Nota: Este material complementar, disponível em https://prettore.github.io/lectures.html representa uma cópia resumida de conteúdos bibliográficos disponíveis gratuitamente na Internet.

Arquivos Textos e Binários

Introdução

O processamento de arquivos é uma parte fundamental da programação, permitindo que os programas armazenem e recuperem dados de forma persistente. Diferentemente das variáveis em memória, cujos valores são perdidos quando o programa termina, os arquivos permitem que os dados sejam preservados entre diferentes execuções do programa. Existem dois tipos principais de arquivos utilizados em programação: arquivos de texto e arquivos binários, cada um com suas características, vantagens e aplicações específicas.
Neste material, exploraremos os conceitos fundamentais de manipulação de arquivos, com foco nas operações básicas como abertura, leitura, escrita e fechamento. Também discutiremos as diferenças entre arquivos de texto e binários, os modos de acesso disponíveis e as funções específicas para manipulação de cada tipo de arquivo. Embora os exemplos sejam principalmente em linguagem C, os conceitos são aplicáveis a diversas linguagens de programação.

Conceito de Arquivos e Manipulação em Programas

Um arquivo é uma coleção de dados armazenados em um dispositivo de armazenamento secundário, como um disco rígido, SSD ou pendrive. Do ponto de vista do sistema operacional, um arquivo é identificado por um nome e um caminho que indica sua localização no sistema de arquivos. Para os programas, um arquivo é acessado através de um ponteiro ou identificador que serve como intermediário entre o programa e o sistema operacional.

A manipulação de arquivos em programas geralmente segue um padrão comum:

1. Abertura do arquivo: Estabelece uma conexão entre o programa e o arquivo, especificando o modo de acesso (leitura, escrita, etc.).
2. Processamento: Realização de operações de leitura e/ou escrita no arquivo.
3. Fechamento do arquivo: Encerra a conexão, liberando recursos do sistema e garantindo que todos os dados sejam salvos corretamente.

Em linguagem C, a manipulação de arquivos é realizada através da biblioteca padrão <stdio.h>, que fornece funções como fopen(), fclose(), fread(), fwrite(), fprintf(), fscanf(), entre outras.

Abertura, Leitura, Escrita e Fechamento de Arquivos

Abertura de Arquivos

A abertura de um arquivo é realizada através da função fopen(), que recebe como parâmetros o nome do arquivo e o modo de abertura. Esta função retorna um ponteiro para o tipo FILE, que será utilizado em todas as operações subsequentes com o arquivo.

Sintaxe em C:

FILE *fopen(const char *filename, const char *mode);	Exemplo:
	
	FILE *arquivo;
	arquivo = fopen("dados.txt", "r"); // Abre o arquivo "dados.txt" para leitura
	if (arquivo == NULL) {
	  printf("Erro ao abrir o arquivo.
	");
	  return 1;
	}

É importante sempre verificar se a abertura do arquivo foi bem-sucedida, pois diversos fatores podem impedir a abertura, como permissões insuficientes, arquivo inexistente (no caso de leitura), ou disco cheio (no caso de escrita).

Leitura de Arquivos

A leitura de dados de um arquivo pode ser realizada de várias formas, dependendo do tipo de dado e da estrutura do arquivo:

1. Leitura formatada com fscanf():

int fscanf(FILE *stream, const char *format, ...);	Exemplo: int idade; char nome[50]; fscanf(arquivo, "%s %d", nome, &idade);

2. Leitura de caracteres com fgetc():

int fgetc(FILE *stream);	Exemplo:
	
	int c;
	while ((c = fgetc(arquivo)) != EOF) {
	  putchar(c); // Imprime o caractere lido
	}

3. Leitura de linhas com fgets():

char *fgets(char *str, int n, FILE *stream);	Exemplo:
	
	char linha[100];
	while (fgets(linha, sizeof(linha), arquivo) != NULL) {
	  printf("%s", linha); // Imprime a linha lida
	}

4. Leitura de blocos de **dados** com fread():

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);	Exemplo:
	
	struct pessoa {
	  char nome[50];
	  int idade;
	};
	struct pessoa p;
	fread(&p, sizeof(struct pessoa), 1, arquivo);

Escrita em Arquivos

A escrita em arquivos também pode ser realizada de várias formas:

1. Escrita formatada com fprintf():

int fprintf(FILE *stream, const char *format, ...);	Exemplo: fprintf(arquivo, "Nome: %s, Idade: %d ", "João", 30);

2. Escrita de caracteres com fputc():

int fputc(int c, FILE *stream);	Exemplo: fputc('A', arquivo);

3. Escrita de strings com fputs():

int fputs(const char *str, FILE *stream);	Exemplo: fputs("Olá, mundo!", arquivo);

4. Escrita de blocos de **dados** com fwrite():

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);	Exemplo: struct pessoa p = {"Maria", 25}; fwrite(&p, sizeof(struct pessoa), 1, arquivo);

Fechamento de Arquivos

Após concluir as operações com um arquivo, é essencial fechá-lo usando a função fclose():

int fclose(FILE *stream);	Exemplo: fclose(arquivo);

O fechamento de arquivos é importante por várias razões:
1. Libera recursos do sistema operacional.
2. Garante que todos os dados em buffer sejam efetivamente escritos no arquivo.
3. Permite que outros programas acessem o arquivo (em sistemas que implementam bloqueio de arquivos).
4. Previne corrupção de dados.

Modos de Abertura de Arquivos

Os modos de abertura determinam as operações permitidas em um arquivo e como o arquivo será tratado durante a abertura. Os principais modos são:
1. "r" - Abre um arquivo para leitura. O arquivo deve existir.
2. "w" - Cria um arquivo para escrita. Se o arquivo já existir, seu conteúdo será truncado (apagado).
3. "a" - Abre um arquivo para escrita no final (append). Se o arquivo não existir, ele será criado.
4. "r+" - Abre um arquivo para leitura e escrita. O arquivo deve existir.
5. "w+" - Cria um arquivo para leitura e escrita. Se o arquivo já existir, seu conteúdo será truncado.
6. "a+" - Abre um arquivo para leitura e escrita no final. Se o arquivo não existir, ele será criado.
Para arquivos binários, adiciona-se o caractere "b" ao modo (ex: "rb", "wb", "ab", etc.).
Exemplo em C:

```
FILE *arquivo_texto = fopen("dados.txt", "r");       // Abre para leitura (texto)
FILE *arquivo_binario = fopen("dados.bin", "wb");    // Abre para escrita (binário)
FILE *arquivo_append = fopen("log.txt", "a");        // Abre para append (texto)
FILE *arquivo_rw = fopen("config.txt", "r+");        // Abre para leitura e escrita (texto)
```

Diferenças entre Arquivos Texto e Binário

Arquivos de Texto

Os arquivos de texto armazenam dados em formato legível por humanos, utilizando caracteres ASCII ou Unicode. Cada byte ou sequência de bytes representa um caractere, e os dados são organizados em linhas separadas por caracteres especiais

Características dos arquivos de texto:

1. Legibilidade: Podem ser abertos e editados em editores de texto simples.
2. Portabilidade: Podem ser transferidos entre diferentes sistemas, embora possam ocorrer problemas com caracteres de fim de linha.
3. Representação de números: Números são armazenados como sequências de caracteres (ex: o número 123 é armazenado como os caracteres '1', '2' e '3').
4. Tamanho: Geralmente ocupam mais espaço que arquivos binários equivalentes, pois cada dígito de um número é armazenado como um caractere.
5. Processamento: A conversão entre representação interna e texto pode adicionar overhead de processamento.

Exemplo de escrita em arquivo de texto em C:

```
FILE *arquivo = fopen("dados.txt", "w");
fprintf(arquivo, "Nome: João Idade: 30 Altura: 1.75");
fclose(arquivo);
```

Arquivos Binários

Os arquivos binários armazenam dados em formato nativo da máquina, sem conversão para texto. Os bytes são interpretados diretamente como valores binários, sem considerar representações de caracteres.

Características dos arquivos binários:

1. Eficiência: Geralmente ocupam menos espaço e são processados mais rapidamente.
2. Precisão: Preservam a representação exata dos dados, sem perdas de precisão em conversões.
3. Não legíveis: Não podem ser facilmente visualizados ou editados em editores de texto.
4. Portabilidade limitada: Podem não ser compatíveis entre diferentes arquiteturas de computadores devido a diferenças na representação interna de dados (endianness, tamanho de tipos, etc.).
5. Acesso direto: Facilitam o acesso aleatório a registros de tamanho fixo.

Exemplo de escrita em arquivo binário em C:

```
struct pessoa {
    char nome[50];
    int idade;
    float altura;
};
struct pessoa p = {"João", 30, 1.75};
```

```
FILE *arquivo = fopen("dados.bin", "wb");
fwrite(&p, sizeof(struct pessoa), 1, arquivo);
close(arquivo);
```

Comparação Prática

Para ilustrar a diferença entre arquivos de texto e binários, considere o armazenamento de um número inteiro, como 12345:
- Em um arquivo de texto, seriam necessários 5 bytes para armazenar os caracteres '1', '2', '3', '4' e '5'.
- Em um arquivo binário, seriam necessários apenas 4 bytes (assumindo um int de 32 bits), com o valor armazenado diretamente em formato binário.

Manipulação com Ponteiros de Arquivo

Além das funções básicas de leitura e escrita, a biblioteca padrão C oferece funções para manipular a posição atual no arquivo (ponteiro de arquivo):
1. fseek() - Posiciona o ponteiro de arquivo em uma posição específica:

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

Os valores possíveis para whence são:
- fseek(arquivo, 0, SEEK_SET); // Posiciona o ponteiro no início do arquivo
- fseek(arquivo, 10, SEEK_CUR); // Avança 10 bytes a partir da posição atual
- fseek(arquivo, -5, SEEK_END); // Posiciona o ponteiro 5 bytes antes do fim do arquivo

2. ftell() - Retorna a posição atual do ponteiro de arquivo:

long ftell(FILE *stream);	Exemplo: long posicao = ftell(arquivo); printf("Posição atual: %ld bytes ", posicao);

3. rewind() - Reposiciona o ponteiro de arquivo no início:

void rewind(FILE *stream);	Exemplo: rewind(arquivo); // Equivalente a fseek(arquivo, 0, SEEK_SET);

4. feof() - Verifica se o fim do arquivo foi atingido:

int feof(FILE *stream);	Exemplo:
	while (!feof(arquivo)) {
	  // Processa o arquivo
	}

Acesso Aleatório vs. Sequencial

Os arquivos podem ser acessados de duas formas principais:
1. Acesso Sequencial: Os dados são lidos ou escritos em sequência, do início ao fim do arquivo. É o modo mais comum para arquivos de texto.
2. Acesso Aleatório: Os dados podem ser acessados diretamente em qualquer posição do arquivo, sem necessidade de ler os dados anteriores. É particularmente útil para arquivos binários com registros de tamanho fixo.
Exemplo de acesso aleatório a registros em um arquivo binário:

```
struct pessoa {
    char nome[50];
    int idade;
    float altura; };
// Abre o arquivo binário
FILE *arquivo = fopen("pessoas.bin", "rb");
// Calcula o tamanho de cada registro
size_t tamanho_registro = sizeof(struct pessoa);
// Acessa diretamente o terceiro registro (índice 2)
fseek(arquivo, 2 * tamanho_registro, SEEK_SET);
// Lê o registro
struct pessoa p;
fread(&p, tamanho_registro, 1, arquivo);
printf("Nome: %s, Idade: %d, Altura: %.2f
", p.nome, p.idade, p.altura);
fclose(arquivo);
```

Tratamento de Erros

O tratamento adequado de erros é crucial na manipulação de arquivos. Além de verificar o retorno de fopen(), é importante verificar o resultado de outras operações:
A função perror() é particularmente útil, pois imprime uma mensagem descritiva do erro baseada no valor da variável global errno.

Arquivos Temporários

Em algumas situações, é necessário criar arquivos temporários que serão utilizados apenas durante a execução do programa. A biblioteca padrão C oferece funções específicas para isso:
1. tmpfile() - Cria um arquivo temporário no modo "wb+":

```
FILE *temp = tmpfile();
if (temp != NULL) {
    // Usa o arquivo temporário
    fclose(temp);  // O arquivo é automaticamente removido ao ser fechado
}
```

2. tmpnam() - Gera um nome único para um arquivo temporário:

```
char nome_temp[L_tmpnam];
if (tmpnam(nome_temp) != NULL) {
    printf("Nome de arquivo temporário: %s
", nome_temp);
    // Usa o nome para criar um arquivo
}
```

Redirecionamento de Entrada e Saída Padrão

É possível redirecionar a entrada e saída padrão (stdin, stdout, stderr) para arquivos. A biblioteca padrão C define três fluxos de arquivo que estão sempre disponíveis:
1. stdin - Entrada padrão (geralmente o teclado)
2. stdout - Saída padrão (geralmente a tela)
3. stderr - Saída de erro padrão (geralmente a tela)

Bufferização de Arquivos

Por padrão, as operações de arquivo em C são bufferizadas para melhorar o desempenho. Isso significa que os dados não são imediatamente escritos no disco, mas armazenados em um buffer na memória até que seja necessário escrevê-los (quando o buffer fica cheio, quando o arquivo é fechado, ou quando fflush() é chamado).
A função fflush() força a escrita dos dados bufferizados:

- fflush(arquivo); // Força a escrita dos dados bufferizados

É possível controlar o modo de bufferização com a função setvbuf():

- setvbuf(arquivo, NULL, _IONBF, 0); // Desativa a bufferização
- setvbuf(arquivo, NULL, _IOLBF, BUFSIZ); // Bufferização por linha
- setvbuf(arquivo, NULL, _IOFBF, BUFSIZ); // Bufferização completa

Boas Práticas na Manipulação de Arquivos

1. Sempre verifique o resultado das operações de arquivo.
2. Feche os arquivos após o uso.
3. Trate adequadamente os erros.
4. Use nomes de arquivo significativos e extensões apropriadas.
5. Documente o formato dos arquivos, especialmente para arquivos binários.
6. Considere a portabilidade ao trabalhar com arquivos binários.
7. Faça backup de arquivos importantes antes de modificá-los.
8. Use funções de alto nível (fprintf, fscanf) para arquivos de texto e funções de baixo nível (fread, fwrite) para arquivos binários.
9. Considere o uso de bibliotecas específicas para formatos complexos (XML, JSON, etc.).
10. Implemente mecanismos de recuperação para casos de falha durante operações críticas.

Conclusão

A manipulação de arquivos é uma habilidade essencial para qualquer programador, permitindo a criação de aplicações que podem armazenar e recuperar dados de forma persistente. A escolha entre arquivos de texto e binários depende das necessidades específicas da aplicação, considerando fatores como legibilidade, eficiência, precisão e portabilidade.

Os arquivos de texto são ideais para dados que precisam ser lidos ou editados por humanos, como arquivos de configuração, logs e dados em formato CSV. Já os arquivos binários são mais adequados para armazenar grandes volumes de dados estruturados, onde a eficiência e a precisão são prioritárias.

Independentemente do tipo de arquivo escolhido, é fundamental seguir boas práticas de programação, como verificar erros, fechar arquivos adequadamente e documentar o formato dos dados. Com o domínio dessas técnicas, o programador estará preparado para desenvolver aplicações robustas que podem persistir e recuperar dados de forma confiável.

Referências

- 📖 Livros e Apostilas
  - KERNIGHAN, B. W.; RITCHIE, D. M. C: A Linguagem de Programação Padrão ANSI. Campus.
  - DEITEL, H. M.; DEITEL, P. J. C: Como Programar. Pearson.
  - BACKES, A. Linguagem C: Completa e Descomplicada. Elsevier.
  - ZIVIANI, N. Projeto de Algoritmos com Implementações em Pascal e C. Cengage Learning.
- 🌐 Recursos Online
  - IC-Unicamp - Manipulação de Arquivos em C - https://www.ic.unicamp.br/~oliveira/doc/mc102_2s2004/aula16.pdf
  - PUCRS - Arquivos em C - https://www.inf.pucrs.br/~pinho/Laprol/Arquivos/Arquivos.htm
  - USP - Arquivos em C - https://www.ime.usp.br/~pf/algoritmos/aulas/io.html
  - UFMG - Manipulação de Arquivos - http://www.decom.ufop.br/romildo/
- 📽️ Vídeos e Cursos
  - Curso em Vídeo - Manipulação de Arquivos em C - https://www.youtube.com/watch?v=PkFSpFQ5I-g
  - Programação Descomplicada - Arquivos em C - https://www.youtube.com/watch?v=LNu-0bzxpos

Anexo - Resumo Estruturado/Mapa Mental Gerado por IA

1. 📖 Introdução

- Persistência de dados
- Diferença entre variáveis em memória x arquivos
- Tipos: **texto** e **binário**

---

2. 📂 Conceito de Arquivo e Manipulação

- Arquivo = coleção de dados em armazenamento secundário
- Identificação: **nome + caminho**
- Acesso via **ponteiro / identificador**
- Etapas da manipulação:
    1. **Abertura** (fopen)
    2. **Processamento** (leitura/escrita)
    3. **Fechamento** (fclose)

---

3. 🛠️ Operações Básicas

Abertura

- fopen(nome, modo) → retorna FILE*
- Modos: `"r"`, `"w"`, `"a"`, `"r+"`, `"w+"`, `"a+"` (+ `b` p/ binários)

Leitura

- fscanf (formatada)
- fgetc (caractere)
- fgets (linha)
- fread (blocos)

Escrita

- fprintf (formatada)
- fputc (caractere)
- fputs (string)
- fwrite (blocos)

Fechamento

- fclose(arquivo) → libera recursos e evita corrupção

---

4. 🔍 Texto vs. Binário

Texto

- Legíveis em editores
- Portáveis
- Mais espaço
- Conversão de números → caracteres

Binário

- Eficiência (menos espaço)
- Precisão (sem conversão)
- Não legíveis diretamente
- Menor portabilidade (endianness, tipos)
- Facilita acesso aleatório

---

5. 🎯 Manipulação Avançada

- **fseek / ftell / rewind** → controle do ponteiro
- **feof** → detectar fim do arquivo

Acesso

- **Sequencial** → do início ao fim
- **Aleatório** → acesso direto a registros

---

6. ⚠️ Recursos Extras

- **Tratamento de erros**: perror, errno
- **Arquivos temporários**: tmpfile, tmpnam
- **Redirecionamento I/O**: stdin, stdout, stderr
- **Bufferização**: fflush, setvbuf

---

7. ✅ Boas Práticas

- Sempre verificar erros
- Fechar arquivos corretamente
- Usar nomes e extensões significativas
- Documentar formatos
- Considerar portabilidade (binários)
- Backups antes de alterações
- Funções de alto nível p/ texto, baixo nível p/ binário

---

8. 🏁 Conclusão

- Escolha texto x binário depende de:
    - Legibilidade
    - Eficiência
    - Precisão
    - Portabilidade
- Fundamentais para aplicações robustas e persistentes