Programação de Computadores

Arquivos

Neste tópico abordaremos a manipulação de dados em memória secundária em C.

Prof. Ciro Cirne Trindade
Prof. Thiago Ferauche

Introdução

- O armazenamento de dados em variáveis é temporário
  - Os dados são perdidos quando o programa termina
- Os arquivos são utilizados para conservação permanente dos dados
  - Os arquivos são armazenados em dispositivos de memória secundária, especialmente em discos

Streams e Arquivos

- O C provê um nível de abstração entre o programador e o dispositivo utilizado
- Esta abstração é chamada *stream* e o dispositivo real é chamado arquivo
  - Uma *stream* é uma variável que permite ao programador enviar e receber dados de um dispositivo

Streams

- O sistema de arquivos de C é projetado para trabalhar com uma ampla variedade de dispositivos
  - Terminal, discos e fitas
- Embora os dispositivos sejam diferentes, o sistema de arquivos de C transforma-os em um dispositivo lógico chamado *stream*
- Todas as *streams* comportam-se de forma semelhante

Streams

- Existem 2 tipos de *streams*:
  - Texto
  - Binário

Streams de texto

- Uma *stream* de texto é uma sequência de caracteres
  - Organizado em linhas terminadas por um caractere de nova linha (opcional na última linha)
- Certas traduções podem ocorrer conforme exigido pelo sistema
  - Por exemplo, uma nova linha pode ser convertida em um par retorno de carro/nova linha

Streams de texto

- Poderá não haver uma relação um para um entre os caracteres que são escritos (ou lidos) e aqueles nos dispositivos externos
  - Consequentemente, o número de caracteres escritos (ou lidos) pode não ser o mesmo que aqueles que se encontram no dispositivo externo

Streams binárias

- Uma *stream* binária é uma sequência de bytes
  - Correspondência de um para um com aqueles encontrados no dispositivo externo
  - Não ocorre nenhuma tradução de caracteres
  - O número de bytes escritos (ou lidos) é o mesmo no dispositivo externo

Arquivos

- Em C, um arquivo pode ser qualquer coisa, desde um arquivo de dados até um terminal ou uma impressora
- Nem todos os arquivos apresentam os mesmos recursos
  - Por exemplo, um arquivo em disco pode suportar acesso aleatório enquanto um teclado não
- Todas as *streams* são iguais, mas não todos os arquivos

Arquivos

- Um arquivo de dados termina com um marcador de final de arquivo (EOF – *end of file*)
- Você associa uma *stream* a um arquivo específico realizando uma operação de abertura de arquivo
  - Uma vez o arquivo aberto, informações podem ser trocadas entre ele e o seu programa

Arquivos

- Se um arquivo suporta acesso aleatório, abrir esse arquivo também inicializa o *indicador de posição no arquivo*
- Este indicador é atualizado automaticamente após operações de leitura ou escrita no arquivo

Arquivos

- Por questões de eficiência, as operações de leitura e escrita em arquivos são bufferizadas
- Um arquivo é desassociado de uma *stream* específica por meio de uma operação de fechamento
  - O conteúdo da *stream* (buffer) é descarregado no dispositivo
  - Os arquivos são fechados automaticamente quando o programa termina

Arquivos

- Cada *stream* associada a um arquivo tem uma estrutura de controle de arquivo do tipo **`FILE`**
  - Definida em stdio.h
  - Contém um descritor de arquivo, incluindo informações a respeito do buffer usado para I/O no arquivo

Abrindo um arquivo

- A função **`fopen()`** abre uma *stream* para uso e associa um arquivo a ele
  - Devolve um ponteiro para o arquivo associado ou **`NULL`**
- Protótipo de **`fopen()`**

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

  - *nomearq* é uma string com o nome do arquivo a ser aberto
  - *modo* é o modo de abertura do arquivo

Modos de abertura do arquivo

- **`r`** abre um arquivo texto para leitura
- **`w`** cria um arquivo de texto para escrita. Se o arquivo existir, elimina seu conteúdo
- **`a`** anexa a um arquivo texto. Abre ou cria um arquivo para gravação no final do arquivo
- **`r+`** abre um arquivo texto para leitura/escrita
- **`w+`** cria um arquivo de texto para leitura/escrita. Se o arquivo existir, elimina seu conteúdo
- **`a+`** anexa a um arquivo texto. Abre ou cria um arquivo, a escrita será no final do arquivo

Modos de abertura do arquivo

- **rb** abre um arquivo binário para leitura
- **wb** cria um arquivo de binário para escrita. Se o arquivo existir, elimina seu conteúdo
- **ab** anexa a um arquivo binário. Abre ou cria um arquivo para gravação no final do arquivo
- **r+b** abre um arquivo binário para leitura/escrita
- **w+b** cria um arquivo de binário para leitura/escrita. Se o arquivo existir, elimina seu conteúdo
- **a+b** anexa a um arquivo binário. Abre ou cria um arquivo, a escrita será no final do arquivo

Abrindo um arquivo

- Para abrir um arquivo chamado teste, permitindo escrita, pode-se escrever:

Devolve `NULL` se houver um erro de abertura

```
FILE * fp;
if ((fp = fopen("teste","w")) == NULL) {
   printf("Erro de abertura do arquivo");
   return 1;
}
```

Fechando um arquivo

- A função **`fclose()`** fecha um *stream* que foi aberto por meio de uma chamada a **`fopen()`**
- Protótipo de **`fclose()`**

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

  - Aonde *fp* é o ponteiro de arquivo devolvido por **`fopen()`**
  - Um valor de retorno igual a zero indica uma operação de fechamento bem sucedida

fgetc() e fputc()

- As funções **fgetc()** e **fputc()** podem ser usadas para ler e escrever caracteres/bytes em um arquivo, respectivamente
- Protótipos de `fgetc()` e `fputc()`
  - **`int fgetc(FILE * fp);`**
  - **`int fputc(int ch, FILE * fp);`**
    - Devolve o caractere escrito, ou `EOF` se houver um erro

Exemplo: exibindo o conteúdo de um arquivo texto

```
#include <stdio.h>
int main() {
    FILE * fp;
    char nomearq[40], ch;
    printf("Entre com o nome do arquivo: ");
    scanf("%39[^\n]", nomearq);
    if ((fp = fopen(nomearq, "r")) == NULL) {
        printf("Erro de abertura do arquivo %s\n",
                nomearq);
        return 1;
    }
    while ((ch = fgetc(fp)) != EOF) {
        fputc(ch, stdout);
    }
    fclose(fp);
    return 0;
}
```

Usando `feof()`

- Em arquivos binários, uma marca igual a **`EOF`** pode ser lida apesar do final do arquivo físico não ter sido atingido
- A função **`feof()`** indica quando o final de um arquivo binário foi de fato atingido
- Protótipo de **`feof()`**

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

  - Devolve verdadeiro ser o final do arquivo foi atingido

fprintf() e fscanf()

- Essas funções operam exatamente como **printf()** e **scanf()** exceto por operarem com arquivos
- Protótipos de **fprintf()** e **fscanf()**

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

  - Devolve o número de caracteres impressos

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

  - Devolve o número de itens lidos

Trabalhando com strings: `fputs()` e `fgets()`

- As funções **`fputs()`** e **`fgets()`** efetuam as operações de leitura e escrita de strings de e para um arquivo em disco
- Protótipo de **`fputs()`**

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

  - Devolve **`EOF`** se houver um erro, ou 1
- Protótipo de **`fgets()`**

```
char * fgets(char * str, int tamanho,
  FILE * fp);
```

  - Devolve **`NULL`** se ocorrer um erro, ou *str*

rewind()

- Posiciona o indicador de posição de arquivo no início do arquivo especificado como seu argumento
- Protótipo de **rewind()**

```
void rewind(FILE * fp);
```

Apagando arquivos

- A função **`remove()`** apaga o arquivo especificado
- Protótipo de **`remove()`**:

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

  - Recebe como argumento o nome do arquivo a ser apagado
  - Devolve zero, caso seja bem-sucedida, e um valor diferente de zero, caso contrário

Exemplo de um programa que apaga um arquivo através de **remove()**

```
#include <stdio.h>

int main(int argc, char * argv[]) {
   char sn;
   if (argc != 2) {
      fprintf(stderr, "Uso: %s <nomearq>\n", argv[0]);
      return 1;
   }
   printf("Apagar %s? (S/N): ", argv[1]);
   scanf(" %c", &sn);
   if (sn == 'S' || sn == 's') {
      if (remove(argv[1])) {
         fprintf(stderr, "O arquivo %s não pode ser
  apagado.\n", argv[1]);
         return 2;
      }
   }
   return 0;
}
```

Renomeando arquivos

- A função **`rename()`** renomeia o arquivo especificado
- Protótipo de **`rename()`**:

```
int rename(const char * nomeantigo,
  const char * novonome);
```

  - Recebe como argumentos o nome do arquivo que será renomeado e o novo nome desse arquivo
  - Devolve zero, caso seja bem-sucedida, e um valor diferente de zero, caso contrário

Exemplo de um programa que renomeia um arquivo através de **rename()**

```
#include <stdio.h>

int main(int argc, char * argv[]) {
   char sn;
   if (argc != 3) {
      fprintf(stderr, "Uso: %s <nomeantigo> <novonome>\n",
               argv[0]);
      return 1;
   }
   if (rename(argv[1], argv[2])) {
      fprintf(stderr, "Nao foi possível renomear o arquivo
  %s!\n", argv[1]);
      return 2;
   }
   return 0;
}
```

`fread()` e `fwrite()`

- As funções **`fread()`** e **`fwrite()`** permitem a leitura e escrita de blocos de qualquer tipo de dado
- Os dados são armazenados no formato binário

fread() e fwrite()

- Protótipo de **fread()**:

```
size_t fread(void * var, size_t
  num_bytes, size_t cont, FILE * fp);
```

Definido no stdio.h

  - Aonde *var* é o endereço de memória que receberá os dados do arquivo, *num_bytes* é o número de bytes a ler, *cont* determina quantos itens serão lidos e *fp* é o ponteiro para o arquivo
  - Devolve o número de itens lidos

fread() e fwrite()

- Protótipo de **fwrite()**:

```
size_t fwrite(void * var, size_t
  num_bytes, size_t cont, FILE * fp);
```

  - Aonde *var* é o endereço de memória da variável que contém os dados, *num_bytes* é o número de bytes a escrever, *cont* determina quantos itens serão escritos e *fp* é o ponteiro para o arquivo
  - Devolve o número de itens escritos

fread() e fwrite()

- As aplicações mais úteis de **fread()** e **fwrite()** envolver ler e escrever tipos de dados definidos pelo usuário, especialmente estruturas
- Considere a seguinte estrutura

```
typedef struct {
    int num_conta;
    char nome[60];
    float saldo;
} cliente;
```

fread()

- Para ler uma estrutura do tipo **cliente** de um arquivo faríamos:

```
FILE * fp;
cliente c;
if((fp = fopen("contas.dat","rb")) ==
     NULL) {
   fprintf(stderr,"Erro!\n");
   return 1;
}
fread(&c, sizeof(cliente), 1, fp);
```

fwrite()

- Para escrever uma estrutura do tipo **cliente** em um arquivo faríamos:

```
FILE * fp;
cliente c = { 123, "Fulano de Tal",
                   1000.0 };
if ((fp = fopen("conta.dat","ab")) ==
      NULL) {
    fprintf(stderr,"Erro!\n");
    return 1;
}
fwrite(&c, sizeof(cliente), 1, fp);
```

fseek()

- Modifica o indicador de posição de arquivo em arquivos que permitem acesso aleatório
- Protótipo de **fseek()**:

```
int fseek(FILE * fp, long
  num_bytes, int origem);
```

  - *num_bytes* é o número de bytes a partir de *origem* que se tornará a nova posição corrente
  - Devolve zero quando bem-sucedida

fseek()

- *Origem* é uma constante definida em stdio.h
  - **SEEK_SET** a partir do início do arquivo
  - **SEEK_CUR** a partir da posição atual
  - **SEEK_END** a partir do final do arquivo
- Exemplo:

```
fseek(fp,9*sizeof(cliente),SEEK_SET);
```

  - Move o indicador de posição de arquivo para a décima estrutura

ftell()

- Devolve o valor atual do indicador de posição do arquivo
- Protótipo de **`ftell()`**:

```
long ftell(FILE * fp);
```

  - Em caso de falha a função `ftell()` devolve -1

Referências

- SCHILDT, Herbert. *C Completo e Total.* 3. ed., Makron Books, 1996.
- DEITEL, H.M.; DEITEL, P.J.. *Como programar em C*. 2. ed., LTC, 1999.
- PRATA, Stephen. *C Primer Plus*. 6. ed., Addison Wesley, 2014.