Programação Estruturada

Aula 22 - Arquivos

Yuri Malheiros (yuri@ci.ufpb.br)

Streams

- Em C, o termo stream significa qualquer fonte de entrada ou qualquer destino de saída
- Nós podemos obter entradas do teclado e escrever a saída na tela
- Podemos ter outros streams, normalmente representado por arquivos armazenados em HD, Pendrives, etc.

Ponteiros para arquivos

- Para acessar streams em C usamos ponteiros para arquivos
- Seu tipo é `FILE *`
    - `FILE` é declarado em `stdio.h`

Streams padrão

- `stdio.h` fornece três streams padrão
- Eles já estão prontos para uso, não precisamos declará-los
- Os streams padrão são:
  - `stdin` - entrada padrão
  - `stdout` - saída padrão
  - `stderr` - erro padrão

Streams padrão

- As funções `printf`, `scanf`, `puts`, `gets`, etc. recebem entradas do `stdin` e enviam a saída para o `stdout`
- Por padrão, o `stdin` representa o teclado e o `stdout` e `stderr` a tela
- Entretanto, podemos mudar esse comportamento padrão usando um mecanismo chamado de redirecionamento

Streams padrão

- Um programa (meu_exemplo) pode obter sua entrada de um arquivo (entrada.dat) ao invés do teclado
- Para isso, escrevemos na linha de comando:
    - `./meu_exemplo < entrada.dat`
- Essa técnica é chamada de redirecionamento de entrada
- O `stdin` passa a ser representado pelo entrada.dat

Streams padrão

- O redirecionamento de saída é similar
- Podemos redirecionar o `stdout` para um arquivo da seguinte forma:
    - `./meu_exemplo > saida.dat`
- Todos os dados escritos no `stdout` vão para saida.dat ao invés de aparecer na tela

Streams padrão

- Nós podemos combinar os redirecionamentos
  - `./meu_exemplo < entrada.dat > saida.data` ou
  - `./meu_exemplo > saida.data < entrada.dat`

Arquivo texto e arquivo binário

- `stdio.h` suporta dois tipos de arquivos: texto e binário
- Os bytes num arquivo texto representam caracteres
- Assim é possível um humano ler o arquivo
- O código fonte de um programa C é um arquivo texto

Arquivo texto e arquivo binário

- Os bytes em um arquivo binário não necessariamente representam caracteres, eles podem representar inteiros, floats, etc.
- Um arquivo executável é um arquivo binário

Arquivo texto e arquivo binário

- Arquivos texto têm duas características que os arquivos binários não possuem
- Arquivos texto são divididos em linhas
    - Cada linha linha é terminada por um ou dois caracteres especiais (essa diferença depende do SO)
- Arquivos texto podem ter um marcador de fim de arquivo

Operações em arquivos

- O redirecionamento é uma maneira simples de ler e escrever em arquivos, mas ela não resolve todos os casos
- Não sabemos os nomes dos arquivos
- Não temos como ler ou escrever em múltiplos arquivos
- Quando o redirecionamento não for suficiente, nós usamos as operações em arquivos fornecidas pela `stdlib.h`

Abrindo um arquivo

- Para abrir um arquivo, usamos a função `fopen`
- `FILE *fopen(const char *filename, const char *mode)`
- `filename` é o nome do arquivo a ser aberto
- `mode` especifica quais operações nós vamos realizar no arquivo
  - `"r"`, por exemplo, indica que vamos fazer leitura

Abrindo um arquivo

- O nome do arquivo pode conter o caminho para esse arquivo
  - `/home/yuri/documentos/arquivo.txt`
- Para o Windows, precisamos ter cuidado, pois o caminho para um arquivo tem o formato:
  - `c:\documentos\teste.txt` e `\` é o início de caracteres especiais num string
- Existem duas maneiras de resolver esse problema
- Usar `\\` ao invés de `\`
  - `c:\\documentos\\teste.txt`
- Usar `/` ao invés de `\`
  - `c:/documentos/teste.txt`

Abrindo um arquivo

- `fopen` retorna um ponteiro para um arquivo
- Quando o programa não consegue abrir o arquivo, `fopen` retorna um ponteiro nulo
    - Isso acontece quando um arquivo não existe ou não temos permissão para acessá-lo

Modos

- Os modos que podemos passar no `fopen` são:
- `"r"` - abre o arquivo para leitura, o arquivo precisa existir
- `"w"` - cria um arquivo em branco para escrita, se o arquivo existir, o seu conteúdo é apagado
- `"a"` - abre o arquivo para escrita a partir do seu final, se o arquivo não existir, um novo arquivo é criado
- `"r+"` - abre o arquivo para leitura e escrita
- `"w+"` - cria um arquivo em branco para leitura e escrita
- `"a+"` - abrir o arquivo para leitura e escrita a partir do seu final

Modos

- Para abrir arquivos binários, precisamos incluir a letra `b` nos modos
- Dessa forma, temos os modos: `"rb"` `"wb"` `"ab"` `"rb+"` `"wb+"` `"ab+"`

Fechando um arquivo

- A função `fclose` permite que o programa feche um arquivo que não está sendo mais usado
- `int fclose(FILE *stream)`
- `stream` é o arquivo que vai ser fechado e deve ser um ponteiro criado pela função `fopen`
- Se o arquivo for fechado com sucesso, a função retorna `0`, se não, ela retorna a macro `EOF`
    - `EOF` é definida no `stdio.h`

Exemplo

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

#define FILE_NAME "exemplo.dat"

int main(void) {
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Não foi possível abrir o arquivo %s\n", FILE_NAME);
    exit(1);
  }

  fclose(fp);

  return 0;
}
```

Funções printf e fprintf

- As funções `printf` e `fprintf` escrevem dados num stream de saída
- A diferença entre `printf` e `fprintf` é que `printf` escreve sempre no `sdtout` e o `fprintf` escreve num stream indicado como seu primeiro argumento
- `printf("Total: %d\n", 10);` - escreve no `stdout`
- `fprintf(fp, "Total: %d\n", 10);` - escreve em `fp`
- Chamar `printf` é o mesmo que chamar `fprintf` para o stream `stdout`

Exemplo

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

#define FILE_NAME "exemplo.dat"

int main(void) {
  FILE *fp;

  fp = fopen(FILE_NAME, "w");
  if (fp == NULL) {
    printf("Não foi possível abrir o arquivo %s\n", FILE_NAME);
    exit(1);
  }

  fprintf(fp, "Total: %d\n", 10);

  fclose(fp);

  return 0;
}
```

Funções scanf e fscanf

- As funções `scanf` e `fscanf` leem dados de um stream de entrada
- A diferença entre `scanf` e `fscanf` é que `scanf` escreve sempre no `stdin` e o `fscanf` escreve num stream indicado como seu primeiro argumento
- `scanf("%d%d, &i, &j);` - lê do `stdin`
- `fscanf(fp, "%d%d, &i, &j);` - lê de `fp`
- Chamar `scanf` é o mesmo que chamar `fscanf` para o stream `stdin`

Exemplo

```c
int main(void) {
  FILE *fp;
  int n;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Não foi possível abrir o arquivo %s\n", FILE_NAME);
    exit(1);
  }

  while (fscanf(fp, "%d", &n) == 1)
      printf("%d\n", n);

  fclose(fp);

  return 0;
}
```

Leitura e escrita de caracteres

- Funções de escrita:
- `int putc(int c, FILE *stream);`
- `int putchar(int c);`
- Note que o caractere é tradado como `int` não como `char`
    - Um dos motivos é que a macro que indica o fim do arquivo (`EOF`) é um número negativo
- As funções retornam a quantidade de caracteres escritos ou `EOF` se acontecer um erro

Leitura e escrita de caracteres

- Funções de leitura:
- `int getc(FILE *stream);`
- `int getchar(void);`
- Para ler os caracteres de um arquivo, um por um:

```
while ((ch = getc(fp)) != EOF) { ... }
```

Exemplo

- Vamos criar um programa que copia um arquivo para outro
- Devemos poder executar: `./copiar arquivo1.txt arquivo2.txt`
- Isso copiará o conteúdo de `arquivo1.txt` para `arquivo2.txt`

Leitura e escrita de linhas

- Funções de escrita:
- `int fputs(const char *s, FILE *stream);`
- `int puts(const char *s);`
- Cuidado que `puts` adiciona uma quebra de linha no final do string e `fputs` não

Leitura e escrita de linhas

- Funções de leitura:
- `char *fgets(char *s, int n, FILE *stream)`
- `char *gets(char *s);`
- `fgets` lê o string até o fim de uma linha ou até alcançar o número `n` de caracteres
    - Isso é uma vantagem em termos de segurança

Leitura e escrita de blocos

- As funções `fread` e `fwrite` permitem um programa ler e escrever blocos de dados maiores
- Elas costumam ser usados com arquivos binários

Leitura e escrita de blocos

- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);`
- `fwrite` copia um array na memória para um stream
- `ptr` é um endereço de um array
- `size` é o tamanho de cada elemento do array
- `nmemb` é o número de elementos que serão escritos
- `stream` é um ponteiro para um arquivo

```
fwrite(a, sizeof(a[0]), sizeof(a) / sizeof(a[0]), fp);
```

Leitura e escrita de blocos

- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`
- `fread` lê os elementos de um array de um stream
- `ptr` é um endereço de um array
- `size` é o tamanho de cada elemento do array
- `nmemb` é o número de elementos que serão escritos
- `stream` é um ponteiro para um arquivo

```
fread(a, sizeof(a[0]), sizeof(a) / sizeof(a[0]), fp);
```