Acesso a arquivos

Vejamos algumas das funções mais usuais para operações de entrada/saída em arquivos na linguagem C. A maioria das funções aqui descritas está declarada no arquivo de cabeçalho "stdio.h".

Conceitos básicos

Tipos de arquivos

Um arquivo armazena uma sequência de bytes, cuja interpretação fica a cargo da aplicação. Contudo, para facilitar a manipulação de arquivos, a linguagem C considera dois tipos de arquivos:

- **arquivos de texto**: contêm sequências de bytes representando caracteres de texto, separadas por caracteres de controle como **\n** ou **\r**. São usados para armazenar informação textual, como código-fonte, páginas Web, arquivos de configuração, etc.
- **arquivos binários**: contêm sequências de bytes, cuja interpretação fica totalmente a cargo da aplicação. São usualmente empregados para armazenar imagens, vídeos, músicas, dados compactados, etc.

Streams e descritores

As operações sobre arquivos em C podem ser feitas de duas formas:

- por **streams**: acesso em nível mais abstrato, independente de sistema operacional e portanto portável. Permite entrada/saída formatada, mas pode ter um desempenho um pouco inferior ao acesso de baixo nível. *Streams* são acessadas através de variáveis do tipo "FILE *".
- por **descritores**: acesso através dos descritores de arquivo fornecidos pelo sistema operacional, pouco portável mas com melhor desempenho.

No que segue são apresentadas as funções de acesso que fazem uso de *streams*, que valem para qualquer sistema operacional. Explicações sobre entrada/saída usando descritores UNIX podem ser encontradas nesta página.

Entrada e saída padrão

Cada programa em execução tem acesso a três arquivos padrão definidos no arquivo de cabeçalho "stdio.h", que são:

- "FILE * stdin": a entrada padrão, normalmente associada ao teclado do terminal, usada para informar dados ao programa ("scanf", por exemplo).
- "FILE * stdout": a saída padrão, normalmente associada à tela do terminal, usada para as saídas normais do programa ("printf", por exemplo).
- "FILE * stderr": a saída de erro, normalmente associada à tela do terminal, usada para mensagens de erro.

Abertura e fechamento de arquivos

Antes de ser usado, um arquivo precisa ser "aberto" pela aplicação (com execção dos arquivos padrão

descritos acima, que são abertos automaticamente). Isso é realizado através da chamada "fopen":

```
#include <stdio.h>
FILE * fopen (const char *filename, const char *opentype)
```

fopen() abre um arquivo indicado por "filename" e retorna um ponteiro para o *stream*. A *string* "opentype" define o modo de abertura do arquivo:

- **r** : abre um arquivo existente para leitura (*read*);
- **w** : abre um arquivo para escrita (*write*). Se o arquivo já existe, seu conteúdo é descartado. Senão, um novo arquivo vazio é criado;
- **a** : abre um arquivo para concatenação (*append*). Se o arquivo já existe, seu conteúdo é preservado e as escritas serão concatenadas no final do arquivo. Senão, um novo arquivo vazio é criado;
- **r+** : abre um arquivo existente para leitura e escrita. O conteúdo anterior do arquivo é preservado e o ponteiro é posicionado no início do arquivo;
- **w+** : abre um arquivo para leitura e escrita. Se o arquivo já existe, seu conteúdo é descartado. Senão, um novo arquivo vazio é criado;
- **a+** : abre um arquivo para escrita e concatenação. Se o arquivo já existe, seu conteúdo é preservado e as escritas serão concatenadas no final do arquivo. Senão, um novo arquivo vazio é criado. O ponteiro de leitura é posicionado no início do arquivo; as escritas são efetuadas no seu final.

```
#include <stdio.h>
int fclose (FILE *stream)
```

fclose() fecha um *stream*. Dados de saída que estão em *buffer* são escritos, enquanto que os dados de entrada são descartados.

```
#include <stdio.h>
int fcloseall (void)
```

fcloseall() fecha todos os *streams* abertos, de forma similar a "fclose". Isso também é efetuado quando a função "main" termina ou quando a função "exit" é invocada.

```
#include <stdio.h>
FILE * freopen (const char *filename, const char *opentype, FILE *stream)
```

freopen() fecha e abre novamente um *stream*, permitindo alterar o arquivo e/ou modo de acesso.

Exemplo: abre o arquivo "x" para leitura:

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

int main ()
{
   FILE *arq ;

   arq = fopen ("x", "r") ;

   if ( arq == NULL )
   {
      perror ("Erro ao abrir arquivo x") ;
      exit (1) ; // encerra o programa com status 1
   }

   fclose (arq) ;
```

```
   exit (0) ;
}
```

Exemplo: abre o arquivo "x" para leitura/escrita, criando-o se não existir:

```
#include <stdio.h>

int main (int argc, char *argv[])
{
   FILE *arq ;

   arq = fopen ("x", "w+") ;

   if ( arq == NULL )
   {
      perror ("Erro ao abrir/criar arquivo x") ;
      exit (1) ; // encerra o programa com status 1
   }

   fclose (arq) ;
   exit (0) ;
}
```

Arquivos-texto

Arquivos-texto contêm sequências de bytes representando um texto simples (sem formatações especiais, como negrito, itálico, etc), como código-fonte ou uma página em HTML, por exemplo.

Em um arquivo-texto, os caracteres do texto são representados usando uma codificação padronizada, para que possam ser abertos por diferentes aplicações em vários sistemas operacionais. As codificações de caracteres mais usuais são:

- ASCII: criada em 1963 para representar os caracteres comuns da língua inglesa, usando 7 bits (valores entre 0 e 127);
- ISO-8859-X: conjunto de extensões da codificação ASCII para suportar outras línguas com alfabeto latino, como o Português. Os caracteres entre 0 e 127 são os mesmos do código ASCII, enquanto os caracteres entre 128 e 255 são específicos. Por exemplo, a extensão ISO-8859-1 contém os caracteres acentuados e cedilhas da maior parte das linguagens latinas (Português, Espanhol, Francês etc), e por isso, em algumas aplicações, o ISO-8859-1 é apelidado de *iso-latin*;
- UTF8: codificação baseada no padrão Unicode, e pode representar mais de um milhão de caracteres em todas as línguas conhecidas, além de sinais gráficos (como *emojis*). Os caracteres em UTF8 podem usar entre 1 e 4 bytes para sua representação, o que torna sua manipulação mais complexa.

Diga *man ascii* *man iso_8859-1* *man utf-8* para as páginas de manual dos códigos.

Além dos caracteres em si, as codificações geralmente suportam **caracteres de controle**, que permitem representar algumas estruturas básicas de formatação do texto, como quebras de linha. Alguns caracteres de controle presentes nas codificações acima são:

```
nome             valor  representação
---------------  -----  -------------
null               0    NUL, \0, ^@
bell               7    BEL, \a, ^G
backspace          8    BS,  \b, ^H
tab                9    HT,  \t, ^I
line feed         10    LF,  \n, ^J
form feed         12    FF,  \f, ^L
carriage return   13    CR,  \r, ^M
```

```
escape          27    ESC,   , ^[
```

Saída simples

As funções abaixo permitem gravar caracteres ou *strings* simples em *streams*.

```
#include <stdio.h>

int fputc   (int c, FILE *stream)   // escreve o caractere equivalente a "c" em "arq"
int putc    (int c, FILE *stream)   // idem, implementada como macro
int putchar (int c)                 // idem, em "stdout"

int fputs (const char *s, FILE *stream)  // escreve a string "s" no stream indicado
int puts  (const char *s)                // idem, em "stdout"
```

Entrada simples

As funções abaixo permitem ler caracteres isolados de um *stream*. O valor lido é um "int" indicando o caractere lido ou então o valor especial "EOF" (*End-Of-File*):

```
#include <stdio.h>

int fgetc (FILE *stream)  // Lê o próximo caractere do stream indicado
int getc  (FILE *stream)  // Idem, implementado como macro (mais rápido)
int getchar ()            // Idem, sobre ''stdin''
```

Para a leitura de *strings*, **gets()** lê caracteres de "stdin" até encontrar um *newline* e os armazena na *string* "s". O caractere *newline* é descartado.

```
#include <stdio.h>
char * gets (char *s)
```

Atenção: a função "gets" é perigosa, pois não provê segurança contra *buffer overflow* na *string* "s", que pode ocorrer se a *string* lida da entrada for mais longa do que o espaço alocado. Sempre que possível, use a função "fgets" ou "getline".

fgets() lê uma linha de caracteres do *stream* e a deposita na *string* "s". O tamanho da linha é limitado em "count-1" caracteres, aos quais é adicionado o "'\0'" que marca o fim da *string*. O *newline* é incluído.

```
#include <stdio.h>
char * fgets (char *s, int count, FILE *stream)
```

Saída formatada

As operações de entrada e saída formatada usam padrões para formatação dos diversos tipos de dados descritos em livros de programação em C e no manual da GLibC.

printf() escreve usando a formatação definida em "template" no *stream* de saída padrão "stdout".

```
#include <stdio.h>
int printf (const char *template, ...)
```

fprintf() é idêntica a "printf", mas usa o stream indicado.

```
#include <stdio.h>
int fprintf (FILE *stream, const char *template, ...)
```

sprintf() é similar a "printf", mas a saída é depositada na *string* "s".

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

```
int sprintf (char *s, const char *template, ...)
```

Atenção: o programador deve garantir que s contenha espaço suficiente para receber a saída; caso contrário, pode ocorrer um *buffer overflow* com consequências imprevisíveis. As funções "snprintf" e "asprintf" são mais seguras e evitam esse problema.

Entrada formatada

scanf() lê dados do stream "stdin" de acordo com a formatação definida na *string* "template". Os demais argumentos são ponteiros para as variáveis onde os dados lidos são depositados. Retorna o número de dados lidos ou "EOF".

```
#include <stdio.h>
int scanf (const char *template, ...)
```

fscanf() é Similar a "scanf", mas usando como entrada o *stream* indicado.

```
#include <stdio.h>
int fscanf (FILE *stream, const char *template, ...)
```

sscanf() é similar a "scanf", mas usando como entrada a *string* "s".

```
#include <stdio.h>
int sscanf (const char *s, const char *template, ...)
```

Erros e fim de *stream*

A macro **EOF** define o valor retornado por algumas funções para indicar fim do arquivo ou erro.

```
int EOF
```

feof() retorna valor não nulo se o *stream* chegou ao seu fim.

```
#include <stdio.h>
int feof (FILE *stream)
```

ferror() retorna um valor não nulo se ocorreu um erro em algum acesso anterior ao *stream*.

```
#include <stdio.h>
int ferror (FILE *stream)
```

Além de ajustar o indicador de erro do *stream*, as funções de acesso a *streams* também ajustam a variável "errno".

Arquivos binários

Todos os arquivos contêm sequências de bytes, mas costuma-se dizer que um arquivo é "binário" quando seu conteúdo não é informação textual, e não representa texto codificado em ASCII ou outra codificação.

Arquivos binários são usados para armazenar informações mais complexas do que texto puro, como imagens, música, filmes, código executável, etc. A interpretação do conteúdo de um arquivo binário é de responsabilidade do programador da aplicação que faz uso do arquivo binário.

A linguagem C oferece funções para ler e escrever blocos de bytes em arquivos, ou que efetuam a cópia desses bytes da memória da aplicação para o arquivo, e vice-versa.

Leitura/escrita de blocos

A funções a seguir permitem ler/escrever blocos de bytes em arquivos binários:

fread() lê até "count" blocos de tamanho "size" bytes cada um e os deposita no vetor "data", a partir do *stream* indicado. Retorna o número de blocos lidos.

```
#include <stdio.h>
size_t fread (void *data, size_t size, size_t count, FILE *stream)
```

fwrite() escreve até "count" blocos de tamanho "size" bytes do vetor "data" no *stream* indicado. Retorna o número de blocos escritos.

```
#include <stdio.h>
size_t fwrite (const void *data, size_t size, size_t count, FILE *stream)
```

Essas funções também podem ser usadas para ler/escrever em arquivos-texto, pois texto é uma sequência de bytes.

Exemplo de uso

Este exemplo manipula um arquivo binário contendo um fichário de alunos; são implementadas (em arquivos separados) as operações de escrita de ficha, listagem do fichário e ordenação do fichário:

```
#ifndef __FICHA__
#define __FICHA__

#define TAMNOME  100

typedef struct Ficha_t
{
  char nome[TAMNOME];
  int idade ;
  char sexo ;
  float grr ;
} Ficha_t ;

#endif
```

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

#define FICHARIO "fichario.dat"

int main (int argc, char *argv[])
{
  FILE *arq ;
  Ficha_t ficha ;
  int ret ;

  // abre o fichário em modo "append"
  arq = fopen (FICHARIO, "a") ;
  if (!arq)
  {
    perror ("Erro ao abrir fichario") ;
    exit (1) ;
  }

  // lê dados da ficha
  printf ("Nome : ") ;
  scanf ("%[^\n]", ficha.nome) ;
  printf ("Idade: ") ;
  scanf ("%d", &ficha.idade) ;
```

```
  printf ("Sexo: ") ;
  scanf (" %1c", &ficha.sexo) ;
  printf ("GRR  : ") ;
  scanf ("%f", &ficha.grr) ;
  printf ("\n") ;

  // escreve a ficha no final do fichario
  ret = fwrite (&ficha, sizeof(Ficha_t), 1, arq) ;
  if (ret)
    printf ("Gravou com sucesso!\n") ;
  else
    printf ("Erro ao gravar...\n") ;

  // fecha o fichario
  fclose (arq) ;
  return (0) ;
}
```

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

#define FICHARIO "fichario.dat"

int main (int argc, char *argv[])
{
  FILE *arq ;
  Ficha_t ficha ;

  // abre o fichário em modo leitura
  arq = fopen (FICHARIO, "r") ;
  if (!arq)
  {
    perror ("Erro ao abrir fichario") ;
    exit (1) ;
  }

  // lê uma ficha do fichario
  fread (&ficha, sizeof(ficha), 1, arq) ;
  while (! feof(arq))
  {
    // imprime a ficha lida
    printf ("Nome : %s\n", ficha.nome) ;
    printf ("Idade: %d\n", ficha.idade) ;
    printf ("Sexo : %c\n", ficha.sexo) ;
    printf ("GRR  : %f\n", ficha.grr) ;
    printf ("\n") ;

    // le a proxima ficha
    fread (&ficha, sizeof(ficha), 1, arq) ;
  }

  // fecha o fichario
  fclose (arq) ;
  return (0) ;
}
```

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

#define FICHARIO "fichario.dat"
#define MAXFICHAS 1000

Ficha_t fichario[MAXFICHAS] ;
```

```
int numfichas ;

int main (int argc, char *argv[])
{
  FILE *arq ;
  Ficha_t aux ;
  int i, j, menor ;

  // abre o fichário em leitura/escrita, preservando o conteúdo
  arq = fopen (FICHARIO, "r+") ;
  if (!arq)
  {
    perror ("Erro ao abrir fichario") ;
    exit (1) ;
  }

  // lê fichas do fichario, todas de uma vez
  numfichas = fread (fichario, sizeof(Ficha_t), MAXFICHAS, arq) ;
  printf ("Encontrei %d fichas no fichário\n", numfichas) ;

  // ordena as fichas por idade (por seleção)
  for (i=0; i < numfichas-1; i++)
  {
    // encontra o menor elemento no restante do vetor
    menor = i ;
    for (j=i+1; j < numfichas; j++)
      if (fichario[j].idade < fichario[menor].idade)
        menor = j ;

    // se existe menor != i, os troca entre si
    if (menor != i)
    {
      aux = fichario[i] ;
      fichario[i] = fichario[menor] ;
      fichario[menor] = aux ;
    }
  }

  // retorna o ponteiro ao início do fichário
  rewind (arq) ;

  // escreve as fichas do fichario, todas de uma vez
  fwrite (fichario, sizeof(Ficha_t), numfichas, arq) ;

  // fecha o fichario
  fclose (arq) ;
  return (0) ;
}
```

Posicionamento no arquivo

Para cada arquivo aberto em uma aplicação, o sistema operacional mantém um contador interno indicando a posição da próxima operação de leitura ou escrita. Esse contador é conhecido como **ponteiro de posição** (embora não seja realmente um ponteiro).

Por *default*, as operações em um arquivo em C ocorrem em posições sucessivas dentro do arquivo: cada leitura (ou escrita) corre **após** a leitura (ou escrita) precedente, até atingir o final do arquivo. Essa forma de acesso ao arquivo é chamada de **acesso sequencial**.

Por vezes uma aplicação precisa ler ou escrever em posições específicas de um arquivo, ou precisa voltar a ler uma posição do arquivo que já percorreu anteriormente. Isso ocorre frequentemente em aplicações que manipulam arquivos muito grandes, como vídeos ou bases de dados. Para isso é necessária uma forma

de **acesso direto** a posições específicas do arquivo.

Em C, o acesso direto a posições específicas de um arquivo é feita através de funções de **posicionamento de ponteiro**, que permitem alterar o valor do ponteiro de posição do arquivo, antes da operação de leitura/escrita desejada.

As funções para acessar o ponteiro de posição de um arquivo em C são:

fseek() ajusta posição do ponteiro no *stream* indicado.

```
#include <stdio.h>
int fseek (FILE *stream, long int offset, int whence)
```

O ajuste é definido por "offset", enquanto o valor de "whence" indica se o ajuste é relativo ao início do arquivo ("SEEK_SET"), à posição corrente ("SEEK_CUR") ou ao final do arquivo ("SEEK_END"). Ver também "fseeko" e "fseeko64".

```
                              // posiciona o ponteiro de "arq":
fseek (arq, 1000, SEEK_SET) ; // 1000 bytes após o início
fseek (arq,  300, SEEK_END) ; // 300 bytes antes do fim
fseek (arq, -500, SEEK_CUR) ; // 500 bytes antes da posição atual
```

rewind() reposiciona o ponteiro no início (posição 0) do *stream* indicado.

```
#include <stdio.h>
void rewind (FILE *stream)
```

ftell() informa a posição corrente de leitura/escrita em "stream" (ver também "ftello" e "ftello64" no manual).

```
#include <stdio.h>
long int ftell (FILE *stream)
```

Todas as funções de manipulação do ponteiro de arquivo consideram as *posições em bytes* a partir do início do arquivo, nunca em registros ou estruturas.

Exercícios

1. Escreva um programa em C para informar o número de caracteres presentes em um arquivo de texto (dica: man wc).

2. Escreva um programa em C para ler um arquivo "minusc.txt" e escrever um arquivo "maiusc.txt" contendo o mesmo texto em maiúsculas (dica: man tr).

3. Escreva um programa "mycp" para fazer a cópia de um arquivo em outro: "mycp arq1 arq2". Antes da cópia, "arq1" deve existir e "arq2" não deve existir. Mensagens de erro devem ser geradas caso essas condições não sejam atendidas, ou o nome dado a "arq2" seja inválido. Defina cuidadosamente o que seja *inválido*.

4. O comando "grep" do UNIX imprime na saída padrão (stdout) as linhas de um arquivo de entrada que contenham uma determinada *string* informada como parâmetro. Escreva esse programa em C (dica: use a função "strstr").

5. Escreva três programas C separados para:

    a. escrever um arquivo com 10 milhões de inteiros "long" aleatórios, armazenados em modo binário;

b. ler o arquivo de inteiros em um vetor, ordenar o vetor e reescrever o arquivo;

c. escrever na tela os primeiros 10 números e os últimos 10 números contidos no arquivo.

6. Mais exercícios ne capítulo 11 da apostila do NCE