1. Conceitos gerais

1.1 Leitura de ficheiros com stdlib de C

A leitura de ficheiros em C consiste essencialmente em três passos:

1. Abertura do ficheiro
   - `fopen()`
   - É necessário especificar o filename e modo de abertura do ficheiro
   - Se a operação é feita com sucesso, é returnado um file pointer, `FILE*`
   - Em caso de erro, é returnado `NULL`
2. Leitura do ficheiro até chegar ao fim
   - Diferentes abordagens (`fgetc()` , `fread()` , `fscanf()` , `fgets()` , ...)
   - Questão chave: como saber quando que chegamos ao final do ficheiro?
     - `feof()`
     - A própria função usada para leitura pode ter um retorno especial
       - E.g. `fgetc()` retorna `EOF` (End Of File)
3. Fechar o ficheiro
   - `fclose`
   - Particularmente importante para fazer *flush* dos dados que estão buffered e ainda não foram efetivamente escritos em disco

Estratégia usada para a leitura de ficheiros depende do tipo de processamento feito, performance vs complexidade do código, entre outros.

De forma análoga, para operações de escrita temos abordagens equivalentes: `fputc()` , `fwrite()` , `fprintf()` , `fputs()` , ...

1.2 Estratégias para leitura de ficheiros

1.2.1 Leitura carater a carater

Resumo: leitura em loop até chegar ao final do ficheiro; a cada iteração obtem-se o carater seguinte do ficheiro

- Abordagem simples
- Abordagem limitada porque apenas temos informação de um carater de cada vez; pode ser incoveniente se precisarmos de mais informação para o processamento que estamos a fazer (e.g. palavra completa)
  - Workaround: guardar os carateres num buffer auxiliar, mas existem formas melhores de atingir este fim (e.g. `fread()`)
- Overhead: chamadas sucessivas à função `fgetc` (uma invocação por carater + `EOF`)

```c
/* read file char by char */
char *c;
while ((c = fgetc(f)) != EOF) {
    printf("%c", c);
}
```

Exemplo completo em `f3/demos/char_by_char.c`.

1.2.2 Leitura do ficheiro inteiro

Carregar todo o ficheiro para a memória, resultando numa string com o texto completo

- Não é recomendável para ficheiros grandes, consumo elevado de memória
- Raramente é justificável ser preciso o ficheiro completo

Em termos de implementação, é necessário:

- Determinar tamanho do ficheiro; não existe uma forma direta no standard de C, mas é possível usando o `fseek()` e `ftell()`
  - Com o `fseek()` avançamos para o fim do ficheiro na *stream*
  - O `ftell()` retorna a posição atual no ficheiro, em bytes; Como estamos no fim do ficheiro, então é equivalente ao tamanho do ficheiro em bytes
  - NOTA: Não esquecer de mover para o inicio do ficheiro antes de começar a ler, usando novamente o `fseek()`
- Ler um bloco de bytes que corresponde ao tamanho do ficheiro; para tal, usar `fread()`

```c
/* determine file size in bytes */
fseek(f, 0, SEEK_END);              // move to end
filesize = ftell(f);                // get current position in bytes
fseek(f, 0, SEEK_SET);              // restore, go back to beginning
/* allocate memory */
char *buf = malloc(filesize + 1);   // extra byte for '\0'
/* read file */
fread(buf, sizeof(char), filesize, f);
buf[filesize] = '\0';               // insert '\0'
```

Exemplo completo em `f3/demos/load_full_file.c`.

1.2.3 Leitura em blocos

Definir um buffer auxiliar de tamanho fixo, `N`, e ler o ficheiro em loop em blocos de `N` bytes. É uma generalização da abordagem carater a carater com o `fgetc()`

- Mais eficiente
  - reduz o número de chamadas a funções
  - gasto de memória é constante (O(1)) e baixo (valores tipicos de `N` são 256, 512, 1024, ...)
- Mais informação em memória; poderá ser importante para alguns tipos de processamento de texto

```c
/* read file by blocks */
char buf[MAX_SIZE];
size_t num_bytes;
while ((num_bytes = fread(buf, sizeof(char), MAX_SIZE - 1, f)) > 0) {
    buf[num_bytes] = '\0';  // insert the null-terminating null
    printf("%s", buf);      // print the string
}
```

Cuidados a ter:

- O `fread()`, e análogo `fwrite()`, são genéricos; i.e., as funções não sabem o que vocês estão a ler/escrever; vocês dizem precisamente quantos bytes a ler, ou quantos bytes a escrever
- O `fread()` não sabe como interpretar os bytes que lê. É uma string? É um pixel de uma imagem?
- Isto implica que **o buffer onde o `fread()` escreve o texto nunca vai conter um `\0`, logo não é uma string válida em C**.
- Solução:
  - Se o buffer tem tamanho `MAX`, no `fread` apenas pedir `MAX-1` bytes/carateres, reservando espaço para um `\0`
  - O `fread` retorna o número de elementos/carateres lido
  - Com base no no valor de retorno, escrever o `\0` manualmente

Exemplo completo em `f3/demos/blocks.c`.

1.2.4 Leitura em blocos/linhas

Leitura em blocos, mas em vez de usar o `fread()`, usar o `fgets()`; poderá ser mais indicado para ficheiros de texto.

Principais diferenças para o `fread()`:

- Insere o `'\0'`
- Permite leitura por linhas, porque retorna quando chega a `N` carateres (tamanho do bloco), ou quando encontra um `\n`
- Não retorna o número de bytes/carateres lidos; No entanto, o conteúdo no buffer é uma string válida e portanto poderiam usar o `strlen()` para ir buscar o tamanho

```c
/* read file by blocks */
char buf[MAX_SIZE];
while (fgets(buf, MAX_SIZE, f) != NULL) {
    printf("%s", buf); // print the string
}
```

Exemplo completo em `f3/demos/string_blocks.c`.

2. Exercicios

Q1

a)

- Basta usar o `tolower()`, que converte um `char` para minúscula

b) e c)

- Usar a função `strstr` para encontrar substrings
- Retorna `NULL` se não encontrar nada, ou retorna um apontador para o inicio da primeira ocorrência

- Para encontrar mais que uma ocorrência é necessário "avançar" a string, de forma a que a segunda chamada à função comece a pesquisar depois da primeira ocorrência

Exemplo:

- String: `"ndjasndABCdmasksdmaABCdkmsad"`
- Substring a pesquisar: `"ABC"`

A primeira chamada a `strstr` retorna um apontador para o primeiro `A` `"ndjasnd[A]BCdmasksdmaABCdkmsad"`.

- Se voltarmos a chamar `strstr` com o apontador original, o resultado é o mesmo. É necessário que na segunda chamada se mova o apontador de forma a que `strstr` só analise a partir do marcador `|>`, i.e. `"ndjasndA|>BCdmasksdmaABCdkmsad"`. - Ou então, podemos saltar toda a ocorrência de substring, avançando todo o `"ABC"` encontrado: `"ndjasndABC|>dmasksdmaABCdkmsad"`

Q2

Implementar leitura em blocos, como explicado acima.

Q3

- Validar os argumentos
- Leitura do ficheiro em blocos
- Fazer as transformações no texto, para cada bloco, usando `tolower()` e `toupper()`
- Imprimir de imediato o bloco devidamente transformado

Q4

- Tip: vejam o `man fopen` e as várias opções disponíveis para as flags. Existem certas flags que permitem criar o ficheiro caso não exista

Q5

Este exercicio é mais fácil fazendo processamento carater a carater. No entanto, nada vos impede de ler o ficheiro em blocos, que é a estratégia mais eficiente ☺

Só para clarificar o conceito de carater, palavra e linha:

- palavra: eu considerei uma sequência de carateres alfanuméricos e com hífens (e.g., "pombo-correio"); qualquer carater que seja diferente destes marca o fim de uma palavra
- linhas: apenas fiz contagem do carater `\n` (line feed); no entanto, em Windows é frequente ser usada uma sequência diferente de carateres para uma mudança de linha, `\r\n`;
    - keep it simple!

Recursos:

- The Carriage Return and Line Feed Characters

Q6

Work in progress ☕

3. Food for thought

Dentro da pasta `q1` :

- Inspeciona os ficheiros de texto `1a_iso.txt` e `1a_utf.txt`. Visivelmente parecem iguais e ambos contêm 25 carateres.
- Executa os seguintes comandos:

```
gcc 1a.c -o mylower
./mylower "$(cat 1a_iso.txt)"
./mylower "$(cat 1a_utf.txt)"
```

Os comandos executam o programa `mylower` e é passado o conteúdo dos dos ficheiros `1a_iso.txt` e `1a_utf.txt` como argumento.

O output deve ter este aspeto:

```
➜  q1 ./mylower "$(cat 1a_utf.txt)"
DEBUG: string length = 30
isto É um teste çom Ç à À
➜  q1 ./mylower "$(cat 1a_iso.txt)"
DEBUG: string length = 25
isto  um teste om  ?
```

- No primeiro exemplo, o output tem tamanho 30, embora apenas existam 25 carateres no ficheiro?! Além disso, a string não é convertida devidamente para as minúsculas
- No segundo exemplo, o tamanho é 25 (como esperado), mas aparecem carateres estranhos no terminal
- Nota: Dependendo do encoding do vosso terminal, as observações podem ser o inverso; de qualquer forma, em nenhum dos casos os carateres são todos convertidos para minúscula como esperado

A explicação para isto tem haver com o encoding usado para guardar o texto dos ficheiros. Provavelmente já ouviste falar de ASCII (https://www.asciitable.com/), mas existem muitos outros standards. Relativamente aos ficheiros exemplo:

- o `1a_utf.txt` usa `UTF-8` e cada carater pode variar entre 1 a 4 bytes. Esta é a justificação para o tamanho de 30 bytes.
- o `1a_iso.txt` usa uma extensão da tabela de ASCII, pelo que cada carater é 1 byte
- Tip: Podes usar o comando `file -i <filename>` para determinar o encoding de um ficheiro

```
➜  q1 file -i 1a_utf.txt
1a_utf.txt: text/plain; charset=utf-8
➜  q1 file -i 1a_iso.txt
1a_iso.txt: text/plain; charset=iso-8859-1
```

> *Eventually, ISO released [...] ISO 8859 describing its own set of eight-bit ASCII extensions. The most popular is **ISO 8859-1**, also called ISO Latin 1, which contained characters sufficient for the **most common Western European languages**. Variations were standardized for other*

> *languages as well: ISO 8859-2 for Eastern European languages and ISO 8859-5 for Cyrillic languages, for example.*
>
> *Source: https://en.wikipedia.org/wiki/Extended_ASCII*

Problema:

- **Existem diferentes formas de representar texto**
- Funções standard de C que dependem do idioma (e.g., converter minúsculas para maiúsculas e vice-versa), por defeito, **apenas lidam com o ASCII original de 7 bits**
  - Nesta representação, carateres alfanuméricos são suportados, assim como de pontuação e outros especiais
  - No entanto, **carateres como `Ç`, ou `Á` não são suportados**!
  - Como tal, os carateres especiais não são reconhecidos e não são convertidos.
- Encodings em que cada carater usa 2 ou mais bytes são problemáticos...
  - Repara que a função `tolower`, por exemplo, apenas recebe um único byte; portanto poderá não receber toda a informação relativamente a um carater
  - Seria preciso utilizar bibliotecas ou fazer uma implementação manual

Ok, processamento de texto tem vários desafios. E agora?

Existe algum suporte em C para lidar com certos encodings de 1 byte e respetivos idiomas configurando o `locale`, (e.g. `setlocale(LC_ALL, "pt_PT.iso88591")`). No entanto, isto apenas funciona assumindo que o encoding apenas usa um byte por carater, como é o caso de ASCII. Para UTF-8, em que um carater pode usar 2 ou mais bytes, não há suporte nativo, não existe uma resposta objetiva. Depende do fim que se pretende atingir. A melhor aposta, é provavelmente usar bibliotecas que já existem!

Para o contexto de Sistemas Operativos, não se preocupem com estas questões 😄 A minha sugestão é evitarem usar carateres especiais para os inputs dos vossos programas.

Recursos extra:

- Extended ASCII
- UTF-8 strings in C