entender a diferença entre os termos *streams* e *arquivos*. O sistema de E/S de C fornece uma interface consistente ao programador C, independentemente do dispositivo real que é acessado. Isto é, o sistema de E/S de C provê um nível de abstração entre o programador e o dispositivo utilizado. Essa abstração é chamada de *stream* e o dispositivo real é chamado de *arquivo*. É importante entender como streams e arquivos se integram.

Streams

O sistema de arquivos de C é projetado para trabalhar com uma ampla variedade de dispositivos, incluindo terminais, acionadores de disco e acionadores de fita.

Embora cada um dos dispositivos seja muito diferente, o sistema de arquivo com buffer transforma-os em um dispositivo lógico chamado de stream. Todas as streams comportam-se de forma semelhante. Pelo fato de as streams serem amplamente independentes do dispositivo, a mesma função pode escrever em um arquivo em disco ou em algum outro dispositivo, como o console. Existem dois tipos de streams: texto e binária.

Streams de Texto

Uma *stream de texto* é uma seqüência de caracteres. O padrão C ANSI permite (mas não exige) que uma stream de texto seja organizada em linhas terminadas por um caractere de nova linha. Porém, o caractere de nova linha é opcional na última linha e é determinado pela implementação (a maioria dos compiladores C não termina streams de texto com caracteres de nova linha). Em uma stream de texto, certas traduções podem ocorrer conforme exigido pelo sistema host. Por exemplo, uma nova linha pode ser convertida em um par retorno de carro/alimentação de linha. Portanto, poderá não haver uma relação de um para um entre os caracteres que são escritos (ou lidos) e aqueles nos dispositivos externos. Além disso, devido a possíveis traduções, o número de caracteres escritos (ou lidos) pode não ser o mesmo que aquele encontrado no dispositivo externo.

Streams Binárias

Uma *stream binária* é uma seqüência de bytes com uma correspondência de um para um com aqueles encontrados no dispositivo externo — isto é, não ocorre nenhuma tradução de caracteres. Além disso, o número de bytes escritos (ou lidos) é o mesmo que o encontrado no dispositivo externo. Porém, um número definido pela implementação de bytes nulos pode ser acrescentado a uma stream binária. Esses bytes nulos poderiam ser usados para aumentar a informação para que ela preenchesse um setor de um disco, por exemplo.

Arquivos

Em C, um *arquivo* pode ser qualquer coisa, desde um arquivo em disco até um terminal ou uma impressora. Você associa uma stream com um arquivo específico realizando uma operação de abertura. Uma vez o arquivo aberto, informações podem ser trocadas entre ele e o seu programa.

Nem todos os arquivos apresentam os mesmos recursos. Por exemplo, um arquivo em disco pode suportar acesso aleatório enquanto um teclado não pode. Isso revela um ponto importante sobre o sistema de E/S de C: todas as streams são iguais, mas não todos os arquivos.

Se o arquivo pode suportar acesso aleatório (algumas vezes referido como *solicitação de posição*), abrir esse arquivo também inicializa o *indicador de posição no arquivo* para o começo do arquivo. Quando cada caractere é lido ou escrito no arquivo, o indicador de posição é incrementado, garantindo progressão através do arquivo.

Um arquivo é desassociado de uma stream específica por meio de uma operação de fechamento. Se um arquivo aberto para saída for fechado, o conteúdo, se houver algum, de sua stream associada será escrito no dispositivo externo. Esse processo é geralmente referido como *descarga* (*flushing*) da stream e garante que nenhuma informação seja acidentalmente deixada no buffer de disco. Todos os arquivos são fechados automaticamente quando o programa termina normalmente, com **main()** retornando ao sistema operacional ou uma chamada a **exit()**. Os arquivos não são fechados quando um programa quebra (*crash*) ou quando ele chama **abort ()**.

Cada stream associada a um arquivo tem uma estrutura de controle de arquivo do tipo **FILE**. Essa estrutura é definida no cabeçalho STDIO.H. Nunca modifique esse bloco de controle de arquivo.

A separação que C faz entre streams e arquivos pode parecer desnecessária ou estranha, mas a sua principal finalidade é a consistência da interface. Em C, você só precisa pensar em termos de stream e usar apenas um sistema de arquivos para realizar todas as operações de E/S. O compilador C converte a entrada ou saída em linha em uma stream facilmente gerenciada.

Fundamentos do Sistema de Arquivos

O sistema de arquivos C ANSI é composto de diversas funções inter-relacionadas. As mais comuns são mostradas na Tabela 9.1. Essas funções exigem que o cabeçalho STDIO.H seja incluído em qualquer programa em que são utilizadas. Note que a maioria das funções começa com a letra "f". Isso é uma convenção do padrão C UNIX, que definiu dois sistemas de arquivos. As funções de E/S do UNIX não começavam com um prefixo e a maioria das funções do sistema de E/S formatado tinha o prefixo "f". O comitê do ANSI escolheu manter essa convenção de nomes para manter uma continuidade.

Tabela 9.1 As funções mais comuns do sistema de arquivo com buffer.

Nome	Função
fopen()	Abre um arquivo
fclose()	Fecha um arquivo
putc()	Escreve um caractere em um arquivo
fputc()	O mesmo que putc()
getc()	Lê um caractere de um arquivo
fgetc()	O mesmo que getc()
fseek()	Posiciona o arquivo em um byte específico
fprintf()	É para um arquivo o que **printf()** é para o console
fscanf()	É para um arquivo o que **scanf()** é para o console
feof()	Devolve **verdadeiro** se o fim de arquivo for atingido
ferror()	Devolve **verdadeiro** se ocorreu um erro
rewind()	Recoloca o indicador de posição de arquivo no início do arquivo
remove()	Apaga um arquivo
fflush()	Descarrega um arquivo

O arquivo de cabeçalho STDIO.H fornece os protótipos para as funções de E/S e define estes três tipos: **size_t**, **fpos_t** e **FILE**. O tipo **size_t** é essencialmente o mesmo que um unsigned, assim como o **fpos_t**. O tipo **FILE** é discutido na próxima seção.

STDIO.H também define várias macros. As relevantes a este capítulo são: **NULL**, **EOF**, **FOPEN_MAX**, **SEEK_SET**, **SEEK_CUR** e **SEEK_END**. A macro **NULL** define um ponteiro nulo. A macro **EOF** é geralmente definida como -1 e é o valor devolvido quando uma função de entrada tenta ler além do final do arquivo. **FOPEN_MAX** define um valor inteiro que determina o número de arquivos que podem estar abertos ao mesmo tempo. As outras macros são usadas com **fseek()**, que é uma função que executa acesso aleatório em um arquivo.

O Ponteiro de Arquivo

O ponteiro é o meio comum que une o sistema C ANSI de E/S. *Um ponteiro de arquivo* é um ponteiro para informações que definem várias coisas sobre o arquivo, incluindo seu nome, status e a posição atual do arquivo. Basicamente, o ponteiro de arquivo identifica um arquivo específico em disco e é usado pela stream associada para direcionar as operações das funções de E/S. Um ponteiro de arquivo é uma variável ponteiro do tipo **FILE**. Para ler ou escrever arquivos, seu programa precisa usar ponteiros de arquivo. Para obter uma variável ponteiro de arquivo, use um comando como este:

```
FILE *fp;
```

Abrindo um Arquivo

A função **fopen()** abre uma stream para uso e associa um arquivo a ela. Ela retorna o ponteiro de arquivo associado a esse arquivo. Mais freqüentemente (e para o resto desta discussão) o arquivo é um arquivo em disco. A função **fopen()** tem este protótipo:

FILE *fopen(const char* *nomearq*, const char* *modo*);

onde *nomearq* é um ponteiro para uma cadeia de caracteres que forma um nome válido de arquivo e pode incluir uma especificação de caminho de pesquisa (*path*). A string apontada por *modo* determina como o arquivo será aberto. A Tabela 9.2 mostra os valores legais para *modo*. Strings como "r+b" também podem ser representadas como "rb+".

Como exposto, a função **fopen()** devolve um ponteiro de arquivo. Seu programa nunca deve alterar o valor desse ponteiro. Se ocorrer um erro quando estiver tentando abrir um arquivo, **fopen()** devolve um ponteiro nulo.

Como mostra a Tabela 9.2, um arquivo pode ser aberto no modo texto ou binário. Em muitas implementações, no modo texto, seqüências de retorno de carro/alimentação de linha são traduzidas para caracteres de nova linha na entrada. Na saída, ocorre o inverso: novas linhas são traduzidas para retornos de carro/alimentações de linha. Nenhuma tradução desse tipo ocorre em arquivos binários.

Tabela 9.2 Os valores legais para modo.

Modo	Significado
r	Abre um arquivo-texto para leitura
w	Cria um arquivo-texto para escrita
a	Anexa a um arquivo-texto
rb	Abre um arquivo binário para leitura
wb	Cria um arquivo binário para escrita
ab	Anexa a um arquivo binário
r+	Abre um arquivo-texto para leitura/escrita
w+	Cria um arquivo-texto para leitura/escrita
a+	Anexa ou cria um arquivo-texto para leitura/escrita
r+b	Abre um arquivo binário para leitura/escrita
w+b	Cria um arquivo binário para leitura/escrita
a+b	Anexa a um arquivo binário para leitura/escrita

Para abrir um arquivo chamado TEST, permitindo escrita, pode-se digitar:

```
FILE *fp;

fp = fopen("test","w");
```

Embora tecnicamente correto, você geralmente verá o código anterior escrito desta forma:

```
FILE *fp;

if((fp = fopen("test", "w"))==NULL) {
  printf("arquivo não pode ser aberto\n");
  exit(1);
}
```

Este método detectará qualquer erro na abertura de um arquivo, tal como um disco cheio ou protegido contra gravação, antes de que seu programa tente gravar nele. Em geral, você sempre deve confirmar o sucesso de **fopen** antes de tentar qualquer outra operação sobre o arquivo.

Se você usar **fopen()** para abrir um arquivo com permissão para escrita, qualquer arquivo já existente com esse nome será apagado e um novo arquivo será iniciado. Se nenhum arquivo com esse nome existe, um será criado. Se você deseja adicionar ao final do arquivo, deve usar o modo "a". Arquivos já existentes só podem ser abertos para operações de leitura. Se o arquivo não existe, um erro é devolvido. Finalmente, se um arquivo é aberto para operações de leitura/escrita, ele não será apagado se já existe e, se não existe, ele será criado.

O número de arquivos que pode ser aberto em um determinado momento é especificado por **FOPEN_MAX**. Este número geralmente é superior a 8, mas você deve verificar no manual do seu compilador qual é o seu valor exato.

Fechando um Arquivo

A função **fclose()** fecha uma stream que foi aberta por meio de uma chamada a **fopen()**. Ela escreve qualquer dado que ainda permanece no buffer de disco no arquivo e, então, fecha normalmente o arquivo em nível de sistema operacional. Uma falha ao fechar uma stream atrai todo tipo de problema, incluindo perda de dados, arquivos destruídos e possíveis erros intermitentes em seu programa. Uma **fclose()** também libera o bloco de controle de arquivo associado à stream, deixando-o disponível para reutilização. Em muitos casos, há um limite do sistema operacional para o número de arquivos abertos simultaneamente, assim, você deve fechar um arquivo antes de abrir outro.

A função **fclose()** tem este protótipo:

int fclose(FILE *fp);

onde *fp* é o ponteiro de arquivo devolvido pela chamada a **fopen()**. Um valor de retorno zero significa uma operação de fechamento bem-sucedida. Qualquer outro valor indica um erro. A função padrão **ferror()** (discutida em breve) pode ser utilizada para determinar e informar qualquer problema. Geralmente, **fclose()** falhará quando um disco tiver sido retirado prematuramente do acionador ou não houver mais espaço no disco.

Escrevendo um Caractere

O padrão C ANSI define duas funções equivalentes para escrever caracteres: **putc()** e **fputc()**. (Tecnicamente, **putc()** é implementada como macro.) Há duas funções idênticas simplesmente para preservar a compatibilidade com versões mais antigas de C. Este livro usa **putc()**, mas você pode usar **fputc()**, se desejar.

A função **putc()** escreve caracteres em um arquivo que foi previamente aberto para escrita por meio da função **fopen()**. O protótipo para essa função é

int putc(int *ch*, FILE *fp);

onde *fp* é um ponteiro de arquivo devolvido por **fopen()** e *ch* é o caractere a ser escrito. O ponteiro de arquivo informa a **putc()** em que arquivo em disco escrever. Por razões históricas, *ch* é definido como um **int**, mas apenas o byte menos significativo é utilizado.

Se a operação **putc()** foi bem-sucedida, ela devolverá o caractere escrito. Caso contrário, ela devolve **EOF**.

Lendo um Caractere

O padrão ANSI define duas funções para ler um caractere — **getc()** e **fgetc()** — para preservar a compatibilidade com versões anteriores de C. Este livro usa **getc()** (que é implementada como uma macro), mas você pode usar **fgetc()** se desejar.

A função **getc()** lê caracteres de um arquivo aberto no modo leitura por **fopen()**. O protótipo de **getc()** é

int getc(FILE *fp);

onde *fp* é um ponteiro de arquivo do tipo **FILE** devolvido por **fopen()**. Por razões históricas, **getc()** devolve um inteiro, mas o byte mais significativo é zero.

A função **getc()** devolve **EOF** quando o final do arquivo for alcançado. O código seguinte poderia ser utilizado para ler um arquivo-texto até que a marca de final de arquivo seja lida.

```
do {
  ch = getc(fp);
} while(ch!=EOF);
```

No entanto, **getc()** também retorna **EOF** quando ocorre um erro. Você pode usar **ferror()** para determinar precisamente o que ocorreu.

Usando fopen(), getc(), putc() e fclose()

As funções **fopen()**, **getc()**, **putc()** e **fclose()** constituem o conjunto mínimo de rotinas de arquivo. O programa seguinte, KTOD, é um exemplo simples da utilização de **putc()**, **fopen()** e **fclose()**. Ele simplesmente lê caracteres do teclado e os escreve em um arquivo em disco até que o usuário digite um cifrão **($)**. O nome do arquivo é especificado na linha de comando. Por exemplo, se você chamar esse programa KTOD, digitando **KTOD TEST**, você poderá escrever linhas de texto no arquivo TEST.

```
/* KTOD: Do teclado para o disco. */
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
  FILE *fp;
  char ch;

  if(argc!=2) {
    printf("Você esqueceu de digitar o nome do arquivo.\n");
    exit(1);
  }

  if((fp=fopen(argv[1], "w"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  do {
    ch = getchar();
    putc(ch, fp);
  } while(ch!='$');

  fclose(fp);
}
```

O programa complementar DTOS lê qualquer arquivo ASCII e mostra o conteúdo na tela.

```
/* DTOS: Um programa que lê arquivos e mostra-os na tela. */
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
  FILE *fp;
  char ch;

  if(argc!=2) {
    printf("Você esqueceu de digitar o nome do arquivo.\n");
    exit(1);
  }

  if((fp=fopen(argv[1], "r"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  ch = getc(fp);  /* lê um caractere */

  while (ch!=EOF) {
    putchar(ch);  /* imprime na tela */
    ch = getc(fp);
  }

  fclose(fp);
}
```

Tente esses dois programas. Primeiro use KTOD para criar um arquivo-texto. Então, leia seu conteúdo usando DTOS.

Usando feof()

Como exposto anteriormente, o sistema de arquivo com buffer também pode operar com dados binários. Quando um arquivo é aberto para entrada binária, um valor inteiro igual à marca de **EOF** pode ser lido. Isso poderia fazer com que a rotina de entrada indicasse uma condição de fim de arquivo apesar de o final físico do arquivo não ter sido alcançado. Para resolver esse problema, C inclui a função **feof()**, que determina quando o final de arquivo foi atingido na leitura de dados binários. A função **feof()** tem este protótipo:

int feof(FILE *fp);

Assim como as outras funções de arquivo, seu protótipo está em STDIO.H. Ela devolve verdadeiro se o final do arquivo foi atingido; caso contrário, devolve 0. Assim, a rotina seguinte lê um arquivo binário até que o final do arquivo seja encontrado:

```
while(!feof(fp)) ch = getc(fp);
```

Obviamente, você pode aplicar esse método tanto para arquivo-texto como para arquivos binários.

O programa seguinte, que copia arquivos-textos ou binários, contém um exemplo de **feof()**. Os arquivos são abertos no modo binário e **feof()** verifica o final de arquivo.

```
/* Copia um arquivo. */
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
  FILE *in, *out;
  char ch;

  if(argc!=3) {
    printf("Você esqueceu de informar o nome do arquivo.\n");
    exit(1);
  }

  if((in=fopen(argv[1], "rb"))==NULL) {
    printf("O arquivo-fonte não pode ser aberto.\n");
    exit(1);
  }
  if((out=fopen(argv[2], "wb"))==NULL) {
    printf("O arquivo-destino não pode ser aberto.\n");
    exit(1);
  }

  /* Esse código copia de fato o arquivo */
  while(!feof(in)) {
    ch = getc(in);
    if(!feof(in)) putc(ch, out);
  }
```

```
  fclose(in);
  fclose(out);
}
```

Trabalhando com Strings: fputs() e fgets()

Além de **getc()** e **putc()**, C suporta as funções relacionadas **fputs()** e **fgets()**, que efetuam as operações de leitura e gravação de strings de caractere de e para um arquivo em disco. Essas funções operam de forma muito semelhante a **putc()** e **getc()**, mas, em lugar de ler ou escrever um único caractere, elas operam com strings. São os seguintes os seus protótipos:

int fputs(const char **str*, FILE **fp*);
char *fgets(char **str*, int *length*, FILE **fp*);

Os protótipos para **fgets()** e **fputs()** estão em STDIO.H.

A função **fputs()** opera como **puts()**, mas escreve a string na stream especificada. **EOF** será devolvido se ocorrer um erro.

A função **fgets()** lê uma string da stream especificada até que um caractere de nova linha seja lido ou que *length-1* caracteres tenham sido lidos. Se uma nova linha é lida, ela será parte da string (diferente de **gets()**). A string resultante será terminada por um nulo. A função devolverá um ponteiro para **str** se bem-sucedida ou um ponteiro nulo se ocorrer um erro.

No programa seguinte, a função **fputs()** lê strings do teclado e escreve-as no arquivo chamado TEST. Para terminar o programa, insira uma linha em branco. Como **gets()** não armazena o caractere de nova linha, é adicionado um antes que a string seja escrita no arquivo para que o arquivo possa ser lido mais facilmente.

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

void main(void)
{
  char str[80];
  FILE *fp;

  if((fp = fopen("TEST", "w"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }
```

```
  do {
    printf("Digite uma string (CR para sair):\n");
    gets(str);
    strcat(str, "\n");  /* acrescenta uma nova linha */
    fputs(str, fp);
  } while(*str!='\n');
}
```

rewind()

A função **rewind()** reposiciona o indicador de posição de arquivo no início do arquivo especificado como seu argumento. Isto é, ela "rebobina" o arquivo. Seu protótipo é

void rewind(FILE *fp);

onde *fp* é um ponteiro válido de arquivo. O protótipo para **rewind()** está em STDIO.H.

Para ver um exemplo de **rewind()**, você pode modificar o programa da seção anterior para que ele mostre o conteúdo do arquivo recém-criado. Para fazer isso, o programa rebobina o arquivo depois de completada a entrada e, então, usa **fgets()** para ler de volta o arquivo. Note que o arquivo precisa ser aberto no modo leitura/escrita usando "w+" como parâmetro de modo.

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

void main(void)
{
  char str[80];
  FILE *fp;

  if((fp = fopen("TEST", "w+"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  do {
    printf("Digite uma string (CR para sair):\n");
    gets(str);
    strcat(str, "\n");  /* acrescenta uma nova linha */
```

```
    fputs(str, fp);
  } while(*str!='\n');

  /* agora, lê e mostra o arquivo */
  rewind(fp); /* reinicializa o indicador de posição de arquivo
                 para o começo do arquivo. */
  while(!feof(fp)) {
    fgets(str, 79, fp);
    printf(str);
  }
}
```

ferror()

A função **ferror()** determina se uma operação com arquivo produziu um erro. A função **ferror()** tem este protótipo:

```
int ferror(FILE *fp);
```

onde *fp* é um ponteiro válido de arquivo. Ela retorna verdadeiro se ocorreu um erro durante a última operação no arquivo; caso contrário, retorna falso. Como toda operação modifica a condição de erro, **ferror()** deve ser chamada imediatamente após cada operação com arquivo; caso contrário, um erro pode ser perdido. O protótipo para **ferror()** está em STDIO.H.

O programa seguinte ilustra **ferror()** removendo tabulações de um arquivo-texto e substituindo pelo número apropriado de espaços. O tamanho da tabulação é definido por **TAB_SIZE**. Observe como **ferror()** é chamada após cada operação no disco. Para utilizar o programa, execute-o após especificar os nomes dos arquivos de entrada e saída na linha de comando.

```
/* O programa substitui espaços por tabulações em um arquivo-
   texto e fornece verificação de erros. */

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

#define TAB_SIZE 8
#define IN 0
#define OUT 1

void err(int e);

void main(int argc, char *argv[])
```

```
{
  FILE *in, *out;
  int tab, i;
  char ch;

  if(argc!=3) {
    printf("uso: detab <entrada> <saída>\n");
    exit(1);
  }

  if((in = fopen(argv[1], "rb"))==NULL) {
    printf("O arquivo %s não pode ser aberto.\n", argv[1]);
    exit(1);
  }

  if((out = fopen(argv[2], "wb"))==NULL) {
    printf("O arquivo %s não pode ser aberto.\n", argv[2]);
    exit(1);
  }

  tab = 0;
  do {
    ch = getc(in);
    if(ferror(in)) err(IN);

    /* se encontrou um tab, então */
      envia o número apropriado de espaços */
    if(ch=='\t') {
      for(i=tab; i<8; i++) {
        putc(' ', out);
        if(ferror(out)) err(OUT);
      }
      tab = 0;
    }
    else {
      putc(ch, out);
      if(ferror(out)) err(OUT);
      tab++;
      if(tab==TAB_SIZE) tab = 0;
      if(ch=='\n' || ch=='\r') tab = 0;
    }
  } while(!feof(in));
  fclose(in);
  fclose(out);
}
```

```
void err(int e)
{
  if(e==IN) printf("Erro na entrada.\n");
  else printf("Erro na saída.\n");
  exit(1);
}
```

Apagando Arquivos

A função **remove()** apaga o arquivo especificado. Seu protótipo é

int remove(const char **filename*);

Ela devolve zero, caso seja bem-sucedida, e um valor diferente de zero, caso contrário.

O programa seguinte apaga um arquivo especificado na linha de comando. Porém, ele primeiro lhe dá uma chance de mudar de idéia. Um utilitário como esse poderia ser útil a novos usuários de computador.

```
/* Verificação dupla antes de apagar. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

main(int argc, char *argv[])
{
  char str[80];
  if(argc!=2) {
    printf("uso: xerase <nomearq>\n");
    exit(1);
  }

  printf("apaga %s? (S/N): ", argv[1]);
  gets(str);

  if(toupper(*str)=='S')
    if(remove(argv[1])) {
      printf("O arquivo não pode ser apagado.\n");
      exit(1);
    }
  return 0; /* retorna sucesso ao SO */
}
```

Esvaziando uma Stream

Para esvaziar o conteúdo de uma stream de saída, deve-se utilizar a função **fflush()**, cujo protótipo é mostrado aqui:

```
int fflush(FILE *fp);
```

Essa função escreve o conteúdo de qualquer dado existente no buffer para o arquivo associado a *fp*. Se **fflush()** for chamada com um valor nulo, todos os arquivos abertos para saída serão descarregados.

A função **fflush()** devolve 0 para indicar sucesso; caso contrário, devolve **EOF**.

fread() e fwrite()

Para ler e escrever tipos de dados maiores que um byte, o sistema de arquivo C ANSI fornece duas funções: **fread()** e **fwrite()**. Essas funções permitem a leitura e a escrita de blocos de qualquer tipo de dado. Seus protótipos são

```
size_t fread(void *buffer, size_t num_bytes,
             size_t count, FILE *fp);
size_t fwrite(const void *buffer, size_t num_bytes,
             size_t count, FILE *fp);
```

Para **fread()**, *buffer* é um ponteiro para uma região de memória que receberá os dados do arquivo. Para **fwrite()**, *buffer* é um ponteiro para as informações que serão escritas no arquivo. O número de bytes a ler ou escrever é especificado por *num_bytes*. O argumento *count* determina quantos itens (cada um de comprimento *num_byte*) serão lidos ou escritos. (Lembre-se de que o tipo **size_t** é definido em STDIO.H e é aproximadamente o mesmo que unsigned.) Finalmente, *fp* é um ponteiro para uma stream aberta anteriormente. Os protótipos das duas funções estão definidos em STDIO.H.

A função **fread()** devolve o número de itens lidos. Esse valor poderá ser menor que *count* se o final do arquivo for atingido ou ocorrer um erro. A função **fwrite()** devolve o número de itens escritos. Esse valor será igual a *count* a menos que ocorra um erro.

Usando fread() e fwrite()

Quando o arquivo for aberto para dados binários, **fread()** e **fwrite()** podem ler e escrever qualquer tipo de informação. Por exemplo, o programa seguinte escreve e em seguida lê de volta um **double**, um **int** e um **long** em um arquivo em disco. Observe como **sizeof** é utilizado para determinar o comprimento de cada tipo de dado.

```
/* Escreve alguns dados não-caracteres em um arquivo em disco
   e lê de volta. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  FILE *fp;
  double d = 12.23;
  int i = 101;
  long l = 123023L;

  if((fp=fopen("test", "wb+"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  fwrite(&d, sizeof(double), 1, fp);
  fwrite(&i, sizeof(int), 1, fp);
  fwrite(&l, sizeof(long), 1, fp);

  rewind(fp);

  fread(&d, sizeof(double), 1, fp);
  fread(&i, sizeof(int), 1, fp);
  fread(&l, sizeof(long), 1, fp);

  printf("%f %d %ld", d, i, l);

  fclose(fp);

}
```

Como esse programa ilustra, o buffer pode ser (e geralmente é) simplesmente a memória usada para guardar uma variável. Nesse programa, os valores de retorno de **fread()** e **fwrite()** são ignorados. Em um contexto real, porém, seus valores de retorno deveriam ser verificados à procura de erros.

Uma das mais úteis aplicações de **fread()** e **fwrite()** envolve ler e escrever tipos de dados definidos pelo usuário, especialmente estruturas. Por exemplo, dada esta estrutura:

```
struct struct_type {
     float balance;
     char name[80];
```

```
} cust;
```

a sentença seguinte escreve o conteúdo de **cust** no arquivo apontado por **fp**.

```
fwrite (&cust, sizeof(struct struc_type), 1, fp);
```

Exatamente para ilustrar como é fácil escrever grandes quantidades de dados usando **fread()** e **fwrite()**, um programa simples de lista postal foi desenvolvido. A lista será armazenada em uma matriz de estruturas deste tipo:

```
struct list_type {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[10];
} list[SIZE];
```

O valor de **SIZE** determina quantos endereços podem ser armazenados na lista.

Quando o programa é executado, o campo **nome** de cada estrutura é inicializado com um nulo na primeira posição. Por convenção, o programa assume que a estrutura não é usada se o nome tem comprimento 0.

As rotinas **save()** e **load()**, mostradas em breve, são utilizadas para salvar e carregar o banco de dados da lista postal. Observe como pouco código está contido em cada rotina devido à força de **fread()** e **fwrite()**. Observe, também, como essas funções verificam os valores de retorno de **fread()** e **fwrite()** devido aos erros.

```
/* Salva a lista. */
void save(void)
{
  FILE *fp;
  register int i;

  if((fp=fopen("maillist", "wb"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    return;
  }


  for(i=0; i<SIZE; i++)
    if(*list[i].name)
      if(fwrite(&list[i],
        sizeof(struct list_type), 1, fp)!=1)
```

```
            printf("Erro de escrita no arquivo.\n");

  fclose (fp);
}

/* Carrega o arquivo.*/
void load(void)
{
  FILE *fp;
  register int i;
  if((fp=fopen("maillist", "rb"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    return;
  }

  init_list();
  for(i=0; i<SIZE; i++)
    if(fread(&list[i],
      sizeof(struct list_type), 1, fp)!=1) {
        if(feof(fp)) break;
        printf("Erro de leitura no arquivo.\n");

    }

  fclose (fp);
}
```

As duas rotinas confirmam uma operação com arquivo bem-sucedida verificando o valor de retorno de **fread()** ou **fwrite()**. Além disso, **load()** deve verificar explicitamente o final de arquivo via **feof()**, porque **fread()** retorna o mesmo valor caso o fim de arquivo seja atingido ou ocorra um erro.

O programa de lista postal completo é mostrado em breve. Você pode querer utilizá-lo como um núcleo para melhorias adicionais, incluindo a capacidade de apagar nomes e fazer buscas por endereços.

```
/* Um programa de lista postal muito simples. */

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 100

struct list_type {
```

```
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[10];
} list[SIZE];
int menu(void);
void init_list(void), enter(void);
void display(void), save(void);
void load(void);

void main(void)
{
  char choice;

  init_list();

  for(;;) {
    choice = menu();
    switch(choice) {
      case 'e': enter();
        break;
      case 'd': display();
        break;
      case 's': save();
        break;
      case 'l': load();
        break;
      case 'q': exit(0);
    }
  }
}

/* Inicializa a lista. */
void init_list(void)
{
  register int t;

  for(t=0; t<SIZE; t++) *list[t].name = '\0';
  /* um nome de comprimento zero significa vazio */
}

/* Põe os nomes na lista. */
void enter(void)
{
```

```
  register int i;

  for(i=0; i<SIZE; i++)
    if(!*list[i].name) break;

  if(i==SIZE) {
    printf("lista cheia\n");
    return;
  }

  printf("nome: ");
  gets(list[i].name);

  printf("rua: ");
  gets(list[i].street);

  printf("cidade: ");
  gets(list[i].city);

  printf("estado: ");
  gets(list[i].state);

  printf("CEP: ");
  gets(list[i].zip);
}

/* Mostra a lista. */
void display(void)
{
  register int t;

  for(t=0; t<SIZE; t++) {
    if(*list[t].name) {
      printf("%s\n", list[t].name);
      printf("%s\n", list[t].street);
      printf("%s\n", list[t].city);
      printf("%s\n", list[t].state);
      printf("%s\n\n", list[t].zip);
    }
  }
}

/* Salva a lista. */
void save(void)
{
```

```
  FILE *fp;
  register int i;

  if((fp=fopen("maillist", "wb"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    return;
  }

  for(i=0; i<SIZE; i++)
    if(*list[i].name)

      if(fwrite(&list[i],
        sizeof(struct list_type), 1, fp)!=1)
          printf("Erro de escrita no arquivo.\n");
 fclose (fp);
}

/* Carrega o arquivo. */
void load(void)
{
  FILE *fp;
  register int i;

  if((fp=fopen("maillist", "rb"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    return;
  }

  init_list();
  for(i=0; i<SIZE; i++)
    if(fread(&list[i],
      sizeof(struct list_type), 1, fp)!=1) {
        if(feof(fp)) break;
        printf("Erro de leitura no arquivo.\n");
    }

  fclose (fp);
}

/* Obtém uma seleção do menu. */
menu(void)
{
  char s[80];
```

```
  do {
    printf("(I)nserir\n");
    printf("(V)isualizar\n");
    printf("(C)arregar\n");
    printf("(S)alvar\n");
    printf("(T)erminar\n");
    printf("escolha: ");
    gets(s);
  } while(!strchr("ivcst", tolower(*s)));
  return tolower(*s);
}
```

fseek() e E/S com Acesso Aleatório

Operações de leitura e escrita aleatórias (ou randômicas) podem ser executadas utilizando o sistema de E/S bufferizado com a ajuda de **fseek()**, que modifica o indicador de posição de arquivo. Seu protótipo é mostrado aqui:

int fseek(FILE *fp, long *numbytes*, int *origin*);

Aqui, *fp* é um ponteiro de arquivo devolvido por uma chamada a **fopen()**. *numbytes*, um inteiro longo, é o número de bytes a partir de *origin*, que se tornará a nova posição corrente, e *origin* é uma das seguintes macros definidas em STDIO.H.

Origin	Nome da Macro
Início do arquivo	SEEK_SET
Posição atual	SEEK_CUR
Final do arquivo	SEEK_END

Portanto, para mover *numbytes* a partir do início do arquivo, *origin* deve ser **SEEK_SET**. Para mover da posição atual, deve-se utilizar **SEEK_CUR** e para mover a partir do final do arquivo, deve-se utilizar **SEEK_END**. A função **fseek()** devolve 0 quando bem-sucedida e um valor diferente de zero se ocorre um erro.

O fragmento seguinte ilustra **fseek()**. Ele procura e mostra um byte específico em um arquivo especificado. O nome do arquivo e o byte a ser buscado devem ser especificados na linha de comando.

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

void main(int argc, char *argv[])
{
  FILE *fp;
```

```
  if(argc!=3) {
    printf("Uso: SEEK nomearq byte\n");
    exit(1);
  }

  if((fp=fopen(argv[1], "r"))==NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  if(fseek(fp, atol(argv[2]), SEEK_SET)) {
    printf("Erro na busca.\n");
    exit(1);
  }

  printf("O byte em %ld é %c.\n", atol(argv[2]), getc(fp));
  fclose(fp);
}
```

fseek() pode ser utilizada para efetuar movimentações em múltiplos de qualquer tipo de dado simplesmente multiplicando o tamanho dos dados pelo número do item que se deseja alcançar. Por exemplo, se você tiver um arquivo de lista postal produzido pelo exemplo da seção anterior, o fragmento de código seguinte move-se para o décimo endereço.

```
fseek(fp, 9*sizeof(struct list_type), SEEK_SET);
```

fprintf() e fscanf()

Como extensão das funções básicas de E/S já discutidas, o sistema de E/S com buffer inclui **fprintf()** e **fscanf()**. Essas funções comportam-se exatamente como **printf()** e **scanf()** exceto por operarem com arquivos. Os protótipos de **fprintf()** e **fscanf()** são

```
int fprintf(FILE *fp, const char *control_string,...);
int fscanf(FILE *fp, const char *control_string,...);
```

onde *fp* é um ponteiro de arquivo devolvido por uma chamada a **fopen()**. **fprintf()** e **fscanf()** direcionam suas operações de E/S para o arquivo apontado por *fp*.

Como exemplo, o programa seguinte lê uma string e um inteiro do teclado e os grava em um arquivo em disco chamado TEST. O programa então lê o arquivo e exibe a informação na tela. Após executar este programa, examine o arquivo TEST. Você verá que ele contém texto legível.

```
/* Exemplo de fscanf() - fprintf() */
#include <stdio.h>
#include <io.h>
#include <stdlib.h>

void main(void)
{
  FILE *fp;
  char s[80];
  int t;

  if((fp=fopen("test", "w")) == NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  printf("Digite uma string e um número: ");
  fscanf(stdin, "%s%d", s, &t); /* lê do teclado */

  fprintf(fp, "%s %d", s, t); /* escreve no arquivo */
  fclose(fp);

  if((fp=fopen("test", "r")) == NULL) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  fscanf(fp, "%s%d", s, &t); /* lê do arquivo */
  fprintf(stdout, "%s %d", s, t); /*  imprime na tela */
}
```

Um aviso: embora **fprintf()** e **fscanf()** geralmente sejam a maneira mais fácil de escrever e ler dados diversos em arquivos em disco, elas não são sempre a escolha mais apropriada. Como os dados são escritos em ASCII e formatados como apareceriam na tela (e não em binário), um tempo extra é perdido a cada chamada. Assim, se há preocupação com velocidade ou tamanho do arquivo, deve-se utilizar **fread()** e **fwrite()**.

As Streams Padrão

Sempre que um programa em C começa a execução, três streams são abertas automaticamente. Elas são a entrada padrão (**stdin**), a saída padrão (**stdout**) e a saída de erro padrão (**stderr**). Normalmente, essas streams referem-se ao console, mas podem ser redirecionadas pelo sistema operacional para algum outro dispositivo em ambientes que suportam redirecionamento de E/S. (E/S redirecionadas são suportadas pelo UNIX, OS/2 e DOS, por exemplo.)

Como as streams padrões são ponteiros de arquivos, elas podem ser utilizadas pelo sistema de E/S bufferizado para executar operações de E/S no console. Por exemplo, **putchar()** poderia ser definida desta forma:

```
putchar (char c)
{
   return putc(c,stdout);
}
```

Em geral, **stdin** é utilizada para ler do console e **stdout** e **stderr**, para escrever no console. **stdin**, **stdout** e **stderr** podem ser utilizadas como ponteiros de arquivo em qualquer função que use uma variável do tipo **FILE** *. Por exemplo, você pode usar **fputs()** para escrever uma string no console usando uma chamada como esta:

```
fputs ("ola aqui", stdout);
```

Tenha em mente que **stdin**, **stdout** e **stderr** não são variáveis no sentido normal e não podem receber nenhum valor usando **fopen()**. Além disso, da mesma maneira que são criados automaticamente no início do seu progama, os ponteiros são fechados automaticamente no final; você não deve tentar fechá-los.

A Conexão de E/S pelo Console

Recorde, conforme o Capítulo 8, que C faz uma pequena distinção entre E/S pelo console e E/S com arquivo. As funções de E/S pelo console, descritas no Capítulo 8, na realidade direcionam suas operações de E/S para **stdin** ou **stdout**. Essencialmente, as funções de E/S pelo console são simplesmente versões especiais de suas funções semelhantes para arquivos. Elas existem apenas como conveniência para o programador.

Como descrito na seção anterior, você pode realizar E/S pelo console usando qualquer uma das funções do sistema de arquivos de C. Contudo, o que

pode surpreendê-lo é ser possível efetuar E/S de arquivo em disco, usando funções de E/S pelo console, como **printf()**. Isso ocorre porque todas as funções de E/S pelo console, descritas no Capítulo 8, operam em **stdin** e **stdout**. Em ambientes que permitem redirecionamento de E/S, isso significa que **stdin** e **stdout** poderiam referir-se a um dispositivo diferente do teclado e da tela. Por exemplo, considere este programa:

```
#include <stdio.h>

void main(void)
{
  char str[80];

  printf("Digite uma string: ");
  gets(str);
  printf(str);
}
```

Assuma que esse programa é chamado de TEST. Se você executa TEST normalmente, ele mostra sua mensagem na tela, lê uma string do teclado e mostra essa string no vídeo. Porém, em um ambiente que suporta redirecionamento de E/S, **stdin**, **stdout** ou ambas podem ser redirecionadas para um arquivo. Por exemplo, em ambiente DOS ou OS/2, executar TEST desta forma:

```
TEST > OUTPUT
```

faz com que a saída de TEST seja escrita em um arquivo chamado OUTPUT. Executar TEST desta forma:

TEST <INPUT > OUTPUT

direciona **stdin** para o arquivo chamado INPUT e envia a saída para o arquivo chamado OUTPUT.

Quando um programa em C termina, qualquer stream redirecionada é reposta no seu estado padrão.

Usando freopen() para Redirecionar as Streams Padrão

As streams padrão podem ser redirecionadas utilizando-se a função **freopen()**. Essa função associa uma stream existente a um novo arquivo. Assim, você pode usá-la para associar uma stream padrão com um novo arquivo. Seu protótipo é

FILE *freopen (const char *nomearq,
const char *modo, FILE *stream);

onde *nomearq* é um ponteiro para o nome do arquivo que se deseja associar à stream apontada por *stream*. O arquivo é aberto usando o valor de *modo*, que pode ter os mesmos valores usados em **fopen()**. **Freopen()** retorna *stream* no caso de sucesso, NULL se falhar.

O programa seguinte usa **freopen()** para redirecionar **stdout** para um arquivo chamado OUTPUT:

```
#include stdio,h>

void main(void)
{
  char str[80];

  freopen("OUTPUT", "w", stdout);

  printf("Digite uma string: ");
  gets(str);
  printf(str);
}
```

Em geral, redirecionar as streams padrão usando **freopen()** é útil em situações especiais, como em depuração. Porém, efetuar operações de E/S em disco utilizando **stdin** e **stdout** redirecionados não é tão eficiente quanto utilizar funções como **fread()** e **fwrite()**.

O Sistema de Arquivo Tipo UNIX

Como C foi originalmente desenvolvida sobre o sistema operacional UNIX, ela inclui um segundo sistema de E/S com arquivos em disco que reflete basicamente as rotinas de arquivo em disco de baixo nível do UNIX. O sistema de arquivo tipo UNIX usa funções que são separadas das funções do sistema de arquivo com buffer. Elas são mostradas na Tabela 9.3.

Lembre-se de que o sistema de arquivo tipo UNIX é, algumas vezes, chamado de sistema de arquivo sem buffer. Isso porque deve-se fornecer e gerenciar todos os buffers de disco — as rotinas não farão isso por você. Portanto, um sistema tipo UNIX não contém funções como **getc()** e **putc()** (que lêem e escrevem caracteres de ou para uma stream de dados). Em vez disso, ele contém as funções **read()** e **write()**, que lêem ou escrevem um buffer completo de informação a cada chamada.

Tabela 9.3 As funções de E/S tipo UNIX sem buffer.

Nome	Função
read()	Lê um buffer de dados
write()	Escreve um buffer de dados
open()	Abre um arquivo em disco
creat()	Cria um arquivo em disco
close()	Fecha um arquivo em disco
lseek()	Move ao byte especificado em um arquivo
unlink()	Remove um arquivo do diretório

Recorde que o sistema de arquivo sem buffer não é definido pelo padrão C ANSI e seu uso provavelmente diminuirá nos próximos anos. Por essa razão, não é recomendado para novos projetos. No entanto, muitos programas em C existentes usam-no e ele é suportado por virtualmente todo compilador C.

O arquivo de cabeçalho usado pelo sistema de arquivo tipo UNIX é chamado IO.H em muitas implementações. Para algumas funções, será necessário incluir também o arquivo de cabeçalho FNCTL.H .

NOTA: Muitas implementações de C não permitem que você use as funções de arquivo do ANSI e as funções de arquivo tipo UNIX no mesmo programa. Apenas por segurança, use um sistema ou outro.

open()

Ao contrário do sistema de E/S de alto nível, o sistema de baixo nível não utiliza ponteiros de arquivo do tipo **FILE**, mas descritores de arquivo do tipo **int**. O protótipo para **open()** é

int open(const char **nomearq*, int *modo*);

onde *nomearq* é qualquer nome de arquivo válido e *modo* é uma das seguintes macros que são definidas no arquivo de cabeçalho FNCTL.H.

Modo	Efeito
O_RDONLY	Lê
O_WRONLY	Escreve
O_RDWR	Lê/Escreve

Muitos compiladores possuem modos adicionais — como texto, binário etc. —, verifique, portanto, o seu manual do usuário. Uma chamada bem-sucedida a **open()** devolve um inteiro positivo. Um valor de retorno -1 significa que o arquivo não pode ser aberto.

A chamada a **open()** é geralmente escrita desta forma:

```
int fd;
if((fd = open (filename, mode))== -1) {
  printf("Não pode abrir arquivo.\n");
  exit(1);
}
```

Na maioria das implementações, a operação falha se o arquivo especificado no comando **open()** não está no disco. (Isto é, ela não cria um novo arquivo.) Para criar um novo arquivo, você normalmente chama **creat()**, que será descrito a seguir. Porém, dependendo da implementação exata do seu compilador C, você pode ser capaz de usar **open()** para criar um arquivo que ainda não existe. Verifique seu manual do usuário.

creat()

Se seu compilador não lhe permite criar um novo arquivo usando **open()**, ou se você quer garantir a portabilidade, você deve usar **creat()** para criar um novo arquivo para ser gravado. O protótipo de **creat()** é

int creat(const char **filename*, int *mode*);

onde *filename* é qualquer nome válido de arquivo. O argumento *mode* especifica um código de acesso para o arquivo. Consulte o manual de usuário de seu compilador para detalhes específicos. **creat()** retorna um descritor de arquivo válido se for bem-sucedida, ou -1 no caso de erro.

close()

O protótipo para **close()** é

int close(int *fd*);

Aqui, *fd* deve ser um descritor de arquivo válido, previamente obtido por meio de uma chamada a **open()** ou **create()**. **close()** devolve -1 se for incapaz de fechar o arquivo. Ela devolve 0 caso seja bem-sucedida.

A função **close()** libera o descritor de arquivo para que ele possa ser reutilizado com outro arquivo. Há sempre algum limite no número de arquivos que pode existir simultaneamente, assim, você deve fechar um arquivo quando ele não for mais necessário. Uma operação de fechamento faz com que qualquer informação nos buffers internos do disco do sistema seja escrita no disco. Uma falha no fechamento de um arquivo geralmente leva à perda de dados.

read() e write()

Uma vez que o arquivo tenha sido aberto para escrita, ele pode ser acessado por **write()**. O protótipo para a função **write()** é

int write(int *fd*, const void **buf*, unsigned *size*);

Toda vez que uma chamada a **write()** é executada, são escritos *size* caracteres no arquivo em disco especificado por *fd* do buffer apontado por *buf*. O buffer pode ser uma região alocada na memória ou uma variável.

A função **write()** devolve o número de bytes escritos após uma operação de escrita bem-sucedida. Se ocorre alguma falha, muitas implementações devolvem um **EOF**, mas verifique o seu manual do usuário.

A função **read()** é o complemento de **write()**. Seu protótipo é

int read(int *fd*, void **buf*, unsigned *size*);

onde *fd*, *buf* e *size* são os mesmos de **write()**, exceto por **read()** colocar os dados lidos no buffer apontado por *buf*. Caso **read()** seja bem-sucedida, ela devolve o número de caracteres realmente lido. Ela devolve 0 se o final físico do arquivo for ultrapassado e -1 se ocorrerem erros.

O programa seguinte ilustra alguns aspectos do sistema de E/S estilo UNIX. Ele lê linhas de texto do teclado e escreve-as em um arquivo em disco. Depois que elas são escritas, o programa as lê de volta.

```
/* Lê e escreve usando E/S sem buffer */
#include <stdio.h>
#include <io.h>
#include <stdlib.h>
#include <string.h>
#include <fnctl.h>

#define BUF_SIZE 128

void input(char *buf, int fd1);
void display(char *buf, int fd2);

void main(void)
{
  char buf[BUF_SIZE];
  int fd1, fd2;

  if((fd1=open("test", O_WRONLY))==-1) { /*abre para escrita */
    printf("O arquivo não pode ser aberto.\n");
```

```
    exit(1);
  }

  input(buf, fd1);
  /* agora fecha o arquivo e lê de volta */
  close(fd1);

  if((fd2=open("test", O_RDONLY))==-1) { /*abre para leitura */
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  display(buf, fd2);
  close(fd2);
}

/* Insere texto. */
void input(char *buf, int fd1)
{
  register int t;
  do {
    for(t=0; t<BUF_SIZE; t++) buf[t] = '\0';
    gets(buf); /* lê caracteres do teclado */
    if(write(fd1, buf, BUF_SIZE)!= BUF_SIZE) {
      printf("Erro de escrita.\n");
      exit(1);
    }
  } while(strcmp(buf, "quit"));
}

/* Mostra o arquivo. */
void display(char *buf, int fd2)
{
  for(;;) {
    if(read(fd2, buf, BUF_SIZE)==0) return;
    printf("%s\n", buf);
  }
}
```

unlink()

Se você deseja excluir um arquivo, use **unlink()**. Seu protótipo é

int unlink(const char **nomearq*);

onde *nomearq* é um ponteiro de caracteres para algum nome válido de arquivo. **unlink()** devolve zero se for bem-sucedida e -1 caso seja incapaz de excluir o arquivo. Isso pode acontecer se o arquivo não se encontra no disco ou se o disco está protegido para escrita.

Acesso Aleatório Usando lseek()

O sistema de arquivos tipo UNIX suporta acesso aleatório (ou randômico) via chamadas a **lseek()**. O protótipo para **lseek()** é

long lseek(int *fd*, long *offset*, int *origin*);

onde *fd* é um descritor de arquivo devolvido por uma chamada a **creat()** ou **open()**. O *offset* é geralmente um long, mas verifique o manual do usuário do seu compilador C. *origin* pode ser uma destas macros (definidas em IO.H): **SEEK_SET**, **SEEK_CUR** ou **SEEK_END**. Esses são os efeitos de cada valor para *origin*:

SEEK_SET:	Move-se *offset* bytes a partir do início do arquivo.
SEEK_CUR:	Move-se *offset* bytes a partir da posição atual.
SEEK_END:	Move-se *offset* bytes a partir do final do arquivo.

A função **lseek()** devolve a posição atual do arquivo medida a partir do início do arquivo. Em caso de falha, é devolvido -1.

O programa mostrado aqui utiliza **lseek()**. Para executá-lo, especifique um arquivo na linha de comando. Será solicitado a você o buffer que deseja ler. Digite um número negativo para sair. Você pode desejar uma modificação no tamanho do buffer para coincidir com o tamanho do setor do seu sistema, embora isso não seja necessário. Aqui, o tamanho do buffer é 128:

```
/* Demonstra lseek(). */
#include <stdio.h>
#include <io.h>
#include <stdlib.h>
#include <fcntl.h>

#define BUF_SIZE 128

void main(int argc, char *argv[])
{
  char buf[BUF_SIZE+1], s[10];
  int fd, sector;

  if(argc!=2) {
    printf("uso: dump <sector>\n");
```

```
    exit(1);
  }

  buf[BUF_SIZE] = '\0'; /* o buffer termina com um nulo */

  if((fd=open(argv[1], O_RDONLY))==-1) {
    printf("Arquivo não pode ser aberto.\n");
    exit(1);
  }

  do {
    printf("\nBuffer: ");
    gets(s);

    sector = atoi(s); /* obtém o setor a ler */

    if(lseek(fd, (long)sector*BUF_SIZE, 0)==-1L)
      printf("Erro na busca\n");

    if(read(fd, buf, BUF_SIZE)==0) {
      printf("Setor fora da faixa\n");
    }
    else
      printf(buf);
  } while(sector>=0);
  close(fd);
}
```