Arquivos

Jander Moreira

1 Primeiras palavras

Uma das primeiras coisas são ensinadas a respeito de computadores são seus elementos constituintes: processador, memória, unidades de entrada e saída. E também que a memória, chamada **memória principal** (ou RAM), é onde ficam armazenados os dados e que ela é... volátil. Isso quer dizer que, sem energia elétrica nos circuitos, os bits não são registrados e, como todos sabem, se a energia parar, tudo o que está na memória é perdido.

É claro que, desde o início, armazenar os dados em algum lugar que fosse permanente foi essencial. Assim, foram criados dispositivos mecânicos (papel e fita perfurada), magnéticos (disquetes e discos rígidos), ópticos (CDs e DVDs) e de estado sólido *(pendrives* e SSDs). Toda essa classe de dispositivos é enquadrada em uma categoria usualmente chamada de **memória secundária**.

Os dispositivos de memória secundária se caracterizam por:

- Grande capacidade de armazenamento;
- Armazenamento em caráter permanente (i.e., não volátil);
- Custo por byte relativamente baixo;
- Velocidade de acesso menor, quando comparada à memória principal.

Para armazenar os dados nesses dispositivos, os sistemas operacionais organizam o espaço onde é possível gravar os dados, criando os **sistemas de arquivos**. Cada SO possui seu próprio sistema de arquivo (ou muitas vezes, seus próprios sistemas de arquivos, no plural). Sistemas baseados no MS Windows usam, por exemplo, FAT e NTFS; as várias distribuições Linux usam Ext3, Ext4 e ReiserFS, entre outras.

Embora cada sistema de arquivos use sua própria organização interna e haja bastante variação de um para outro em termos de estruturação, todos fornecem um modelo abstrato semelhante: os dados são guardados em **arquivos**, sendo cada arquivo descrito por uma série de características, como nome, data da gravação, tamanho e direitos de acesso, por exemplo.

Sob o enfoque de programação, o acesso a dados em arquivos é feito por meio de solicitações ao sistema operacional, o qual se encarrega de localizar o arquivo, verificar os direitos de acesso e localizar os dados gravados no dispositivo.

Este texto apresenta uma visão geral sobre a manipulação de dados gravados em arquivos, mostrando quais os conceitos essenciais envolvidos e mostrando como isso pode ser feito por meio de programas escritos em C. A seção 2 discorre sobre as relações

importantes entre memória principal e secundária; na seção 3 são apresentados os conceitos de programação envolvidos, caracterizando dois tipos de arquivos: binário e texto; as considerações finais estão na seção 4.

2 Memória principal X memória secundária

É sabido que há muitas diferenças entre a memória principal (MP) e a memória secundária (MS). A Tabela 1 apresenta uma comparação geral entre os dois tipos de memória.

Tabela 1. Comparação entre algumas características entre MP e MS.

Característica	Memória principal	Memória secundária
Permanência de armazenamento	Volátil	Permanente
Velocidade de acesso	Rápida	Lenta
Capacidade	Menor	Maior
Custo por MiB	Maior	Menor
Acesso pela CPU	Direto	Transferência para MP

No contexto deste texto, o ponto mais relevante é o acesso pelo processador. Todos os bytes existentes na memória principal estão disponíveis para ser usados pelo processador. Esse é o caso das variáveis que são usadas nos programas. A CPU, porém, não tem acesso aos dados armazenados em um disco rígido, por exemplo. Para que os dados possam ser usados, é preciso que o SO os copie para memória principal.

Desse modo, manipulações de arquivo podem ser descritas como transferências de dados entre a memória secundária e a memória principal, sendo que todas as manipulações efetivas ocorrem na memória principal.

Para exemplificar esse conceito, considera-se a existência de um arquivo no disco rígido de um computador. Esse arquivo contém dados de uma agenda, com informações como nome, telefone fixo, telefone celular, endereço de e-mail etc. para cada pessoa registrada. Supondo-se que se queira imprimir na tela uma relação de todos os contatos do arquivo, listando as informações nome, telefone celular e endereço de e-mail, um programa precisa, para cada pessoa, solicitar ao SO que copie os dados que estão no disco para variáveis na MP e, então, usar essas variáveis para escrever os dados na tela.

Usando-se o mesmo arquivo de agenda, a alteração do número de telefone de algum dos contatos precisa copiar os dados do contato para variáveis na MP, alterar os dados das variáveis e, então, copiar essas informações modificadas de volta para o arquivo.

Assim, qualquer manipulação de dados em disco exige que os bytes que serão trabalhados ocorra na memória principal, fazendo-se uso de cópias (transferência) de dados entre a MP e a MS.

3 Manipulação de arquivos

Para que um programa possa manipular dados em arquivos, é preciso que esse arquivo exista ou seja criado em um dispositivo de memória secundária. Os bytes usados para representar os dados do arquivo devem existir de verdade, ou seja, deve haver o ajuste da superfície magnética de um disco rígido ou a “queima” da superfície de um CD

para registrar os *bits* que devem ser armazenados. O gerenciamento dessa parte é feito pelo sistema operacional, que cuida do que é necessário para a manutenção do **arquivo físico**.

O termo **arquivo físico** é usado para descrever o arquivo real, formado por um conjunto de bytes em algum dispositivo, para o qual o usuário tem acesso por meio de um nome de arquivo. Quem dá acesso ao arquivo físico é o SO. Assim, quando se vê o ícone de um arquivo em uma interface gráfica, ele é apenas uma representação visual do arquivo físico que existe registrado nas trilhas e setores de um disco rígido dentro do gabinete de um computador pessoal, por exemplo.

Os programas, para que tenham acesso aos bytes armazenados em um arquivo físico, precisam fazer solicitações ao SO. Na prática, um conjunto de **chamadas de sistema** é disponibilizada para esse fim. Em programas em C, essas chamadas de sistema são na forma de funções.

Existem algumas manipulações básicas para que o programa tenha acesso aos dados que estão em disco. Elas estão sumarizadas na Tabela 2.

Tabela 2. Manipulações básicas para acesso aos dados em arquivos.

Manipulação	Descrição
Abertura de arquivo	Solicitação do direito de acesso ao arquivo físico, seja para usar um arquivo existente quanto para criar um arquivo novo.
Fechamento de arquivo	Solicitação do encerramento do direito de acesso ao arquivo.
Leitura de dados	Solicitação de cópia de um conjunto de dados (bytes) do dispositivo para variáveis na memória principal.
Escrita de dados	Solicitação da cópia de um conjunto de bytes armazenados em variáveis na memória principal para o dispositivo.

Ao se **abrir** um arquivo, o sistema operacional verifica questões básicas, como se o usuário tem permissões para acesso aos dados do arquivo físico específico, se o arquivo existe (no caso de acesso para recuperar dados) ou se há espaço em disco (no caso da criação de um novo arquivo), entre outras. Além disso, o próprio SO reserva memória para controlar o acesso ao arquivo em questão, criando para isso um **descritor de arquivo** interno. Esse descritor controla coisas como nome do arquivo, tamanho do arquivo em bytes, a porção do arquivo que será lida ou onde haverá a próxima gravação etc.

Quando um arquivo não precisa mais ser usado por um programa, ele é **fechado**, o que significa que o SO é informado que não mais precisa manter na memória o descritor daquele arquivo. Os dados ficam efetivamente gravados no dispositivo e informações como data do último acesso são registradas pelo sistema de arquivos.

Enquanto um arquivo está aberto, é possível ter acesso aos seus dados. Como o processador não tem acesso aos dados do arquivo diretamente, é usada a operação de **leitura** para copiar os bytes que formam um dado no arquivo para a MP, especificamente para uma variável. Com o dados na variável, o programa pode prosseguir com o processamento que precisa fazer.

O processo inverso à leitura é a **escrita** (ou **gravação**). Essa manipulação corresponde a pegar os dados (bytes) que formam uma variável na memória e copiá-los para o dispositivo. Isso permite tanto alterar uma informação já existente no arquivo (sobrepondo os dados) ou aumentando-se o arquivo (acrescentando-se uma nova informação no final dele).

Nos programas, uma variável é usada para representar o arquivo físico, a qual é usada nas chamadas das funções disponíveis. Essa variável corresponde ao conceito de **arquivo lógico**. Assim, o arquivo lógico pode ser visto apenas como uma representação no programa do arquivo físico.

Como cada uma dessas manipulações é feita será apresentado nas seções seguintes. Primeiramente é feita a distinção de arquivos binários e arquivos texto (seção 3.1). Na seção 3.2 são apresentados os arquivos binários, que serão o foco deste texto, enquanto a seção 3.3 discorre sobre arquivos texto. Em cada uma dessas seções serão apresentadas as funções em C que permitem a manipulação adequada dos dados em disco.

3.1 Arquivos binários e arquivos texto

De forma geral, os arquivos são classificados em arquivos binários e arquivos texto.

Um **arquivo texto** é aquele que contém caracteres “legíveis”, com textos e números. Um exemplo de arquivo texto é o código de um programa em C, com textos em comentários, espaços, tabulações e mudanças de linha. Os arquivos texto são usados para fácil visualização do conteúdo por humanos e usam os caracteres para representar as informações se deseja armazenar. Cada caractere corresponde a um byte[a], usando-se tanto para representar as letras e dígitos como as mudanças de linha (“enter”) e as tabulações, por exemplo.

Arquivos binários, por sua vez, não correspondem a arquivos legíveis. Os dados armazenados são guardados usando sua representação binária. Um exemplo ajuda a entender a diferença principal entre os tipos de arquivos. O valor inteiro 127, em um arquivo texto, será representado por três caracteres: '1', '2' e '7', ocupando três bytes no arquivo. Utilizando a tabela ASCII diretamente, seriam usados os bytes indicados na Figura 1(a). Em um arquivo binário, o valor 127 pode usar a representação equivalente do tipo **int** da linguagem C. Nesse caso, o valor usaria quatro bytes, os quais estão apresentados na Figura 1(b). Como os bytes usados não representam caracteres alfanuméricos na tabela, mas caracteres de controle, o número 127 não fica “legível”.

00110001 00110010 00110111

(a)

01111111 00000000 00000000 00000000

(b)

Figura 1. Representações do número 127: (a) Caracteres da tabela ASCII para os dígitos 1, 2 e 7; (b) Bytes usados pelo tipo *int* da linguagem C.

(a) Em princípio, cada caractere ocupa um byte e será usada a tabela ASCII para essa codificação. Existem, porém, codificações diferentes, que podem usar mais de um byte para cada caractere, especialmente em caso de representação de letras acentuadas e símbolos específicos, com € ou £.

Há vantagens e desvantagens em cada uma das representações de arquivos. Se, por um lado, um arquivo texto é facilmente legível[a], por outro ele não permite versatilidade, pois o valor 10 ocuparia dois bytes, enquanto o valor 100000 ocuparia seis. Caso sejam escritos os valores 143 e 37 em um arquivo texto, seu conteúdo seria "14337", o que não permitiria saber exatamente o valor que foi armazenado. Esse caso exigiria o controle de separação entre os valores, usando-se um espaço, tabulação ou um ponto-e-vírgula, por exemplo.

Por sua vez, representações binárias possuem o mesmo número de bytes para simbolizar um dado tipo de dados (e.g., **int** tem sempre quatro bytes), de forma que vários inteiros em um arquivo não precisam ser separados entre si. Uma vantagem adicional dessa representação é que, todos os dados tendo o mesmo número de bytes, é possível saber em que posição cada um deles está, pois os tamanhos são constantes. Por outro lado, para se ter acesso visual à informação armazenada, é preciso ter um programa que leia o valor no formato binário utilizado e o escreva no formato legível.

O enfoque deste texto será sobre arquivos binários, pois são usados para manipulações mais sofisticadas (como bancos de dados, por exemplo). Os arquivos texto são de estrutura mais direta e, assim, serão abordados de forma mais superficial.

3.2 Arquivos binários

Arquivos binários são arquivos formados pelas representações binárias dos dados. Neste texto, o uso das diversas funções em C disponíveis para acesso a arquivos serão vistas por meio de exemplos, de forma a facilitar sua compreensão[b].

O acesso aos dados do arquivo somente é possível se o arquivo estiver aberto. Assim, inicialmente, é preciso aprender a abrir e fechar um arquivo. O Programa 1 apresenta o essencial para abrir (e fechar) um arquivo, mesmo sem acesso aos dados.

Programa 1

```
1.        /*
2.            Abertura e fechamento de arquivo
3.            Jander 2011
4.        */
5.
6.        #include <stdio.h>
7.        #include <errno.h>
8.
9.        int main(){
10.           FILE *descritor;
11.           char nomeArquivo[100];
12.
13.           printf("Nome do arquivo: ");
14.           gets(nomeArquivo);
15.
16.           /* abertura do arquivo */
17.           descritor = fopen(nomeArquivo, "r+");
18.
19.           /* verificacao */
20.           if(!descritor){
```

(a) Nesse caso, um editor de texto simples pode ser usado para abrir o arquivo e ver seu conteúdo.

(b) A abordagem que é apresentada neste texto para arquivos binários é restrita a algumas condições, delimitando um pouco a visão geral de arquivos. Essa opção visa consolidar os conceitos que são apresentados, mas é importante dizer que há uma versatilidade de manipulação maior que a usada nesta discussão.

```
21.                 printf("Erro de abertura do arquivo\n");
22.                 perror("> Resultado da abertura");
23.                 printf("> Erro: %d\n", errno);
24.           }
25.           else{
26.                 printf("Arquivo aberto com sucesso\n");
27.
28.                 /* fechamento do arquivo */
29.                 fclose(descritor);
30.           }
31.
32.           return 0;
33.      }
```

O **include** necessário para acesso às funções de entrada e saída para arquivos é o já conhecido **stdio.h**. Além desse, é feita a inclusão do arquivo **errno.h**, na linha 7. Esse último é usado para acesso à variável **errno**, usada na linha 23[a].

Na linha 10 é declarada a variável usada para controle de arquivo lógico. Em C, a variável é um ponteiro para um descritor de arquivo, indicado pelo tipo **FILE**.

O comando de abertura do arquivo está na linha 17. A função de abertura é a **fopen()**, que tem como primeiro argumento o nome do arquivo. No exemplo, o segundo argumento indica que o acesso ao arquivo preservará seu conteúdo e e também permitirá alterações (escrita). Mas detalhes sobre as opções de abertura são apresentados na sequência. A função **fopen()** retorna, em caso de sucesso, um ponteiro para o descritor de arquivos do sistema operacional; em caso de falha, retorna um ponteiro nulo (NULL).

O sucesso ou insucesso da abertura do arquivo pode ser verificado pelo valor do ponteiro. Na linha 20, a expressão é verdadeira se o ponteiro for nulo (equivale a **descritor == NULL**).

O código das linhas 21 a 23 é executado em caso de erro de abertura, mostrando no exemplo algumas mensagens. Os destaques são feitos para a função **perror()** da linha 22 e para a variável **errno** da linha 23. A função **perror()** apresenta uma mensagem indicada pelo programador seguida de um texto descrevendo o erro encontrado (usualmente em inglês). À variável **errno**, em caso de erro, é atribuído um código específico, que pode ser consultado para cuidar da situação específica[b]. É importante notar que essa variável **não é modificada em caso de sucesso**. Assim, se **errno** tiver valor 2 (que indica erro) e for feita uma abertura de arquivo com sucesso, o conteúdo continua valendo 2, de forma que essa variável **não pode ser usada** para avaliar se a operação foi bem sucedida. Além disso, qualquer outra operação de entrada e saída pode alterar seu conteúdo, o que inclui as funções **gets()**, **printf()**, **scan()** etc.

Se a abertura do arquivo ocorrer com sucesso, então é apresentada uma mensagem indicando o sucesso (linha 26) e, em seguida, o arquivo é fechado na linha 29. A função de fechamento é a **fclose()**, cujo único argumento é o ponteiro do arquivo retornado pela **fopen()**. O fechamento de um arquivo somente é uma operação válida se ele estiver aberto.

O Programa 1 não faz acesso aos dados, de forma que pode ser experimentado[c] com qualquer arquivo sem risco de danificá-lo. Sugere-se que seja testado passando-se

(a) Em alguns compiladores, a inclusão do **errno.h** não é necessária, sendo a variável **errno** definida em **stdio.h**. Há diferenças basicamente entre as plataformas Linux e Windows.

(b) A página http://manpages.ubuntu.com/manpages/hardy/pt/man3/errno.3.html mostra os valores de **errno**.

(c) Observa-se que no Windows os caminhos usam a barra reversa (e.g., **c:\user\nome.txt**). Em C, a barra reversa tem funções especiais, como indicar a mudança de linha com **\n**. Assim, nessa plataforma, é preciso usar \\ para se ter a barra simples: **fopen("c:\\user\\nome.txt", "r")**, por exemplo.

nomes de arquivos que existam, que não existam, aos quais não se tenha acesso (arquivos do sistema, por exemplo). Para cada um deles, verificar a mensagem apresentada e o código de erro.

Quando um programa em C termina, todos os descritores são automaticamente liberados e os arquivos fechados. Não é de bom prática de programação, porém, não usar o comando de fechamento explicitamente no programa.

Agora em mais detalhes, pode-se destacar que a função **fopen()** tem o seguinte formato seguinte. O argumento **path** é o caminho do arquivo, que pode ser explicitado tanto relativo quanto completo. O argumento **mode** é o modo de abertura que se deseja. A Tabela 3 mostra quais as opções para o modo de abertura.

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

Tabela 3. Modos para abertura de arquivos em C.

Modo	Descrição
w	Um arquivo vazio é criado[a], permitindo-se apenas comandos de escrita. Leitura de dados não são permitidas.
w+	Um arquivo vazio é criado, permitindo-se comandos de escrita e leitura de dados.
r	Um arquivo é aberto, se ele existir e houver permissões corretas, permitindo apenas operações de leitura. O ajuste é feito para que o acesso seja feito no início dos dados.
r+	Um arquivo é aberto, se ele existir e houver permissões corretas, permitindo operações de leitura e escrita. O ajuste é feito para que o acesso seja feito no início dos dados.
a	Um arquivo é aberto, se ele existir e houver permissões corretas, permitindo apenas operações de leitura. O ajuste é feito para que o acesso seja feito no fim dos dados.
a+	Um arquivo é aberto, se ele existir e houver permissões corretas, permitindo operações de leitura e escrita. O ajuste é feito para que o acesso seja feito no fim dos dados.

A função **fclose()**, definida abaixo, fecha o arquivo e tem, como único parâmetro, um ponteiro para um descritor válido. O valor zero é retornado em caso de sucesso, ou **EOF** em caso de falha, também ajustando o valor de **errno**.

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

Se o primeiro exemplo mostra como abrir e fechar um arquivo, verificando se houve sucesso na abertura, ainda não foram vistas manipulações de conteúdo. Para exemplificar as funções usadas para leitura e escrita, será considerado um arquivo de dados para guardar dados sobre as notas de alunos. Para cada aluno será registrado seu RA e uma nota.

Foi criado um arquivo de cabeçalho, denominado **registro.h**, o qual contém a definição da **struct** que é usada para os dados. Na listagem Programa 2 é apresentado o conteúdo desse arquivo, que será usado em outros programas.

(a) Se já existir um arquivo com o nome especificado, todo seu conteúdo é apagado e um arquivo vazio é criado. É, portanto, preciso cuidado para que não sejam perdidos dados.

Programa 2

```
1.        /*
2.            Definicoes para os dados do arquivo
3.            Jander 2011
4.        */
5.
6.        #ifndef _REGISTRO_H
7.        #define _REGISTRO_H
8.
9.        typedef struct{
10.                int ra;
11.                float nota;
12.       } tEntrada;
13.
14.       #endif
```

A criação do arquivo inicial de dados é feita pela leitura dos dados do teclado e, para cada um dos registros preenchidos, ele será gravado no arquivo. O Programa 3 contém o código de criação do arquivo.

Programa 3

```
1.        /*
2.            Criacao de um arquivo de agenda
3.            Jander 2011
4.        */
5.
6.        #include <stdio.h>
7.        #include "registro.h"
8.
9.        int main(){
10.           tEntrada aluno;
11.           FILE *arquivo;
12.
13.           /* criacao de uma lista de notas nova (vazia!) */
14.           arquivo = fopen("notas.dat", "w");
15.
16.           if(!arquivo)
17.                perror("Criacao do arquivo falhou");
18.           else{
19.                printf("RA (0 para terminar): ");
20.                scanf("%d", &aluno.ra);
21.
22.                while(aluno.ra != 0){
23.                     printf("Nota: ");
24.                     scanf("%f", &aluno.nota);
25.
26.                     /* copia dos dados para o arquivo */
27.                     fwrite(&aluno, sizeof(aluno),1, arquivo);
28.
29.                     /* proximo aluno */
30.                     printf("RA (0 para terminar): ");
31.                     scanf("%d", &aluno.ra);
32.                }
33.
34.
```

```
35.                 /* fecha o arquivo */
36.                 fclose(arquivo);
37.           }
38.
39.           return 0;
40.       }
```

No Programa 4, a inclusão do arquivo de cabeçalho é indicada na linha 7. É possível notar que o nome do arquivo está entre aspas, mostrando que o arquivo se encontra no mesmo diretório que o código fonte.

As variáveis utilizadas são um registro para guardar os dados (linha 10) e o ponteiro para o descritor de arquivo (linha 11).

A tentativa de criar um arquivo vazio é feita na linha 14 e o sucesso dessa operação é checado logo em seguida, no **if** da linha 16. O nome do arquivo é sempre **notas.dat** e o modo de abertura usado foi com "**w**", indicando a criação de um arquivo vazio.

Se houver falha na criação, por exemplo pela falta de autorização em criar arquivos no diretório atual, então apenas uma mensagem é apresentada usando-se a função **perror()** da linha 17.

Caso a abertura ocorra com sucesso, passa a haver um arquivo no diretório atual vazio (tamanho de zero byte). Se já houvesse um arquivo de mesmo nome, seu conteúdo teria sido perdido e o arquivo vazio o substituiria.

Os comandos das linhas 19 a 32 compõem uma repetição que faz a leitura dos dois campos do registro (RA e nota), terminando quando for digitado um RA igual a zero. Feita a leitura de um registro completo, ele é gravado no arquivo, o que é feito pela função **fwrite**(), na linha 27. Os parâmetros para essa função são, nesta ordem: o endereço dos dados que serão copiados para o arquivo, o número de bytes que devem ser copiados, a quantidade de itens (no caso, apenas um registro) e qual arquivo lógico será usado.

Neste ponto, um novo conceito deve ser apresentado: a **posição corrente** do arquivo. A posição corrente é a posição do próximo byte que será lido ou escrito. Quando o arquivo é aberto com "w" (também para "w+", "r" e "r+"), a posição corrente é o byte **zero**. Ao ser executado o comando **fwrite**, são copiados **sizeof(aluno)** bytes da posição indicada por **&aluno** para o arquivo, fazendo que a posição corrente passe a valer **sizeof(aluno)**. Por exemplo, se a posição corrente for a posição zero e foram copiados 8 bytes para o arquivo, eles ocuparão as posições de 0 a 7, sendo a nova posição corrente ao final do comando a posição 8. Em uma nova escrita, outros 8 bytes serão transferidos, ou seja, escritos no arquivo nas posições de 8 a 15, alterando a posição corrente para o byte 16. Desse modo, ao se escrever cada registro no arquivo, a posição corrente é automaticamente alterada para que seja o próximo byte depois do último escrito. O resultado disso é que cada escrita simplesmente acrescenta seus bytes ao fim do arquivo, aumentando seu tamanho.

Finalmente, quando é digitado um valor para **aluno.ra** igual a zero, a repetição **while** da linha 22 termina e o arquivo é, na sequência, fechado (linha 36).

O resultado final deste programa é um arquivo formado pelos registros escritos, um após o outro, em quantidade correspondente ao número de itens digitados. Além disso, é preciso lembrar que, cada vez que é executado, o programa destrói o arquivo anterior, perdendo os dados, e começando novamente.

Agora que se dispõe do arquivo criado, é preciso ver uma forma de se recuperar a informação digitada. O próximo passo, então, é o Programa 4, cuja função é simplesmente listar o conteúdo do arquivo.

Programa 4

```
1.        /*
2.            Listagem dos dados do arquivo de notas
3.            Jander 2011
4.        */
5.
6.        #include <stdio.h>
7.        #include "registro.h"
8.
9.        int main(){
10.           tEntrada aluno;
11.           int cont = 0;
12.           FILE *arquivo;
13.
14.           /* abertura do arquivo para leitura */
15.           arquivo = fopen("notas.dat", "r");
16.
17.           if(!arquivo)
18.                perror("O arquivo nao foi aberto");
19.           else{
20.                while(fread(&aluno, sizeof(aluno),
21.                                          1, arquivo) != 0){
22.                     printf("Registro #%d\n", cont++);
23.                     printf("  > RA: %d\n", aluno.ra);
24.                     printf("  > Nota: %.1f\n\n", aluno.nota);
25.                }
26.
27.                /* fecha o arquivo */
28.                fclose(arquivo);
29.           }
30.
31.           return 0;
32.       }
```

No Programa 4, o primeiro ponto de destaque é a abertura do arquivo (linha 15), que usa o modo "r", que significa que o arquivo tem seus dados preservados (isto é, o arquivo não se torna vazio) e somente admite operações de leitura de dados, como é o caso da listagem pretendida. Nessa abertura, a posição corrente é a posição zero, ou seja, o primeiro byte do arquivo.

O segundo ponto importante, que é o centro do programa, está na linha 20. Nela aparece o comando **fread()**. Esse comando possui os mesmos parâmetros do **fwrite()** e podem, neste caso, ser descritos da seguinte forma: o primeiro parâmetro é o endereço de memória para onde os dados obtidos do arquivo devem ser copiados (o endereço da variável); o segundo é o número de bytes que devem ser copiados (**sizeof(aluno)** bytes); o próximo é o número de itens que devem ser copiado (apenas um registro de cada vez); e finalmente o descritor do arquivo de onde os dados devem ser obtidos.

O comando **fread()** copia o número de bytes solicitado do arquivo para a variável, iniciando com o byte da posição corrente. Se for considerado que o registro possui, por exemplo, 8 bytes, na primeira leitura serão copiados os bytes de 0 a 7 para a variável. A posição corrente passa automaticamente a ser a posição 8, ou seja, o próximo byte a ser

lido. Na segunda leitura, são copiados os bytes do segundo registro, sucessivamente até o último. A cada leitura, a posição corrente passa automaticamente para o início do próximo registro.

Além deste modo de operação, é possível ver que a função **fread()** retorna um valor, o qual é verificado ainda na linha 21, dentro do **if**. Ao ser chamada a função, é especificado o número de itens que serão lidos, que no exemplo é 1. Quando bem sucedida, a função retorna o número de itens lidos, ou seja, o valor 1. Após a leitura do último registro de todo o arquivo, a posição corrente corresponde ao primeiro byte após o último registro, ou seja, ao primeiro byte "depois do fim do arquivo", onde não há mais dados. Quando é feita a leitura nessa condição, o **fread()** falha, pois não há mais dados, e retorna o número de itens lidos: zero. Assim, quando a função retorna zero é um indicativo que o arquivo terminou e que a repetição deve parar.

Dentro da repetição, cada vez que o comando de leitura é bem sucedido, a variável **aluno** contém uma cópia dos dados do arquivo e pode ter seu conteúdo escrito na tela. Essa é a função das linhas 23 e 24. A variável **cont** é apenas um contador auxiliar para numerar os registros, iniciando-se em zero.

Terminados os dados do arquivo, o **while** termina e o arquivo é fechado (linha 28).

O código C do Programa 4 é a forma padrão de varrer o arquivo todo, registro a registro, obtendo-se cada registro individualmente. Fica como sugestão, fazer alterações nesse programa para que, ao invés de listar, contar quantos alunos possuem nota maior ou igual a 6,0, por exemplo. Ou então, somar todas as notas e calcular a média da turma.

Os comandos de leitura e escrita são definidos na sequência. O parâmetro **ptr** é um ponteiro que indica para onde os dados do arquivo serão copiados ao se fazer a leitura pelo **fread()** ou de onde virão os dados que serão gravados no arquivo pelo **fwrite()**. Em ambas as funções, **size** corresponde ao número de bytes que cada item tem, usualmente usando-se o operador **sizeof** para fazer o cálculo. O parâmetro **nmemb** é o número de itens que devem ser copiados a cada operação e, para leitura de registros um a um, usa-se o valor 1, mas outras quantidades podem ser indicadas[a]. O último dos parâmetros é o descritor do arquivo que deve ser usado na operação.

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

O último exemplo deste texto corresponde a uma manipulação no arquivo anterior, com alteração dos dados. O objetivo do Programa 5 é obter RA e nota de um aluno do teclado e, em seguida, procurar no arquivo se o RA já está presente. Se existir, a nota do aluno deve ser substituída pela nova nota; se não existir, um novo registro deve ser acrescentado ao arquivo, correspondendo aos dados do "novo" aluno.

(a) Se, no lugar de uma variável simples, for passado o endereço de um vetor de 10 posições, os comandos **fread()** ou **fwrite()** podem especificar **nmemb** igual a 10, fazendo, em um único comando, a cópia dos dados de 10 registros. O número de registros efetivamente copiados pode ser verificado consultando-se o valor de retorno da função.

Programa 5

```
1.        /*
2.            Obter dados de um aluno e atualiza-lo no arquivo;
3.            se o RA nao for encontrado, deve ser acrescentado
4.            ao arquivo
5.            Jander 2011
6.        */
7.
8.        #include <stdio.h>
9.        #include "registro.h"
10.
11.       int main(){
12.           tEntrada aluno, novoAluno;
13.           FILE *arquivo;
14.           int achou; // logico
15.
16.           /* criacao de uma lista de notas nova (vazia!) */
17.           arquivo = fopen("notas.dat", "r+");
18.
19.           if(!arquivo)
20.                perror("O arquivo nao foi aberto");
21.           else{
22.                /* leitura dos dados do aluno*/
23.                printf("RA: ");
24.                scanf("%d", &novoAluno.ra);
25.                printf("Nota: ");
26.                scanf("%f", &novoAluno.nota);
27.
28.                /* busca pelo registro do aluno */
29.                achou = 0; // falso
30.                while(!achou && fread(&aluno, sizeof(aluno),
31.                                      1, arquivo) != 0)
32.                     if(aluno.ra == novoAluno.ra)
33.                          achou = 1; // verdadeiro
34.
35.                /* verifica o fim do while */
36.                if(achou){
37.                     /* retorna a posição corrente
38.                        para o inicio do registro */
39.                     fseek(arquivo, -sizeof(aluno), SEEK_CUR);
40.                     /* escreve todo o registro novo
41.                        sobre o anterior */
42.                     fwrite(&novoAluno, sizeof(novoAluno),
43.                                        1, arquivo);
44.                     printf("Registro alterado\n");
45.                }
46.                else{
47.                     /* aproveita que a posicao corrente
48.                        esta no fim do arquivo e escreve */
49.                     fwrite(&novoAluno, sizeof(novoAluno),
50.                                        1, arquivo);
51.                     printf("Novo registro acrescentado\n");
52.                }
53.
54.                /* fecha o arquivo */
55.                fclose(arquivo);
56.           }
57.
58.           return 0;
59.       }
```

O Programa 5, inicialmente, abre o arquivo e verifica o sucesso dessa operação. O modo usado no **fopen()** da linha 17 é "r+", pois os dados devem fica preservados, mas como se pretende fazer alterações, o **+** indica que comandos de escrita também são permitidos.

Se houver sucesso na abertura do arquivo de dados, então são lidos os dados do aluno do teclado (linhas 23 a 26).

A busca pelo dado no arquivo é feita pelo **while** da linha 30, o qual tem duas condições de término possíveis: ao se encontrar um registro cujo RA tenha valor igual ao RA procurado ou ao se chegar ao final do arquivo. A variável **achou** é usada para indicar se o registro foi encontrado.

Ao final da repetição, é verificada a condição da saída. Se a variável **achou** for verdadeira, então o registro foi localizado. Nesse caso, é preciso lembrar que o registro em questão acabou de ser lido e que a posição corrente está no início do registro seguinte. Se for executado um **fwrite()** nesse momento, é o registro seguinte que terá seus dados sobrescritos, o que não é desejável. Assim, para que o resultado desejado seja conseguido, é necessária a volta da posição corrente para o início do registro correto. Isso é feito pelo comando **fseek()**, cuja única função é modificar a posição corrente. No exemplo, o primeiro parâmetro da função é o descritor do arquivo, o segundo é a indicação de como deve ser modificada a posição corrente (em número de bytes) e o último parâmetro é o ponto de referência que deve ser usado. Deste modo, o **fseek()** da linha 39 pode ser interpretado da seguinte forma: altere a posição corrente de **arquivo** voltando **sizeof(novoAluno)** bytes – o sinal negativo indica que há o retrocesso – em relação à posição atual (SEEK_CUR). Quando, finalmente o comando da linha 47 é executado, o registro correto é sobrescrito com o novo conteúdo.

Outra possibilidade é que **achou** seja falso, ou seja, que o RA especificado não esteja no arquivo. Assim um novo registro deve ser escrito. Para que seja acrescentado um novo registro ao arquivo, deve ser executado um comando **fwrite()**, mas a posição corrente deve ser o primeiro byte depois do último registro, senão haverá sobreposição de dados. No caso do exemplo, a posição corrente já está no local esperado, pois se não foi encontrado o registro com um RA igual, então o **while** terminou porque todos os registros foram lidos. Nessa situação a repetição terminou quando se tentou fazer a leitura de um registro e a posição corrente já estava no fim do arquivo, retornando valor 0 para o **fread()**. Assim, neste caso, basta que o comando **fwrite()** seja executado para acrescentar o novo registro. Isso é feito na linha 49.

Por fim, o arquivo é fechado na linha 55.

Sugere-se que o programa, depois de testado e usado algumas vezes, seja modificado de forma que o modo da linha 17 seja apenas "r". Deve-se notar que, embora o programa pareça funcionar normalmente, os dados do arquivo não são modificados no arquivo.

A função **fseek**() tem sua especificação apresentada na sequência. O primeiro parâmetro, **stream**, é o descritor de arquivo. O valor de **offset** indica o deslocamento que deve ser feito (em bytes). O último parâmetro, **whence**, é o ponto de referência que é usado para o deslocamento (Tabela 4).

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

Tabela 4. Referências para o parâmetro whence do fseek().

Referência (whence)	Descrição
SEEK_CUR	O deslocamento em bytes é calculado em relação à posição corrente, isto é, em relação à posição atual. Valores positivos para **offset** indicam avanço na posição corrente, enquanto valores negativos indica retrocesso.
SEEK_SET	O deslocamento é feito em relação ao início do arquivo, ou seja, em relação à posição zero. Neste caso, **offset** deve ser maior ou igual a zero.
SEEK_END	O deslocamento é feito em relação ao fim do arquivo. Os valores de **offset** devem ser negativos, pois valores positivos indica posições além do fim do arquivo.[a]

O comando **fseek()** tem várias utilizades, ilustradas na Tabela 5.

Tabela 5. Exemplos de uso do fseek().

Comando	Resultado
fseek(arq, 0, SEEK_SET)	Posiciona no início do arquivo, ou seja, no byte zero.
fseek(arq, 0, SEEK_END)	Posiciona no fim do arquivo. Se um comando **fwrite()** for executado, um novo registro será acrescentado no fim do arquivo.
fseek(arq, 0, SEEK_CUR)	Não faz nada.
fseek(arq, -sizeof(reg), SEEK_END)	Posiciona no início do último registro do arquivo.
fseek(arq, n*sizeof(reg), SEEK_SET)	Posiciona no início do registro **n**, sendo que o primeiro registro é o registro 0.

Em muitas situações, é preciso marcar a localização de um registro. Nesse caso, é possível fazer uso da função **ftell()**, a qual retorna a posição corrente. Assim, pode-se guardar o valor de uma posição em uma variável e, posteriormente, usar o **fseek()** para retornar a ela. A função **ftell()** retorna a posição corrente em bytes e seu único parâmetro é o descritor de arquivo.

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

3.3 Arquivos texto

Assim como arquivos binários, os arquivos texto são abertos para serem manipulados e devem ser fechados quando não forem mais utilizados. As funções **fopen()** e **fclose()** são usadas da mesma forma que a descrita na seção anterior, bem como as verificações de erros são as mesmas.

A diferença começa quando o conceito de registro, da forma como visto em arquivos binários, deixa de existir. Os arquivos texto não permitem **fseek()**, por exemplo[b].

(a) Programas em C aceitam, em grande parte, que posições além do fim do arquivo sejam especificadas. Ao se escrever além do final, os bytes intermediários, que não existiam, são preenchidos com valor zero. Esse comportamento, entretanto, pode variar conforme o compilador e o sistema operacional utilizados.

(b) A função **fseek()** funciona, porém como o conteúdo é texto, não se sabe onde posicionar. Faz sentido, usualmente, voltar ao início do arquivo ou ir para seu final.

Em geral, o processamento de arquivos texto é a realização da leitura do início ao fim.

Uma forma interessante de ver os arquivos texto é enxergá-los como o teclado, quando forem feitas leituras, ou como tela, quando forem feitas escritas. As funções disponíveis para as manipulações são similares às usadas para leitura e escrita usuais. Por exemplo, no lugar do **printf()** é empregado o **fprintf()**.

Para exemplificar o uso de arquivos texto, é apresentado dois programa que simula a existência de um arquivo de configuração. Uma parte do programa ilustra como algumas configurações podem ser salvas em um arquivo texto.

Programa 6

```
1.        /*
2.            Apresenta calculo de uma funcao de segundo grau
3.            para alguns valores de x
4.            Jander 2011
5.        */
6.
7.        #include <stdio.h>
8.
9.        // definicao dos parametros configuraveis
10.       typedef struct{
11.                 int casas;  // para resultados
12.                 float a, b, c; // coeficientes
13.       } tConfig;
14.
15.       void leiaConfiguracao(tConfig *config){
16.           FILE *arq;
17.           char linha[100];
18.
19.           arq = fopen("config.ini", "r");
20.           if(!arq){
21.                 // se nao abriu, usa valores padrao
22.                 config->casas = 2;
23.                 config->a = config->b = config->c = 1;
24.           }
25.           else{
26.                 // recupera dados do arquivo
27.                 fgets(linha, 100, arq); // linha com comentario
28.                 fscanf(arq, "casas=%d", &config->casas);
29.                 fgetc(arq); // pula \n
30.                 fscanf(arq, "a=%f", &config->a);
31.                 fgetc(arq); // pula \n
32.                 fscanf(arq, "b=%f", &config->b);
33.                 fgetc(arq); // pula \n
34.                 fscanf(arq, "c=%f", &config->c);
35.                 fgetc(arq); // pula \n
36.
37.                 fclose(arq);
38.           }
39.       }
40.
41.       void graveConfiguracao(tConfig config){
42.           FILE *arq;
43.
44.           arq = fopen("config.ini", "w");
45.           if(!arq)
```

```
46.                printf("Aviso: falha ao salvar configuracoes");
47.           else{
48.                fprintf(arq, "# Nao mude a ordem das linhas");
49.                fprintf(arq, " nem coloque espacos\n");
50.                fprintf(arq, "casas=%d\n", config.casas);
51.                fprintf(arq, "a=%f\n", config.a);
52.                fprintf(arq, "b=%f\n", config.b);
53.                fprintf(arq, "c=%f\n", config.c);
54.
55.                fclose(arq);
56.           }
57.      }
58.
59.      void altereConfiguracao(tConfig *config){
60.
61.           printf("Equacao: ax^2 + bx + c\n");
62.           printf("Novo valor para a: ");
63.           scanf("%f", &config->a);
64.           printf("Novo valor para b: ");
65.           scanf("%f", &config->b);
66.           printf("Novo valor para c: ");
67.           scanf("%f", &config->c);
68.           printf("Numero de casas decimais: ");
69.           scanf("%d", &config->casas);
70.      }
71.
72.      void mensagem(){
73.
74.           printf("\nDigite o valor de x\n");
75.           printf(" ou -999 para terminar\n");
76.           printf(" ou -888 para alterar as configuracoes");
77.           printf("\n    valor: ");
78.      }
79.
80.
81.      int main(){
82.           float x;
83.           tConfig config;
84.           char formato[100];
85.
86.           // leitura das configuracoes
87.           leiaConfiguracao(&config);
88.           sprintf(formato,
89.                "%%.%dfx^2 + %%.%dfx + %%.%df = %%.%df\n",
90.                config.casas, config.casas,
91.                config.casas, config.casas);
92.
93.           mensagem();
94.           scanf("%f", &x);
95.           while(x != -999){
96.                // verifica -888
97.                if(x == -888){
98.                     // modifica as configuracoes
99.                     altereConfiguracao(&config);
100.                    sprintf(formato,
101.                    "%%.%dfx^2+ %%.%dfx + %%.%df = %%.%df\n",
102.                    config.casas, config.casas,
103.                    config.casas, config.casas);
104.               }
105.               else
106.                    // mosta o resutado calculado para o x
```

```
107.                     printf(formato, config.a,
108.                        config.b, config.c,
109.                        config.a*x*x + config.b*x + config.c);
110.
111.               // proximo x
112.               mensagem();
113.               scanf("%f", &x);
114.          }
115.
116.          graveConfiguracao(config);
117.
118.          return 0;
119.     }
```

O programa exemplificado apenas usa valores dos coeficientes *a*, *b* e *c* de uma equação de segundo grau e calculo o valor da equação para um dado valor *x*. Os pontos de destaque no programa são as funções **leiaConfiguracao()** e **graveConfiguracao()**.

Em **leiaConfiguracao()**, é feita a tentativa de abrir um arquivo de configurações, que é um arquivo texto, no formato abaixo. Isso é feito na linha 19. O arquivo sempre tem o nome de **config.ini**.

Arquivo config.ini

```
# Nao mude a ordem das linhas nem coloque espacos
casas=2
a=1.000000
b=2.000000
c=5.000000
```

Se a houver problemas com a abertura do arquivo, valores padrão são usados. Se o arquivo abrir, então os dados são lidos dele. Na linha 27 é feita a leitura da linha de comentário, que é colocada em um literal **linha**. A função **fgets()** é usada. Essa função opera de forma similar à função **gets()**, porém permite especificar o tamanho máximo da leitura (segundo parâmetro) e também o descritor de arquivo de onde virão os dados.

As linhas 28, 30, 32 e 34 usam a função **fscanf()** para realizar a leitura das demais linhas. O formato dessa função é igual ao da função **scanf()**, sendo que apenas se acrescenta o primeiro parâmetro, que é o descritor de arquivo. No caso do programa, na cadeia de entrada são descritos, além do **%d** e do **%f**, também o restante do texto que existe na linha do arquivo. Assim o comando consegue separar o texto do valor numérico pretendido. O **fscanf()** não consome o caractere de mudança de linha que existe no arquivo e, assim, é necessária a execução do **fgetc()**, cuja função é apenas pegar o próximo caractere do arquivo (ou seja, o “enter” que existe no fim da linha). Assim são lidos os valores para o número de casas decimais que o programa usa e também os coeficientes da equação que foram salvos.

Em **graveConfiguracao()** é feita a criação de um novo arquivo, usando-se o modo "w" do **fopen()**. Esse arquivo sobrescreve uma versão anterior do **config.ini**, se ela existir. As linhas de 48 a 53 criam as linhas que compõem o arquivo. A função **fprintf()** é empregada para esta escrita. A não ser pelo primeiro parâmetro, que é o descritor do arquivo para onde os textos serão enviados, os demais parâmetros do **fprintf()** são usados da mesma forma que para o **printf()**.

Durante a execução do programa é possível modificar tanto os coeficiente da equação quanto o número de casas decimais usadas para os resultados. Quando o

programa for finalizado (para **x** igual a -999), os valores são salvos no aquivo e, então, usados na próxima execução.

Um último destaque é a função **sprintf()**, que é usada nas linhas 88 e 100. Essa função é usada da mesma forma que o **printf()**, exceto que o texto gerado pelo comando é enviado para o literal especificado no primeiro parâmetro.

Para esse programa exemplo, é interessante observar que o arquivo **config.ini** pode ser editado em um editor de texto comum, alterando-se os valores ali contidos. É preciso observar, porém, que nenhuma verificação de consistência nos dados é realizada, de forma que o arquivo, se em formato incorreto, não trará dados de configuração consistentes para o programa.

4 Considerações finais

O uso de arquivos é importante para a manutenção permanente de dados. Neste texto foram apresentadas a formas principais de acesso aos dados em arquivos, tanto no formato binário como no formato texto.

Conseguir manipular dados em arquivos não é uma tarefa trivial, de forma que entender o modo de funcionamento de cada função especificada é essencial para que os resultados corretos sejam obtidos. Um erro em um **fseek()**, por exemplo, pode fazer que uma escrita de dados “destrua” algumas partes do arquivo, o que leva a uma perda de definitiva dos dados.

Quando se começa a programar para arquivos, os deslizes, grandes ou pequenos, são comuns. É sempre preciso lembrar que, se um erro corromper os dados do arquivo, nenhuma outra execução do programa que use esses dados corrompidos gerará resultados coerentes. Uma sugestão é que sempre seja mantida uma cópia do arquivo de dados correto e, havendo corrupção dos dados, ele seja substituído pela cópia de segurança para que se prossiga com os testes.

Vale lembrar, finalmente, que em todas as funções de arquivo os valores de **errno** podem ser consultados para verificar, em caso de insucesso, a razão da falha.