

Capítulo 10

Arquivos

Neste capítulo, apresentaremos alguns conceitos básicos sobre arquivos e alguns detalhes da forma de tratamento de arquivos em disco na linguagem C. A finalidade desta apresentação é discutir formas variadas para salvar e recuperar informações em arquivos. Com isto, será possível implementar funções para escrever e ler em disco as informações armazenadas nas estruturas de dados que temos discutido.

Um arquivo em disco representa um elemento de informação do dispositivo de memória secundária. A memória secundária (disco) difere da memória principal em diversos aspectos. As duas diferenças mais relevantes são: eficiência e persistência. Enquanto o acesso a dados armazenados na memória principal é muito eficiente do ponto de vista de desempenho computacional, o acesso a informações armazenadas em disco é, em geral, extremamente ineficiente. Para contornar essa situação, os sistemas operacionais trabalham com *buffers*, que representam áreas da memória principal usadas como meio de transferência das informações de/para o disco. Normalmente, trechos maiores (alguns Kbytes) são lidos e armazenados no buffer a cada acesso ao dispositivo. Dessa forma, uma subsequente leitura de dados do arquivo, por exemplo, possivelmente não precisará acessar o disco, pois o dado requisitado pode já se encontrar no buffer. Os detalhes de como esses acessos se realizam dependem das características do dispositivo e do sistema operacional empregado.

Outra grande diferença entre memória principal e secundária (disco) consiste no fato de as informações em disco serem persistentes, em geral sendo lidas por programas e pessoas diferentes dos que as escreveram, o que faz com que seja mais prático atribuir nomes aos elementos de informação armazenados no disco (em vez de endereços), falando assim em arquivos e diretórios (pastas). Cada arquivo é identificado por seu *nome* e pelo *diretório* no qual se encontra armazenado numa determinada unidade de disco. Os nomes dos arquivos são, em geral, compostos pelo *nome em si* seguido de uma *extensão*. A extensão pode ser usada para identificar a natureza da informação armazenada no arquivo ou para identificar o programa que gerou (e é capaz de interpretar) o arquivo. Assim, a extensão “.c” é usada para identificar arquivos que têm códigos fontes da linguagem C, e a extensão “.txt” é, em geral, usada para identificar arquivos textos.

Um arquivo pode ser visto de duas maneiras: em “modo texto”, como um texto composto de uma sequência de caracteres, ou em “modo binário”, como uma sequência de bytes (números binários). Podemos optar por salvar (e recuperar) informações em disco usando um dos dois modos, texto ou binário. Uma vantagem do arquivo texto é que pode ser lido por uma pessoa e editado com editores de textos convencionais. Em

contrapartida, com o uso de um arquivo binário é possível salvar (e recuperar) grandes quantidades de informação de forma mais eficiente. O sistema operacional pode tratar arquivos “textos” de maneira diferente da utilizada para arquivos “binários”. Em casos especiais, pode ser interessante tratar arquivos de um tipo como se fossem do outro, tomando os cuidados apropriados.

Para minimizar a dificuldade com que arquivos são manipulados, os sistemas operacionais oferecem um conjunto de serviços para ler e escrever informações em disco. A linguagem C disponibiliza esses serviços para o programador por meio de um conjunto de funções. Os serviços que mais nos interessam são:

- **Abertura de arquivos:** o sistema operacional encontra o arquivo com o nome dado e prepara o buffer na memória.
- **Leitura do arquivo:** o sistema operacional recupera o trecho solicitado do arquivo. Como o buffer contém parte da informação do arquivo, parte ou toda a informação solicitada pode vir do buffer.
- **Escrita no arquivo:** o sistema operacional acrescenta ou altera o conteúdo do arquivo. A alteração no conteúdo é feita inicialmente no buffer para depois ser transferida para o disco.
- **Fechamento de arquivo:** toda a informação constante do buffer é atualizada no disco (ou descartada, se for um arquivo de leitura) e a área do buffer utilizada na memória é liberada.

Uma das informações que é mantida pelo sistema operacional é um *cursor* que indica a posição de trabalho no arquivo. O arquivo representa uma memória sequencial. O cursor indica a posição corrente nesta memória. Na leitura, é recuperada do arquivo a informação que segue a posição do cursor. Em geral, o cursor percorre a sequência de informação existente no arquivo, do início até o fim, conforme os dados vão sendo recuperados (lidos) para a memória. Na escrita, a informação é salva na posição que segue o cursor, podendo sobreescriver informações já existentes; normalmente, no entanto, os dados são acrescentados quando o cursor se encontra no fim do arquivo.

Nas seções subsequentes, vamos apresentar as funções mais utilizadas em C para acessar arquivos e discutir diferentes estratégias para tratar arquivos. Todas as funções da biblioteca padrão de C que manipulam arquivos encontram-se na biblioteca de entrada e saída (`stdio.h`).

10.1 Funções para abrir e fechar arquivos

A função básica para abrir um arquivo é `fopen`:

```
FILE* fopen (char* nome_arquivo, char* modo);
```

`FILE` é um tipo definido pela biblioteca padrão que representa uma abstração do arquivo. Quando abrimos um arquivo, a função tem como valor de retorno um ponteiro para o tipo `FILE`, e todas as operações subsequentes nesse arquivo receberão este ponteiro como parâmetro de entrada. Se o arquivo não puder ser aberto, a função tem como retorno o valor `NULL`.

Devemos passar o nome do arquivo a ser aberto. O nome do arquivo pode ser relativo, e o sistema procura o arquivo a partir do diretório corrente (diretório de trabalho do programa), ou pode ser absoluto, e, para tanto, especificamos o nome completo do arquivo, incluindo os diretórios, desde o diretório raiz.

Existem diferentes modos de abertura de um arquivo. Podemos abrir um arquivo para *leitura* ou para *escrita*, e devemos especificar se o arquivo será aberto em modo *texto* ou em modo *binário*. O parâmetro *modo* da função *fopen* é uma cadeia de caracteres em que se espera a ocorrência de caracteres que identifiquem o modo de abertura. Os caracteres interpretados no modo são:

- **r** (*read*): Indica modo para leitura.
- **w** (*write*): Indica modo para escrita.
- **a** (*append*): Indica modo para escrita ao final do existente.
- **t** (*text*): Indica modo texto.
- **b** (*binary*): Indica modo binário.

Se o arquivo já existe e solicitamos a sua abertura para escrita com modo **w**, o arquivo é destruído e um novo, inicialmente vazio, é criado. Quando solicitamos com modo **a**, ele é preservado e novos conteúdos podem ser escritos no seu fim. Com ambos os modos, se o arquivo não existe, um novo é criado. Se solicitarmos a abertura de um arquivo para leitura, ele já deve existir, caso contrário, a função falha e tem como retorno o valor **NULL**. A função também tem **NULL** como valor de retorno, se tentarmos abrir um arquivo para escrita numa área (diretório) que não temos acesso de escrita. Se quisermos abrir um arquivo para poder simultaneamente ler e escrever, acrescentamos o caractere **+** no modo de abertura. Assim, **r+** indica leitura e escrita num arquivo já existente e **w+** indica leitura e escrita num novo arquivo.

Os modos **b** e **t** podem ser combinados com os demais. Maiores detalhes podem ser encontrados nos manuais da linguagem C. Quando abrimos um arquivo, testamos o sucesso da abertura antes de qualquer outra operação, por exemplo:

```
...
FILE* fp = fopen("entrada.txt", "rt");
if (fp == NULL) {
    printf("Erro na abertura do arquivo!\n");
    exit(1);
}
...
```

Neste fragmento de código, solicitamos a abertura do arquivo de nome *entrada.txt* para leitura em modo texto. Em seguida, testamos se a abertura do arquivo foi realizada com sucesso.

Após ler/escrever as informações de um arquivo, devemos fechá-lo. Para fechar um arquivo, devemos usar a função *fclose*, que espera como parâmetro o ponteiro do arquivo que se deseja fechar. O protótipo da função é:

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

O valor de retorno dessa função é zero, se o arquivo for fechado com sucesso, ou a constante `EOF` (definida pela biblioteca), que indica a ocorrência de um erro. Naturalmente, só podemos fechar um arquivo previamente aberto. Depois de fechado, não temos mais acesso ao seu conteúdo.

10.2 Arquivos em modo texto

Nesta seção, vamos descrever as principais funções para manipular arquivos em modo texto. Também discutiremos algumas estratégias para a organização de dados em arquivos.

10.2.1 Funções para ler dados

A principal função de C para a leitura de dados em arquivos em modo texto é a função `fscanf`, similar à função `scanf` que temos usado para capturar valores entrados via teclado. No caso da `fscanf`, os dados são capturados de um arquivo previamente aberto para leitura. A cada leitura, os dados correspondentes são transferidos para a memória e o cursor do arquivo avança, passando a apontar para o próximo dado do arquivo (que pode ser capturado numa leitura subsequente). O protótipo da função `fscanf` é:

```
int fscanf (FILE* fp, char* formato, ...);
```

Conforme pode ser observado, o primeiro parâmetro deve ser o ponteiro para o arquivo do qual os dados serão lidos. Os demais parâmetros são os já discutidos para a função `scanf`: o formato e a lista de endereços de variáveis que armazenarão os valores lidos. A função `fscanf` (e isso também vale para a função `scanf`) tem como valor de retorno o número de dados lidos com sucesso. Este valor de retorno nos permitirá verificar se a leitura foi feita com sucesso.

Outra função de leitura muito usada em modo texto é a função `fgetc` que, dado o ponteiro do arquivo, captura o próximo caractere do arquivo (e o cursor avança para o próximo caractere). O protótipo dessa função é:

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

Apesar do tipo do valor de retorno ser `int`, o valor retornado é o código do caractere lido em caso de sucesso. Se o fim do arquivo for alcançado, a constante `EOF` (*end of file*) é retornada.

Outra função também utilizada para ler linhas de um arquivo é a função `fgets`. Essa função recebe como parâmetros três valores: a cadeia de caracteres que armazenará o conteúdo lido do arquivo, o número máximo de caracteres que deve ser lido e o ponteiro do arquivo. O protótipo da função é:

```
char* fgets (char* s, int n, FILE* fp);
```

A função lê do arquivo uma sequência de caracteres, até que um caractere '`\n`' seja encontrado ou que o máximo de caracteres especificado seja alcançado. A especificação de um número máximo de caracteres é importante para evitar que se invada memória.

quando a linha do arquivo for maior do que supúnhamos. Assim, se dimensionarmos nossa cadeia de caracteres, que receberá o conteúdo da linha lida, com 121 caracteres, passaremos esse valor para a função, que lerá no máximo 120 caracteres, pois o último será ocupado pelo finalizador de cadeia de caracteres, o caractere '\0'. O valor de retorno dessa função é o ponteiro da própria cadeia de caracteres passada como parâmetro, ou `NULL`, no caso de ocorrer erro de leitura (por exemplo, quando alcançar o final do arquivo).

É importante salientar que a informação lida é *sempre* aquela que segue o cursor do arquivo. Quando abrimos um arquivo para leitura, o cursor é automaticamente posicionado no início do arquivo. A cada leitura, o cursor avança e passa a apontar para a posição imediatamente após a informação lida. Assim, numa próxima leitura, captura-se a próxima informação do arquivo.

10.2.2 Funções para escrever dados

Dentre as funções que existem para escrever (salvar) dados em um arquivo texto, vamos considerar as duas mais frequentemente utilizadas: `fprintf` e `fputc`, que são análogas, mas para escrita, às funções que vimos para leitura.

A função `fprintf` é similar à função `printf` que temos usado para imprimir dados na saída padrão (em geral, a tela do monitor). A diferença consiste na presença do parâmetro que indica o arquivo para o qual o dado será salvo. O valor de retorno dessa função representa o número de bytes escritos (o mesmo vale para a `printf`). O protótipo da função é dado por:

```
int fprintf(FILE* fp, char* formato, ...);
```

A função `fputc` escreve um caractere no arquivo. O protótipo é:

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

No primeiro parâmetro, especificamos o código do caractere que queremos escrever (salvar). O valor de retorno dessa função é o próprio caractere escrito, ou `EOF`, se ocorrer um erro na escrita.

10.3 Estruturação de dados em arquivos textos

Existem diferentes formas para estruturar os dados em arquivos em modo texto e diferentes formas para capturar as informações contidas neles. A forma de estruturar e a forma de tratar as informações dependem da aplicação. A seguir, apresentaremos três formas para representar e acessar dados armazenados em arquivos: caractere a caractere, linha a linha e usando palavras-chave.

10.3.1 Acesso caractere a caractere

Para exemplificar o acesso caractere a caractere, vamos discutir duas aplicações simples. Inicialmente, vamos considerar o desenvolvimento de um programa que conta o número de linhas de um arquivo (para simplificar, vamos supor um arquivo fixo, com o nome `entrada.txt`). Para calcular o número de linhas do arquivo, podemos ler, caractere a

caractere, todo o conteúdo do arquivo, contando o número de ocorrências do caractere que indica mudança de linha, isto é, o número de ocorrências do caractere '\n':

```

/* Conta número de linhas de um arquivo */

#include <stdio.h>

int main (void)
{
    int c;
    int nlinhas = 0;      /* contador do número de linhas */

    /* abre arquivo para leitura */
    FILE *fp = fopen("entrada.txt","rt");
    if (fp==NULL) {
        printf("Não foi possível abrir arquivo.\n");
        return 1;
    }

    /* lê caractere a caractere */
    while ((c = fgetc(fp)) != EOF) {
        if (c == '\n')
            nlinhas++;
    }

    /* fecha arquivo */
    fclose(fp);

    /* exibe resultado na tela */
    printf("Número de linhas = %d\n", nlinhas);

    return 0;
}

```

Neste programa, como capturamos caractere a caractere, usamos a função `fgetc`, declarando a variável `c` como sendo do tipo `int` (pois `fgetc` retorna um `int`). Note que atribuímos e testamos o valor de `c` dentro da condição do `while`. Isso é válido porque, como vimos, a atribuição em C também é uma expressão, que resulta no valor atribuído (que então é testado neste nosso código).

Como alternativa, podemos reescrever o código usando a função `fscanf` para fazer a leitura dos caracteres:

```

...
char c;
...
while (fscanf("%c",&c)==1) {
    if (c == '\n')
        nlinhas++;
}
...

```

Neste segundo código, prosseguimos com a leitura enquanto a função `fscanf` retornar 1, sinalizando que uma leitura foi realizada com sucesso; no caso, o caractere foi lido com sucesso.

Vamos agora considerar outro exemplo. Vamos escrever um programa que lê o conteúdo do arquivo e cria um arquivo com o mesmo conteúdo, mas com todas as letras minúsculas convertidas para maiúsculas. Os nomes dos arquivos serão fornecidos, via teclado, pelo usuário. Uma possível implementação desse programa é mostrada a seguir:

```
/* Converte arquivo para maiúsculas */

#include <stdio.h>
#include <ctype.h> /* função toupper */

int main (void)
{
    int c;
    char entrada[121]; /* armazena nome do arquivo de entrada */
    char saída[121]; /* armazena nome do arquivo de saída */

    /* pede ao usuário os nomes dos arquivos */
    printf("Digite o nome do arquivo de entrada: ");
    scanf("%120s", entrada);
    printf("Digite o nome do arquivo de saída: ");
    scanf("%120s", saída);

    /* abre arquivos para leitura e para escrita */
    FILE* e = fopen(entrada,"rt");
    if (e == NULL) {
        printf("Não foi possível abrir arquivo de entrada.\n");
        return 1;
    }
    FILE* s = fopen(saída,"wt");
    if (s == NULL) {
        printf("Não foi possível abrir arquivo de saída.\n");
        fclose(e);
        return 1;
    }

    /* lê da entrada e escreve na saída */
    while ((c = fgetc(e)) != EOF)
        fputc(toupper(c),s);

    /* fecha arquivos */
    fclose(e);
    fclose(s);

    return 0;
}
```

Novamente, poderíamos ter usado as funções `fscanf` e `fprintf` para fazer a leitura e a escrita dos caracteres. A função `toupper` da biblioteca `ctype.h` recebe um caractere

e tem como valor de retorno o caractere maiúsculo correspondente; se o caractere de entrada não for uma letra minúscula, o caractere retornado é o mesmo da entrada.

Por fim, vale salientar que a linguagem C oferece a função `ungetc`, que nos permite “devolver” o último caractere lido. Se devolvermos um caractere, este será capturado numa próxima leitura. Trata-se de uma função muito útil quando nossa aplicação precisa “ver”, sem avançar com o cursor, qual a informação que se segue, e então decidir que procedimento deve ser adotado.

10.3.2 Acesso linha a linha

Em diversas aplicações, é mais adequado tratar o conteúdo do arquivo linha a linha. Um caso simples que podemos mostrar consiste em procurar a ocorrência de uma subcadeia de caracteres dentro de um arquivo (análogo a o que é feito pelo utilitário `grep` dos sistemas Unix). Se a subcadeia for encontrada, apresentamos como saída o número da linha da primeira ocorrência.

Para implementar esse programa, vamos utilizar a função `strstr`, que procura a ocorrência de uma subcadeia numa cadeia de caracteres maior. A função tem como valor de retorno o endereço da primeira ocorrência, ou `NULL`, se a subcadeia não for encontrada. O protótipo dessa função é:

```
char* strstr (char* s, char* sub);
```

Nossa implementação consistirá em ler, linha a linha, o conteúdo do arquivo, contando o número da linha. Para cada linha, verificamos a ocorrência da subcadeia, interrompendo a leitura em caso afirmativo.

```
/* Procura ocorrência de subcadeia no arquivo */

#include <stdio.h>
#include <string.h> /* função strstr */

int main (void)
{
    char entrada[121]; /* armazena nome do arquivo de entrada */
    char subcadeia[121]; /* armazena subcadeia */

    /* pede ao usuário o nome do arquivo e a subcadeia */
    printf("Digite o nome do arquivo de entrada: ");
    scanf("%120s",entrada);

    printf("Digite a subcadeia: ");
    scanf("%120s",subcadeia);

    /* abre arquivo para leitura */
    FILE* fp = fopen(entrada,"rt");
    if (fp == NULL) {
        printf("Não foi possível abrir arquivo de entrada.\n");
        return 1;
    }
```

```
/* lê linha a linha */
int n = 0;           /* número da linha corrente */
int achou = 0;        /* indica se achou subcadeia */
char linha[121];     /* armazena cada linha do arquivo */
while (fgets(linha, 121, fp) != NULL) {
    n++;
    if (strstr(linha, subcadeia) != NULL) {
        achou = 1;
        break;
    }
}

/* fecha arquivo */
fclose(fp);

/* exibe saída */
if (achou)
    printf("Achou na linha %d.\n", n);
else
    printf("Nao achou.");

return 0;
}
```

Como outro exemplo de arquivos manipulados linha a linha, vamos considerar um arquivo no qual estão armazenadas informações descrevendo figuras geométricas, a saber: retângulos, triângulos e círculos. Para ter essas descrições num arquivo, temos que escolher um formato apropriado, o qual nos permita recuperar a informação salva. Para exemplificar um formato válido, vamos adotar uma formatação por linha: em cada linha, salvamos um caractere que indica o tipo da figura (r, t ou c), seguido dos parâmetros que a definem: base e altura para os retângulos e triângulos, e raio para os círculos. Para enriquecer o formato, podemos considerar que as linhas iniciadas com o caractere # representam comentários e devem ser desconsideradas na leitura. Por fim, linhas em branco são permitidas e desprezadas. Um exemplo do conteúdo de um arquivo com esse formato é apresentado a seguir (note a presença de linhas em branco e de linhas que são comentários).

```
# Lista de figuras geométricas

r 2.0 1.2
c 5.8
# t 1.23 12
t 4 1.02

c 5.1
```

Para recuperar as informações contidas num arquivo com esse formato, podemos ler do arquivo cada uma das linhas e depois ler os dados contidos na linha. Para tanto,

```

/* lê linha a linha */
int n = 0;           /* número da linha corrente */
int achou = 0;        /* indica se achou subcadeia */
char linha[121];     /* armazena cada linha do arquivo */
while (fgets(linha,121,fp) != NULL) {
    n++;
    if (strstr(linha,subcadeia) != NULL) {
        achou = 1;
        break;
    }
}

/* fecha arquivo */
fclose(fp);

/* exibe saída */
if (achou)
    printf("Achou na linha %d.\n", n);
else
    printf("Nao achou.");

return 0;
}

```

Como outro exemplo de arquivos manipulados linha a linha, vamos considerar um arquivo no qual estão armazenadas informações descrevendo figuras geométricas, a saber: retângulos, triângulos e círculos. Para ter essas descrições num arquivo, temos que escolher um formato apropriado, o qual nos permita recuperar a informação salva. Para exemplificar um formato válido, vamos adotar uma formatação por linha: em cada linha, salvamos um caractere que indica o tipo da figura (r, t ou c), seguido dos parâmetros que a definem: base e altura para os retângulos e triângulos, e raio para os círculos. Para enriquecer o formato, podemos considerar que as linhas iniciadas com o caractere # representam comentários e devem ser desconsideradas na leitura. Por fim, linhas em branco são permitidas e desprezadas. Um exemplo do conteúdo de um arquivo com esse formato é apresentado a seguir (note a presença de linhas em branco e de linhas que são comentários).

```

# Lista de figuras geométricas

r  2.0  1.2
c  5.8
# t  1.23  12
t 4  1.02
c  5.1

```

Para recuperar as informações contidas num arquivo com esse formato, podemos ler do arquivo cada uma das linhas e depois ler os dados contidos na linha. Para tanto,

precisamos apresentar uma função adicional muito útil. Trata-se da função que permite ler dados de uma cadeia de caracteres. A função `sscanf` é similar às funções `scanf` e `fscanf`, mas captura os valores armazenados numa cadeia de caracteres. O protótipo dessa função é:

```
int sscanf (char* s, char* formato, ...);
```

A primeira cadeia de caracteres passada como parâmetro representa a cadeia de caracteres da qual os dados serão lidos. Com essa função, é possível ler uma linha de um arquivo e depois ler as informações contidas na linha. (Analogamente, existe a função `sprintf`, que permite escrever dados formatados numa cadeia de caracteres.)

Faremos a interpretação do arquivo da seguinte forma: para cada linha lida do arquivo, tentaremos ler do conteúdo da linha um caractere (desprezando eventuais caracteres brancos iniciais) seguido de dois números reais. Se nenhum dado for lido com sucesso, significa que temos uma linha vazia e devemos desprezá-la. Se pelo menos um dado (no caso, o caractere) for lido com sucesso, podemos interpretar o tipo da figura geométrica armazenada na linha ou detectar a ocorrência de um comentário. Se for um retângulo ou um triângulo, os dois valores reais também deverão ter sido lidos com sucesso. Se for um círculo, apenas um valor real deverá ter sido lido com sucesso. O fragmento de código a seguir ilustra essa implementação. Supõe-se que `fp` representa um ponteiro para um arquivo com formato válido aberto para leitura, em modo texto.

```
FILE* fp;
...
char linha[121];
while (fgets(linha, 121, fp)) {
    char c;
    float v1, v2;
    int n = sscanf(linha, " %c %f %f", &c, &v1, &v2);
    if (n>0) {
        switch(c) {
            case '#':
                /* desprezar linha de comentário */
                break;
            case 'r':
                if (n!=3) {
                    ... /* tratar erro de formato do arquivo */
                }
                else {
                    ... /* tratar retângulo: base = v1, altura = v2 */
                }
                break;
            case 't':
                if (n!=3) {
                    ... /* tratar erro de formato do arquivo */
                }
                else {
                    ... /* tratar triângulo: base = v1, altura = v2 */
                }
                break;
        }
    }
}
```

```

        case 'c':
            if (n!=2) {
                ... /* tratar erro de formato do arquivo */
            }
            else {
                ... /* tratar círculo: raio = v1 */
            }
            break;
        default:
            ... /* tratar erro de formato do arquivo */
            break;
    }
}
...

```

Note que o processamento linha a linha como descrito garante que todos os parâmetros de uma mesma figura estão numa única linha do arquivo, junto com o caractere identificador. A leitura linha a linha pode também ser útil para tratar ausência de valores. Para exemplificar, vamos considerar um arquivo que representa um conjunto de pontos no espaço 3D. Esses pontos são descritos pelas suas três coordenadas: x , y e z . Um formato bastante flexível para esse arquivo considera que cada ponto é dado em uma linha e permite a omissão da terceira coordenada, se ela for igual a zero. Dessa forma, o formato atende também a descrição de pontos no espaço 2D. Um exemplo desse formato é ilustrado a seguir:

2.3 4.5 6.0
1.2 10.4
7.4 1.3 9.6
5.4 12.2
4.5 8.6
8.8 4.2 7.0
...

Para interpretar esse formato, devemos ler cada uma das linhas e tentar ler três valores reais de cada linha (aceitando o caso em que apenas dois valores são lidos com sucesso).

10.3.3 Acesso via palavras-chave

Quando os objetos num arquivo têm descrições de tamanhos variados, é comum adotar uma formatação com o uso de palavras-chave. Cada objeto é precedido por uma palavra-chave que o identifica. A interpretação desse tipo de arquivo pode ser feita lendo-se as palavras-chave e interpretando a descrição do objeto correspondente. Para ilustrar, vamos considerar que, além de retângulos, triângulos e círculos, também temos polígonos quaisquer no nosso conjunto de figuras geométricas. Cada polígono pode ser descrito pelo número de vértices que o compõe, seguido das respectivas coordenadas desses vértices:

```
RETANGULO
```

```
 b   h
```

```
TRIANGULO
```

```
 b   h
```

```
CIRCULO
```

```
 r
```

```
POLIGONO
```

```
 n
```

```
 x1  y1
```

```
 x2  y2
```

```
 . . .
```

```
 xn  yn
```

O fragmento de código a seguir ilustra a interpretação desse formato, onde `fp` representa o ponteiro para o arquivo aberto para leitura:

```
FILE* fp;
...
char palavra[121];
while (fscanf(fp,"%120s",palavra) == 1)
{
    if (strcmp(palavra,"RETANGULO")==0) {
        /* interpreta retângulo */
    }
    else if (strcmp(palavra,"TRIANGULO")==0) {
        /* interpreta triângulo */
    }
    else if (strcmp(palavra,"CIRCULO")==0) {
        /* interpreta círculo */
    }
    else if (strcmp(palavra,"POLIGONO")==0) {
        /* interpreta polígono */
    }
    else {
        /* trata erro de formato */
    }
}
```

10.4 Arquivos em modo binário

Os arquivos em modo binário servem para salvar (e depois recuperar) o conteúdo da memória principal diretamente no disco. A memória é salva copiando-se o conteúdo de cada byte da memória para o arquivo. Uma das grandes vantagens de se usar arquivos binários é que podemos salvar (e recuperar) uma grande quantidade de dados de forma bastante eficiente. Nesta seção, vamos apenas apresentar as funções básicas para a manipulação de arquivos binários.

10.4.1 Funções para salvar e recuperar

Para escrever (salvar) dados em arquivos binários, usamos a função `fwrite`. O protótipo dessa função pode ser simplificado por:

```
int fwrite (void* p, int tam, int nelem, FILE* fp);
```

O primeiro parâmetro dessa função representa o endereço de memória cujo conteúdo deseja-se salvar em arquivo. O parâmetro `tam` indica o tamanho, em bytes, de cada elemento e o parâmetro `nelem` indica o número de elementos. Por fim, passa-se o ponteiro do arquivo binário para o qual o conteúdo da memória será copiado (na posição que segue o cursor).

A função `fread`, para ler (recuperar) dados de arquivos binários, é análoga, sendo que agora o conteúdo do disco é copiado para o endereço de memória passado como parâmetro. O protótipo da função pode ser ilustrado por:

```
int fread (void* p, int tam, int nelem, FILE* fp);
```

Para exemplificar a utilização dessas funções, vamos considerar que uma aplicação tem um conjunto de pontos armazenados num vetor. O tipo que define o ponto pode ser:

```
typedef struct ponto Ponto;
struct ponto {
    float x, y, z;
};
```

Uma função para salvar o conteúdo de um vetor de pontos pode receber como parâmetros o nome do arquivo, o número de pontos no vetor e o ponteiro para o vetor. Uma possível implementação dessa função é:

```
void salva (char* arquivo, int n, Ponto* vet)
{
    FILE* fp = fopen(arquivo, "wb");
    if (fp==NULL) {
        printf("Erro na abertura do arquivo.\n");
        exit(1);
    }
    fwrite(vet, sizeof(Ponto), n, fp);
    fclose(fp);
}
```

Assumindo a correta alocação do vetor, a função para recuperar os dados salvos pode ser:

```
void carrega (char* arquivo, int n, Ponto* vet)
{
    FILE* fp = fopen(arquivo, "rb");
    if (fp==NULL) {
        printf("Erro na abertura do arquivo.\n");
        exit(1);
    }
    fread(vet, sizeof(Ponto), n, fp);
    fclose(fp);
}
```

Outra grande vantagem oferecida pelo uso de arquivos binários consiste na possibilidade de recuperar apenas parte da informação armazenada. Num arquivo binário, nós, programadores, temos o controle de quantos bytes ocupa cada informação armazenada no arquivo. Com isso, podemos alterar a posição do cursor do arquivo, o que nos permite posicioná-lo para ler uma informação. A função que nos permite movimentar o cursor do arquivo tem o seguinte protótipo:

```
int fseek (FILE* fp, long offset, int origem);
```

O primeiro parâmetro indica o arquivo no qual estamos reposicionando o cursor. O segundo parâmetro indica quantos bytes iremos avançar e o terceiro parâmetro indica em relação a que posição estamos avançando o cursor: em relação à posição corrente (`SEEK_CUR`), em relação ao início do arquivo (`SEEK_SET`) ou em relação ao final do arquivo (`SEEK_END`).

Para exemplificar, vamos considerar a existência de um arquivo de pontos no espaço 3D salvo como exemplificado. Vamos então escrever uma função que, dado um ponteiro para o arquivo aberto para leitura, faça a captura do ponto de índice i armazenado. Uma possível implementação dessa função é mostrada a seguir:

```
Ponto le_ponto (FILE* fp, int i)
{
    Ponto p;
    fseek(fp,i*sizeof(Ponto),SEEK_SET); /* posicione o cursor */
    fread(&p,sizeof(Ponto),1,fp);        /* recupere a informação do ponto
    return p;
}
```

Exercícios

1. Considere um arquivo texto com as notas dos alunos de uma disciplina. Cada linha do arquivo contém a matrícula de um aluno (cadeia de nove caracteres), seguida pelos valores de suas três notas (P_1 , P_2 e P_3). Considere ainda que podem existir linhas em branco no arquivo. Um exemplo desse formato é:

9010087-2 2.0 4.3 6.5
8820324-3 7.0 8.2 8.6
9210478-5 6.0 7.5 7.8
9020256-8 3.0 0.5 4.2

Escreva uma função que receba como parâmetros o número de matrícula de um aluno e o nome de um arquivo com as notas de uma disciplina no formato descrito, e retorne a média do aluno na disciplina. A média de um aluno é calculada pela fórmula $(P_1 + P_2 + P_3)/3$. O protótipo da função deve ser:

```
float media (char* mat, char* nome_arquivo);
```

Caso o número de matrícula passado como parâmetro não seja encontrado no arquivo, a função deve retornar -1.0 . Se não for possível abrir o arquivo de entrada, a função deve imprimir a mensagem “Erro” e terminar a execução do programa.

2. Considere a existência de um arquivo texto, denominado “turma.txt”, com um cadastro de uma turma. Para cada aluno da turma, existem duas linhas no arquivo: na primeira linha, encontra-se o nome do aluno; na linha seguinte, encontram-se as duas notas obtidas pelo aluno. Considere que podem existir linhas em branco no arquivo. Um exemplo deste arquivo é mostrado a seguir:

```
Fulano Pereira
9.0 8.0
Beltrano Silva
4.0 5.0

Sicrano Santos
3.0 7.0
Fulana Souza
4.0 4.0

Maria Paula
6.0 7.0
```

Escreva um programa completo que leia o conteúdo do arquivo (“turma.txt”) com o formato anterior e crie outro arquivo, com o nome “aprovados.txt”, com a lista dos alunos que obtiveram médias maiores ou iguais a 5.0 , seguindo a mesma ordem do arquivo de entrada. Cada linha do arquivo de saída deve conter o nome e a média obtida pelo aluno. Se esse arquivo anterior fosse usado num exemplo, a saída obtida seria:

```
Fulano Pereira      8.5
Sicrano Santos    5.0
Maria Paula        6.5
```

Se o arquivo de entrada não puder ser aberto, deve-se imprimir a mensagem “Erro” na tela e abortar o programa. Pode-se considerar que sempre será possível abrir o arquivo de saída.

3. Considere um tipo que representa um funcionário de uma empresa, definido pela estrutura a seguir:

```
typedef struct funcionario Funcionario;
struct funcionario {
    char nome[81];      /* nome do funcionário */
    float valor_hora;   /* valor da hora de trabalho em Reais */
    int horas_mes;      /* horas trabalhadas em um mês */
};
```

Escreva uma função em C que preencha um vetor de ponteiros para `Funcionario` com os dados lidos de um arquivo texto. Essa função deve receber como parâmetros o vetor de ponteiros para `Funcionario` e o nome do arquivo de entrada. Nesse arquivo de entrada, os dados de cada funcionário são armazenados em duas linhas: uma com o seu nome (cadeia com até 80 caracteres), e outra com o valor de sua hora de trabalho e com o número de horas trabalhadas em um mês (nessa ordem). Um exemplo desse formato é mostrado a seguir.

```

João da Silva
15.0 160
Manuel Santos
15.0 80
Fulana de Tal
23.5 40

```

A função implementada deve ter o seguinte protótipo:

```
void carrega (int n, Funcionario** vet, char* arquivo);
```

Considere que: não existem linhas em branco no arquivo; o parâmetro `vet` já vem com todas as suas posições inicializadas com `NULL`; o comprimento do vetor (parâmetro `n`) é no mínimo igual à quantidade de registros de funcionários no arquivo de entrada; se não for possível abrir o arquivo, a função deve imprimir a mensagem “Erro” e terminar a execução do programa.

4. Considere um tipo que representa um aluno de um curso, definido pela estrutura a seguir:

```

typedef struct aluno Aluno;
struct aluno {
    char mat[8];           /* matrícula do aluno */
    char nome[81];          /* nome do aluno */
    float cr;               /* coeficiente de rendimento */
};

```

Considere ainda a existência de um arquivo texto no qual se tem a lista dos alunos do curso. A primeira informação do arquivo é um número inteiro que indica a quantidade de alunos listados no arquivo. A seguir, em cada linha, encontra-se os dados de cada um dos alunos listados: número de matrícula, nome entre aspas simples e coeficiente de rendimento. Um exemplo de um arquivo com uma lista de alunos é apresentado a seguir:

```

4
0011234  'Fulano Pereira'  8.6
0111224  'Beltrano Silva'   7.1
0421233  'Sicrano Santos'   5.6
0314231  'Fulana Souza'     9.4

```

Escreva uma função em C que leia um arquivo com este formato e crie dinamicamente um vetor de `Aluno`. Essa função recebe como parâmetro o nome do arquivo de entrada. Ela deve então ler o número de alunos existentes, alocar dinamicamente o vetor e ler o conteúdo do arquivo para o vetor. A função recebe ainda um parâmetro adicional que representa um ponteiro de uma variável inteira, no qual deve-se armazenar o número de alunos existentes. A função deve retornar o vetor alocado (e devidamente preenchido), obedecendo ao seguinte protótipo:

```
Aluno* carrega (char* arquivo, int *n);
```

Se ocorrer erro na abertura do arquivo, a função deve retornar `NULL`. Se o arquivo existir, considere que existe pelo menos um aluno listado e que não existem linhas em branco no arquivo. Considere ainda que os nomes dos alunos não terão mais do que 80 caracteres.