AULA 17

ARQUIVOS

Nos programas que fizemos até aqui, a entrada e a saída de dados sempre ocorreram em uma janela de terminal ou console do sistema operacional. Na linguagem C, não existem palavras-chaves definidas para tratamento de operações de entrada e saída. Essas tarefas são realizadas através de funções. Pouco usamos outras funções de entrada e saída da linguagem C além das funções `scanf` e `printf`, localizadas na biblioteca padrão de entrada e saída, com arquivo-cabeçalho `stdio.h`. Isso significa que o fluxo de dados de entrada e saída dos programas que fizemos sempre passa pela memória principal. Nesta aula, baseada na referência [7], aprenderemos funções que realizam tarefas de entrada e saída armazenados em um dispositivo de memória secundária em arquivos.

17.1 Seqüências de caracteres

Na linguagem C, o termo **seqüência de caracteres**, do inglês *stream*, significa qualquer fonte de entrada ou qualquer destinação para saída de informações. Os programas que produzimos até aqui sempre obtiveram toda sua entrada a partir de uma seqüência de caracteres, em geral associada ao teclado, e escreveram sua saída em outra seqüência de caracteres, associada com o monitor.

Programas maiores podem necessitar de seqüências de caracteres adicionais, associadas a arquivos armazenados em uma variedade de meios físicos tais como discos e memórias ou ainda portas de rede e impressoras. As funções de entrada e saída mantidas em `stdio.h` trabalham do mesmo modo com todas as seqüências de caracteres, mesmo aquelas que não representam arquivos físicos.

O acesso a uma seqüência de caracteres na linguagem C se dá através de um **ponteiro de arquivo**, cujo tipo é `FILE *`, declarado em `stdio.h`. Algumas seqüências de caracteres são representadas por ponteiros de arquivos com nomes padronizados, como veremos a seguir. Podemos ainda declarar ponteiros de arquivos conforme nossas necessidades, como fazemos abaixo:

```
FILE *pt1, *pt2;
```

Veremos mais sobre arquivos na linguagem C e as funções de suporte associadas a eles a partir da seção 17.3. Em particular, veremos arquivos do sistema com nomes padronizados na seção 17.3.5.

A biblioteca representada pelo arquivo-cabeçalho `stdio.h` suporta dois tipos de arquivos: texto e binário. Os bytes em um **arquivo-texto** representam caracteres, fazendo que seja possível examinar e editar o seu conteúdo. O arquivo-fonte de um programa na linguagem C, por exemplo, é armazenado em um arquivo-texto. Por outro lado, os bytes em um **arquivo-binário** não representam necessariamente caracteres. Grupos de bytes podem representar outros tipos de dados tais como inteiros e números com ponto flutuante. Um programa executável, por exemplo, é armazenado em um arquivo-binário.

Arquivos-texto possuem duas principais características que os diferem dos arquivos-binários: são divididos em linhas e podem conter um marcador especial de fim de arquivo. Cada linha do arquivo-texto normalmente termina com um ou dois caracteres especiais, dependendo do sistema operacional.

17.2 Redirecionamento de entrada e saída

Como já fizemos nos trabalhos da disciplina e em algumas aulas, a leitura e escrita em arquivos podem ser facilmente executadas nos sistemas operacionais em geral. Como percebemos, nenhum comando especial teve de ser adicionado aos nossos programas na linguagem C para que a leitura e a escrita fossem executadas de/para arquivos. O que fizemos até aqui foi redirecionar a entrada e/ou a saída de dados do programa. Como um exemplo simples, vejamos o programa 17.1, onde um número inteiro na base decimal é fornecido como entrada e na saída é apresentado o mesmo número na base binária.

Programa 17.1: Conversão de um número inteiro na base decimal para base binária.

```
#include <stdio.h>

int main(void)
{
   int pot10, numdec, numbin;

   scanf("%d", &numdec);
   pot10 = 1;
   numbin = 0;
   while (numdec > 0) {
      numbin = numbin + (numdec % 2) * pot10;
      numdec = numdec / 2;
      pot10 = pot10 * 10;
   }
   printf("%d\n", numbin);

   return 0;
}
```

Supondo que o programa 17.1 tenha sido armazenado no arquivo-fonte `decbin.c` e o programa executável equivalente tenha sido criado após a compilação com o nome `decbin`, então, se queremos que a saída do programa executável `decbin` seja armazenada no arquivo `resultado`, podemos digitar em uma linha de terminal o seguinte:

```
prompt$ ./decbin > resultado
```

Esse comando instrui o sistema operacional a executar o programa `decbin` redirecionando sua saída, que normalmente seria apresentada no terminal, para um arquivo com nome `resultado`. Dessa forma, qualquer informação a ser apresentada por uma função de saída, como `printf`, não será mostrada no terminal, mas será escrita no arquivo `resultado`, conforme especificado na linha de comandos do terminal.

Por outro lado, podemos redirecionar a entrada de um programa executável, de tal forma que chamadas a funções que realizam entrada de dados, não mais solicitem essas informações ao usuário a partir do terminal, mas as obtenha a partir de um arquivo. Por exemplo, o programa 17.1 usa a função `scanf` para ler um número inteiro a partir de um terminal. Podemos redirecionar a entrada do programa `decbin` quando está sendo executado, fazendo que essa entrada seja realizada a partir de um arquivo. Por exemplo, se temos um arquivo com nome `numero` que contém um número inteiro, podemos digitar o seguinte em uma linha de terminal:

```
prompt$ ./decbin < numero
```

Com esse redirecionamento, o programa 17.1, que solicita um número a ser informado pelo usuário, não espera até que um número seja digitado. Ao contrário, pelo redirecionamento, a entrada do programa é tomada do arquivo `numero`. Ou seja, a chamada à função `scanf` tem o efeito de ler um valor do arquivo `numero` e não do terminal, embora a função `scanf` não "saiba" disso.

Podemos redirecionar a entrada e a saída de um programa simultaneamente da seguinte maneira:

```
prompt$ ./decbin < numero > resultado
```

Observe então que esse comando faz com que o programa `decbin` seja executado tomando a entrada de dados a partir do arquivo `numero` e escrevendo a saída de dados no arquivo `resultado`.

O redirecionamento de entrada e saída é uma ferramenta útil, já que, podemos manter um arquivo de entradas e realizar diversos testes sobre um programa executável a partir desse arquivo de entradas. Além disso, se temos, por exemplo, um arquivo alvo que contém soluções correspondentes às entradas, podemos comparar esse arquivo de soluções com as saídas de nosso programa, que também podem ser armazenadas em um arquivo diferente. Utilitários para comparação de conteúdos de arquivos disponíveis no sistema operacional Linux, tais como `diff` e `cmp`, são usados para atingir esse objetivo. Por exemplo,

```
prompt$ diff resultado solucao
```

17.3 Funções de entrada e saída da linguagem C

Até esta aula, excluindo o redirecionamento de entrada e saída que vimos na seção 17.2, tínhamos sempre armazenado quaisquer informações na memória principal do nosso sistema computacional. Como já mencionado, uma grande quantidade de problemas pode ser resolvida com as operações de entrada e saída que conhecemos e com o redirecionamento de entrada e saída que acabamos de aprender na seção 17.2. Entretanto, existem problemas onde há necessidade de obter, ou armazenar, dados de/para dois ou mais arquivos. Os arquivos são mantidos em um dispositivo do sistema computacional conhecido como memória secundária, que pode ser implementado como um disco rígido, um disquete, um disco compacto (CD), um disco versátil digital (DVD), um cartão de memória, um disco removível (*USB flash memory*), entre outros. A linguagem C tem um conjunto de funções específicas para tratamento de arquivos que se localiza na biblioteca padrão de entrada e saída, cujo arquivo-cabeçalho é `stdio.h`. Na verdade, essa biblioteca contém todas as funções que fazem tratamento de qualquer tipo de entrada e saída de um programa, tanto da memória principal quanto secundária. Como exemplo, as funções `printf`, `scanf`, `putchar` e `getchar`, que conhecemos bem, encontram-se nessa biblioteca.

17.3.1 Funções de abertura e fechamento

Para que se possa realizar qualquer operação sobre um arquivo é necessário, antes de tudo, de **abrir** esse arquivo. Como um programa pode ter de trabalhar com diversos arquivos, então todos eles deverão estar abertos durante a execução desse programa. Dessa forma, há necessidade de identificação de cada arquivo, o que é implementado na linguagem C com o uso de um **ponteiro para arquivo**.

A função `fopen` da biblioteca padrão de entrada e saída da linguagem C é uma função que realiza a abertura de um arquivo no sistema. A função recebe como parâmetros duas cadeias de caracteres: a primeira é o identificador/nome do arquivo a ser aberto e a segunda determina o modo no qual o arquivo será aberto. O nome do arquivo deve constar no sistema de arquivos do sistema operacional. A função `fopen` devolve um ponteiro único para o arquivo que pode ser usado para identificar esse arquivo a partir desse ponto do programa. Esse ponteiro é posicionado no início ou no final do arquivo, dependendo do modo como o arquivo foi aberto. Se o arquivo não puder ser aberto por algum motivo, a função devolve o ponteiro com valor `NULL`. Um ponteiro para um arquivo deve ser declarado com um tipo pré-definido `FILE`, também incluso no arquivo-cabeçalho `stdio.h`. A interface da função `fopen` é então descrita da seguinte forma:

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

As opções para a cadeia de caracteres `modo`, parâmetro da função `fopen`, são descritas na tabela a seguir:

modo	Descrição
`r`	modo de leitura de texto
`w`	modo de escrita de texto†
`a`	modo de adicionar texto‡
`r+`	modo de leitura e escrita de texto
`w+`	modo de leitura e escrita de texto†
`a+`	modo de leitura e escrita de texto‡
`rb`	modo de leitura em binário
`wb`	modo de escrita em binário†
`ab`	modo de adicionar em binário‡
`r+b` ou `rb+`	modo de leitura e escrita em binário
`w+b` ou `wb+`	modo de leitura e escrita em binário†
`a+b` ou `ab+`	modo de leitura e escrita em binário‡

onde:

† trunca o arquivo existente com tamanho 0 ou cria novo arquivo;

‡ abre ou cria o arquivo e posiciona o ponteiro no final do arquivo.

Algumas observações sobre os modos de abertura de arquivos se fazem necessárias. Primeiro, observe que se o arquivo não existe e é aberto com o modo de leitura (`r`) então a abertura falha. Também, se o arquivo é aberto com o modo de adicionar (`a`), então todas as operações de escrita ocorrem o final do arquivo, desconsiderando a posição atual do ponteiro do arquivo. Por fim, se o arquivo é aberto no modo de atualização (`+`) então a operação de escrita não pode ser imediatamente seguida pela operação de leitura, e vice-versa, a menos que uma operação de reposicionamento do ponteiro do arquivo seja executada, tal como uma chamada a qualquer uma das funções `fseek` , `fsetpos` , `rewind` ou `fflush` .

Por exemplo, o comando de atribuição a seguir

```
ptarq = fopen("entrada", "r");
```

tem o efeito de abrir um arquivo com nome `entrada` no modo de leitura. A chamada à função `fopen` devolve um identificador para o arquivo aberto que é atribuído ao ponteiro `ptarq` do tipo `FILE` . Então, esse ponteiro é posicionado no primeiro caracter do arquivo. A declaração prévia do ponteiro `ptarq` deve ser feita da seguinte forma:

```
FILE *ptarq;
```

A função `fclose` faz o oposto que a função `fopen` faz, ou seja, informa o sistema que o programador não necessita mais usar o arquivo. Quando um arquivo é fechado, o sistema realiza algumas tarefas importantes, especialmente a escrita de quaisquer dados que o sistema possa ter mantido na memória principal para o arquivo na memória secundária, e então dissocia o identificador do arquivo. Depois de fechado, não podemos realizar tarefas de leitura ou escrita no arquivo, a menos que seja reaberto. A função `fclose` tem a seguinte interface:

```
int fclose(FILE *ptarq)
```

Se a operação de fechamento do arquivo apontado por `ptarq` obtém sucesso, a função `fclose` devolve o valor 0 (zero). Caso contrário, o valor `EOF` é devolvido.

17.3.2 Funções de entrada e saída

A função `fgetc` da biblioteca padrão de entrada e saída da linguagem C permite que um único caracter seja lido de um arquivo. A interface dessa função é apresentada a seguir:

```
int fgetc(FILE *ptarq)
```

A função `fgetc` lê o próximo caracter do arquivo apontado por `ptarq`, avançando esse ponteiro em uma posição. Se a leitura é realizada com sucesso, o caracter lido é devolvido pela função. Note, no entanto, que a função, ao invés de especificar o valor de devolução como sendo do tipo `unsigned char`, especifica-o como sendo do tipo `int`. Isso se deve ao fato de que a leitura pode falhar e, nesse caso, o valor devolvido é o valor armazenado na constante simbólica `EOF`, definida no arquivo-cabeçalho `stdio.h`. O valor correspondente à constante simbólica `EOF` é obviamente um valor diferente do valor de qualquer caracter e, portanto, um valor negativo. Do mesmo modo, se o fim do arquivo é encontrado, a função `fgetc` também devolve `EOF`.

A função `fputc` da biblioteca padrão de entrada e saída da linguagem C permite que um único caracter seja escrito em um arquivo. A interface dessa função é apresentada a seguir:

```
int fputc(int caracter, FILE *ptarq)
```

Se a função `fputc` tem sucesso, o ponteiro `ptarq` é incrementado e o caracter escrito é devolvido. Caso contrário, isto é, se ocorre um erro, o valor `EOF` é devolvido.

Existe outro par de funções de leitura e escrita em arquivos com identificadores `fscanf` e `fprintf`. As interfaces dessas funções são apresentadas a seguir:

```
int fscanf(FILE *ptarq, char *formato, ...)
```

e

```
int fprintf(FILE *ptarq, char *formato, ...)
```

Essas duas funções são semelhantes às respectivas funções `scanf` e `printf` que conhecemos bem, a menos de um parâmetro a mais que é informado, justamente o primeiro, que é

o ponteiro para o arquivo que ser quer realizar as operações de entrada e saída formatadas. Dessa forma, o exemplo de chamada a seguir

```
fprintf(ptarq, "O número %d é primo\n", numero);
```

realiza a escrita no arquivo apontado por `ptarq` da mensagem entre aspas duplas, substituindo o valor numérico correspondente armazenado na variável `numero`. O número de caracteres escritos no arquivo é devolvido pela função `fprintf`. Se um erro ocorrer, então o valor `-1` é devolvido.

Do mesmo modo,

```
fscanf(ptarq, "%d", &numero);
```

realiza a leitura de um valor que será armazenado na variável `numero` a partir de um arquivo identificado pelo ponteiro `ptarq`. Se a leitura for realizada com sucesso, o número de valores lidos pela função `fscanf` é devolvido. Caso contrário, isto é, se houver falha na leitura, o valor `EOF` é devolvido.

Há outras funções para entrada e saída de dados a partir de arquivos, como as funções `fread` e `fwrite`, que não serão cobertas nesta aula. O leitor interessado deve procurar as referências bibliográficas do curso.

17.3.3 Funções de controle

Existem diversas funções de controle que dão suporte a operações sobre os arquivos. Dentre as mais usadas, listamos uma função que descarrega o espaço de armazenamento temporário da memória principal para a memória secundária e as funções que tratam do posicionamento do ponteiro do arquivo.

A função `fflush` faz a descarga de qualquer informação associada ao arquivo que esteja armazenada na memória principal para o dispositivo de memória secundária associado. A função `fflush` tem a seguinte interface:

```
int fflush(FILE *ptarq)
```

onde `ptarq` é o ponteiro para um arquivo. Se o ponteiro `ptarq` contém um valor nulo, então todos espaços de armazenamento temporários na memória principal de todos os arquivos abertos são descarregados nos dispositivos de memória secundária. Se a descarga é realizada com sucesso, a função devolve o valor 0 (zero). Caso contrário, a função devolve o valor `EOF`.

Sobre as funções que tratam do posicionamento do ponteiro de um arquivo, existe uma função específica da biblioteca padrão da linguagem C que realiza um teste de final de arquivo. Essa função tem identificador `feof` e a seguinte interface:

```
int feof(FILE *ptarq)
```

O argumento da função `feof` é um ponteiro para um arquivo do tipo `FILE`. A função devolve um valor inteiro diferente de 0 (zero) se o ponteiro `ptarq` está posicionado no final do arquivo. Caso contrário, a função devolve o valor 0 (zero).

A função `fgetpos` determina a posição atual do ponteiro do arquivo e tem a seguinte interface:

```
int fgetpos(FILE *ptarq, fpos_t *pos)
```

onde `ptarq` é o ponteiro associado a um arquivo e `pos` é uma variável que, após a execução dessa função, conterá o valor da posição atual do ponteiro do arquivo. Observe que `fpos_t` é um novo tipo de dado, definido no arquivo-cabeçalho `stdio.h`, adequado para armazenamento de uma posição qualquer de um arquivo. Se a obtenção dessa posição for realizada com sucesso, a função devolve 0 (zero). Caso contrário, a função devolve um valor diferente de 0 (zero).

A função `fsetpos` posiciona o ponteiro de um arquivo em alguma posição escolhida e tem a seguinte interface:

```
int fsetpos(FILE *ptarq, fpos_t *pos)
```

onde `ptarq` é o ponteiro para um arquivo e `pos` é uma variável que contém a posição para onde o ponteiro do arquivo será deslocada. Se a determinação dessa posição for realizada com sucesso, a função devolve 0 (zero). Caso contrário, a função devolve um valor diferente de 0 (zero).

A função `ftell` determina a posição atual de um ponteiro em um dado arquivo. Sua interface é apresentada a seguir:

```
long int ftell(FILE *ptarq)
```

A função `ftell` devolve a posição atual no arquivo apontado por `ptarq`. Se o arquivo é binário, então o valor é o número de *bytes* a partir do início do arquivo. Se o arquivo é de texto, então esse valor pode ser usado pela função `fseek`, como veremos a seguir. Se há sucesso na sua execução, a função devolve a posição atual no arquivo. Caso contrário, a função devolve o valor `-1L`.

A função `fseek` posiciona o ponteiro de um arquivo para uma posição determinada por um deslocamento. A interface da função é dada a seguir:

```
int fseek(FILE *ptarq, long int desloca, int a_partir)
```

O argumento `ptarq` é o ponteiro para um arquivo. O argumento `desloca` é o número de *bytes* a serem saltados a partir do conteúdo do argumento `apartir`. Esse conteúdo pode ser um dos seguintes valores pré-definidos no arquivo-cabeçalho `stdio.h`:

`SEEK_SET`	A partir do início do arquivo
`SEEK_CUR`	A partir da posição atual
`SEEK_END`	A partir do fim do arquivo

Em um arquivo de texto, o contéudo de `apartir` dever ser `SEEK_SET` e o conteúdo de `desloca` deve ser 0 (zero) ou um valor devolvido pela função `ftell`. Se a função é executada com sucesso, o valor 0 (zero) é devolvido. Caso contrário, um valor diferente de 0 (zero) é devolvido.

A função `rewind` faz com que o ponteiro de um arquivo seja posicionado para o início desse arquivo. A interface dessa função é a seguinte:

```
void rewind(FILE *ptarq)
```

17.3.4 Funções sobre arquivos

Há funções na linguagem C que permitem que um programador remova um arquivo do disco ou troque o nome de um arquivo. A função `remove` tem a seguinte interface:

```
int remove(char *nome)
```

A função `remove` elimina um arquivo, com nome armazenado na cadeia de caracteres `nome`, do sistema de arquivos do sistema computacional. O arquivo não deve estar aberto no programa. Se a remoção é realizada, a função devolve o valor 0 (zero). Caso contrário, um valor diferente de 0 (zero) é devolvido.

A função `rename` tem a seguinte interface:

```
int rename(char *antigo, char *novo)
```

A função `rename` faz com que o arquivo com nome armazenado na cadeia de caracteres `antigo` tenha seu nome trocado pelo nome armazenado na cadeia de caracteres `novo`. Se a função realiza a tarefa de troca de nome, então `rename` devolve o valor 0 (zero). Caso contrário, um valor diferente de 0 (zero) é devolvido e o arquivo ainda pode ser identificado por seu nome antigo.

17.3.5 Arquivos do sistema

Sempre que um programa na linguagem C é executado, três arquivos ou seqüências de caracteres são automaticamente abertos pelo sistema, identificados pelos ponteiros `stdin`,

`stdout` e `stderr`, definidos no arquivo-cabeçalho `stdio.h` da biblioteca padrão de entrada e saída. Em geral, `stdin` está associado ao teclado e `stdout` e `stderr` estão associados ao monitor. Esses ponteiros são todos do tipo `FILE`.

O ponteiro `stdin` identifica a entrada padrão do programa e é normalmente associado ao teclado. Todas as funções de entrada definidas na linguagem C que executam entrada de dados e não têm um ponteiro do tipo `FILE` como um argumento tomam a entrada a partir do arquivo apontado por `stdin`. Assim, ambas as chamadas a seguir:

```
scanf("%d", &numero);
```

e

```
fscanf(stdin, "%d", &numero);
```

são equivalentes e lêem um número do tipo inteiro da entrada padrão, que é normalmente o terminal.

Do mesmo modo, o ponteiro `stdout` se refere à saída padrão, que também é associada ao terminal. Assim, as chamadas a seguir:

```
printf("Programar é bacana!\n");
```

e

```
fprintf(stdout, "Programa é bacana!\n");
```

são equivalentes e imprimem a mensagem acima entre as aspas duplas na saída padrão, que é normalmente o terminal.

O ponteiro `stderr` se refere ao arquivo padrão de erro, onde muitas das mensagens de erro produzidas pelo sistema são armazenadas e também é normalmente associado ao terminal. Uma justificativa para existência de tal arquivo é, por exemplo, quando as saídas todas do programa são direcionadas para um arquivo. Assim, as saídas do programa são escritas em um arquivo e as mensagens de erro são escritas na saída padrão, isto é, no terminal. Ainda há a possibilidade de escrever nossas próprias mensagens de erro no arquivo apontado por `stderr`.

17.4 Exemplos

Nessa seção apresentaremos dois exemplos que usam algumas das funções de entrada e saída em arquivos que aprendemos nesta aula.

O primeiro exemplo, apresentado no programa 17.2, é bem simples e realiza a cópia do conteúdo de um arquivo para outro arquivo.

Programa 17.2: Um exemplo de cópia de um arquivo.

```
#include <stdio.h>

#define MAX 100

int main(void)
{
   char nome_base[MAX+1], nome_copia[MAX+1];
   int c;
   FILE *ptbase, *ptcopia;

   printf("Informe o nome do arquivo a ser copiado: ");
   scanf("%s", nome_base);

   printf("Informe o nome do arquivo resultante: ");
   scanf("%s", nome_copia);

   ptbase = fopen(nome_base, "r");
   if (ptbase != NULL) {
      ptcopia = fopen(nome_copia, "w");
      if (ptcopia != NULL) {
         c = fgetc(ptbase);
         while (c != EOF) {
            fputc(c, ptcopia);
            c = fgetc(ptbase);
         }
         fclose(ptbase);
         fclose(ptcopia);
         printf("Arquivo copiado\n");
      }
      else {
         printf("Impossível abrir o arquivo %s para escrita\n", nome_copia);
      }
   }
   else {
      printf("Impossível abrir o arquivo %s para leitura\n", nome_base);
   }

   return 0;
}
```

O segundo exemplo, apresentado no programa 17.3, mescla o conteúdo de dois arquivos texto em um arquivo resultante. Neste exemplo, cada par de palavras do arquivo resultante é composto por uma palavra de um arquivo e uma palavra do outro arquivo. Se um dos arquivos contiver menos palavras que o outro arquivo, esse outro arquivo é descarregado no arquivo resultante.

Programa 17.3: Um segundo exemplo de uso de arquivos.

```
#include <stdio.h>

#define MAX 100

int main(void)
{
   char nome1[MAX+1], nome2[MAX+1], palavra1[MAX+1], palavra2[MAX+1];
   int i1, i2;
   FILE *pt1, *pt2, *ptr;

   printf("Informe o nome do primeiro arquivo: ");
   scanf("%s", nome1);
   printf("Informe o nome do segundo arquivo: ");
   scanf("%s", nome2);
   pt1 = fopen(nome1, "r");
   pt2 = fopen(nome2, "r");
   if (pt1 != NULL && pt2 != NULL) {
      ptr = fopen("mesclado", "w");
      if (ptr != NULL) {
         i1 = fscanf(pt1, "%s", palavra1);
         i2 = fscanf(pt2, "%s", palavra2);
         while (i1 != EOF && i2 != EOF) {
            fprintf(ptr, "%s %s ", palavra1, palavra2);
            i1 = fscanf(pt1, "%s", palavra1);
            i2 = fscanf(pt2, "%s", palavra2);
         }
         while (i1 != EOF) {
            fprintf(ptr, "%s ", palavra1);
            i1 = fscanf(pt1, "%s", palavra1);
         }
         while (i2 != EOF) {
            fprintf(ptr, "%s ", palavra2);
            i2 = fscanf(pt2, "%s", palavra2);
         }
         fprintf(ptr, "\n");
         fclose(pt1);
         fclose(pt2);
         fclose(ptr);
         printf("Arquivo mesclado\n");
      }
      else
         printf("Não é possível abrir um arquivo para escrita\n");
   }
   else
      printf("Não é possível abrir os arquivos para leitura\n");

   return 0;
}
```

Exercícios

17.1 Escreva um programa que leia o conteúdo de um arquivo cujo nome é fornecido pelo usuário e copie seu conteúdo em um outro arquivo, trocando todas as letras minúsculas por letras maiúsculas.

17.2 Suponha que temos dois arquivos cujas linhas são ordenadas lexicograficamente. Por exemplo, esses arquivos podem conter nomes de pessoas, linha a linha, em ordem alfabética. Escreva um programa que leia o conteúdo desses dois arquivos, cujos nomes são fornecidos pelo usuário, e crie um novo arquivo resultante contendo todas as linhas dos dois arquivos ordenadas lexicograficamente. Por exemplo, se os arquivos contêm as linhas

Arquivo 1	**Arquivo 2**
Antônio	Carlos
Berenice	Célia
Diana	Fábio
Solange	Henrique
Sônia	
Zuleica	

o arquivo resultante deve ser

Antônio
Berenice
Carlos
Célia
Diana
Fábio
Henrique
Solange
Sônia
Zuleica