Ficha 7 – Ficheiros – API Buffered I/O

Tópicos abordados:

– Ficheiros

– Exercícios

Duração prevista: 1,5 aulas

©2019: {vitor.carreira, patricio, mfrade, loureiro, nfonseca, rui, nuno.costa, leonel.santos, luis.correia, miguel.negrao}@ipleiria.pt

1 Standard I/O - API Buffered I/O ("Alto nível")

Tal como analisado na ficha que abordou o acesso a ficheiros através da API de baixo nível, para efetuar leituras ou escritas mais eficientes ou de informação textual é recomendado recorrer à API de alto nível que está orientada à manipulação de *streams* de *bytes* e utiliza ponteiros do tipo FILE. Esta API será apresentada nas próximas secções.

1.1 Tipo de dados FILE

Na API de alto nível, um ficheiro é representado pela estrutura FILE. Esta estrutura armazena todos os dados necessários para trabalhar com um ficheiro. A declaração da variável do tipo ficheiro é efetuada do seguinte modo:

```
FILE *nome_da_variável;
```

Exemplo:

```
#include <stdio.h>
...
FILE *fptr = NULL;
...
```

As operações efetuadas sobre ficheiros obedecem à seguinte sequência:

- Abertura;
- Operações de leitura / escrita;
- Fecho;

1.1 Abertura

A linguagem C distingue dois tipos de ficheiros:

- Ficheiros de texto – leitura / escrita formatada (requer API de alto nível)
- Ficheiros binários – leitura / escrita de baixo nível (não requer a API de alto nível, embora possa ser empregue a API de alto nível)

Antes de efetuar qualquer operação de escrita / leitura é necessário proceder à abertura de um ficheiro. A abertura de um ficheiro consiste em associar uma estrutura FILE ao ficheiro pretendido, indicando o nome do ficheiro e o tratamento pretendido (modo de abertura). Para tal, utiliza-se a função ***fopen*** definida na biblioteca ***stdio.h***.

```
#include <stdio.h>
FILE *fopen(char *filename, char *mode);
```

Esta função abre o ficheiro *filename* utilizando o modo de abertura especificado pelo parâmetro *mode*. Caso se verifique um erro (o ficheiro não existe, não tem permissões, etc.…), a função devolve a macro NULL.

Em C, o acesso aos dados de um ficheiro é efetuado de forma sequencial. A estrutura FILE possui um conjunto de campos, entre os quais se inclui um apontador que indica a próxima posição de leitura/escrita. Sempre que se faz uma leitura ou escrita, o apontador é atualizado de acordo com o número de *bytes* lidos ou escritos. A posição inicial do apontador é definida de acordo com o modo de abertura. Os modos de abertura possíveis para um ficheiro encontram-se descritos no quadro seguinte.

Modo de abertura	**Descrição**
"r"	Abre o ficheiro unicamente para leitura e posiciona o apontador de leitura/escrita para o início do ficheiro. O ficheiro tem de existir. Apenas são autorizadas operações de leitura.
"w"	Abre ou cria um ficheiro unicamente para escrita. O ficheiro é criado se não existir. Se o ficheiro existe o seu conteúdo é apagado. Posiciona o apontador de leitura/escrita para o início do ficheiro. Apenas são autorizadas operações de escrita.
"a"	Abre o ficheiro para adição, i.e., todos os dados escritos são

	adicionados ao ficheiro. Cria o ficheiro se este não existir e posiciona o apontador de leitura/escrita para o final do ficheiro. Apenas estão autorizadas operações de escrita.
"r+"	Abre o ficheiro para escrita e leitura. O ficheiro tem de existir. Posiciona o apontador de leitura/escrita para o início do ficheiro.
"w+"	Abre o ficheiro para escrita e leitura. O ficheiro é criado se não existir. Caso exista o seu conteúdo é apagado. Posiciona apontador de leitura/escrita para o início do ficheiro.
"a+"	Abre o ficheiro para leitura e adição. Cria o ficheiro se este não existir. Posiciona o apontador de leitura/escrita para o início do ficheiro. No entanto, todas as operações de escrita ocorrem sempre no final do ficheiro.

Atualmente, num sistema Unix, não existe qualquer diferença entre ficheiros binários ou de texto na abertura de um ficheiro. Apenas as funções utilizadas para escrita ou leitura diferem (ver próximas secções). No entanto, para manter compatibilidade com código anterior à norma C89, é possível indicar que se pretende abrir um ficheiro para modo binário acrescentando um "*b"* no modo de abertura. A letra "*b*" pode ser colocada em qualquer posição após o primeiro símbolo. Sendo assim, as combinações possíveis para modos de abertura de ficheiros binários (para efeitos de retro compatibilidade) são: "rb", "wb", "ab", "r+b" ou "rb+", "w+b" ou "wb+" e "a+b" ou "ab+".

O quadro seguinte ilustra de forma resumida as propriedades dos modos de abertura:

Propriedades	r	w	a	r+	w+	a+
Ficheiro tem de existir	*			*		
Apaga toda a informação do ficheiro se este existe		*			*	
Operação de leitura	*			*	*	*
Operação de escrita na posição atual do apontador		*		*	*	
Operação de escrita sempre no fim do ficheiro			*			*

Exemplos:

```
FILE *fptr = NULL;

(....)
fptr = fopen("dados.txt", "r");
if (fptr == NULL) {
  WARNING("Não foi possível abrir o ficheiro para leitura");
}
(....)
```

Figura 1 – Exemplo função *fopen*

1.1.1 Diferenciação entre ficheiros de texto e ficheiros binários

Nos sistemas operativos **Windows** existe a distinção entre a abertura de um ficheiro em modo binário e um ficheiro em modo texto. Assim, quando se abre um ficheiro considerado de modo texto, é necessário acrescentar a letra "*t*" ao modo de abertura. Similarmente, a abertura de um ficheiro em modo binário é assinalada com o uso da letra "*b*" no modo de abertura. Por exemplo, indicar "rb" em modo de abertura assinala a abertura do ficheiro para leitura em modo binário, ao passo que o uso de "rt" assinala abertura do ficheiro para leitura em modo de texto.

Num sistema Windows, quando um ficheiro é aberto em modo de texto ("t"), existe uma conversão automática i) na escrita de '\n' para '\r\n' e ii) na leitura de '\r\n' para '\n'. Significa isso, que num sistema operativo Windows, o seguinte código:

```
fprintf(file_F, "linha\n");
```

irá resultar na escrita de '\r\n' para o ficheiro, pois é assim que é identificado uma mudança de linha num ficheiro processado em modo de texto ("t"). Ocorre o inverso quando se efetua uma leitura em modo de texto: '\r\n' é convertido para '\n'.

Em sistemas Unix, o uso dos tipos "t" e "b" não produz efeito, pois não existe distinção entre modo de texto e modo binário.

1.2 Fecho

A razão pela qual é necessário fechar um ficheiro após a sua utilização resume-se a dois motivos. O primeiro prende-se com o facto de que cada processo dispõe de um limite máximo de ficheiros (descritores) que pode manter abertos. Se um programa necessita de manipular muitos ficheiros e não os fecha após a sua utilização, o limite máximo é atingido e futuras tentativas de abrir novos ficheiros resultarão em erros. O segundo motivo tem a ver com o facto das operações de escrita e leitura serem otimizadas através da utilização de um *buffer*. No caso de uma operação de escrita, os dados não são escritos imediatamente no dispositivo de armazenamento. A escrita apenas ocorre quando o *buffer* se encontra cheio ou despejado explicitamente. Se o ficheiro não for fechado, a informação que se encontra no *buffer* ficará perdida. Para fechar um ficheiro utiliza-se a função ***fclose***.

```
#include <stdio.h>
int fclose(FILE *fptr)
```

Esta função fecha o ficheiro apontado por *fptr*. Após a chamada da função pode-se utilizar novamente o ponteiro *fptr* para outro ficheiro. Em caso de erro a função devolve **EOF**. Caso contrário, devolve 0 (zero).

Exemplo:

```
#include <stdio.h>
(...)
FILE *fptr = NULL;
fptr = fopen("dados.txt", "r");
if (fptr != NULL) {
  /* Operações de leitura */
  (...)
  fclose(fptr);
} else {
  WARNING("O ficheiro não existe ou não tem permissões de
leitura");
}
(...)
```

Figura 2 – Exemplo da função *fclose*

1.3 Fim de ficheiro (EOF)

A função ***feof*** permite determinar se se chegou ao fim do ficheiro.

```
#include <stdio.h>
int feof(FILE *fptr)
```

Esta função testa se o apontador de leitura/escrita da estrutura *fptr* se encontra posicionado no fim do ficheiro. Devolve 0 (valor lógico falso) se não está posicionado no fim do ficheiro *fptr*. Caso contrário devolve um valor diferente de zero (valor lógico verdade).

Segue-se um pequeno exemplo da aplicação desta função:

```
#include <stdio.h>
(...)
FILE *fptr = NULL;
fptr = fopen("dados.txt", "r");

if (fptr != NULL) {
  while ( !feof(fptr) ) {
    /* Codigo de leitura */
  }
  fclose(fptr);
}
(...)
```

Figura 3 – Exemplo da função *feof*

1.4 Escrita / Leitura

Em C, o acesso aos dados (leitura ou escrita) de um ficheiro é efetuado de forma sequencial. Como já foi referido, a estrutura FILE utilizada para manipular o ficheiro

contém, entre outros campos, um apontador que indica a posição da próxima leitura/escrita. Com exceção do modo "a+", após uma operação de escrita ou leitura, o apontador avança de acordo com o número de *bytes* escritos ou lidos. No caso do modo de abertura "a+", inicialmente o apontador de leitura/escrita aponta para o início do ficheiro. No entanto, após uma operação de escrita, o apontador ficará a apontar para o fim do ficheiro.

As funções de escrita e leitura em ficheiros são divididas em dois grupos: funções de escrita/leitura de texto e funções de escrita/leitura de dados binários.

1.4.1 Escrita de texto

Existem várias funções para a escrita de texto em ficheiros. As mais comuns são:

```
#include <stdio.h>

int fputc(int ch, FILE *fptr)
int fputs(const char *text, FILE *fptr)
int fprintf(FILE *fptr, const char *format, ...)
```

A função ***fputc*** escreve o caractere *ch* na posição apontada pelo apontador de leitura/escrita da estrutura *fptr*. Em caso de erro devolve EOF. Caso contrário, devolve o caractere escrito. O apontador de leitura/escrita avança um *byte*.

Exemplo:

```
#define ERR_IO 1

/* Função que utilizando fgetc e fputc efetua a cópia de um
ficheiro de texto caractere a caractere (nota: não é uma
solução otimizada) */
void copia(const char *origem, const char *destino) {
  FILE *fptr1 = NULL, *fptr2 = NULL;
  int ch;

  fptr1 = fopen(origem, "r");
  fptr2 = fopen(destino, "w");

  if (fptr1 == NULL)
    ERROR(ERR_IO, "Ficheiro %s não existe", origem);
  if (fptr2 == NULL)
    ERROR(ERR_IO, "Erro de abertura do ficheiro %s para
escrita", destino);

  while ( (ch = fgetc(fptr1)) != EOF ){
    if (fputc(ch, fptr2) == EOF) {
      ERROR(ERR_IO, "Erro de escrita");
      break;
    }
  }
  fclose(fptr1);
  fclose(fptr2);

}
```

Figura 4 – Exemplo da função *fputc*

A função ***fputs*** escreve a *string text* (exceto o terminador ´\0') na posição apontada pelo apontador de leitura/escrita da estrutura *fptr*. Esta função devolve **EOF** em caso de erro. Caso contrário, devolve um valor positivo. O apontador de leitura/escrita é incrementado em ***strlen***(*text*).

Exemplo:

```
#define MAX 100
#define ERR_IO 1

/* Função que vai acrescentando num ficheiro o que o utilizador
vai introduzindo. A função termina após a introdução da palavra
"fim" */
void teclado_para_ficheiro(char *nome_ficheiro) {

  FILE *fptr = NULL;
  char buffer[MAX], *text;

  fptr = fopen("log.txt", "a");
  if (fptr == NULL)
    ERROR(ERR_IO, "Erro ao abrir o ficheiro para acrescimo");

  text = fgets(buffer, MAX, stdin);
  while (text != NULL && strcmp(buffer, "fim\n") != 0) {
    if (fputs(buffer, fptr) == EOF) {
      WARNING("Erro de escrita\n");
      break;
    }
    text = fgets(buffer, MAX, stdin);
  }
  fclose(fptr);
}
```

Figura 5 – Exemplo da função *fputs*

Lab 1

Compile o programa da listagem anterior e execute-o. Qual a razão de ser necessário utilizar a *string* "fim\n" em vez de apenas a *string* "fim" para sair do ciclo?

A função ***fprintf*** é muito semelhante à função ***printf***. A única diferença é que a função ***fprintf*** efetua a escrita formatada a partir da posição do apontador de leitura/escrita da estrutura *fptr* enquanto a função ***printf*** efetua a escrita no terminal (*stdout*). Em caso de erro devolve um valor negativo. Caso contrário, devolve o número de carateres escritos (exceto os terminadores '\0' das *strings*). O apontador de leitura/escrita é incrementado de acordo com o número de carateres escritos.

Suponha que a seguinte estrutura se encontra definida:

```
typedef struct {
  char nome[40];
  long numero;
} aluno_t;
```

O seguinte código irá escrever num ficheiro de texto os dados dos alunos presentes na tabela dados.

```
  FILE *fptr = NULL;
  aluno_t dados[2] = {{"Pedro", 22}, {"Luisa", 20}};
  int i;

  fptr = fopen("dados.txt", "w"); /* "wt" em Windows */
  if (fptr == NULL)
    ERROR(ERR_IO, "Erro ao abrir o ficheiro para escrita");

  for (i = 0; i < 2; i++) {
    if(fprintf(fptr,"%s %ld\n", dados[i].nome,dados[i].numero)< 0){
      WARNING("Erro de escrita");
      return -1;    }
  }
  fclose(fptr);
```

Figura 6 – Exemplo da função *fprintf*

Após a execução do código anterior o ficheiro "dados.txt" fica com os seguintes dados:

```
Pedro 22
Luísa 20
```

Lab 2

Compile e execute o programa **lab2**. Experimente retirar as permissões de escrita ao ficheiro "dados.txt" (*chmod -w dados.txt*). Execute o programa novamente. O que aconteceu?

1.4.2 Leitura de texto

Existem várias funções para a leitura de ficheiros de texto. As mais comuns são:

```
#include <stdio.h>

int fgetc(FILE *fptr);
char *fgets(char *text, int max_size, FILE *fptr)
ssize_t getline(char **lineptr, size_t *n, FILE *fptr)
ssize_t getdelim(char **lineptr, size_t *n, int delim, FILE *fptr)
int fscanf(FILE *fptr, char *format, ...)
```

A função ***fgetc*** efetua a leitura de um caractere, a partir da posição do apontador de leitura/escrita da estrutura *fptr*. Em caso de erro de leitura ou fim de ficheiro a função devolve EOF (indica fim de ficheiro). Caso contrário, devolve o caractere lido. O apontador de leitura/escrita é incrementado em 1.

Exemplo:

```
/* Função que conta o número de dígitos existentes num ficheiro
```

```
@return -1 em caso de erro; número de dígitos em caso de
sucesso*/
int conta_digito(const char *nome_ficheiro) {
  FILE *fptr = NULL;
  int conta = 0, ch;

  fptr = fopen(nome_ficheiro, "r");
  if (fptr == NULL)
    ERROR(ERR_IO, "Erro ao abrir o ficheiro para escrita");

  while ( (ch=fgetc(fptr)) != EOF)
    if (isdigit(ch))
      conta++;
  fclose(fptr);

  return conta;
}
```

Figura 7 – Exemplo da função *fgetc*

A função anterior não distingue um erro de leitura do fim de ficheiro dado que devolve o mesmo valor (**EOF**) para as duas situações. Outra forma de escrever esta função de modo a distinguir as duas situações seria utilizando a função ***feof***. Neste caso o código seria o seguinte:

```
int conta_digito(char *nome_ficheiro) {
  FILE *fptr = NULL;
  fptr = fopen(nome_ficheiro, "r");
  if (fptr == NULL)
    ERROR(ERR_IO, "Erro ao abrir o ficheiro para leitura");

  int conta = 0;
  int ch;
  while ( (ch=fgetc(fptr)) != EOF)
    if (isdigit(ch))
      conta++;

  /* Verifica se está no final do ficheiro.
     Se não está é porque existe um erro de leitura */
  if (!feof(fptr)) {
    fprintf(stderr, "%s: Erro de leitura\n", nome_ficheiro);
    return -1;
  }
  fclose(fptr);
  return conta;
}
```

Figura 8 – Exemplo da função *fgetc distinguindo erro de leitura de fim de ficheiro*

Lab 3

Execute o código do laboratório 3 passando como argumento o ficheiro "dados.txt". Execute novamente o programado com o argumento “/proc/self/mem”.

A função ***fgets*** efetua a leitura, a partir da posição do apontador de leitura/escrita, de uma cadeia de carateres para a *string text* parando quando se verificar uma das seguintes condições:

- encontrou o caractere fim de linha (‘\n’);
- encontrou o caractere fim de ficheiro (EOF);
- leu *max_size –1* carateres;

Em caso de erro de leitura ou fim de ficheiro a função devolve NULL. Caso contrário, devolve o ponteiro *texto* colocando um ‘\0’ após o último caractere lido a fim de terminar corretamente a *string*. O apontador de leitura/escrita é incrementado de acordo com o número de carateres lidos.

Exemplo:

```
/* Função que mostra o conteúdo de um ficheiro texto numerando
cada linha. Assume-se cada linha tem um máximo de 100 carateres
*/
#define MAX 100
void mostra_ficheiro(char *nome_ficheiro)
{
  FILE *fptr = NULL;
  int linha = 1;
  char str[MAX];

  fptr = fopen(nome_ficheiro, "r");
  if (fptr == NULL)
    ERROR(ERR_IO, "Erro ao abrir o ficheiro para leitura");

  while (fgets(str, MAX, fptr)!= NULL) {
    printf("Linha %d: %s", linha, str);
    linha++;
  }

  if (!feof(fptr))
    printf("Erro de leitura\n");

  fclose(fptr);
}
```

Figura 9 – Exemplo da função *fgets*

A função ***getline*** efetua a leitura, a partir da posição do apontador de leitura/escrita, de uma linha inteira de um ficheiro, guardando o texto lido (incluindo o caractere fim de linha “\n”, se for encontrado algum, e um carater de terminação *'\0'*) num *buffer* armazenando o seu endereço no ponteiro ***lineptr**.[1]

[1] As funções **getline** e **getdelim** não fazem parte da biblioteca C, pelo que poderão existir situações em que, mesmo usando a opção -std=c11, no processo de compilação, o compilador não encontre a função. Uma solução para o problema é adicionar #define _GNU_SOURCE no ficheiro .c, onde as funções são invocadas.

Se, antes da chamada da função ***getline***, ***lineptr** for definido como NULL e se a ***n** for atribuído o valor de 0, a função irá alocar um *buffer* para armazenar a linha lida. Esta funcionalidade requer que o *buffer* seja libertado no programa, mesmo que a função falhe.

Em alternativa à situação anterior, poderá ser alocado ***n** *bytes* de espaço em memória para o *buffer* através da função *malloc* apontando o mesmo para ***lineptr**. Caso o tamanho alocado não seja suficiente para guardar a linha lida, a função redimensiona o *buffer*, através da função *realloc*, e atualiza o ***lineptr** e o ***n**.

Em ambos os casos, em caso de sucesso, ***lineptr** e ***n** serão sempre atualizados com o endereço de memória do *buffer* e com o tamanho em *bytes* alocado para o mesmo, respetivamente.

Se ocorrer um erro ou for atingido o fim de ficheiro (EOF) sem que tenham sido lidos quaisquer *bytes*, a função retorna -1. Caso contrário, devolve o número de carateres lidos, incluindo o '\n' e excluindo o '\0'. O apontador de leitura/escrita é incrementado de acordo com o número de carateres lidos.

Exemplo:

```
/* Programa que lê o ficheiro /etc/motd linha a linha e
apresenta a linha lida assim como o seu tamanho e número.*/
#include <stdio.h>
#include <stdlib.h>
#define ERR_IO 1
int main(void)
{
  FILE *fptr;
   char *lineptr = NULL;
   size_t n = 0;
   ssize_t res;

   fptr = fopen("/etc/motd", "r");
   if (fptr == NULL)
     ERROR(ERR_IO, "Erro ao abrir o ficheiro para leitura");

  while ((res = getline(&lineptr, &n, fptr)) != -1) {
     printf("Encontrada linha com o tamanho: %zu :\n", res);
     printf("%s", lineptr);
  }

   free(lineptr);
   fclose(fptr);
   exit(0);
}
```

Figura 10 – Exemplo da função *getline*

A função ***getdelim*** funciona de forma idêntica à função ***getline*** mas poderá ser indicado um outro delimitador de linha diferente do '\n' através do campo **delim**.

Tal como a função ***getline***, apenas será adicionado no final da string lida o caractere delimitador caso o mesmo seja encontrado antes do fim de ficheiro (EOF).

Exercício 1

Desenvolva um programa que mostre o conteúdo do ficheiro **/etc/passwd** linha a linha, numerando as mesmas aquando da sua impressão no ecrã. Pretende-se que desenvolva duas funções que efetuem a leitura do conteúdo do ficheiro linha a linha: uma que utilize a função ***fgets*** e a outra que recorra à função ***getline***.

A função ***fscanf*** é muito semelhante à função ***scanf***. A única diferença é que a função ***fscanf*** efetua a leitura formatada a partir da posição do apontador de leitura/escrita da estrutura *fptr* enquanto a função ***scanf*** efetua a leitura formatada a partir do teclado (*stdin*).

A função devolve EOF se chegar ao fim do ficheiro. Caso contrário, devolve o número de campos lidos corretamente. Sempre que um campo é lido com sucesso, o apontador de leitura/escrita avança para o próximo campo. Se o número de campos lidos for inferior ao esperado então, o formato do ficheiro não está em conformidade com a *string* de formatação *format*.

Suponha que se pretende elaborar um programa que processe um ficheiro com a seguinte informação:

```
Pedro 22 1.72
Ana  20 1.65
Luísa 20 1.66
```

E no final, o programa deve mostrar a média das idades. Uma solução possível seria a seguinte:

```
void le_ficheiro(char *nome_ficheiro) {
  FILE *fptr = NULL;
  int idade, numero_entradas;
  float altura;
  char nome[40];
  float soma;

  fptr = fopen(nome_ficheiro, "r");
  if (fptr == NULL)
    ERROR(ERR_IO, "Erro ao abrir o ficheiro para leitura");

  numero_entradas = 0;
  soma = 0;
  while (fscanf(fptr,"%s %d %f",nome, &idade, &altura) == 3) {
    soma += idade;
```

```
    numero_entradas++;
    printf("Entrada %d: Nome = %s Idade = %d Altura = %f\n",
numero_entradas, nome, idade, altura);
  }
  if (!feof(fptr)){
    WARNING("%s: Erro de leitura\n", nome_ficheiro);
    return;
  }
  else if (numero_entradas > 0)
    printf("Media das idades: %f\n", soma / numero_entradas);
  fclose(fptr);

}
```

Figura 11 – Exemplo da função *fscanf*

Lab 4

Execute o código do laboratório 4. Altere o conteúdo do ficheiro **dados.txt** e coloque um campo a mais no primeiro registo. Comente os resultados.

1.4.3 Escrita de dados binários

Suponha que no exemplo da figura 6 a tabela dados continha 100 elementos. Se pretendêssemos escrever todos os dados da tabela para um ficheiro poderíamos utilizar a função ***fprintf*** para tal. No entanto, a utilização dessa função implica um elevado desperdício de espaço em disco. Suponha ainda que à estrutura é acrescentado um campo do tipo enumerado e um campo data (estrutura). O problema agora agrava-se, pois seria necessário escrever esses dados utilizando novamente a função ***fprintf***, aumentando assim a complexidade do programa.

A escrita binária resolve este tipo de problemas. Na escrita binária pode escrever-se qualquer tipo de informação dado que apenas se vai escrever o número de *bytes* associado ao tipo de dados a serem escritos. Por exemplo, um inteiro irá ocupar sempre o mesmo número de *bytes* independentemente do seu valor (e.g. 1234 ou 123456789 irão ocupar o mesmo espaço, considerando que uma variável do tipo ***int*** usa 4 octetos). Ao contrário dos ficheiros de texto, a informação contida num ficheiro binário não é diretamente legível, isto é, não é possível visualizar a informação através de um editor de texto.

Para efetuar a escrita binária utiliza-se a função ***fwrite*** da biblioteca ***stdio.h***.

```
#include <stdio.h>
int fwrite(void *dataptr, int size, int nelements, FILE *fptr);
```

Esta função escreve, a partir da posição do apontador de leitura/escrita da estrutura *fptr*, *nelements* de tamanho *size* armazenados no endereço apontado por *dataptr*. Em caso de erro devolve um valor inferior a *nelements*. Caso contrário, devolve o número de elementos escritos. O apontador de leitura/escrita é incrementado de acordo com o número de *bytes* escritos.

```
typedef struct {
  char nome[40];
  long numero;
} aluno_t;

  (...)
  FILE *fptr;
  aluno_t grupo[20];
  (...)
  /* Exemplo da utilização da função fwrite para escrever a
tabela grupo de uma só vez. Note que o nome de uma tabela é um
ponteiro para o primeiro elemento => não é necessário utilizar
o operador & */
  if (fwrite(grupo, sizeof(aluno_t), 20, fptr) != 20)
    WARNING("Erro de escrita");
  (...)
```

Figura 12 – Exemplo da função *fwrite (1)*

Como já abordado em fichas anteriores, para determinar o tamanho ocupado em memória por um determinado tipo de dados utiliza-se o operador ***sizeof***. Este operador calcula o tamanho (em *bytes*) ocupado por qualquer tipo de dados. A sua sintaxe é a seguinte:

```
  sizeof(tipo de dados)
```

Onde o tipo de dados pode ser qualquer tipo. Exemplos:

```
sizeof(int) – devolve o espaço ocupado por um inteiro
sizeof(double) – devolve o espaço ocupado por um double
sizeof(aluno_t) – devolve o espaço ocupado por uma estrutura aluno_t
```

Exemplo:

Considere as seguintes estruturas,

```
typedef enum {informatica, informatica_saude} curso_t;

typedef struct {
  int dia, mes, ano;
} data_t;

typedef struct {
  char nome[40];
  long numero;
  data_t inscricao;
  curso_t curso;
} aluno_t;
```

O seguinte código irá escrever num ficheiro binário os dados dos alunos presentes na tabela dados, um de cada vez.

```
int main(void) {
  FILE *fptr = NULL;
  aluno_t dados[2] = {
     {"Pedro", 5672, {10, 10, 1995}, informatica},
     {"Luisa", 7823, {25, 9, 1996}, informatica_saude}
  };
  int i;
  fptr = fopen("dados.dat", "wb");
  if (fptr == NULL)
     ERROR(ERR_IO, "Erro ao abrir o ficheiro para escrita");

  for (i=0;i<2;i++){
     if (fwrite(&dados[i], sizeof(aluno_t), 1, fptr) != 1) {
     ERROR(ERR_IO, "%s: Erro de leitura\n", nome_ficheiro);    }
  }
  fclose(fptr);
}
```

Figura 13 – Exemplo da função *fwrite (2)*

O código anterior pode ser reescrito de forma a chamar a função ***fwrite*** uma única vez. Neste caso, teríamos:

```
(...)
  fptr = fopen("dados.dat", "wb");
  if (fptr == NULL)
     ERROR(ERR_IO, "Erro ao abrir o ficheiro para escrita");

  if (fwrite(dados, sizeof(aluno_t), 2, fptr) != 2){
     ERROR(ERR_IO, "%s: Erro de leitura\n", nome_ficheiro);
  }
  fclose(fptr);

(...)
```

Lab 5

Execute o código do laboratório 5. Escreva para o terminal o conteúdo do ficheiro dados.dat (*cat dados.dat*). Comente os resultados. Compare o tamanho do ficheiro *dados.txt* com o ficheiro *dados.dat*. Comente os resultados.

1.4.4 Leitura de dados binários

A leitura de dados binários processa-se através da função ***fread*** da biblioteca ***stdio.h***.

```
#include <stdio.h>
int fread(void *dataptr, int size, int nelements, FILE *fptr)
```

Esta função efetua a leitura de *nelements* de tamanho *size*, a partir da posição indicada pelo apontador de leitura/escrita da estrutura *fptr*, para o bloco de memória apontado por *dataptr*. Em caso de erro de leitura ou fim de ficheiro a função devolve 0 (zero) ou um valor inferior a *nelements*. Caso contrário, devolve o número de elementos lidos. Após a operação, o apontador de leitura/escrita é incrementado de acordo com o número de *bytes* lidos.

Exemplo:

```
typedef struct {
  char nome[40];
  long numero;
} aluno_t;

int main(int argc, char **argv) {
  FILE *fptr = NULL;
  aluno_t aluno;

  fptr = fopen("dados.dat", "rb");
  if (fptr == NULL)
    ERROR(ERR_IO, "Erro ao abrir o ficheiro para leitura");

  /* Exemplo da utilização da função fread para ler um elemento
de cada vez */
  while (fread(&aluno, sizeof(aluno_t), 1, fptr) != 1) {
    printf("\nDados lidos\n: ");
    printf("Nome: %s\nNumero: %s\n: ", aluno.nome,
aluno.numero);
  }
  if (!feof(fptr))
    WARNING("Erro de leitura");
  fclose(fptr);
}
```

Figura 14 – Exemplo da função *fread*

1.4.5 Acesso aleatório

Até ao momento os acessos a um ficheiro têm-se processado de forma sequencial, i.e., no caso da leitura o apontador de leitura/escrita é posicionado no início do ficheiro e, após cada operação de leitura, vai avançando até chegar ao final do ficheiro. Ora, existem situações em que apenas pretendemos ler ou escrever parte de um ficheiro. Para isso é necessário aceder a um ficheiro de forma aleatória. O acesso aleatório consiste em possibilitar o acesso a um determinado elemento sem ter de passar pelos elementos anteriores. A função utilizada para este fim é a função ***fseek*** que se encontra definida na biblioteca ***stdio.h***.

```
#include <stdio.h>
int fseek(FILE *fptr, long offset, int origin)
```

Esta função posiciona o apontador de leitura/escrita da estrutura *fptr* na posição *offset* relativamente à origem *origin*. Devolve 0 (zero) em caso de sucesso e um valor diferente de 0 (zero) em caso de erro. Os valores que o parâmetro *origin* pode tomar são:

`SEEK_SET` – deslocamento efetuado a partir do início do ficheiro

`SEEK_CUR` – deslocamento efetuado a partir da posição atual

`SEEK_END` – deslocamento efetuado a partir do fim do ficheiro

Alguns exemplos:

```
/* Nota: as constantes do tipo long terminam em L */
(...)
fseek(fptr, 0L, SEEK_SET);/* Vai para o início do ficheiro   */
fseek(fptr, 0L, SEEK_END);/* Vai para o fim do ficheiro */
fseek(fptr, n, SEEK_SET); /* Vai para o byte na posição n  */
fseek(fptr, n, SEEK_CUR); /* Salta n bytes a partir da posição
atual */
fseek(fptr, -n, SEEK_CUR);/* Recua n bytes a partir da posição
atual */
(...)
```

Figura 15 - Exemplo da função *fseek*

Para além da função ***fseek,*** duas funções podem ser utilizadas para manipular o apontador de leitura/escrita de um ficheiro:

```
#include <stdio.h>
void rewind(FILE *fptr);
long ftell(FILE *fptr);
```

A função ***rewind*** posiciona o apontador de escrita/leitura no início do ficheiro. É equivalente a:

```
  fseek(fptr, 0L, SEEK_SET)
```

A função ***ftell*** devolve a posição atual do apontador de leitura/escrita. Se o apontador estiver posicionado no final do ficheiro, esta função devolve o tamanho do ficheiro em *bytes*. Em caso de erro a função devolve -1 (-1L).

Por exemplo, a função seguinte calcula o tamanho de um ficheiro (**nota**: existem chamadas mais apropriadas para o efeito):

```
long tamanho_ficheiro(char *nome) {
  FILE *fptr = NULL;
  long tam;

  fptr=fopen(nome, "rb");
  if (fptr == NULL)
    ERROR(ERR_IO, "Erro ao abrir o ficheiro para leitura");
```

```
  /* Posiciona fptr no final do ficheiro */
  fseek(fptr, 0L, SEEK_END);
  tam = ftell(fptr);
  fclose(fptr);
  return tam;
}
```

Figura 16 – Exemplo da função *ftell*

Como nota final é importante referir que o acesso aleatório aplica-se usualmente a ficheiros binários dado que neste caso se sabe o tamanho ocupado por cada entrada no ficheiro. Segue-se um exemplo mais completo de como se poderia consultar uma entrada específica do ficheiro do exemplo da figura 14.

```
int main(void) {
  FILE *fptr = NULL;
  int nelem, pos;
  aluno_t aluno;

  fptr = fopen("dados.dat", "rb");
  if (fptr == NULL)
    ERROR(ERR_IO, "Erro ao abrir o ficheiro para leitura");

  /* Primeiro calcula-se o número de elementos do ficheiro
     O número de elementos é dado por:
       tamanho do ficheiro / sizeof(tamanho de cada elemento)
   */
  fseek(fptr, 0L, SEEK_END);
  nelem = ftell(fptr) / sizeof(aluno_t);

  printf("Qual a ficha que deseja ver? (0-%d)\n", nelem-1);
  scanf("%d", &pos);

  while (pos >= 0 && pos < nelem) {
    /* Posiciona o ponteiro na ficha desejada. Nota: a ficha
       nº pos está na posição pos*sizeof(aluno_t)
    */
    fseek(fptr, pos*sizeof(aluno_t), SEEK_SET);

    if (fread(&aluno, sizeof(aluno_t), 1, fptr) != 1) {
      ERROR(ERR_IO, "%s: Erro de leitura\n", nome_ficheiro);

    }
    printf("Ficha nº %d:\n", pos);
    printf("Nome: %s\nNumero: %ld\n", aluno.nome,
aluno.numero);

    /* Pede próxima ficha */
    printf("Qual a ficha que deseja ver? (0-%d)\n", nelem-1);
    scanf("%d", &pos);
  }
  fclose(fptr);
}
```

Figura 17 – Exemplo de acesso aleatório a ficheiros

Lab 6

Execute o código do laboratório 6. O que acontece se o utilizador introduzir uma letra em vez de uma posição válida?

1.4.6 *Flush* - libertação de *buffer*

Tal como já foi indicado, quer nas aulas teóricas quer no início desta ficha, as operações de leitura e escrita utilizam *buffers* definidos na área do utilizador o que implica que, por exemplo, os dados escritos para um ficheiro através das funções analisadas anteriormente apenas sejam efetivamente escritos na memória do sistema operativo (modo *system*) após o *buffer* se encontrar cheio ou ser despejado explicitamente.

Por forma a garantir/forçar o despejo de um *buffer* definido na área do utilizador para operações de escrita, utilizam-se as seguintes funções:

A função ***fflush*** que se encontra definida na biblioteca ***stdio.h***.

```
#include <stdio.h>
int fflush(FILE *stream);
```

Esta função força a escrita dos dados existentes em *buffers* de área de utilizador, através da chamada ao sistema *write()*, para *buffers* do sistema operativo na área do kernel. Como parâmetro de entrada, a função recebe o ponteiro do tipo FILE para o ficheiro. Como saída, a função devolve 0 em caso de sucesso, isto é, despejo do *buffer* feito com sucesso e na totalidade. Caso contrário, devolve **EOF** e atribui o erro à variável *errno*.

Exemplo:

```
#define MAX 100
/* Função que vai acrescentando num ficheiro o que o utilizador
vai introduzindo. A função termina após a introdução da palavra
"fim" */
void teclado_para_ficheiro(char *nome_ficheiro) {
  FILE *fptr = NULL;
  char buffer[MAX], *text;

  fptr = fopen(nome_ficheiro, "a");
  if (fptr == NULL)
    ERROR(ERR_IO, "Erro ao abrir o ficheiro para acrescimo");

  text = fgets(buffer, MAX, stdin);
  while (text != NULL && strcmp(buffer, "fim\n") != 0) {
    if (fputs(buffer, fptr) == EOF) {
      WARNING("Erro de escrita\n");
      break;
    }
      fflush(fptr);
    text = fgets(buffer, MAX, stdin);
  }
  fclose(fptr);
}
```

Figura 18 – Exemplo da função *fflush*

Mesmo utilizando a função ***fflush***, não se consegue garantir que os dados são efetivamente escritos no dispositivo de armazenamento, visto que os mesmos apenas são "transferidos" para os *buffers* do sistema operativo. Assim, para realmente garantir que os dados são efetivamente escritos no dispositivo físico, para além da função ***fflush*** ter-se-á de utilizar a função ***fsync*** que se encontra definida na biblioteca ***unistd.h***.

```
#include <unistd.h>
int fsync(int fd);
```

Esta função vai sincronizar as alterações feitas a um ficheiro, através de um programa, com o dispositivo de armazenamento. Como parâmetro de entrada, a função recebe o descritor de baixo-nível para o ficheiro (do tipo inteiro). Como saída, a função devolve 0 em caso de sucesso. Caso contrário, devolve **-1** e atribui o erro à variável *errno*.

Por forma a obter o descritor do ficheiro aberto, poder-se-á obter o mesmo através da utilização da função ***fileno*** que se encontra definida na biblioteca ***stdio.h***.

```
#include <stdio.h>
int fileno(FILE *stream);
```

Esta função devolve o descritor do ficheiro (no formato ***int***) tendo em conta o ponteiro do tipo FILE para o ficheiro passado como parâmetro de entrada. Esta função devolve **-1** caso detete que o parâmetro de entrada seja inválido.

Exemplo:

```
#define MAX 100
/* Função que vai acrescentando num ficheiro o que o utilizador
vai introduzindo. A função termina após a introdução da palavra
"fim" */
void teclado_para_ficheiro(char *nome_ficheiro) {
  FILE *fptr = NULL;
  char buffer[MAX], *text;
  int fd, ret;

  fptr = fopen(nome_ficheiro, "a");
  if (fptr == NULL)
    ERROR(ERR_IO, "Erro ao abrir o ficheiro para acrescimo");

  fd = fileno(fptr);
  text = fgets(buffer, MAX, stdin);
  while (text != NULL && strcmp(buffer, "fim\n") != 0) {
    if (fputs(buffer, fptr) == EOF) {
      WARNING("Erro de escrita\n");
      break;
    }
    fflush(fptr);
    if(fsync(fd) == -1){
      WARNING("Erro na sincronização do ficheiro\n");
    }
    text = fgets(buffer, MAX, stdin);
```

```
  }
  fclose(fptr);
}
```

Figura 19 – Exemplo da função *fsync* e *fileno*

2 Exercícios

Recorrendo ao *template* de projeto da UC, que também já incluí um *template* de *makefile*, desenvolva os seguintes exercícios e tenha em consideração que deve sempre apresentar mensagens de erro sempre que necessário, bem como validar os argumentos de linha de comando:

1. Escreva um programa que conte e imprima o número de linhas de um ficheiro de texto. O programa deve receber como único argumento, de linha de comando, o nome do ficheiro de texto a examinar.

2. Escreva um programa que recebe um ficheiro de texto, terminado em '.txt', contendo números inteiros separados pelo carater ';' e cria um novo ficheiro binário, com o mesmo nome, mas terminado em '.bin', com a sequência de números lidos em formato binário. Cada número é representado por um inteiro de 4 octetos. Note que pode visualizar o conteúdo do ficheiro binário em formato hexadecimal através do seguinte comando: `od -tx4 ficheiro.bin`

3. Escreva um programa que calcule e apresente a média, mediana e o desvio padrão de uma sequência de números inteiros lidos de um ficheiro binário. O programa deve receber como único argumento de linha de comando o nome do ficheiro binário a examinar.

4. Escreva um programa que calcule o total faturado por cada método de pagamento tendo em conta os dados existentes no ficheiro .CSV que se encontra disponível para *download* na página da UC no Moodle.