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 negativo indica que o arquivo não pode ser aberto.

```
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
informação nos buffers internos do disco do siste
falha no fechamento de um arqui

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=
```

```
    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

```
int unlink
```

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[
  int fd, sector;

  if(ar
```

```
    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);
}
```