Binario vs. Texto en escritura/lectura de ficheros

Binario vs. Texto

- Imaginemos que queremos guardar en el disco duro el número ***425*** en formato ***short int***, mediante sentencias del lenguaje C.
- Hay una diferencia de almacenar ese dato en formato **binario** o **texto**.
- Cuando se almacenan datos de una sola vez en un ordenador de más de 1 byte, estos se pueden escribir:
  - **big-endian:** 0x01A9 se almacena en memoria (disco duro por ejemplo) como {01, A9}. Típico de Motorola.
  - **little-endian:** 0x01A9 se almacena en memoria (disco duro por ejemplo) como {A9, 01}. Típico de Intel.
  - Hay arquitecturas que pueden trabajar con ambos enfoques: middle-endian. Típico de ARM y PowerPc.
- Así en binario se guardan dos bytes a la vez (fwrite).
- En texto internamente byte a byte (fprint).

```c
#include <stdio.h>

main(){

FILE *f1, *f2;
short int n=425;
/* Apertura del fichero original, para lectura en binario*/
f1 = fopen ("bin.dat", "wb");
if (f1==NULL)
{
   perror("No se puede abrir bin.dat");
   return -1;
}

/* Apertura del fichero de destino, para lectura*/
f2 = fopen ("text.dat", "w");
if (f2==NULL)
{
   perror("No se puede abrir text.dat");
   return -1;
}

/* Escritura en binario */
fwrite(&n,sizeof(n), 1, f1);

/* Escritura en texto */
fprintf(f2,"%d",n);

/* Cierre de ficheros */
fclose(f1);
fclose(f2);
}
```

Binario vs. Texto

- Compilo y ejecuto: $ gcc bin-text.c -o bin-text; ./bin-text
- Muestro los bytes de los ficheros bin.dat y text.dat

```
$ hexdump  -C bin.dat
00000000  a9 01                                        |..| (los puntos son caracteres no imprimibles)
00000002
$ hexdump  -C text.dat
00000000  34 32 35                                     |425|
00000003
```

- Escritura en Binario (**fwrite(&n,sizeof(n), 1, f1)**): como lo he ejecutado en un Intel es **little-endian** y el valor hexadecimal A901 almacenado el disco duro representa el número hexadecimal 01A9 que en binario es 0000 0001 1010 1001. Este número en binario es precisamente la representación en complemento a dos del número *short int* n=425. Así $425_{10} \rightarrow 01_{16}\ A9_{16} \rightarrow 0000000110101001_2$ (en representación complemento a 2)

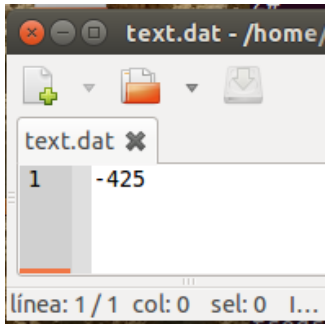
- Escritura en Texto (**fprintf(f2,"%d",n)**): generalmente el modo texto ocupa más, e implica la conversión en RAM de cada dígito del número a ASCII. $425_{10} \rightarrow$ “425” $\rightarrow$ ‘4’ ‘2’ ’5’ $\rightarrow$ 52 50 53 (códigos ASCII) $\rightarrow$ $34_{16}\ 32_{16}\ 35_{16} \rightarrow 00110100_2$ $00110010_2\ 00110101_2$ (binario puro, cuando hay un signo menos se codifica con su código ASCII, 45 (-), es decir 2D en hexadcimal).

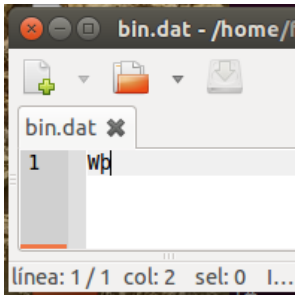
Binario vs. Texto

- Supongamos ahora: **short int n = -425;** (1111 1110 0101 0111 en C-A2, 0xFE57)
- Compilo y ejecuto: $ gcc bin-text.c -o bin-text; ./bin-text
- Muestro los bytes de los ficheros bin.dat y text.dat

```
$ hexdump  -C bin.dat
00000000  57 fe                                          |W.|  (el punto es un carácter no imprimible)

00000002
$ hexdump  -C text.dat
00000000  2d 34 32 35                                    |-425|
00000003
```





- Así la escritura en texto es legible para un editor de texto. La escritura en binario no tiene porque ser legible para un editor de texto.
- La lectura (como es lógico) debe ser coherente con la escritura: binario ↔ binario, texto ↔ texto (es decir `fwrite` ↔ `fread`, `fprintf` ↔ `fscanf`)