# 10. ARQUIVOS

A linguagem C não contém comandos de E/S. 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. Isso torna fácil criar arquivos que satisfaçam qualquer necessidade.

## *10.1. Streams e Arquivos*

Antes de estudar o 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.

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 um algum outro dispositivo, como o console. Existem dois tipos de *streams*: texto e binário.

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 caracter de nova linha. Porém, o caracter de nova linha é opcional na última linha e é determinado pela implementação (a maioria dos compiladores C não termina *streams* de texto com caracter 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.

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.

Em C, um arquivo pode ser qualquer coisa, desde um arquivo em disco até um terminal ou uma impressora. É associada uma *stream* com um arquivo específico realizando uma operação de abertura. Assim, quando o arquivo é aberto, informações podem ser trocadas entre ele e o programa. Cada *stream* associada a um arquivo tem uma estrutura de controle de arquivo do tipo *FILE*. Essa estrutura é definida no cabeçalho *stdio.h*. Abrir um arquivo também inicializa o **indicador de posição no arquivo** para o começo do arquivo. Quando cada caracter é lido ou escrito no arquivo, o indicador de posição é

incrementado, garantindo progressão através do arquivo. Assim, cada *byte* do arquivo possui um endereço único que é o deslocamento relativo ao início do arquivo.

Na operação de fechamento, um arquivo é desassociado de uma *stream* específica. 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()*.

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, é necessário apenas pensar em termos de *stream* e usar 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 [SCH 96].

## 10.2. Arquivo Texto X Arquivo Binário

Como já comentado, é possível gravar dados no formato de caracter (ASCII) ou em binário (*binary file*). Entretanto, a seqüência e interpretação dos dados é de inteira responsabilidade do programador do sistema. Existem dois conjuntos de rotinas para manipulação de arquivos em C: rotinas para manipulação de arquivos "texto" e para manipulação de arquivos "binários". A diferença básica entre os dois conjuntos é que as rotinas de manipulação de arquivos binários armazenam uma cópia dos *bytes* da memória principal para secundária, enquanto que as de manipulação de arquivos texto armazenam a representação ASCII dos valores.

## 10.3. Funções para manipulação de arquivos

Como já comentado, o sistema de arquivos C ANSI é composto de diversas funções inter-relacionadas que exigem a inclusão do cabeçalho *stdio.h*. Este arquivo fornece os protótipos para as funções de E/S e define várias macros, tais como NULL (ponteiro nulo) e EOF(final do arquivo, geralmente definida como -1).

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. Um ponteiro de arquivo é uma variável ponteiro do tipo ***FILE***, usada para ler ou escrever arquivos. Para obter uma variável ponteiro de arquivo, deve-se usar o seguinte comando:

```
FILE *fp;
```

A função ***fopen()*** abre uma *stream* para uso e associa um arquivo a ela, que por sua vez retorna o ponteiro de arquivo associado a esse arquivo cujo valor nunca deve ser alterado. . Se ocorrer um erro quando estiver tentando abrir um arquivo, *fopen()* devolve um ponteiro nulo. Protótipo:

```
FILE *fopen(const char* <nomearq>, const char* <modo>);
```

Onde <nomearq> é um ponteiro para uma *string* que forma um nome de arquivo válido e pode incluir uma especificação de caminho de pesquisa (*path*). A *string* apontada por <modo> determina como o arquivo será aberto. A próxima tabela mostra os valores válidos, onde *strings* como "r+b" também podem ser representadas como "rb+".

De acordo com a tabela, 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 deste tipo ocorre em arquivos binários.

| 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 |

Por exemplo, para abrir um arquivo chamado "teste", permitindo escrita, pode-se digitar:

```
FILE *fp;
if ( (fp = fopen("teste", "w")) == NULL ) {
        printf("Arquivo não pode ser aberto! \n");
        exit(1);
}
```

Assim, será detectado qualquer erro na abertura de um arquivo, tal como um disco cheio ou protegido contra gravação, antes que o programa tente gravar nele.

Se *fopen()* for usado 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. Para adicionar ao final do arquivo, deve-se usar o modo "a". Neste caso, arquivos já existentes só podem ser abertos para operações de leitura e 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 pela macro *FOPEN_MAX*. Este número geralmente é superior a oito, mas é necessário verificar o manual do compilador para verificar qual é o valor exato.

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* causa vários problemas, incluindo perda de dados e arquivos destruídos. Esta função também libera o bloco de controle de arquivo associado à *stream*, deixando-o disponível para reutilização. O protótipo da *fclose()* é:

```
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()* pode ser utilizada para determinar e informar qualquer problema. Geralmente *fclose()* falhará quando um disco tiver sido retirado prematuramente do *drive* ou não houver mais espaço no disco.

O padrão C ANSI define duas funções equivalentes para escrever caracteres: ***putc()*** e ***fputc()***. Há duas funções idênticas simplesmente para preservar a compatibilidade com versões mais antigas de C. Analogamente, também são oferecidas duas funções para ler um caracter: ***getc()*** e ***fgetc()***. Os protótipos para estas funções são:

```
int putc(int ch, FILE *fp);
int fputc(int ch, FILE *fp);
int getc(FILE *fp);
int fgetc(FILE *fp);
```

Os programas a seguir ilustram a utilização destas funções [SCH 96].

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

void main(int argc, char *argv[]){ // recebe o nome do arquivo como parâmetro na chamada do programa
  FILE *fp;
  char ch;
  if(argc!=2) {
    printf("Voce esqueceu de digitar o nome do arquivo\n");
    exit(1);
  }
  if((fp=fopen(argv[1], "w"))==NULL) {
    printf("Arquivo nao pode ser aberto\n");
    exit(1);
  }
 do {
    ch = getchar();
    putc(ch, fp);
  } while(ch!='$');
  fclose(fp);
}
```

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

void main(int argc, char *argv[]) { // recebe o nome do arquivo como parâmetro na chamada do programa
  FILE *fp;
  char ch;
  if(argc!=2) {
    printf("Voce esqueceu de digitar o nome do arquivo\n");
    exit(1);
  }
 if((fp=fopen(argv[1], "r"))==NULL) {
    printf("Arquivo nao 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);
}
```

Como 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. O protótipo para esta função é:

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

Esta função, que pode ser aplicada tanto para arquivo texto como para arquivo binário, devolve verdadeiro se o final do arquivo foi atingido; caso contrário, devolve 0. Assim, a rotina a seguir lê um arquivo binário até que o final do arquivo seja encontrado:

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

Além de *getc()* e *putc()*, C suporta as funções relacionadas ***fputs()*** e ***fgets()***, que efetuam as operações de leitura e gravação de *strings* de e para um arquivo em disco. São os seguintes os seus protótipos:

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

A função *fputs()* opera como *puts()*, mas escreve uma *string* na *stream* especificada. EOF será devolvido se ocorrer um erro. A função *gets()* 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 para nulo se ocorrer um erro [SCH 96].

O próximo exemplo mostra a leitura de arquivos texto usando a função *fgets()* [PIN 00].

```
// Exemplo de uso de arquivo texto.  Este programa lê um arquivo texto e imprime
// o seu conteúdo na tela.
#include <stdio.h>
#include <conio.h>

void main() {
  FILE *arq;
  char Linha[100];
  char *result;
  int i;

  clrscr();
  // Abre um arquivo TEXTO para LEITURA
  arq = fopen("ArqTeste.txt", "rt");
  if (arq == NULL) {  // Se houve erro na abertura
     printf("Problemas na abertura do arquivo\n");
     return;
  }
  i = 1;
  while (!feof(arq)) {
        // Lê uma linha (inclusive com o '\n')
     result = fgets(Linha, 100, arq);  // o 'fgets' lê até 99 caracteres ou até o '\n'
     if (result)  // Se foi possível ler
        printf("Linha %d : %s",i,Linha);
     i++;
  }
  fclose(arq);
}
```

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.

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

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

Esta função 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, para que um erro não seja perdido.

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

```
int remove (const char *filename);
```

Esta função devolve zero se for bem sucedida, e um valor diferente de zero caso contrário.

Para esvaziar o conteúdo de uma *stream* de saída deve-se utilizar a função ***fflush()***, que possui o seguinte protótipo:

*int fflush(FILE *fp);*

Esta 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. Ela retorna 0 para indicar sucesso, e EOF caso contrário.

Para ler e escrever tipos de dados maiores que um *byte*, o sistema de arquivos 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 (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. O tipo *size_t* é essencialmente o mesmo que um *unsigned*. Finalmente, "fp" é um ponteiro para uma *stream* aberta anteriormente.

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. Quando o arquivo for aberto para dados binários, *fread()* e *fwrite()* podem ler e escrever qualquer tipo de informação. Uma das aplicações mais úteis de *fread()* e *fwrite()* envolve ler e escrever tipos de dados definidos pelo usuário, especialmente estruturas. A utilização destas funções estão exemplificadas no próximo programa [SCH 96].

```
// 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("arquivo nao 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);
}
```

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

*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.

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".

Por exemplo, o seguinte programa lê uma *string* e um inteiro do teclado e os grava em um arquivo em disco. Depois, o programa lê o arquivo e exibe a informação na tela.

```
/* 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("arquivo nao pode ser aberto\n");
    exit(1);
  }
  printf("entre com uma string e um numero: ");
  fscanf(stdin, "%s%d", s, &t); /* lê do teclado    */
  fprintf(fp, "%s %d", s, t);   /* grava no arquivo */
  fclose(fp);

  if((fp=fopen("test", "r")) == NULL) {
    printf("arquivo nao 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()*.