

Manipulação de Arquivos Binários em C

Matheus Carvalho Raimundo

mcarvalhor.com

cstdio <stdio.h>

- “Standard Input Output”
- Abrir arquivo, escrever em arquivo, ler de arquivo, renomear arquivo, remover arquivo, ...
- Operações em buffer
- Mais baixo nível: unistd.h e fcntl.h

fopen(nome_arquivo, modo)

Abre um arquivo qualquer para ser manipulado pelo seu programa. Retorna um ponteiro para este arquivo ou NULL em caso de erros.

- **nome_arquivo:** é o caminho do sistema para o arquivo que se deseja abrir. Exemplo:
“C:\Users\matheus\arquivo.txt” ou “/home/matheus/arquivo.txt”
- **modo:** é o modo em que o arquivo será manipulado. É uma combinação dos seguintes caracteres:

“r” -> abre para leitura,	“r+” -> abre para leitura e escrita,
“w” -> abre para escrita,	“w+” -> abre para leitura e escrita,
“a” -> abre para escrita no fim do arquivo,	“a+” -> abre para leitura e escrita no fim do arquivo,
“b” -> binário	

fopen - tabela de modos para arquivo de texto

Modo	Pode ler?	Pode escrever?	Pode usar fseek?	E se o arquivo não existe?	E se o arquivo já existe?	Onde o ponteiro começa?	Lida com binário?
"r"	Sim	Não	Sim	Erro, retorna NULL	Ok	No começo	Não*
"r+"	Sim	Sim, se ã for no final, reescreve	Sim	Erro, retorna NULL	Ok	No começo	Não*
"w"	Não	Sim, se ã for no final, reescreve	Sim	Ele é criado	Ele é removido e criado do zero	No começo	Não*
"w+"	Sim	Sim, se ã for no final, reescreve	Sim	Ele é criado	Ele é removido e criado do zero	No começo	Não*
"a"	Não	Sim, mas só no final do arquivo	Não	Ele é criado	Ok	No fim	Não*
"a+"	Sim	Sim, mas só no final do arquivo	Sim, mas só escreve no fim	Ele é criado	Ok	No fim	Não*

* Por definição da biblioteca, não é pra lidar. Mas alguns sistemas operacionais permitem lidar com arquivo binário. Por garantia, se for manipular arquivo binário, use sempre o "b" (próximo slide).

fopen - tabela de modos para arquivo binário

Modo	Pode ler?	Pode escrever?	Pode usar fseek?	E se o arquivo não existe?	E se o arquivo já existe?	Onde o ponteiro começa?	Lida com binário?
“rb”	Sim	Não	Sim	Erro, retorna NULL	Ok	No começo	Sim
“r+b” ou “rb+”	Sim	Sim, se ñ for no final, reescreve	Sim	Erro, retorna NULL	Ok	No começo	Sim
“wb”	Não	Sim, se ñ for no final, reescreve	Sim	Ele é criado	Ele é removido e criado do zero	No começo	Sim
“w+b” ou “wb+”	Sim	Sim, se ñ for no final, reescreve	Sim	Ele é criado	Ele é removido e criado do zero	No começo	Sim
“ab”	Não	Sim, mas só no final do arquivo	Não	Ele é criado	Ok	No fim	Sim
“a+b” ou “ab+”	Sim	Sim, mas só no final do arquivo	Sim, mas só escreve no fim	Ele é criado	Ok	No fim	Sim

fseek(ponteiro, offset, origem)

Ajusta o ponteiro do arquivo para a posição “offset” a partir de “origem”.

Dica: se você tiver armazenando registros, você pode usar o RRN e essa função para pular para o *byteoffset* de um registro em específico.

- **ponteiro:** o ponteiro retornado por “fopen”.
- **offset:** o número de bytes que vai pular, ou melhor, o *byteoffset*.
- **origem:** SEEK_SET para início do arquivo, SEEK_CUR para posição atual do ponteiro, SEEK_END para fim do arquivo

- **Exemplos:**

- ir para o início do arquivo: `fseek(ponteiro, 0, SEEK_SET);`

- ir para o final do arquivo: `fseek(ponteiro, 0, SEEK_END);`

- ir para o *byteoffset* 1024 do arquivo: `fseek(ponteiro, 1024, SEEK_SET);`

- voltar 4 bytes da posição atual: `fseek(ponteiro, -4, SEEK_CUR)` ou `fseek(ponteiro, -sizeof(int), SEEK_CUR);`

- ir para o *offset* do último byte do arquivo: `fseek(ponteiro, -1, SEEK_END);`

ftell(ponteiro)

Retorna a posição do ponteiro do arquivo (*byteoffset*).

Dica: se você tiver armazenando registros, você pode usar essa posição em equações para calcular em qual RRN o ponteiro está.

- **ponteiro:** o ponteiro retornado pelo “fopen”.

fread(buffer, sizeof, número, ponteiro)

Faz a leitura de **sizeof * número** bytes do arquivo e salva o conteúdo em **buffer**. Retorna o próprio **número**, ou algo diferente se deu erro pra ler os dados.

- **buffer:** uma posição de memória para salvar os dados lidos. Tem que já estar alocado dinamicamente (*malloc*) ou estaticamente com o tamanho mínimo que caiba os bytes lidos.
- **sizeof:** o tamanho de cada elemento lido.
- **número:** o número de elementos a serem lidos.
- **ponteiro:** o ponteiro retornado por “fopen”.

- **Exemplos:**

- ler 1 byte do arquivo: `fread(byte_lido, 1, 1, ponteiro);`

- ler um inteiro do arquivo: `fread(valor, 4, 1, ponteiro)` ou `fread(valor, sizeof(int), 1, ponteiro);`

- ler 10 inteiros do arquivo: `fread(vetor, 4, 10, ponteiro)` ou `fread(vetor, sizeof(int), 10, ponteiro);`

- ler uma string de tamanho = 40 do arquivo: `fread(str, 1, 40, ponteiro)` ou `fread(str, sizeof(char), 40, ponteiro);`

- ler uma struct inteira de uma vez do arquivo: `fread(struct, sizeof(struct), 1, ponteiro);`

fwrite(buffer, sizeof, número, ponteiro)

Faz a escrita de **sizeof * número** bytes no arquivo que estão salvos em **buffer**. Se não tiver no fim do arquivo, vai sobrescrever os dados existentes (exceção se foi aberto com “a”). Se tiver no fim, aumenta o tamanho do arquivo. Retorna o próprio **número**, ou algo diferente se deu erro pra escrever os dados.

- **buffer:** uma posição de memória para salvar os dados lidos. Tem que já estar alocado dinamicamente (*malloc*) ou estaticamente com o tamanho mínimo que caiba os bytes lidos.
- **sizeof:** o tamanho de cada elemento lido.
- **número:** o número de elementos a serem lidos.
- **ponteiro:** o ponteiro retornado por “fopen”.

- **Exemplos:**

escrever 1 byte no arquivo: `fwrite(byte_pra_escrever, 1, 1, ponteiro);`

escrever um inteiro no arquivo: `fwrite(valor, 4, 1, ponteiro)` ou `fwrite(valor, sizeof(int), 1, ponteiro);`

escrever 10 inteiros no arquivo: `fwrite(vetor, 4, 10, ponteiro)` ou `fwrite(vetor, sizeof(int), 10, ponteiro);`

escrever uma string de tamanho 40 no arquivo: `fwrite(str, 1, 40, ponteiro)` ou `fwrite(str, sizeof(char), 40, ponteiro);`

escrever uma struct inteira de uma vez no arquivo: `fwrite(struct, sizeof(struct), 1, ponteiro);`

feof(ponteiro)

Retorna se está no fim do arquivo ou não. Ela apenas verifica um indicador *flag*, e não verifica efetivamente.

- **ponteiro**: o ponteiro retornado pelo “fopen”.

Atenção: muito cuidado com esta função! Recomendação: não use. Ela pode não funcionar como desejado. Na verdade, ela verifica se chegou ao fim do arquivo só se você tentou ler anteriormente e deu algum erro. Ela não verifica efetivamente se é o fim do arquivo.

É muito comum ver esse tipo de erro ao lidar com arquivos:

```
long counter = 0;
while(!feof(ponteiro)) {
    fread(buffer, 1, 1, ponteiro);
    counter++;
    printf("Eu li 1 byte do arquivo!\n");
}
printf("\nFIM. Eu li  %ld bytes do arquivo.\n", counter);
```



Na verdade, o código acima não funciona e pode provocar *bugs*!

Na verdade, a solução ideal e que funciona tinha que ser algo parecido com isso:

```
long counter = 0;
while(fread(buffer, 1, 1, ponteiro) != 0) {
    counter++;
    printf("Eu li 1 byte do arquivo!\n");
}
printf("\nFIM. Eu li  %ld bytes do arquivo.\n", counter);
```



O código acima funciona porque é verificado se chegou no fim do arquivo na própria operação de leitura dos dados. Esse é o ideal.

fclose(ponteiro)

Fecha um arquivo que estava em uso e o libera para que outros programas usem.

Sempre que finalizar a manipulação de um arquivo, não se esqueça de usar essa função para fechar ele. Isso diz pro sistema operacional: “Terminei de mexer nesse arquivo. Se outro programa quiser usar ele, agora pode.”.

Vale lembrar que você não pode abrir o mesmo arquivo sem antes ter fechado ele. Mas se o seu programa se encerrar, o sistema operacional automaticamente fecha o arquivo.

- **ponteiro:** o ponteiro retornado pelo “fopen”.

Outras funções

- remove
- rename
- tmpfile
- fflush
- fgetc
- rewind
- ferror
- ...

Manual do Linux

- Contém documentação detalhada de todas as funções da **stdio.h** (e de outras bibliotecas do C e programas do Linux também).
- Comando: **man CMD**
onde **CMD** é a função ou comando que você tá buscando pelo manual.
- Pressionar **q** fecha o manual.
- Exemplos:
 - > man fopen
 - > man malloc
 - > man ls
 - > man hexdump
 - > man fwrite
 - > man stdio
 - > man meld
 - > man vbindiff

Struct na RAM

No C, podemos definir *structs*, que são um conjunto de dados agrupados na memória RAM. Você não deve se preocupar em como o compilador faz essa alocação na memória, e nem em como os dados ficam organizados. Você só precisa saber que estão juntos na memória RAM, e acessar eles usando os operadores “estrutura.dado” ou “estrutura->dado”.

```
typedef struct {
    long a;
    char b;
    char c;
} estrutura_a;

typedef struct {
    char b;
    long a;
    char c;
} estrutura_b;
```

Qual o tamanho (*sizeof*) da **estrutura_a** e **estrutura_b**? *Spoiler*: pode não ser 10 bytes. E ainda mais: apesar de guardarem as mesmas informações e em mesma quantidade, não necessariamente são de mesmo tamanho.

Struct no disco

Como a gente viu, não dá pra confiar no compilador para organizar os dados de uma *struct*.

Mas então como fazemos para salvar uma *struct* no disco? A gente viu que o compilador pode não deixar do jeito que a gente quer, então como fazemos para os dados ficarem organizados adequadamente no disco?

Solução: salvar campo-a-campo, manualmente.

Ao invés de salvar uma *struct* inteira com *fwrite*, use *fwrite* para cada um dos campos presentes dentro dela. Lembre-se de que, para ler com *fread* essa *struct* salva em disco posteriormente, você também tem que ler campo-a-campo.

```
fwrite(&estrutura_a, sizeof(estrutura_a), 1, ponteiro);
```

```
fread(&estrutura_a, sizeof(estrutura_a), 1, ponteiro);
```



```
fwrite(&estrutura_a.a, sizeof(long), 1, ponteiro);
```

```
fwrite(&estrutura_a.b, sizeof(char), 1, ponteiro);
```

```
fwrite(&estrutura_a.c, sizeof(char), 1, ponteiro);
```

```
fread(&estrutura_a.a, sizeof(long), 1, ponteiro);
```

```
fread(&estrutura_a.b, sizeof(char), 1, ponteiro);
```

```
fread(&estrutura_a.c, sizeof(char), 1, ponteiro);
```



Struct com ponteiros

Cuidado com ponteiros dentro de uma *struct*. Se for possível, **evite usar**. Além de dar mais trabalho (você precisa alocar e desalocar manualmente), você pode se perder na aritmética dos ponteiros e armazenar dados indevidamente.

```
typedef struct {  
    int RRN;  
    char nome[50];  
    int idade;  
} estrutura_a;
```

```
typedef struct {  
    int RRN;  
    char *nome;  
    int idade;  
} estrutura_b;
```

```
fwrite(&estrutura_b.nome, sizeof(char), 50, ponteiro);  
  
fread(&estrutura_b.nome, sizeof(char), 50, ponteiro);
```



```
fwrite(&estrutura_a.nome, sizeof(char), 50, ponteiro); ou  
fwrite(estrutura_a.nome, sizeof(char), 50, ponteiro); ou  
fwrite(estrutura_b.nome, sizeof(char), 50, ponteiro);
```

```
fread(&estrutura_a.nome, sizeof(char), 50, ponteiro); ou  
fread(estrutura_a.nome, sizeof(char), 50, ponteiro); ou  
fread(estrutura_b.nome, sizeof(char), 50, ponteiro);
```



hexdump -Cv binario.bin

Arquivos binários possuem dados que não são visíveis ao editor de texto (caracteres não-ASCII). Sendo assim, pode ser complicado verificar se seu programa tá fazendo o que deveria ou não no arquivo binário.

O Linux tem um programa chamado **hexdump**. Esse programa ajuda a visualizar arquivos binários. Ele basicamente mostra: a posição (*byteoffset*), os valores dos bytes em hexadecimal, e se for um caractere visível válido, mostra seu ASCII.

Basta executar o comando `hexdump -Cv arquivo.bin` onde **arquivo.bin** é o arquivo binário que você deseja analisar.

Use as setas do teclado para mover a tela, e pressione **q** para sair a qualquer momento.

byteoffset	hexadecimal	ASCII
00000160	00 00 00 00 00 00 00 00 08 20 00 00 48 00 00 00H...
00000170	00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00text...
00000180	3C 2C 00 00 00 20 00 00 00 2E 00 00 00 02 00 00	<,.....
00000190	00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60`
000001A0	2E 72 73 72 63 00 00 00 CC 05 00 00 00 60 00 00	.rsrc...i.....
000001B0	00 06 00 00 00 30 00 00 00 00 00 00 00 00 00 000.....
000001C0	00 00 00 00 40 00 00 40 2E 72 65 6C 6F 63 00 00	...@..@.reloc..
000001D0	0C 00 00 00 00 80 00 00 00 02 00 00 00 36 00 00€.....6..
000001E0	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 42@..B
000001F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000200	18 4C 00 00 00 00 00 00 48 00 00 00 02 00 05 00	.L.....H.....
00000210	E0 2D 00 00 CC 1C 00 00 03 00 02 00 01 00 00 06	à-..i.....

Valgrind (memcheck)

O Linux tem um programa chamado **valgrind** ou **memcheck** (o nome varia com a distribuição Linux). Se você está enfrentando erros de memória, é recomendável utilizar esse programa pra saber o que está acontecendo e ajudar a debugar o erro.

Compile seu programa com a flag “-g” do GCC, e depois rode seu programa no valgrind.

Se houverem erros, ele vai mostrar informações e a linha em que o erro foi detectado.

Modularização

A dica mais importante de todas, com certeza é a de modularizar seu código. Crie arquivos *.h bem estruturados e lembre-se de deixar tudo bem documentado.

Por exemplo:

Se você for armazenar registros em um arquivo binário, logo de primeira sabemos duas funções óbvias que precisam ser modularizadas: ler_registro e escrever_registro. Crie um *.h que lida apenas com leitura e escrita de registros, outro que lida com índice, outro com árvore-B, outro com as funcionalidades, ..., e assim por diante.