INSTITUTO DE INFORMÁTICA
Universidade Federal de Goiás

Arquivos

Introdução à Programação

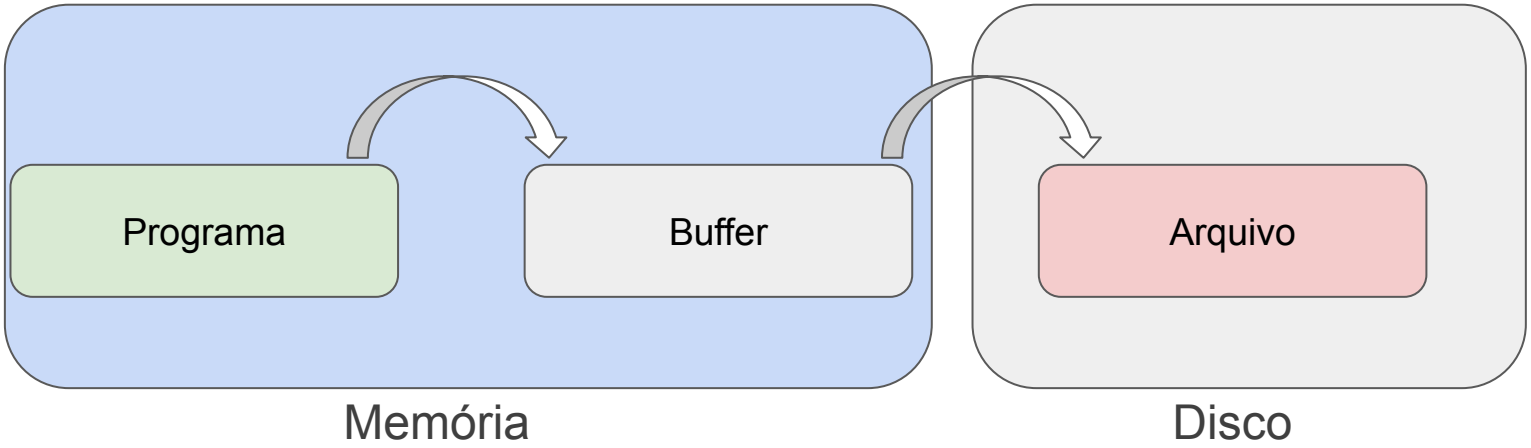
Bruno Brandão

INSTITUTO DE INFORMÁTICA UFG

UFG UNIVERSIDADE FEDERAL DE GOIÁS

Fluxo de Dados

- Para comunicar com dispositivos, programas, e arquivos, utilizamos fluxos de dados.
- É preciso enviar e receber dados (E/S Entrada e Saída)
- O fluxo de dados padroniza essas operações com um buffer, que coordena a comunicação



Fluxo de Dados

- Buffer é preenchido e esvaziado a critério da lógica implementada (biblioteca padrão C)
- Objetivo de minimizar operações em disco de leitura e escrita
- Ler e escrever no HDD é custoso
- Programas diferentes podem estar operando de forma assíncrona

Struct FILE

- O programa interage com o fluxo através de funções de alto nível para abrir, ler, e fechar arquivos
- Estas funções atuam na estrutura FILE que contém:
  - Ponteiro para o buffer
  - Tipo de operação permitida
  - Indicador da posição atual de leitura
  - Códigos de erros
  - etc.
- O fluxo é criado definindo um ponteiro para esta estrutura

```
FILE *arquivo
```

Elementos do Fluxo

- Tipo de Operação
  - **Apenas leitura**
  - **Apenas escrita**
  - **Leitura e escrita**

- Tipo de Acesso
  - **Sequencial**
  - **Aleatório**

- Tipo de Dado
  - **Texto**
  - **Binário**

Função *fopen()*

- “Abrir um arquivo”
- Estabelece um fluxo para operações de leitura, escrita, ou ambas com um arquivo

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

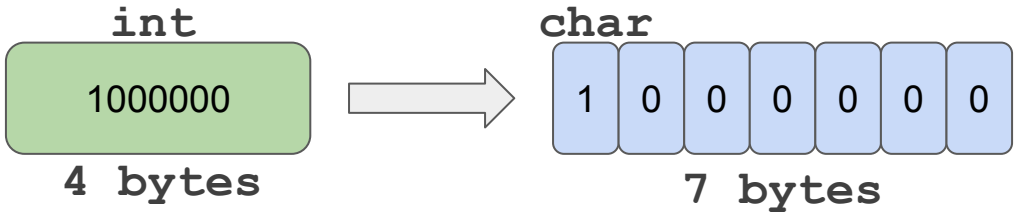
- Recebe uma string com o caminho do arquivo (relativo ou absoluto)
- Recebe o modo, que define o tipo de operação
- Retorna um ponteiro para um FILE se obteve sucesso
- Se não, retorna ponteiro NULL
    - Arquivo não existe
    - Sem permissão de acesso

Modos de Operação

Modo	Descrição
`"r"`	Abre arquivo para leitura. O arquivo deve existir
`"w"`	Abre o arquivo para escrita. Se existir, é sobrescrito, se não, criado.
`"a"`	Abre o arquivo para escrita. Se existir, adiciona conteúdo ao final, se não, criado.
`"r+"`	Abre arquivo para leitura e escrita. O arquivo deve existir
`"w+"`	Abre o arquivo para leitura e escrita. Se existir, é sobrescrito, se não, criado.
`"a+"`	Abre o arquivo para leitura e escrita. Se existir, adiciona conteúdo ao final, se não, criado.
…+`"b"`	Opcional para abrir arquivos binários (“rb”, “wb”, etc.)
…+`"t"`	Opcional para indicar arquivo de texto, na ausência de “b” e “t” o arquivo é tratado como de texto

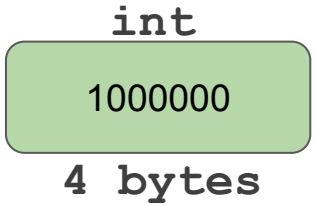
Arquivos de Texto

- Guardam informação em formato de caracteres
- Podem ser abertos em um editor de texto e o conteúdo tem algum sentido
- Fáceis de ler e editar manualmente
- Portabilidade entre sistemas
- Menos eficientes pois cada caractere precisa ser interpretado
- Trabalhar com estruturas complexas é muito trabalhoso



Arquivos Binários

- Guardam informação em sequências de bytes assim como estão na memória RAM.
- Compacto e eficiente na leitura e escrita
- Ideal para estruturas complexas (structs)
- Não é fácil de editar manualmente
- Transferência entre sistemas operacionais requer mais cuidado



Casos de uso

Texto

- Configurações
- Logs de erros ou acessos
- Dados tabulares em CSV (*Comma Separated Values*)
- Anotações / Documentos
- Código fonte

Binário

- Imagens
- Bases de dados
- Guardar estado de um sistema para retomar posteriormente
  - Jogos
  - Backup contra falhas

Leitura de Texto

Exemplo *fopen()*

```c
FILE *file = fopen("example.txt", "r");

if (file == NULL) {
    printf("Error: Could not open file\n");
    return 1;
}
```

Função *fgetc()*

- Lê caractere por caractere de um arquivo de texto

```
int fgetc(FILE *stream);
```

- Recebe o ponteiro para a estrutura FILE
- O arquivo PRECISA ser aberto em modo texto
- Retorna inteiro para acomodar o valor de EOF (-1) (*End-Of-File* não existe na tabela ASCII)
- Retorna EOF se houver um erro ou o arquivo terminou

Exemplo *fgetc()*

```c
FILE *file = fopen("example.txt", "r");
if (file == NULL) {
    perror("Error opening file");
    return 1;
}

int ch;
while ((ch = fgetc(file)) != EOF) {
    putchar(ch); // Print each character to the console
}
```

Função *fscanf()*

- Lê o arquivo através de uma formatação

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

- Recebe o ponteiro para a estrutura FILE
- Recebe a formatação, e as variáveis onde colocar os dados
- Retorna o número de itens lidos
- Se houver erro na entrada ou encontrar o fim do arquivo, retorna EOF
- O arquivo PRECISA ser aberto em modo texto

Exemplo *fscanf()*

```c
FILE *file = fopen("data.txt", "r");
if (file == NULL) {
    perror("Error opening file");
    return 1;
}

int number;
char name[50];
float value;

if (fscanf(file, "%d %s %f", &number, name, &value) == 3) {
    printf("Number: %d, Name: %s, Value: %.2f\n", number, name, value);
} else {
    printf("Error reading file\n");
}

fclose(file);
```

Função *fgets()*

- Lê o arquivo linha por linha

```
char *fgets(char *str, int n, FILE *stream);
```

- Recebe um vetor de caracteres onde guardar a linha
- Recebe o número n de caracteres para ler (incluindo \0)
- Recebe o ponteiro para a estrutura FILE
- Retorna um ponteiro com a string
- Caso chegue no fim do arquivo, ou um erro ocorra, retorna um ponteiro nulo

Função *fgets()*

- **Lê até:**
- Encontrar \n
- Ler (n - 1) caracteres
- Chegar no fim do arquivo (EOF)

- Ao encontrar o \n ele é incluído na string ao contrário de scanf

Exemplo *fgets()*

```c
int main() {
    FILE *file = fopen("example.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    char buffer[100]; // Buffer to store each line
    while (fgets(buffer, sizeof(buffer), file) != NULL) {
        printf("%s", buffer); // Print each line (newline already included)
    }

    fclose(file);
    return 0;
}
```

Leitura de Binário

Função *fread()*

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- **`ptr`** é o ponteiro para onde colocar os dados lidos
- **`size`** é o tamanho em bytes de cada elemento a ser lido
- **`nmemb`** é o número de elementos para ler
- **`stream`** é o ponteiro para o fluxo (FILE)
- Retorna o número de elementos lidos com sucesso. Pode ser menor que **`nmemb`** se houver um erro, ou chegar ao fim do arquivo.

Diferenciando erros e EOFs

- Arquivos binários não possuem EOF, então é preciso utilizar funções para detectar erro ou final de arquivo.

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

- Retorna um valor diferente de zero se chegou ao final
- Caso contrário, zero

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

- Retorna um valor diferente de zero se houve erro
- Caso contrário, zero

feof() e ferror()

- Podem ser usados para arquivos de texto e binários
- As saídas de leitura para arquivos de texto são ambíguas, portanto é importante utilizar `feof` e `ferror` para diferenciá-las e confirmar se é um erro ou fim.
- `feof` NÃO PREVINE o EOF, ele apenas indica se o EOF foi encontrado APÓS uma tentativa de leitura
- Sempre cheque a saída das funções de leitura

Exemplo *fread()*

```c
size_t total_read = 0;
size_t elements_read;
FILE *file = fopen("data.bin", "rb");
if (file == NULL) {
    perror("Error opening file");
    return 1;
}

int buffer[100]; // Buffer to hold 100 integers
while ((elements_read = fread(buffer + total_read, sizeof(int), 10, file)) > 0) {
    total_read += elements_read;
}

if (feof(file)) {
    printf("End of file reached.\n");
} else if (ferror(file)) {
    perror("Error reading file");
}
```

Exemplo *fread()*

Lendo o tamanho do vetor

```c
FILE *file = fopen("data.bin", "rb");
if (file == NULL) {
    perror("Error opening file");
    return 1;
}

char buffer[100];
size_t bytes_read = fread(buffer, 1, sizeof(buffer), file);

if (bytes_read < sizeof(buffer)) {
    if (feof(file)) {
        printf("Reached end of file after reading %zu bytes.\n", bytes_read);
    } else if (ferror(file)) {
        perror("Error reading file");
    }
}

fclose(file);
```

Exemplo *fread()*

Lendo uma estrutura

```c
typedef struct {
    int id;
    char name[20];
    float score;
} Record;

Record records[5];
FILE *file = fopen("records.bin", "rb");
if (file == NULL) {
    perror("Error opening file");
    return 1;
}

size_t count = fread(records, sizeof(Record), 5, file);
printf("Read %zu records:\n", count);
```

Escrita de Texto

Função *fputc()*

- Escreve um caractere por vez no arquivo

```
int fputc(int c, FILE *stream);
```

- Recebe o caractere a ser escrito, como um inteiro
- Recebe o ponteiro para o fluxo (FILE)
- Retorna o caractere escrito
- Em caso de falha, retorna EOF

Exemplo *fputc()*

```c
FILE *file = fopen("output.txt", "w");
if (file == NULL) {
    perror("Error opening file");
    return 1;
}

char message[] = "Hello, world!";
for (int i = 0; message[i] != '\0'; i++) {
    if (fputc(message[i], file) == EOF) {
        perror("Error writing to file");
        fclose(file);
        return 1;
    }
}
```

Função *fprintf()*

- Escrita estruturada no arquivo de texto

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

- Recebe o ponteiro para o fluxo (FILE)
- Recebe a formatação, como no printf
- Retorna o número de caracteres escritos
- Retorna EOF em caso de falha

Exemplo *fprintf()*

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("data.txt", "w");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    int id = 42;
    float score = 98.5;
    char name[] = "Alice";

    fprintf(file, "ID: %d\n", id);
    fprintf(file, "Name: %s\n", name);
    fprintf(file, "Score: %.2f\n", score);

    fclose(file);
    return 0;
}
```

Fluxos Padrão

- Em C, fluxos padrão (*standard streams*) são canais de comunicação pré-configurados entre o programa e o sistema operacional.
- Eles são usados para entrada, saída e mensagens de erro.
- Padronizam a manipulação de entrada e saída.
- Facilitam a interação entre programas e usuários.

Fluxos Padrão

- **stdin (Standard Input)**
  - Responsável pela entrada de dados
  - Funções associadas: `scanf`, `fgets`, `getchar`.
- **stdout (Standard Output)**
  - Responsável pela saída de dados
  - Destinada ao terminal por padrão
  - Funções associadas: `printf`, `putchar`.
- **stderr (Standard Error)**
  - Responsável por mensagens de erro
  - Destinada também ao console, mas um fluxo separado
  - Funções associadas: `fprintf(stderr,...)`

Escrita em Binário

Função *fwrite()*

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- **`ptr`** é o ponteiro para os dados a serem escritos
- **`size`** é o tamanho em bytes de cada elemento a ser escrito
- **`nmemb`** é o número de elementos a serem escritos
- **`stream`** é o ponteiro para o fluxo (FILE)
- Retorna o número de elementos escritos com sucesso
- Caso retorne um valor menor que **`nmemb`** indica um erro

Exemplo *fwrite()*

```c
#include <stdio.h>

typedef struct {
    int id;
    char name[20];
    float score;
} Record;

int main() {
    Record records[] = {
        {1, "Alice", 95.5},
        {2, "Bob", 89.0},
        {3, "Charlie", 92.3}
    };

    FILE *file = fopen("records.bin", "wb");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    size_t count = fwrite(records, sizeof(Record), 3, file);

    printf("Wrote %zu records to the file.\n", count);

    fclose(file);
```

Funções Adicionais

Função *fseek()*

- Movendo o ponteiro da posição do arquivo para escrever em posições diferentes.

```
int fseek(FILE *stream, long offset, int whence);
```

- **`stream`** é o ponteiro para o fluxo (FILE)
- **`offset`** número de bytes para mover o ponteiro
- **`whence`** ponto de referência do movimento, indicado por constantes
  - SEEK_SET: Mover para uma posição a partir do início do arquivo
  - SEEK_CUR: Mover relativo à posição atual
  - SEEK_END: Mover relativo ao final do arquivo
- Retorna zero em caso de sucesso

Função *ftell()*

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

- Retorna a posição atual do ponteiro do arquivo em bytes
- Em caso de erro, retorna -1L (-1 long)

Função *fclose()*

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

- Fecha o fluxo
- Escreve no arquivo tudo que ainda estiver no buffer
- Libera recursos de memória
- Retorna zero em caso de sucesso
- Retorna EOF em caso de falha