Acesso a arquivos

Video desta aula

Aqui serão descritas 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`:

```
#include <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**: contém sequências de bytes representando caracteres de texto, separadas por caracteres de controle como `\n` ou `\r` e usando uma codificação como ASCII, ISO-8859-1 ou UTF-8. São usados para armazenar informação textual, como código-fonte, páginas Web, arquivos de configuração, etc.
- **arquivos binários**: contém 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 um nível mais elevado, independente de sistema operacional e portanto portável. Permite entrada/saída formatada, mas pode ter um desempenho 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.

Na sequência deste texto serão apresentadas as funções de acesso usando *streams*, que são o padrão da linguagem C e valem para qualquer sistema operacional. Explicações sobre entrada/saída usando descritores UNIX podem ser encontradas nesta página.

Entradas e saídas 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 a entrada de dados do 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.

Esses três arquivos não precisam ser abertos, eles estão prontos para uso quando o programa inicia. Geralmente eles estão associados ao terminal onde o programa foi lançado, mas podem ser redirecionados pelo *shell* (mais detalhes aqui).

Abrindo e fechando arquivos

Antes de ser usado, um arquivo precisa ser "aberto" pela aplicação (com exceção dos arquivos padrão descritos acima, que são abertos automaticamente). Isso é realizado através da chamada `fopen`:

```
FILE* fopen (const char *filename, const char *mode)
```

Abre um arquivo indicado por `filename` e retorna um ponteiro para o *stream*. A *string* `mode` 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, enquanto as escritas são efetuadas no seu final.

Os modos `a` e `a+` **sempre** escreverão no final do arquivo, mesmo se o cursor do mesmo for movido para outra posição.

Fecha um *stream*. Os dados de saída em *buffer* são escritos, enquanto dados de entrada são descartados:

```
int fclose (FILE* stream)
```

Fecha e abre novamente um *stream*, permitindo alterar o arquivo e/ou modo de abertura:

```
FILE* freopen (const char *filename, const char *mode, FILE *stream)
```

Exemplo: abrindo o arquivo `x` em leitura:

fopen-read.c

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

int main ()
{
   FILE* arq ;

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

   if ( ! arq )
```

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

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

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

fopen-write.c

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

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

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

   if ( ! arq )
   {
      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 hoje são:

- ASCII: criada em 1963 para representar os caracteres comuns da língua inglesa, usando 7 bits (valores entre 0 e 127).
- ISO-8859: conjunto de extensões da codificação ASCII para suportar outras línguas com alfabeto latino, como o Português. Os caracteres entre 0 e 127 os mesmos da tabela 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 ocidentais (Português, Espanhol, Francês etc).
- UTF-8: codificação baseada no padrão Unicode, capaz de representar mais de um milhão de caracteres em todas as línguas conhecidas, além de sinais gráficos (como *emojis*). Os caracteres em UTF-8 podem usar entre 1 e 4 bytes para sua representação, o que torna sua manipulação mais complexa em programas.

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 ^[

Escrita de arquivos

Escrita simples

Estas funções permitem gravar caracteres ou strings simples em *streams*.

```
int fputc   (int c, FILE* stream)   // escreve o caractere c no stream
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
int puts  (const char *s)                // idem, em "stdout"
```

Um exemplo de uso de operações de escrita simples em arquivo:

escreve-ascii.c

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

int main ()
{
  FILE *arq ;
  unsigned char c ;

  // abre o arquivo em escrita
  arq = fopen ("ascii.txt", "w+") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir/criar arquivo") ;
     exit (1) ;
  }

  // escreve os caracteres ascii
  fputs ("caracteres ASCII:", arq) ;
  for (c = 32; c < 128; c++)
  {
    fputc (c, arq) ;
    fputc (' ', arq) ;
```

```
  }
  fputc ('\n', arq) ;

  // fecha o arquivo
  fclose (arq) ;
}
```

Escrita 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.

Escreve dados usando a formatação definida em `format` no *stream* de saída padrão `stdout`:

```
int printf (const char* format, ...)
```

Idêntico a `printf`, usando o stream indicado:

```
int fprintf (FILE* stream, const char* format, ...)
```

Similar a `printf`, mas a saída é depositada na string `str`:

```
int sprintf (char* str, const char* format, ...)
```

Atenção: o programador deve garantir que `str` tenha tamanho 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.

escreve-tabuada.c

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

int main ()
{
  FILE *arq ;
  int i, j ;

  // abre o arquivo em escrita
  arq = fopen ("tabuada.txt", "w+") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir/criar arquivo") ;
     exit (1) ;
  }

  // escreve o cabeçalho
  fprintf (arq, "Tabuada:\n") ;

  fprintf (arq, "       ") ;
  for (j = 0; j <= 10; j++)
      fprintf (arq, "%4d", j) ;
```

```
  fprintf (arq, "\n") ;

  fprintf (arq, "       ") ;
  for (j = 0; j <= 10; j++)
      fprintf (arq, "----") ;
  fprintf (arq, "\n") ;

  // escreve as linhas de valores
  for (i = 0; i <= 10; i++)
  {
    fprintf (arq, "%4i | ", i) ;
    for (j = 0; j <= 10; j++)
      fprintf (arq, "%4d", i*j) ;
    fprintf (arq, "\n") ;
  }

  // fecha o arquivo
  fclose (arq) ;
}
```

Leitura de arquivos

Leitura simples

Estas funções 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*):

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

Para a leitura de strings  :

```
char* gets (char *s)
```

Lê caracteres de `stdin` até encontrar um *newline* e os armazena na string `s`. O caractere *newline* é descartado.

Atenção: a função `gets` é **perigosa**, pois não provê segurança contra *overflow* na string `s`. Sempre que possível, deve ser usada a função `fgets` ou `getline`.

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 incluso.

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

O exemplo a seguir lê e numera as 10 primeiras linhas de um arquivo:

numera-linhas.c

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

#define LINESIZE 1024

int main ()
{
  FILE *arq ;
  int i ;
  char line[LINESIZE+1] ;

  // abre o arquivo em leitura
  arq = fopen ("dados.txt", "r") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir arquivo") ;
     exit (1) ;
  }

  // lê as 10 primeiras linhas do arquivo
  for (i = 0; i < 10; i++)
  {
    fgets (line, LINESIZE, arq) ;
    printf ("%d: %s", i, line) ;
  }

  // fecha o arquivo
  fclose (arq) ;
}
```

Leitura formatada

Lê dados do *stream* `stdin` de acordo com a formatação definida na string `format`. Os demais argumentos são ponteiros para as variáveis onde os dados lidos são depositados. Retorna o número de dados lidos ou EOF:

```c
int scanf (const char* format, ...)
```

Similar a `scanf`, mas usando como entrada o *stream* indicado:

```c
int fscanf (FILE* stream, const char* format, ...)
```

Similar a `scanf`, mas usando como entrada a string `s`:

```c
int sscanf (const char* s, const char* format, ...)
```

O exemplo a seguir lê 10 valores reais de um arquivo:

le-valores.c

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

int main ()
```

```
{
  FILE *arq ;
  int i ;
  float value ;

  // abre o arquivo em leitura
  arq = fopen ("numeros.txt", "r") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir arquivo") ;
     exit (1) ;
  }

  // lê os 10 primeiros valores do arquivo
  for (i = 0; i < 10; i++)
  {
    fscanf (arq, "%f", &value) ;
    printf ("%d: %f\n", i, value) ;
  }

  // fecha o arquivo
  fclose (arq) ;
}
```

Experimente executá-lo com os dados de entrada abaixo. Pode explicar o que acontece?

numeros.txt

```
10 21 4
  23.7  55 -0.7
6 5723.8, 455
1, 2, 3, 4
```

A função `scanf` considera espaços (espaços em branco, tabulações e novas linhas) como separadores *default* dos campos de entrada. O arquivo `numeros.txt` contém uma vírgula, que não é um separador, então a leitura não pode prosseguir até que a vírgula seja lida.

Um bloco de leitura mais robusto, imune a esse problema, seria:

```
// lê os 10 primeiros valores do arquivo
i = 0 ;
while (i < 10)
{
  ret = fscanf (arq, "%f", &value) ;

  // fim de arquivo ou erro?
  if (ret == EOF)
    break ;

  // houve leitura?
  if (ret > 0)
  {
    printf ("%d: %f\n", i, value) ;
```

```
    i++ ;
  }
  // não houve, tira um caractere e tenta novamente
  else
    fgetc (arq) ;
}
```

Fim de arquivo

Muitas vezes deseja-se ler os dados de um arquivo até o fim, mas não se conhece seu tamanho a priori. Para isso existem funções e macros que indicam se o final de um arquivo foi atingido.

A função recomendada para testar o final de um arquivo é `feof(stream)`. Ela retorna 0 (zero) se o final do arquivo **não foi** atingido. Eis um exemplo de uso:

numera-todas.c

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

#define LINESIZE 1024

int main ()
{
  FILE *arq ;
  int i ;
  char line[LINESIZE+1] ;

  // abre o arquivo em leitura
  arq = fopen ("dados.txt", "r") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir arquivo") ;
     exit (1) ;
  }

  // lê TODAS as linhas do arquivo
  i = 1 ;
  fgets (line, LINESIZE, arq) ;     // tenta ler uma linha
  while (! feof (arq))              // testa depois de tentar ler!
  {
    printf ("%d: %s", i, line) ;
    fgets (line, LINESIZE, arq) ;   // tenta ler a próxima linha
    i++ ;
  }

  // fecha o arquivo
  fclose (arq) ;
}
```

Observe que a função `feof()` indica TRUE somente **após** o final do arquivo ter sido atingido,

ou seja, após uma leitura falhar. Por isso, `feof()` deve ser testada **depois** da leitura e não antes.

A macro EOF representa o valor devolvido por funções de leitura de caracteres como `getchar` e `fgetc` quando o final do arquivo é atingido:

le-eof.c

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

#define LINESIZE 1024

int main ()
{
  FILE *arq ;
  char c ;

  // abre o arquivo em leitura
  arq = fopen ("dados.txt", "r") ;
  if ( ! arq )
  {
     perror ("Erro ao abrir arquivo") ;
     exit (1) ;
  }

  // lê os caracteres até o fim do arquivo
  c = fgetc (arq) ;       // tenta ler um caractere
  while (c != EOF)      // não é o fim do arquivo
  {
    printf ("%c ", c) ;       // tenta ler o próximo
    c = fgetc (arq) ;
  }

  // fecha o arquivo
  fclose (arq) ;
}
```

Por fim, esta função retorna um valor não nulo se ocorreu um erro no último acesso ao *stream*:

```c
int ferror (FILE* stream)
```

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

Exercícios

1. Escreva um programa em C para informar o número de caracteres presentes em um arquivo de texto.
2. Escreva um programa em C que leia um arquivo de texto com números reais (um número por linha) e informe a média dos valores lidos.
3. Escreva um programa em C para ler um arquivo `minusc.txt` e escrever um arquivo `maiusc.txt` contendo o mesmo texto em maiúsculas.

4. O arquivo mapa.txt contém o mapa de um nível do jogo Boulder Dash. Escreva um programa em C que carregue esse mapa em uma matriz de caracteres.
5. 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. Para acessar os nomes dos arquivos na linha de comando use os parâmetros `argc` e `argv` (veja aqui).
6. 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`).

Mais exercícios no capítulo 11 da apostila do NCE/UFRJ.