# 14. Arquivos

W. Celes e J. L. Rangel

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 variadas formas para salvar (e recuperar) informações em arquivos. Com isto, será possível implementar funções para salvar (e recuperar) 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. Desta forma, uma subseqüente 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 estes acessos se realizam dependem das características do dispositivo e do sistema operacional empregado.

A outra grande diferença entre memória principal e secundária (disco) consiste no fato das informações em disco serem persistentes, e em geral são 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 do disco (em vez de endereços), falando assim em arquivos e diretórios (pastas). Cada arquivo é identificado por seu nome e pelo diretório onde encontra-se 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 ".doc" é, no Windows, usada para identificar arquivos gerados pelo editor Word da Microsoft.

Um arquivo pode ser visto de duas maneiras, na maioria dos sistemas operacionais: em "modo texto", como um texto composto de uma seqüência de caracteres, ou em "modo binário", como uma seqüência 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 bastante eficiente. O sistema operacional pode tratar arquivos "texto" de maneira diferente da utilizada para tratar 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 do

disco. A linguagem C disponibiliza esses serviços para o programador através de um conjunto de funções. Os principais serviços que 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 do arquivo 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 e a área do buffer utilizada na memória é liberada.

Uma das informações que é mantida pelo sistema operacional é um ponteiro de arquivo (*file pointer*), que indica a posição de trabalho no arquivo. Para ler um arquivo, este apontador percorre o arquivo, do início até o fim, conforme os dados vão sendo recuperados (lidos) para a memória. Para escrever, normalmente, os dados são acrescentados quando o apontador se encontra no fim do arquivo.

Nas seções subseqüentes, vamos apresentar as funções mais utilizadas em C para acessar arquivos e vamos 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, com interface em `stdio.h`.

## 14.1. 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 subseqüentes nesse arquivo receberão este endereço 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, onde 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 onde espera-se a ocorrência de caracteres que identificam o modo de abertura. Os caracteres interpretados no modo são:

| | | |
|---|---|---|
| `r` | *read-only* | Indica modo apenas para leitura, não pode ser alterado. |
| `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 apagado e um novo, inicialmente vazio, é criado. Quando solicitamos com modo `a`, o mesmo é preservado e novos conteúdos podem ser escritos no seu fim. Com ambos os modos, se o arquivo não existe, um novo é criado.

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

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

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.

## *14.2. Arquivos em modo texto*

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

### Funções para ler dados

A principal função de C para leitura de dados em arquivos em modo texto é a função `fscanf`, similar à função `scanf` que temos usado para capturar valores entrados via o 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 ponteiro do arquivo avança, passando a apontar para o próximo dado do arquivo (que pode ser capturado numa leitura subseqüente). O protótipo da função `fscanf` é:

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

Conforme pode ser observado, o primeiro parâmetro deve ser o ponteiro do 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. Como a função `scanf`, a função `fscanf` também tem como valor de retorno o número de dados lidos com sucesso.

Uma 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. O protótipo dessa função é:

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

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

Uma outra função muito 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 seqüência 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 evitarmos 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 *string* – 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).

### Funções para escrever dados

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

A função `fprintf` é análoga a função `printf` que temos usado para imprimir dados na saída padrão – em geral, o 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 no arquivo. 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);
```

O valor de retorno dessa função é o próprio caractere escrito, ou `EOF` se ocorrer um erro na escrita.

## *14.3. Estruturação de dados em arquivos textos*

Existem diferentes formas para estruturarmos os dados em arquivos em modo texto, e diferentes formas de capturarmos as informações contidas neles. A forma de estruturar e

a forma de tratar as informações dependem da aplicação. A seguir, apresentaremos três formas de representarmos e acessarmos dados armazenados em arquivos: caractere a caractere, linha a linha, e usando palavras chaves.

## 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 as linhas de um determinado 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 */
   FILE *fp;

   /* abre arquivo para leitura */
   fp = fopen("entrada.txt","rt");
   if (fp==NULL) {
      printf("Não foi possivel 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("Numero de linhas = %d\n", nlinhas);

   return 0;
}
```

Como segundo exemplo, vamos considerar o desenvolvimento de 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 saida[121];      /* armazena nome do arquivo de saída */
   FILE* e;              /* ponteiro do arquivo de entrada */
   FILE* s;              /* ponteiro 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 saida: ");
   scanf("%120s",saida);

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

## 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 sub-cadeia de caracteres dentro de um arquivo (análogo a o que é feito pelo utilitário `grep` dos sistemas Unix). Se a sub-cadeia 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 sub-cadeia numa cadeia de caracteres maior. A função retorna o endereço da primeira ocorrência ou `NULL`, se a sub-cadeia não for encontrada. O protótipo dessa função é:

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

A nossa implementação consistirá em ler, linha a linha, o conteúdo do arquivo, contanto o número da linha. Para cada linha, verificamos se a ocorrência da sub-cadeia, interrompendo a leitura em caso afirmativo.

```
/* Procura ocorrência de sub-cadeia no arquivo */

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

int main (void)
{
   int n = 0;             /* número da linha corrente */
   int achou = 0;         /* indica se achou sub-cadeia */
   char entrada[121];     /* armazena nome do arquivo de entrada */
   char subcadeia[121];   /* armazena sub-cadeia */
   char linha[121];       /* armazena cada linha do arquivo */
   FILE* fp;              /* ponteiro do arquivo de entrada */

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

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

   /* lê linha a linha */
   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 segundo exemplo de arquivos manipulados linha a linha, podemos citar o caso em que salvamos os dados com formatação por linha. Para exemplificar, vamos considerar que queremos salvar as informações da lista de figuras geométricas que discutimos na seção 9.3. A lista continha retângulos, triângulos e círculos.

Para salvar essas informações num arquivo, temos que escolher um formato apropriado, que nos permita posteriormente 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 definem a figura, base e altura para os retângulos e triângulos ou 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 na Figura 14.1 (note a presença de linhas em branco e linhas que são comentários):

```
# Lista de figuras geometricas

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

c 5.1
```

Figura 14.1: Exemplo de formatação por linha.

Para recuperarmos 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 introduzir 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 *string*. O protótipo dessa função é:

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

A primeira cadeia de caracteres passada como parâmetro representa a *string* 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 *string*.)

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 abaixo ilustra essa implementação. Supõe-se que `fp` representa um ponteiro para um arquivo com formato válido aberto para leitura, em modo texto.

```
char c;
float v1, v2;
FILE* fp;
char linha[121];
...
while (fgets(linha,121,fp)) {
   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 {
                /* interpretar retângulo: base = v1, altura = v2 */
                ...
            }
        break;
        case 't':
            if (n!=3) {
                /* tratar erro de formato do arquivo */
                ...
            }
            else {
                /* interpretar triângulo: base = v1, altura = v2 */
                ...
              }
        break;
        case 'c':
            if (n!=2) {
                /* tratar erro de formato do arquivo */
                ...
            }
            else {
                /* interpretar círculo: raio = v1 */
                ...
            }
        break;
        default:
            /* tratar erro de formato do arquivo */
            ...
        break;
    }
  }
}
...
```

A rigor, para o formato descrito, não precisávamos fazer a interpretação do arquivo linha a linha. O arquivo poderia ter sido interpretado capturando-se inicialmente um caractere que então indicaria qual a próxima informação a ser lida. No entanto, em algumas situações a interpretação linha a linha ilustrada acima é a única forma possível. Para exemplificar, vamos considerar um arquivo que representa um conjunto de pontos no espaço 3D. Esses pontos podem ser dados 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 essa for igual a zero. Dessa forma, o formato atende também a descrição de pontos no espaço 2D. Um exemplo desse formato é ilustrado abaixo:

```
2.3   4.5  6.0
1.2  10.4
7.4   1.3  9.6
...
```

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

Exercício: Faça um programa que interprete o formato de pontos 3D descrito acima, armazenando-os num vetor.

## Acesso via palavras chaves

Quando os objetos num arquivo têm descrições de tamanhos variados, é comum adotarmos uma formatação com o uso de palavras chaves. Cada objeto é precedido por uma palavra chave que o identifica. A interpretação desse tipo de arquivo pode ser feita lendo-se as palavras chaves 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. A Figura 14.2 ilustra esse formato.

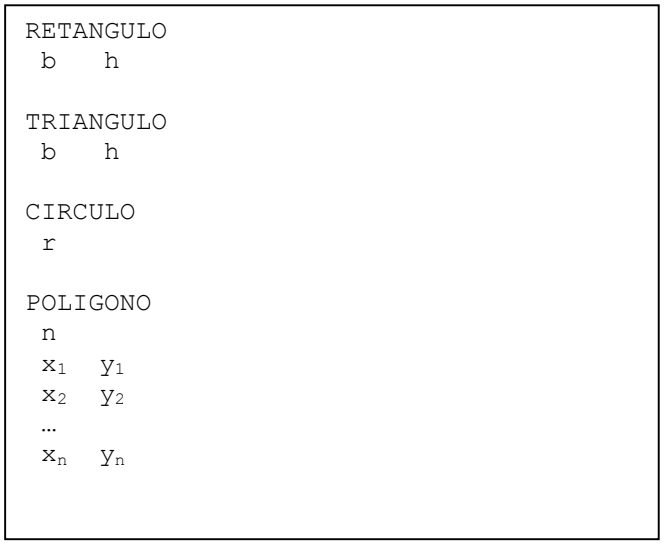


Figura 14.2: Formato com uso de palavras chaves.

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 */
   }
}
```

## 14.4. Arquivos em modo binário

Arquivos em modo binário servem para salvarmos (e depois recuperarmos) o conteúdo da memória principal diretamente no disco. A memória é escrita 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. Neste curso, vamos apenas apresentar as duas funções básicas para manipulação de arquivos binários.

### Função para salvar e recuperar

Para escrever (salvar) dados em arquivos binários, usamos a função `fwrite`. O protótipo dessa função pode ser simplificado por[1]:

```
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.

A função 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 dado 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:

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

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 ilustrada abaixo:

```
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);
}
```

[1] A rigor, os tipos `int` são substituídos pelo tipo `size_t`, definido pela biblioteca padrão, sendo, em geral, sinônimo para um inteiro sem sinal (`unsigned int`).

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);
}
```