CAPÍTULO 12

Arquivos

A finalidade deste capítulo é apresentar o conceito de arquivos e como utilizá-los dentro de um programa escrito em linguagem C. Ao final, o leitor será capaz de:

- Diferenciar entre os tipos de arquivos existentes.
- Declarar um ponteiro para um arquivo.
- Abrir um arquivo.
- Fechar um arquivo.
- Saber a posição dentro de um arquivo.
- Ler e escrever caracteres no arquivo.
- Ler e escrever strings no arquivo.
- Ler e escrever blocos de bytes no arquivo.
- Ler e escrever dados formatados no arquivo.
- Excluir um arquivo.

12.1 DEFINIÇÃO

Um arquivo, de modo abstrato, nada mais é do que uma coleção de bytes armazenados em um dispositivo de armazenamento secundário, que é geralmente um disco rígido, CD, DVD etc. Essa coleção de bytes pode ser interpretada das mais variadas maneiras:

- Caracteres, palavras ou frases de um documento de texto.
- Campos e registros de uma tabela de banco de dados.
- Pixels de uma imagem.
- etc.

O que define o significado de um arquivo em particular é a maneira como as estruturas de dados estão organizadas e as operações são usadas por um programa de processar (ler ou escrever) esse arquivo.

As vantagens de se usar arquivos são muitas:

- É geralmente baseado em algum tipo de armazenamento durável. Ou seja, seus dados permanecem disponíveis para uso mesmo que o programa que o gerou já tenha sido encerrado.
- Permitem armazenar grande quantidade de informação.
- O acesso aos dados pode ser ou não sequencial.
- Acesso concorrente aos dados (ou seja, mais de um programa pode utilizá-lo ao mesmo tempo).

A linguagem C permite manipular arquivos das mais diversas formas. Ela possui um conjunto de funções que podem ser utilizadas pelo programador para criar e escrever em novos arquivos, ler o seu conteúdo, independentemente do tipo de dados que lá estejam armazenados. A seguir, serão apresentados os detalhes necessários para um programador poder trabalhar com arquivos em seu programa.

12.2 TIPOS DE ARQUIVOS

Basicamente, a linguagem C trabalha com apenas dois tipos de arquivos: **arquivos texto** e **arquivos binários**.

Um *arquivo texto* armazena caracteres que podem ser mostrados diretamente na tela ou modificados por um editor de texto simples como o Bloco de Notas.

Os dados gravados em um arquivo texto são gravados exatamente como seriam impressos na tela. Por isso eles podem ser modificados por um editor de texto simples como o Bloco de Notas. No entanto, para que isso ocorra, os dados são gravados como caracteres de 8 bits utilizando a tabela ASCII. Ou seja, durante a gravação dos dados existe uma etapa de "conversão".

Essa "conversão" dos dados faz com que os arquivos texto sejam maiores. Além disso, suas operações de escrita e leitura consomem mais tempo em comparação às dos arquivos binários.

Para entender essa conversão dos dados em arquivos texto, imagine um número inteiro com oito dígitos: 12345678. Esse número ocupa 32 bits na memória. Porém, quando for gravado em um arquivo texto, cada dígito será convertido em seu caractere ASCII, ou seja, 8 bits por dígito. Como resultado final, esse número ocupará 64 bits no arquivo, o dobro do seu tamanho na memória.

Dependendo do ambiente onde o aplicativo é executado, algumas conversões de caracteres especiais podem ocorrer na escrita/leitura de dados em arquivos texto.

Isso ocorre como forma de adaptar o arquivo ao formato de arquivo texto específico do sistema. No modo de arquivo texto, um caractere de **nova linha**, "\n", pode ser convertido pelo sistema no par de caracteres **retorno de carro + nova linha**, "\r\n".

Um *arquivo binário* armazena uma sequência de bits que está sujeita às convenções dos programas que o gerou.

Os dados gravados em um arquivo binário são gravados exatamente como estão organizados na memória do computador. Isso significa que não existe uma etapa de "conversão" dos dados. Portanto, suas operações de escrita e leitura são mais rápidas do que as realizadas em arquivos texto.

Voltemos ao nosso número inteiro com 8 dígitos: 12345678. Esse número ocupa 32 bits na memória. Quando for gravado em um arquivo binário, o conteúdo da memória será copiado diretamente para o arquivo, sem conversão. Como resultado final, esse número ocupará os mesmos 32 bits no arquivo.

São exemplos de arquivos binários os arquivos executáveis, arquivos compactados, arquivos de registros etc.

Para entender melhor a diferença entre esses dois arquivos, imagine os seguintes dados a serem gravados:

```
char nome[20] = "Ricardo";

int i = 30;

float a = 1.74;
```

A Figura 12.1 mostra como seria o resultado da gravação em um arquivo texto e em um arquivo binário. Note que os dados de um arquivo texto podem ser facilmente modificados por um editor de textos.

FIGURA 12.1

Caracteres são legíveis tanto em arquivos texto quanto os arquivos binários.

12.3 SOBRE ESCRITA E LEITURA EM ARQUIVOS

Quanto às operações de escrita e leitura em arquivos, a linguagem C possui uma série de funções prontas para a manipulação de arquivos, cujos protótipos estão reunidos na biblioteca-padrão de entrada e saída, **stdio.h**.

Diferentemente de outras linguagens, a linguagem C não possui funções que automaticamente leiam todas as informações de um arquivo.

Na linguagem C, as funções de escrita e leitura em arquivos se limitam a operações de abrir/fechar e ler/escrever caracteres e bytes. Fica a cargo do programador criar a função que lerá ou escreverá um arquivo de maneira específica.

12.4 PONTEIRO PARA ARQUIVO

A linguagem C usa um tipo especial de ponteiro para manipular arquivos. Quando o arquivo é aberto, esse ponteiro aponta para o registro 0 (o primeiro registro no arquivo). É esse ponteiro que controla o próximo byte a ser acessado por um comando de leitura. É ele também que indica quando chegamos ao final de um arquivo, entre outras tarefas.

Todas as funções de manipulação de arquivos trabalham com o conceito de "ponteiro de arquivo".

Podemos declarar um ponteiro de arquivo da seguinte maneira:

```
FILE *p;
```

Nesse caso, **p** é o ponteiro que nos permitirá manipular arquivos na linguagem C. Um ponteiro de arquivo nada mais é do que um ponteiro para uma área na memória chamada de "buffer". Nela se encontram vários dados sobre o arquivo aberto, como o seu nome e a posição atual.

12.5 ABRINDO E FECHANDO UM ARQUIVO

12.5.1 Abrindo um arquivo

A primeira coisa que devemos fazer ao trabalhar com arquivos é abri-los. Para abrir um arquivo usa-se a função **fopen()**, cujo protótipo é:

```
FILE *fopen(char *nome_do_arquivo,char *modo)
```

A função **fopen()** recebe dois parâmetros de entrada:

- Nome_do_arquivo: uma string contendo o nome do arquivo que deverá ser aberto.
- Modo: uma string contendo o modo de abertura do arquivo.

E retorna:

- **NULL**: no caso de erro.
- O ponteiro para o arquivo aberto.

Caminho absoluto e relativo para o arquivo

No parâmetro **nome_do_arquivo** pode-se trabalhar com caminhos **absolutos** ou **relativos**.

Imagine que o arquivo com que desejamos trabalhar esteja no seguinte local:

```
"C:\Projetos\NovoProjeto\arquivo.txt"
```

O **caminho absoluto** de um arquivo é uma sequência de diretórios separados pelo caractere barra "(\)", que se inicia no diretório raiz e termina com o nome do arquivo. Nesse caso, o **caminho absoluto** do arquivo é a string:

```
"C:\Projetos\NovoProjeto\arquivo.txt"
```

Já o **caminho relativo**, como o próprio nome diz, é relativo ao local onde o programa se encontra. Nesse caso, o sistema inicia a pesquisa pelo nome do arquivo a partir do diretório do programa. Se tanto o programa quanto o arquivo estiverem no mesmo local, o caminho relativo até esse arquivo será

```
".\arquivo.txt"
```

ou

```
"arquivo.txt"
```

Se o programa estivesse no diretório "C:\Projetos", o **caminho relativo** até o arquivo seria:

```
".\NovoProjeto\arquivo.txt"
```

Ao trabalhar com caminhos **absolutos** ou **relativos**, sempre usar duas barras (\\) em vez de uma (\) para separar os diretórios.

Isso é necessário para evitar que alguma combinação de caractere e barra seja confundida com uma **sequência de escape** que não seja a barra invertida. As duas barras (\\) são as sequências de escape da própria barra invertida. Assim, o **caminho absoluto** do arquivo anteriormente definido passa a ser:

```
"C:\\Projetos\\NovoProjeto\\arquivo.txt"
```

Como posso abrir meu arquivo

O modo de abertura do arquivo determina que tipo de uso será feito dele.

O modo de abertura do arquivo diz à função **fopen()** que tipo de uso será feito do arquivo. Pode-se, por exemplo, querer escrever em um arquivo binário ou ler um arquivo texto. A Tabela 12.1 mostra os modos válidos de abertura de um arquivo.

Modo	Arquivo	Função
"r"	Texto	Leitura. Arquivo deve existir.
"w"	Texto	Escrita. Cria arquivo, se não houver. Apaga o anterior, se ele existir.
"a"	Texto	Escrita. Os dados serão adicionados no fim do arquivo ("append").
"rb"	Binário	Leitura. Arquivo deve existir.
"wb"	Binário	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"ab"	Binário	Escrita. Os dados serão adicionados no fim do arquivo ("append").
"r+"	Texto	Leitura/escrita. O arquivo deve existir e pode ser modificado.
"w+"	Texto	Leitura/escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a+"	Texto	Leitura/escrita. Os dados serão adicionados no fim do arquivo ("append").
"r+b"	Binário	Leitura/escrita. O arquivo deve existir e pode ser modificado.
"w+b"	Binário	Leitura/escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a+b"	Binário	Leitura/escrita. Os dados serão adicionados no fim do arquivo ("append").

TABELA 12.1

Note que, para cada tipo de ação que o programador deseja realizar, existe um modo de abertura de arquivo mais apropriado.

O arquivo deve sempre ser aberto em um modo que permita executar as operações desejadas.

Imagine que desejemos gravar uma informação em um arquivo texto. Obviamente, esse arquivo deve ser aberto em um modo que permita escrever nele. Já um arquivo aberto para leitura não permitirá outra operação que não seja a leitura de dados.

Exemplo: abrir um arquivo binário para escrita

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04    FILE *fp;
05    fp = fopen("exemplo.bin","wb");
06    if(fp == NULL)
07        printf("Erro na abertura do arquivo.\n");
08
09    fclose(fp);
10    system("pause");
11    return 0;
12  }
```

FIGURA 12.2

Nesse exemplo, o comando **fopen()** tenta abrir um arquivo de nome "exemplo.bin" no modo de escrita para arquivos binários, "wb". Note que foi utilizado o **caminho relativo** do arquivo. Na sequência, a condição **if(fp == NULL)** testa se o arquivo foi aberto com sucesso.

Finalizando o programa no caso de erro

No caso de um erro, a função **fopen()** retorna um ponteiro nulo (**NULL**).

Caso o arquivo não tenha sido aberto com sucesso, provavelmente o programa não poderá continuar a executar. Nesse caso, utilizamos a função **exit()**, presente na biblioteca **stdlib.h**, para abortar o programa. Seu protótipo é:

```
void exit(int codigo_de_retorno)
```

A função **exit()** pode ser chamada de qualquer ponto do programa. Ela faz com que o programa termine e retorne, para o sistema operacional, o valor definido em **codigo_de_retorno**.

A convenção mais usada é que um programa retorne **zero**, no caso de um término normal, e um número **não nulo**, no caso de ter ocorrido um problema durante a sua execução.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04    FILE *fp;
05    fp = fopen("exemplo.bin","wb");
06    if(fp == NULL){
07        printf("Erro na abertura do arquivo. Fim de programa.\n");
08        system("pause");
09        exit(1);
10    }
11    fclose(fp);
12    system("pause");
13    return 0;
14  }
```

12.5.2 Fechando um arquivo

Sempre que terminamos de usar um arquivo, devemos fechá-lo. Para realizar essa tarefa, usa-se a função **fclose()**, cujo protótipo é:

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

Basicamente, a função **fclose()** recebe como parâmetro o ponteiro **fp**, que determina o arquivo a ser fechado. Como resultado, a função retorna um valor inteiro igual a zero no caso de sucesso no fechamento do arquivo. Um valor de retorno diferente de zero significa que houve erro nessa tarefa.

Por que devemos fechar o arquivo?

Ao fechar um arquivo, todo caractere que tenha permanecido no buffer é gravado. O buffer é uma área intermediária entre o arquivo no disco e o programa em execução. Trata-se de uma região de memória que armazena temporariamente os caracteres a serem gravados em disco. Apenas quando o buffer está cheio é que seu conteúdo é escrito no disco.

Por que utilizar um buffer durante a escrita em arquivo?

O uso de um buffer é uma questão de **eficiência**. Para ler e escrever arquivos no disco rígido é preciso posicionar a cabeça de gravação em um ponto específico do disco rígido. E isso consome tempo. Se tivéssemos que fazer isso para cada caractere lido ou escrito, as operações de leitura e escrita de um arquivo seriam extremamente lentas. Assim, a gravação só é realizada quando há um volume razoável de informações a serem gravadas ou quando o arquivo for fechado.

A função **exit()** fecha todos os arquivos que um programa tiver aberto.

12.6 ESCRITA E LEITURA EM ARQUIVOS

Uma vez aberto um arquivo, pode-se ler ou escrever nele. Para realizar essas tarefas, a linguagem C conta com uma série de funções de escrita e leitura que variam de funcionalidade de acordo com o tipo de dado que se deseja manipular. Desse modo, todas as mais diversas aplicações do programador podem ser atendidas.

12.6.1 Escrita e leitura de caractere

Escrevendo um caractere

As funções mais básicas e fáceis de se trabalhar em um arquivo são as responsáveis pela escrita e leitura de um único caractere. Para se escrever um caractere em um arquivo usamos a função **fputc()**, cujo protótipo é:

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

A função **fputc()** recebe dois parâmetros de entrada:

- **c**: o caractere a ser escrito no arquivo. Note que o caractere é passado como seu valor inteiro.
- **fp**: a variável que está associada ao *arquivo* onde o caractere será escrito.

E retorna:

- A constante **EOF** (em geral, –1), se houver erro na escrita.
- O próprio caractere, se ele foi escrito com sucesso.

Cada chamada da função **fputc()** grava um único caractere **c** no arquivo especificado.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <string.h>
04  int main(){
05    FILE *arq;
06    char string[100];
07    int i;
08    arq = fopen("arquivo.txt","w");
09    if(arq == NULL){
10      printf("Erro na abertura do arquivo");
11      system("pause");
12      exit(1);
13    }
14    printf("Entre com a string a ser gravada no arquivo:");
      gets(string);
15    //Grava a string, caractere a caractere
16    for(i = 0; i < strlen(string); i++)
17      fputc(string[i], arq);
18    fclose(arq);
19    system("pause");
20    return 0;
21  }
```

No exemplo anterior, a função **fputc()** é utilizada para escrever um caractere na posição atual do arquivo, como assinalado pelo indicador de posição interna do arquivo. Em seguida, esse indicador de posição interna é avançado em um caractere de modo a ficar pronto para a escrita do próximo caractere.

A função **fputc()** também pode ser utilizada para escrever um caractere no dispositivo de saída-padrão (geralmente a tela do monitor).

Para usar a função **fputc()** para escrever na tela, basta modificar o arquivo no qual se deseja escrever para a constante **stdout**. Essa constante é um dos arquivos predefinidos do sistema, um ponteiro para o dispositivo de saída-padrão (geralmente o vídeo) das aplicações. Assim, o comando

```
fputc('*', stdout);
```

escreve um "*" na tela do monitor (dispositivo de saída-padrão) em vez de em um arquivo no disco rígido.

Lendo um caractere

Da mesma maneira que é possível gravar um único caractere em um arquivo, também é possível fazer a sua leitura. A função que corresponde à leitura de caracteres é a função **fgetc()**, cujo protótipo é:

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

A função **fgetc()** recebe como parâmetro de entrada apenas a variável que está associada ao *arquivo* de onde o caractere será lido. Essa função retorna:

- A constante **EOF** (em geral, −1), se houver erro na leitura.
- O caractere lido do arquivo, na forma de seu valor inteiro, se ele foi lido com sucesso.

Cada chamada da função **fgetc()** lê um único caractere do arquivo especificado.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04    FILE *arq;
05    char c;
06    arq = fopen("arquivo.txt","r");
07    if(arq == NULL){
08      printf("Erro na abertura do arquivo");
09      system("pause");
10      exit(1);
11    }
12    int i;
13    for(i = 0; i < 5; i++){
14      c = fgetc(arq);
15      printf("%c",c);
16    }
17    fclose(arq);
18    system("pause");
19    return 0;
20  }
```

No exemplo anterior, a função **fgetc()** é utilizada para ler cinco caracteres de um arquivo. Note que a função **fgetc()** sempre retorna o caractere atualmente apontado pelo indicador de posição interna do arquivo especificado.

A cada operação de leitura, o indicador de posição interna do arquivo é avançado em um caractere para apontar para o próximo caractere a ser lido.

Similarmente ao que acontece com a função **fputc()**, a função **fgetc()** também pode ser utilizada para a leitura de caracteres do teclado. Para tanto, basta modificar o arquivo do qual se deseja ler para a constante **stdin**. Essa constante é um dos arquivos predefinidos do sistema, um ponteiro para o dispositivo de entrada-padrão (geralmente o teclado) das aplicações. Assim, o comando

```
char c = fgetc(stdin);
```

lê um caractere do teclado (dispositivo de entrada-padrão) em vez de um arquivo no disco rígido.

O que acontece quando a função **fgetc()** tenta ler um caractere de um arquivo que já acabou?

Nesse caso, precisamos que a função retorne algo indicando que o arquivo acabou. Porém, todos os 256 caracteres da tabela ASCII são "válidos" em um arquivo. Para evitar esse tipo de situação, a função **fgetc()** não devolve um valor do tipo **char**, mas do tipo **int**. O conjunto de valores do tipo **char** está contido dentro do conjunto de valores do tipo **int**. Se o arquivo tiver acabado, a função **fgetc()** devolve um valor inteiro que não possa ser confundido com um valor do tipo **char**.

Quando atinge o final de um arquivo, a função **fgetc()** devolve a constante **EOF** (**End Of File**), que está definida na biblioteca **stdio.h**. Em muitos computadores, o valor de **EOF** é definido como –1.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04    FILE *arq;
05    char c;
06    arq = fopen("arquivo.txt","r");
07    if(arq == NULL){
08      printf("Erro na abertura do arquivo");
09      system("pause");
10      exit(1);
11    }
12    while((c = fgetc(arq)) != EOF)
13      printf("%c",c);
14    fclose(arq);
15    system("pause");
16    return 0;
17  }
```

Nesse exemplo, a função **fgetc()** é utilizada juntamente com a constante **EOF** para ler não apenas alguns caracteres, mas para continuar lendo caracteres enquanto não chegamos ao final do arquivo.

12.6.2 Fim do arquivo

Como visto anteriormente, a constante **EOF** ("End Of File") indica o fim de um arquivo. Porém, quando manipulando dados binários, um valor inteiro igual ao valor da constante **EOF** pode ser lido. Nesse caso, se utilizarmos a constante **EOF** para verificar se chegamos ao final do arquivo, vamos receber a confirmação quando, na verdade, isso ainda não aconteceu. Para evitar esse tipo de situação, a linguagem C inclui a função **feof()**, que determina quando o final de um arquivo foi atingido. Seu protótipo é:

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

Basicamente, a função **feof()** recebe como parâmetro o ponteiro **fp**, que determina o arquivo a ser verificado. Como resultado, a função retorna um valor inteiro igual a **ZERO** se ainda não tiver atingido o final do arquivo. Um valor de retorno diferente de zero significa que o final do arquivo já foi atingido.

Basicamente, a função **feof()** retorna um valor inteiro diferente de zero se o arquivo chegou ao fim; caso contrário, retorna **ZERO**.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04    FILE *fp;
05    char c;
06    fp = fopen("arquivo.txt","r");
07    if(fp==NULL){
08        printf("Erro na abertura do arquivo\n");
09        system("pause");
10        exit(1);
11    }
12    while(!feof(fp)){
13        c = fgetc(fp);
14        printf("%c",c);
15    }
16    fclose(fp);
17    system("pause");
18    return 0;
19  }
```

12.6.3 Arquivos predefinidos

Como visto durante o aprendizado das funções **fputc()** e **fgetc()**, os ponteiros **stdin** e **stdout** podem ser utilizados para acessar os dispositivos de entrada (geralmente o teclado) e saída (geralmente o vídeo) padrão. Porém, existem outros ponteiros que podem ser utilizados.

No início da execução de um programa, o sistema automaticamente abre alguns arquivos predefinidos, entre eles **stdin** e **stdout**.

stdin	Dispositivo de entrada-padrão (geralmente o teclado)
stdout	Dispositivo de saída-padrão (geralmente o vídeo)
stderr	Dispositivo de saída de erro-padrão (geralmente o vídeo)
stdaux	Dispositivo de saída auxiliar (em muitos sistemas, associado à porta serial)
stdprn	Dispositivo de impressão-padrão (em muitos sistemas, associado à porta paralela)

12.6.4 Forçando a escrita dos dados do buffer

Vimos anteriormente que os dados gravados em um arquivo são primeiramente gravados em um buffer, uma área intermediária entre o arquivo no disco e o programa em