Para esta ficha só podem usar funções da biblioteca standard de C! Se conheces uma função e não tens a certeza se é standard, usa o `man`!

Por exemplo, se fizeres `man 3 printf` e desceres a página até a secção **CONFORMING TO** (praticamente no fundo), vais encontrar o seguinte:

> fprintf(), printf(), sprintf(), vprintf(), vfprintf(), vsprintf(): POSIX.1-2001, POSIX.1-2008, C89, C99.

O manual indica que as funções são conforme os standards POSIX, mas também C89 e C99. Logo, a função está standardizada e podem usar.

No entanto, a função `getline` diz:

> Both getline() and getdelim() were originally GNU extensions. They were standardized in POSIX.1-2008.

Esta função não é do standard de C!

Já agora, todas as funções relevantes para esta ficha fazem parte do `stdio.h`, por isso sugiro que vejam o `man stdio` para verem um resumo das funções disponíveis.

Q1

a)

- Primeiro exercicio 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"`

Podem ver nos ficheiros de código uma possível abordagem.

Q3

Há várias formas de resolver o problema, dependendo do contexto:

- Se soubermos que os ficheiros não excedem um certo tamanho, podemos alocar um buffer com esse tamanho máximo e ler todo o ficheiro com uma única chamada ao `fread`
- Se não for possível assumir isso:
  - Ineficiente (`3_memory_monster.c`)
    a. Determinar o tamanho do ficheiro (é possível com standard de C, sem usar o `stat` que é uma system call de Unix).
    b. Alocar buffer desse tamanho com malloc.
    c. Ler todo o ficheiro com uma única chamada a `fread`
    d. Imprimir
    e. Libertar memória
  - Recomendado (`3_chunks.c`)
    - Ler o ficheiro em blocos de tamanho fixo (e.g. 64, 128, 256, 512 bytes são tamanhos comuns. valor ótimo depende do hardware e kernel)
    - Usar `feof()` para saber se chegamos ao final do ficheiro
      - Alternative é usar o retorno do `fread`. Imaginem que pedem para ler `N` elementos, mas o `fread` leu menos que isso. Então significa que chegou ao final do ficheiro. No entanto também pode ser um erro... e por isso deviamos complementar a verificação utilizando o `ferror()`.
      - Minha opinião, usar o `feof()` é mais simples e legível
    - Imprimir de imediato o conteúdo sempre que é lido um novo bloco, evitando assim ocupar memória com todo o conteúdo do ficheiro

Cuidados a ter:

- Quando usam o `fread`, o conteúdo do vosso buffer não tem `\0`, mesmo que chegue ao fim do ficheiro. **Para que o vosso buffer seja uma string válida de C, é necessário meter o `\0` no fim explicitamente**
  - Se o vosso buffer tem tamanho `MAX`, no `fread` apenas peçam `MAX-1` elementos,

reservando espaço para um `\0`
    - Tip: O `fread` retorna o número de elementos/carateres lido, portanto é fácil saber onde meter o `\0`
- Usem `valgrind` para ter a certeza que a vossa implementação está bem
    - Usem ficheiros com conteúdo que excede o tamanho do vosso buffer

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)"
```

Basicamente, está a ser executado o primeiro programa, mas é passado os conteúdos dos ficheiros `1a_iso.txt` e `1a_utf.txt`.

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?? E a string não é convertida devidamente
- No segundo exemplo, o tamanho é 25 como esperado, mas aparecem carateres esquisitos no terminal
- (também pode ser ao contrário, dependendo do encoding do vosso terminal. de qualquer forma, em nenhum dos casos os carateres são todos convertidos para minúscula)

A ideia a reter é que existem diferentes formas de representar texto. Em C tipicamente falamos da tabela de ASCII (https://www.asciitable.com/). O standard original usava 7 bits, suportando 128 carateres. Nesta representação, carateres alfanuméricos são suportados, assim como de pontuação e outros especiais. **Mas carateres como `ç`, ou `Á` não são!** Entretanto surgiram extensões que usam os 8 bits, suportando assim 256 carateres. Diferentes extensões existem

para suportar diferentes linguagens ou conjuntos de carateres visíveis.

Funções como `tolower` apenas suportam o ASCII original de 7 bits, e por isso os carateres especiais não são reconhecidos e como tal não são convertidos.

- Tenham cuidado quando lidam com ficheiros e strings. Se quiserem evitar dores de cabeça, não usem carateres especiais, a.k.a, usem ASCII (7-bits).

Sobre os ficheiros de example: o `1a_iso.txt` usa uma extensão de ASCII, pelo que cada carater é 1 byte. Já o `1a_utf.txt` usa `UTF-8` e cada carater pode variar entre 1 a 4 bytes. Consequentemente, o tamanho de 30.

- Podes usar o comando `file -i

```
➜  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
```

Existe algum suporte limitado em C para lidar com encodings, configurando o `locale`, (e.g. `setlocale(LC_ALL, "pt_PT.iso88591")`). No entanto, para UTF-8, em que um carater pode exigir processar mais que 1 byte, não há suporte nativo e seria preciso utilizar bibliotecas ou fazer uma implementação manual para tal.

Sobre ASCII-Extended, seria relativamente simples fazer as conversões:

Carater	Código minuscula	Código maiúscula
À (grave)	224	192
Á (agudo)	225	193
Â (circunflexo)	226	194
Ã (til)	227	195

Os carateres do mesmo tipo estão agrupados, e portanto o mapeamento é direto usando algumas contas básicas. Por exemplo, converter os 'A's acentuados para minúsculas:

```
Seja 'c' o carater (char)
Seja 'delta' a distância na tabela entre o 'À' e 'à'

Se 'c' >= 192 && 'c' <= 195
Então 'c' = 'c' + delta
```