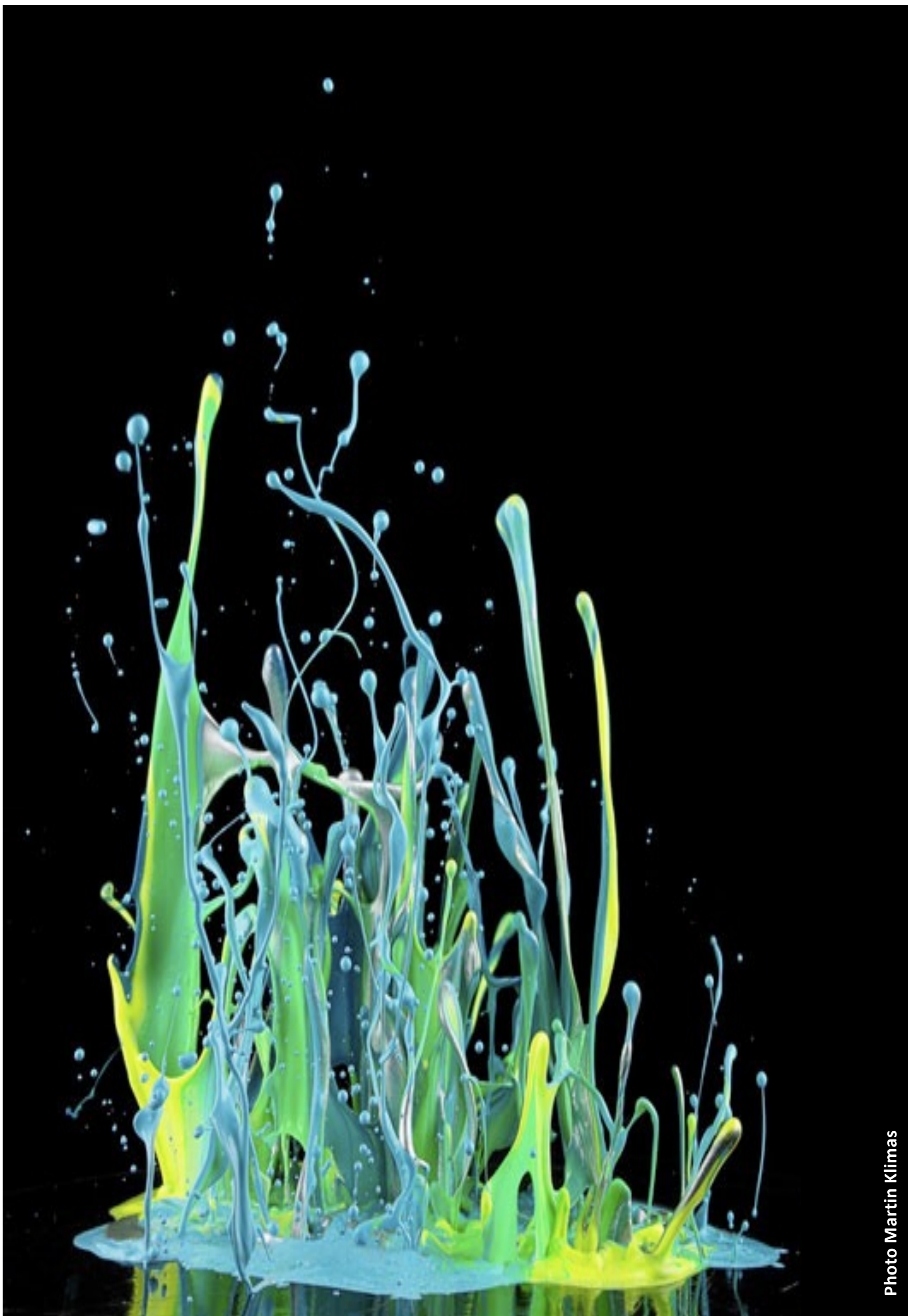


EPFL

Programmation pour Ingénieur

Fichiers

ME 3e semestre

Rev. 2025.r2

Christophe Salzmann

Laboratoire d'Automatique

modifications

- 21.08.2025, r1 – initial
- 24.08.2025, r2 – add recap

Récap - représentation des nombres

0100 0010 0110 1111 0110 1110 0110 1010
0110 1111 0111 0101 0111 0010 0010 0001

B o n

^ ou ö F6, ASCII > 127 platform specific !

Récap - représentation des nombres

```
0100  0010  0110  1111  0110  1110  0110  1010
0110  1111  0111  0101  0111  0010  0010  0001
```

1114599018_{10} 1869967905_{10} $= 6F757221_{16}$

$59.857826232910156 2_{10}$

Recap - sélection/extraction

Ex. comment extraire le jour/mois/année d'un nombre représentant une date en suivant une convention donnée?

Comment extraire le mois de **18092025**

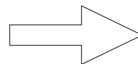
7 6 5 4 3 2 1 0   positions

Par convention les 2 chiffres représentant le mois se trouvent en position 4 et 5

18092025

Masquer pour sélectionner position 4 et 5

00090000



Décaler de 4 positions,

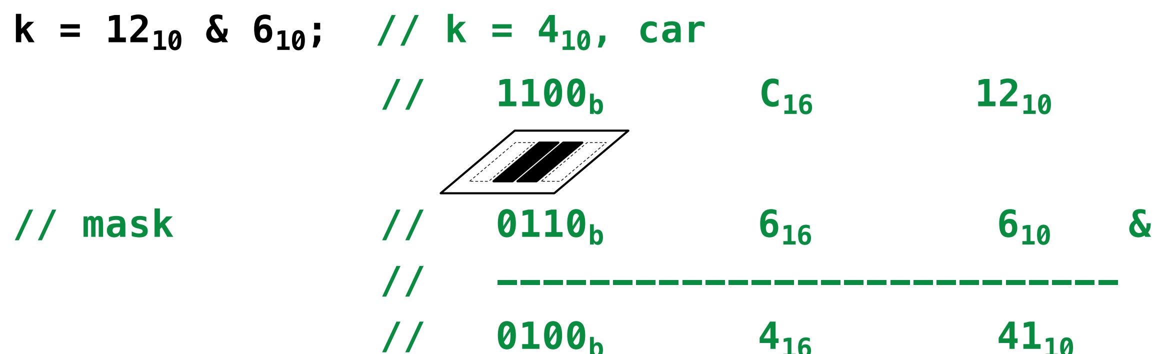
Note: décalage d'une position -> division par 10

09

09 -> septembre

Récap - Masking with &

- L'opérateur **&** effectue l'opération AND **bit à bit**



- La représentation hexadécimale **`0xF`** (ou **`1111`**$_b$) permet de facilement sélectionner 4 bits à la fois, idem pour **`0xFF`** (8 bits) qui permet de sélectionner un **`byte`** ou **`char`** ou octet.
- Ex.

```
0x00125523   (unsigned int 32 bits)
0x000000FF
----------
0x00000023
```

Fonctionne de la même manière pour :

| | *Bitwise OR*
^ *Bitwise exclusive OR*
~ *Bitwise complement*

Récap - Shifting with >>

- L'opérateur >> décale les bits vers la droite (**<<** vers la gauche)

```
c = 1100b >> 1; // c = 0110b décale de 1 bit vers la droite
c = 1100b >> 2; // c = 0011b décale de 2 bits vers la droite
```

Attention aux nombres négatifs! Le bit de signe peut (ou non) être étendu, cela dépend de l'implémentation ☹

- Ex.

```
0x00125523  (unsigned int 32 bits)
0x00FF0000
----------
0x00120000
```

Il faut maintenant décaler 0x00120000 de 16 bit vers la droite

```
0x00120000 >> 16 (bits = 4 pos hexa)
----------
  0x00000012
```

- En c

```
unsigned int pixel = 0x00125523;
unsigned int Red   = (pixel & 0x00FF0000)>> 16;
                                          // Δ précédence des opérations
```

Récap - fichier

- Les données sur le "disque" sont des **1** et des **0**
- Ces **1** et **0** peuvent être interprétés comme du **texte** ou du **binaire**

```
// demo Hex-Text comparison¬// C
hS 24.09.2025¬// O¬#include <std
io.h>¬int main(int argc, const c
har * argv[]) {¬    printf("Hell
o, World!\n");¬    return 0;¬}¬
```

```
2F 2F 20 64 65 6D 6F 20 48 65 78 2D 54 65 78 74 20 63 6F 6D 70 61 72 69 73 6F 6E 0A 2F 2F 20 43
68 53 20 32 34 2E 30 39 2E 32 30 32 35 0A 2F 2F 20 4F 0A 23 69 6E 63 6C 75 64 65 20 3C 73 74 64
69 6F 2E 68 3E 0A 69 6E 74 20 6D 61 69 6E 28 69 6E 74 20 61 72 67 63 2C 20 63 6F 6E 73 74 20 63
68 61 72 20 2A 20 61 72 67 76 5B 5D 29 20 7B 0A 20 20 20 20 70 72 69 6E 74 66 28 22 48 65 6C 6C
6F 2C 20 57 6F 72 6C 64 21 5C 6E 22 29 3B 0A 20 20 20 20 72 65 74 75 72 6E 20 30 3B 0A 7D 0A
```

- Vous demandez à l'OS l'accès au fichier, il décide

- Il y toujours 3 étapes: **ouverture - accès(r/w) – fermeture**
- **Vous devez tester les réponses de l'OS/gestionnaire de fichier**

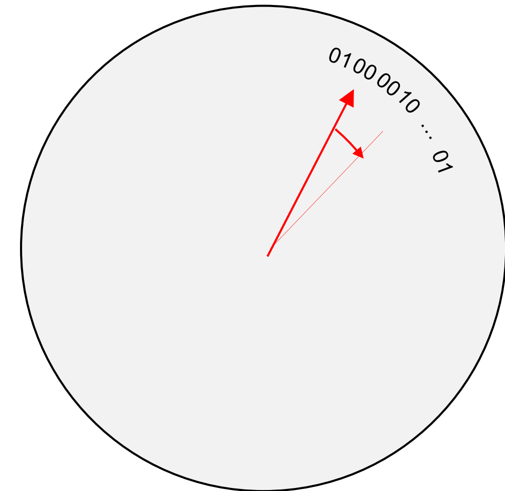
- L'OS/gestionnaire de fichier mémorise la position courante de lecture ou d'écriture

Today

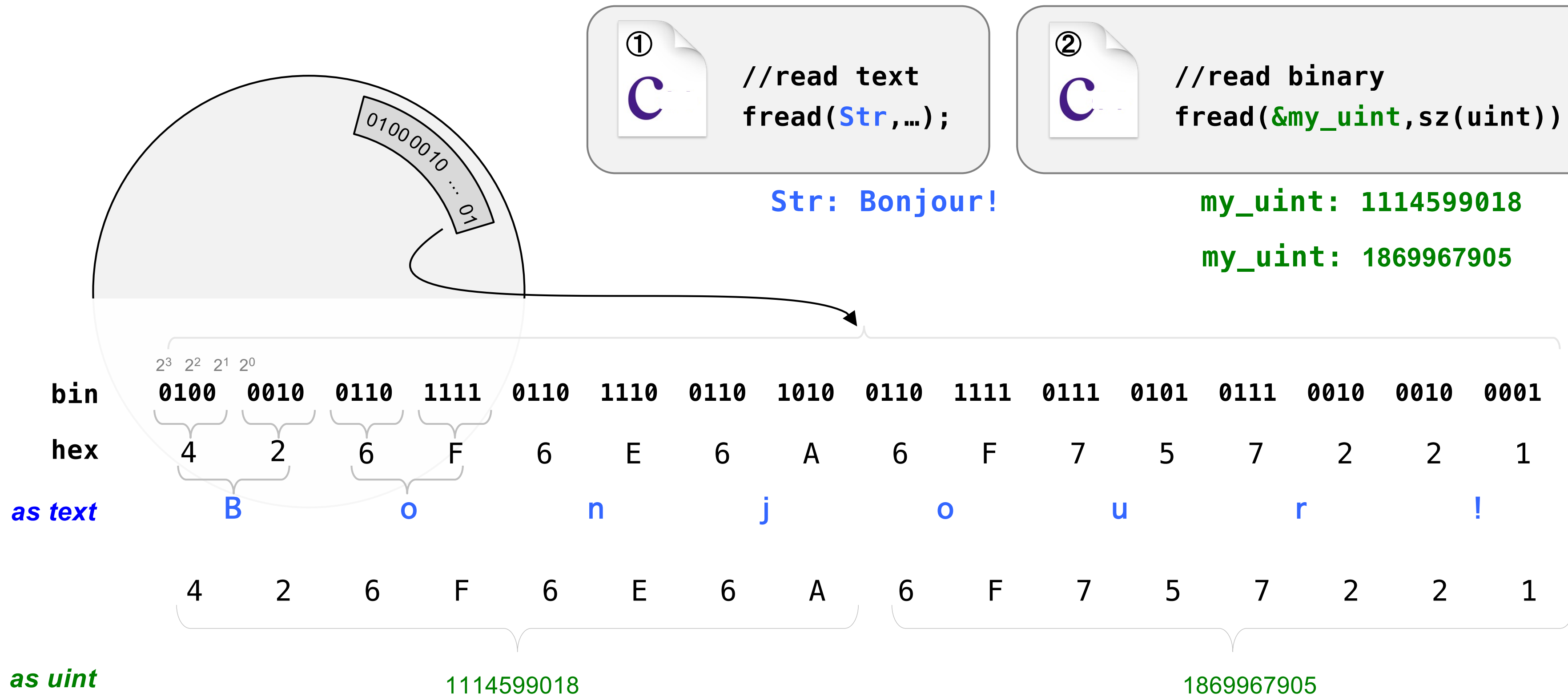
- Fichiers
- 3 étapes de base
- Fichiers binaires / textes
- Gestion des erreurs, `feof()`, etc.
- Attention: caractère fin de ligne
- Attention: Endianess

Fichiers

- Les fichiers sont stockés sur un support (disque dur, SSD, USB-Key, serveur, etc.)
- Vous accédez aux fichiers au travers de l'OS qui gère (en autre) les droits d'accès (r/w/x) aux fichiers, vous n'avez **pas** *de facto* accès à un fichier, l'OS décide.
- L'accès aux fichiers n'est pas instantané, bien que l'OS fasse le maximum pour diminuer la latence.
- Il est possible d'avoir plusieurs accès concurrents en lecture sur un fichier, par contre il devrait n'y avoir qu'un seul accès en écriture.
- Afin d'accélérer l'accès aux fichiers, l'OS peut le mettre en mémoire (cache) et l'écriture dans un fichier en cache se fera quand l'OS "à le temps"
- D'autres processus/programmes peuvent agir sur vos fichiers avant que vous y accédiez, ex: encryption, antis-virus, etc.
- Les données sont stockées sous la forme de '`1`' et '`0`' qu'il faut interpréter correctement, par exemple en regardant l'extension du nom du fichier, ex. `'.txt' -> char, '.bin' -> usigned int`
- A chaque opération de lecture/écriture un pointeur indiquant la position courante est mis à jour



Fichiers – from '0'/'1' to text/number



Conversion bin➔hex: $b_3\,2^3 + b_2\,2^2 + b_1\,2^1 + b_0\,2^0$ ex: **1 1 1 0 -> 8 + 4 + 2 + 0 = 14_{10} = E_{16}** (with 2^3 2^2 2^1 2^0 above the bits)

Interprétation hex as char: using ASCII table (convention), ex: …, 41_{16} ➔ **A**, 42_{16} ➔ **B**, …, 6E ➔ **n**, 6F ➔ **o** , …

Conversion hex_{16}➔num_{10}: $h_7\,16^7 + h_6\,16^6 + \ldots + h_0\,16^0$ ex: 426F6E6A ➔ 1073741824 + 33554432 + … + 10 = **1114599018_{10}**

Fichiers

En C, l'accès aux fichiers se trouvant sur votre disque dur est similaire aux accès aux flots d'entrée/sortie standard **`stdin`** et **`stdout`**. Il y a cependant une différence de taille, ces flots doivent être créés, puis liés avec le fichier pour pouvoir accéder aux données. Une fois les opérations terminées il faut fermer le flot.

Les 3 étapes pour accéder à un fichier:

ouverture/création: demande à l'OS l'accès à un fichier donné

```
fopen();
```

lecture/écriture: demande à l'OS de lire/écrite les *n* prochains bytes

```
fread()/fwrite(); // Il existe d'autres fonctions
```

fermeture: informe l'OS que nous en avons fini avec le fichier

```
fclose();
```

`stdin`, **`stdout`** et **`stderr`** sont 3 flots prédéfinis par l'OS,

`stdin` est lié au clavier et **`stdout`** et **`stderr`** sont liés à la "console",

ex.

```
fprintf(stdout, "hello world");  // affiche "hello world"
```

Fichiers – accès lecture/ecriture

`<stdio.h>` contient plus de 40 fonctions relatives aux fichiers.
Ci-dessous les plus importantes pour la lecture et l'ecriture de fichiers textes et binaires.

Fichier texte	Lecture	Ecriture
1 char	`fgetc()`	`fputc()`
1 ligne	`fgets()`	`fputs()`
Formatted	`fscanf()`	`fprintf()`

Certains caractères peuvent être intepretés

Fichier binaire	Lecture	Ecriture
n char	`fread()`	`fwrite()`

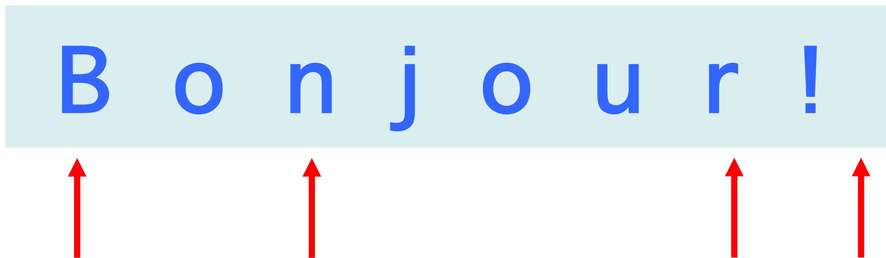
Pas d'interprétation

Sous windows pour accéder aux fonctions fichier standard, ajouter au début de votre code:

```
#pragma warning(disable : 4996)
```

Fichiers – pointeur lecture/écriture

Lors de l'ouverture d'un fichier, un pointeur interne mémorise la position courante dans le fichier. Chaque fois qu'une lecture ou une écriture est faite, le pointeur est mis à jour en fonction du nombre de caractères/bytes lus/écrits.
Note: La fonction `fread()` retourne le nombre de caractères lus.




```
FILE *fp = fopen();

n= fread(..., 2, fp);   n=2

n= fread(..., 4, fp);   n=4

n= fread(..., 3, fp);   n=2

fclose(fp);
```

`ftell()` retourne la position courante
`fseek()` permet de la modifier

Fichiers – ouverture/création

Ouverture: demande à l'OS l'accès à un fichier donné

Création: demande à l'OS de créer un fichier donné et y accéder

```
FILE *fopen(const char * filename, const char * mode);
```

`filename`: chemin absolu (au format de l'OS), ou relatif au chemin courant

`mode`: `r,rb, r+, w, w+, a, a+, …`

`FILE`: pointeur sur le descripteur de fichier (stream/flot),

(`NULL` en cas d'erreur, `errno` contient l'erreur)

`r` ouvre un fichier texte en lecture (**`rb`** binaire). Le pointeur est positionné au début du fichier.

`r+` ouvre un fichier texte en lecture et en écriture. Le pointeur est positionné au début du fichier.

`w` ouvre un fichier texte en écriture, crèe le fichier s'il n'existe pas.
met la taille du fichier à 0. Le pointeur est positionné au début du fichier.

`w+` ouvre un fichier texte en lecture et en écriture, crèe le fichier s'il n'existe pas.
met la taille du fichier à 0. Le pointeur est positionné au début du fichier.

`a` ouvre un fichier texte pour ajout (écriture à la fin), crèe le fichier s'il n'existe pas.
le pointeur est positionné à la fin du fichier.

`a+` ouvre un fichier texte en lecture et pour ajout (écriture à la fin), crèe le fichier s'il n'existe pas.
le pointeur est positionné à la fin du fichier.

Fichiers – fermeture

```
int fclose(FILE *);
```

`FILE:` pointeur sur le descripteur de fichier (stream/flot)

`int: 0` si tout est OK, (en cas d'erreur retourne `EOF`, `errno` contient l'erreur)

Ecrit le contenu encore en cache sur le "disque" et ferme le descripteur de fichier (stream/flot)

Fichiers – lecture/écriture (texte)

lit/écrit 1 caractère

```
int fgetc (FILE *fp);
               // returns an unsigned char cast to an int or EOF
int fputc (int ch, FILE *fp );
```

lit/écrit une chaine de caractères

```
char* fgets (char *str, int n, FILE *fp );
               // lit max n-1 chars, ou jusqu'au retour de ligne
               // n-1 pour laisser de la place pour \0

int fputs (const char *str, FILE *fp );
```

…

Les fonctions ci-dessus permettent de lire des caractères encodés sur 8 bit (ASCII "standard")

Il existe des fonctions similaires (ex. `fputwc()`) qui gèrent les caractères étendus (emoji, etc)

Fichiers txt – ex. fgets()

1234.txt

```
123↲
456↲
abcdefgh↲
```

```c
#include <stdio.h>
#define size 6
int main(void) {
   FILE *f_in;
   char tmp[size];
   f_in = fopen("1234.txt", "r");  // ouverture en lecture 'r'
   if(f_in == NULL){ perror("Erreur lors de l'ouverture\n"); return -1; }

   while (fgets(tmp, size, f_in) != NULL) // lit size-1 char ou jusqu'à fin de ligne
                                          //      jusqu'à la fin de fichier (NULL)
         printf("(%d) %s", strlen(tmp),tmp);

  fclose(f_in);
  return 0;
}
```

Affiche

Avec **\n**

```
(4)123↲
(4)456↲
(5)abcde(4)fgh↲
```

Avec **\r**

```
(5)123↲
4(5)56↲
ab(5)cdefg(2)h↲
```

Fichiers – lecture/écriture (texte)

```
size_t fprintf(FILE *fp, const char *fmtstr, ...)

size_t fscanf(FILE *fp, const char *fmtstr, ...)
```

`fprintf()` fonctionne comme `printf()` mais écrit dans un fichier

`fscanf()` fonctionne comme `scanf()` mais lit dans un fichier

`fp:` pointeur sur le descripteur de fichier (stream/flot)

`fmtstr:` formatting string,

`…:` liste d'arguments (variable)

`size_t :` retourne le nbr de char écrit/lu, si `<0` d'erreur et `errno` contient l'erreur

Fichiers txt – ex. `fprintf()`

Test.txt

Une ligne de test ↲
x: 1234↲

```
#include <stdio.h>

FILE *f_out;
int x = 1234;

int main(void) {
    f_out = fopen("Test.txt", "w+");  // "w+" ouverture en écriture,
              // crèe le fichier s'il n'existe pas,
              // ou vide le contenu du fichier s’il existe
    if(f_out == NULL ){
         printf("Erreur lors de l'ouverture en ecriture\n"); return -1; }
      else {
              fprintf(f_out, "Une ligne de test\n");
              fprintf(f_out, "x: %d\n", x);
              // gestion des erreurs omise
              }
    fclose(f_out);  // fermeture du fichier
    return 0;
 }
```

Une fois les opérations sur le fichier terminées, il faut impérativement **fermer** le flot à l'aide de **`fclose()`**

Fichiers – lecture/écriture (binaire)

Lit/écrit le contenu du/vers fichier sans l'interpreter

```
size_t fread(       void *buf, size_t sz, size_t n, FILE *fp);

size_t fwrite(const void *buf, size_t sz, size_t n, FILE *fp);
```

`fread()` met dans `buf`, les `n` éléments (de taille `sz`) lus dans `fp`

`fwrite()` met (écrit) les `n` éléments (de taille `sz`) de `buf` dans `fp`

- `buf:` pointeur sur le buffer à lire/écrire
- `sz:` taille d'une élément du fichier
- `n:` nombre d'éléments du fichier
- `fp:` pointeur sur le descripteur de fichier (stream/flot)
- `size_t:` si tout est OK retourne `n` (nb éléments écrits/lus), sinon c'est une erreur.
  - `fread()` ne diférencie pas si c'est une erreur ou une fin de fichier.

Fichiers bin – ex. `fread()`

```
#include <stdio.h>
#include <stdlib.h>
#define  SIZE 3
struct intcplx {int r; int i;};

int main(void) {
    struct intcplx b[SIZE];
    FILE *f2 = fopen("file.bin", "rb");   // "rb" ouverture en lecture, binaire
    if(f2 == NULL ){ perror("Erreur lors de l'ouverture en lecture\n");
                     return -1; }
    size_t r2 = fread(b, sizeof b[0], SIZE, f2); // lit l'entier du fichier dans b[]
    fclose(f2);
    printf("Demandé: %d, lus: %z",SIZE,r2); // problème si different!!
    for(size_t i = 0; i < r2; i++)         // affiche les r2 élements lus
        printf("%d+%dj ", b[i].r,b[i].i);
}
```

Fichiers bin – ex. fwrite()

```
#include <stdio.h>
#include <stdlib.h>
#define  SIZE 3
struct intcplx {int r; int i;};

int main(void) {
    struct intcplx a[SIZE]= {{1,0},{0,1},{1,1}}; // a[] contient 3 nb complexes
    FILE *f1 = fopen("file.bin", "wb");       // "wb" ouverture en écriture, binaire
    if(f1 == NULL ){                           // crèe ou efface le fichier
                perror("Erreur lors de l'ouverture en ecriture\n");
                return -1;
                }
    size_t r1 = fwrite(a, sizeof(a[0]), SIZE, f1); // écrit a[] en 1 x
    printf("Ecrit %zu elements %d demandés\n", r1,  SIZE); // problème si different!!
    fclose(f1);
    return 0;
 }
```

Fichiers – `feof()` & gestion de erreur

```
int feof (FILE *fp);
```

`feof()` teste si la fin de fichier a été atteinte ($\neq 0$ atteinte, 0 non-atteinte)

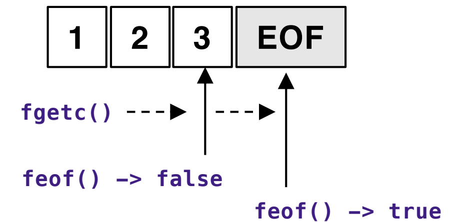
```
int ferror (FILE *fp);
```

`ferror()` teste s'il y a une erreur lors de l'utilisation du fichier (lecture, écritue, etc)

```
int perror (const char *s);
```

`perror()` affiche `s` et un texte decrivant l'erreur courante `errno`

Fichier – Fin de fichier




```
#include <stdio.h>

int main() {
   FILE *f_in;
   int c;

   f_in = fopen("123.txt", "r");  // "123.txt" contient "123" sans retour à la ligne
   if(f_in == NULL ){
        perror("ouverture Fichier.txt");
       }
     else {
        while (!feof(f_in)){                 // feof() fin du fichier ?
            c=fgetc(f_in);                   // lit 1 char
            printf("%c", c);
         }
       fclose(f_in);       // ferme le fichier
  }
}
// affiche '123\377'   '\377' = -1 est le "caractère" EOF
// tester if (c!=EOF) pour éviter l'affichage de '\377'
```

feof() ne sait pas que la fin de fichier a été atteinte tant qu'une function n'essaye pas de lire après la fin du fichier

Caractère de fin de ligne!

- Les informaticiens étant soucieux de se générer du travail pour les 10000 prochaines années ont choisi, de manière consciente ou non, différentes conventions pour plusieurs aspects d'un programme.
- Par exemple la manière d'accéder un nombre en mémoire ou sur le disque, c.f. *endianess*.
- Un autre exemple est la valeur d'un caractère ascii (codé sur 8bit), seuls les 127 premiers caractères sont définis de manière unique, les caractère ascii de 128 à 255 changent d'une plateforme à l'autre. La standardisation à l'aide de caractères codés sur 16bits (unicode) ne s'est pas encore imposée! Il y a également plusieurs autres standardisation ex. UTF-8 …
- Parmi ces caractères le caractère définissant la fin de ligne peut poser problème!
  - **LF**: '\n', char(10), Unix and Unix-like systems (GNU/Linux, Mac OS X, FreeBSD, AIX, Xenix, etc.),
  - **CR+LF**: Microsoft Windows,, CP/M, DOS (MS-DOS, PC-DOS, etc.), …)
  - **LF+CR**: Acorn BBC and RISC OS spooled text output.
  - **CR**: '\r', char(13), Commodore 8-bit machines, Mac OS up to version 9 and OS-9

Fichiers – chemin d'accès

Le chemin d'accès au fichier (path) peut être:

- **absolu**, par ex.

  unix/OSX/linux **/Users/Ch/Desktop/myfile.txt**

  Windows ***C:\Users\Ch\Desktop\myfile.txt***

Attention au **'\'** dans le cas des chemins Windows, il peut aussi être employer pour définir un caractère spécial, comme dans l'exemple "**\tmp**" où il interprété comme '**\t**' définissant une tabulation avec comme résultat: " ⟶| **mp**". Pour éviter toutes mauvaises interprétations, doubler le **'\'**, ex. "**\\tmp**" ou "**/tmp**" quand cela est possible.

L'emplois de chemins absolus peut être problématique car le chemin peut être unique à une machine ou configuration donnée, ex. ci-dessus, si l'utilisateur **Ch** n'existe pas, il ne sera pas possible de lire/écrire dans un sous-dossier de **Ch**

- **relatif** à une position (dossier/folder) courante, par ex.

  Windows/unix/OSX/linux **myfile.txt**

  /SubDir/myfile.txt

Fichiers – chemin courant

Il peut être utile de connaître le chemin (*path*) courant qui n'est pas forcément le même endroit où se trouve l'éxécutable.

La fonction `char* getcwd(char* Buffer, size_t sz)` retourne le *path* courant (ou NULL en cas d'erreur),

Ex.

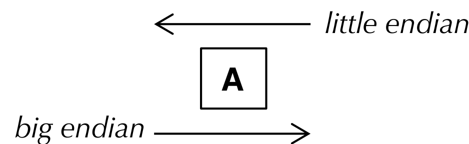
```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>          // pour getcwd()
#include <string.h>

int main(int argc, const char * argv[]) {

   char* buffer;
   if ((buffer = getcwd( NULL,0))== NULL) // _getcwd() Windows
         perror("getcwd error" );
    else {
         printf("Crt path: %s \n",buffer);
         free(buffer);
         }
   printf ("App path: %s\n",argv[0]);
   return 0;
}
```

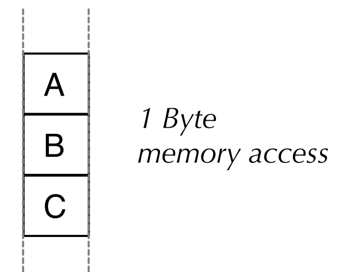
Files I/O – format & endianness

- *Endianness* has been invented by mad computer scientists to annoy freshman students - and others as well.
- It is a source of bugs, we should not have to care about such an issue…
- Endianness defines how data is ordered in memory and thus on disk.
- It is similar to writing: some languages write from left to right and others from right to left. There is no issue as long as direction is known!
- Big endian goes from left to right, the Most Significant Byte first
- **Little endian goes from right to left** (most used, x86), the Least Significant Byte first
- The smallest unit a processor reads/write is (generally) a byte/char.

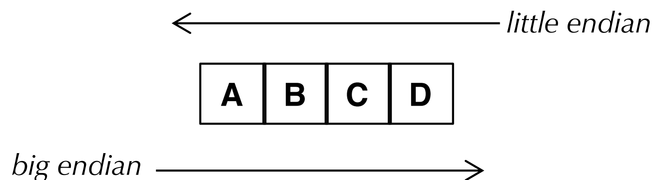


little endian: A

big endian: A

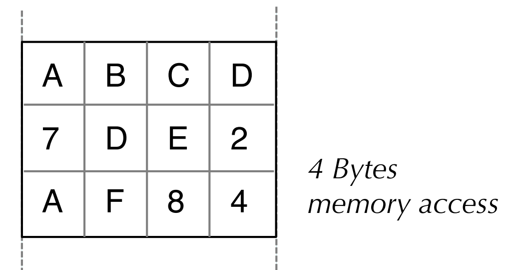


- When the processor writes a longer value, ex. an integer (4 bytes), the order of the writing matters.



little endian: DCBA

big endian: ABCD



Ex. Endianess & binary file (1/2)

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 4
uint32_t BigE[SIZE]= {0xAA000000,0x00BB0000, 0x0000CC00, 0x000000DD};

uint32_t ReverseUInt32 (uint32_t In) {
    uint32_t Out;
    ((char *)&Out)[0] = ((char*)&In)[3];  // ((char*)&in) -> see In and Out as char[]
    ((char *)&Out)[1] = ((char*)&In)[2];
    ((char *)&Out)[2] = ((char*)&In)[1];
    ((char *)&Out)[3] = ((char*)&In)[0];
    return Out;
}

int main(void) {

    FILE *f1 = fopen("file.bin", "wb");    // "wb" ouverture en écriture, binaire
    if(f1 != NULL ){                       // Δ gestion d'erreur minimum !
        fwrite(BigE, sizeof BigE[0], SIZE, f1); // écrit BigE[] en 1 x
        fclose(f1);
    }

    // ... continue on next slide
```

Ex. Endianess & binary file (2/2)

```
...
// ** some Magic on your file occurred here **
//     assume it was save as Big Endian and read as Little Endian
//     ReverseUInt32() will simulate this magic operation

uint32_t LittleE[SIZE];
FILE *f2 = fopen("file.bin", "rb");   // "rb" ouverture en lecture, binaire
if(f2 != NULL){                       // Δ gestion d'erreur minimum !
                                      // lit l'entier du fichier dans LittleE[]
    size_t n=fread(LittleE, sizeof LittleE[0], SIZE, f2);
    fclose(f2);

    for(size_t i = 0; i < n; i++)      // affiche les éléments lus en les inversant
       printf("Inv[%zu] = 0x%08X\n", i, ReverseUInt32(LittleE[i]));
    }
}
```

Affiche

```
Inv[0] = 0x000000AA
Inv[1] = 0x0000BB00
Inv[2] = 0x00CC0000
Inv[3] = 0xDD000000
```

Fichier – full error check

3 appels, 11 lignes de gestion d'erreur

```
int e=0;
uint32_t Buf[6]; //being very specific here

FILE *fp = fopen("file.bin", "rb");   // Δ file.bin contient 4 uint32!

if(fp == NULL){
    perror("Erreur ouverture 'file.bin', will exit");
    exit(EXIT_FAILURE); // ou goto fail:
    }

size_t ret = fread(Buf, sizeof Buf[0], 6, fp);  // essaye de lire 6 uint32

if(ret != 6){                                            // lu 6 uint32 ?
        fprintf(stderr,"Erreur lecture, demandé %d, lu %zu\n",6,ret); //non 4!
        if (feof(fp))                                    // car la fin de fichier atteinte ?
            fprintf(stderr,"Cause: fin de fichier atteinte\n");
        if ((e=ferror(fp)))  {                            // car autre cause ?
            fprintf(stderr,"Cause: erreur %d\n",e);
            clearerr(fp);                                 // efface les erreurs
            }
       // exit(EXIT_FAILURE); à décider en fonction de la logique du code
    }

if (fclose(fp))

       perror("Erreur fermeture");      // et oui c'est possible à la fermeture aussi!
```

Quiz

- Puis-je avoir un fichier contenant à la fois du texte et des valeurs binaires?
- Quelles fonctions appeler pour lire un tel fichier ?
- Quel est l'avantage de sauver un vecteur de valeurs en binaire vs. text ?
- Quel est l'avantage de sauver un vecteur de valeurs en text vs. binaire ?
- Quel est le gain de taille pour un fichier de valeurs en binaire vs. text ?
- J'ai des caractères bizarres en ouvrant un fichier texte créé sur un autre ordinateur, pourquoi ?
- Mon vecteur de short (2 bytes signed)

  { 255, 256, 257} sauvé dans un fichier et lu comme

  { -256, 1, 257} sur une autre machine, pourquoi ?
- Mon fichier text (ASCII) est sauvé en BigEndian et lu sur une autre machine LittleEndian, dois-je faire quelque chose de spécial ?

Quiz

- Puis-je avoir un fichier contenant à la fois du texte et des valeurs binaires?

  Oui, **vous** décider du contenu de votre fichier

- Quelles fonctions appeler pour lire un tel fichier ?

  fread()

- Quel est l'avantage de sauver un vecteur de valeurs en binaire vs. text ?

  `int a= 2147483648;`, en mode **binaire** prend **4** bytes, en mode **text 10** bytes

- Quel est l'avantage de sauver un vecteur de valeurs en text vs. binaire ?

  le mode **text** est lisible par un humain

- Quel est le gain de taille pour un fichier de valeurs en binaire vs. text ?

  variable, dans l'ex. ci-dessus gain de 2.5, cela peut être plus pour des nombres à virgules flottantes

- J'ai des caractères bizarres en ouvrant un fichier texte créé sur un autre ordinateur, pourquoi ?

  les char ACSII > 127 ne sont pas définis de manière uniforme entre les OS

- Mon vecteur de short (2 bytes signed) { 255, 256, 257} sauvé dans un fichier et lu comme { -256, 1, 257} sur une autre machine, pourquoi ? Endianess différente

  255 -> 0x00FF -> swapped -> 0xFF00 -> -256

  256 -> 0x0100 -> swapped -> 0x0001 -> 1

  257 -> 0x0101 -> swapped -> 0x0101 -> 257

- Mon fichier text (ASCII) est sauvé en BigEndian et lu sur une autre machine LittleEndian, dois-je faire quelque chose de spécial ?

  Non ce sont de char (il y a potentiellement un pb avec les char ASCII > 127)

Résumé – 1

- Votre application n'accède pas directement aux fichiers sur votre disque/serveur/cloud, le gestionnaire de fichier/OS se place entre vos requêtes et les fichiers et peut interférer avec les opérations que vous voulez réaliser.
  - ex. pour un fichier donné l'OS peut restreindre l'accès en lecture seul.
- Il est donc primordial de tester les "réponses" à vos appels fichiers.

ex.
```
*fp = fopen(…);
if(fp == NULL){  perror("Err ouverture"); … }
```

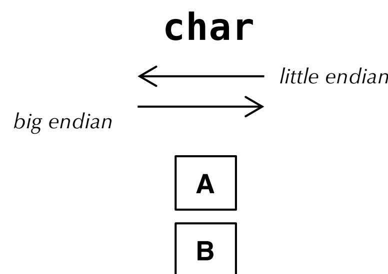
Si le fichier n'existe pas, affiche "**Err ouverture: No such file or directory**"

ex.
```
nLu=fread(…,nDemande,…);
if(nLu < nDemande){/* moins de car que demandé,
                       est-ce normal? */}
```

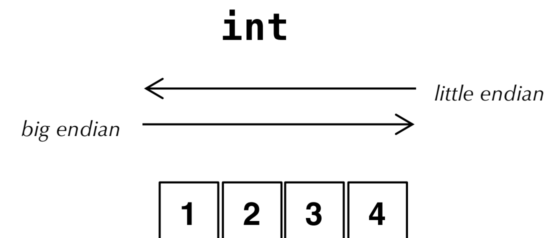
- Les fonctions `ferror()` et `perror()` vos permettent de connaitre/afficher la raison pour laquelle votre appel n'a pas fonctionné, par ex. "**No such file or directory**"

Résumé – 2

- Les fichiers sont composés de '0' et '1', c'est votre application qui décide comment interpréter ces '0' et '1', ex char, int, etc.
- L'accès aux fichiers peut se faire en mode **text** ou **binaire**. En mode **text** certain caractères peuvent être interprété, en mode **binaire** le contenu est retourné inchangé. Lors de l'ouverture du fichier vous devez spécifier si c'est un fichier binaire '**b**' ou non.
- La plus petite unité lue/écrite est le **byte**/char/uchar/8bits. Si vous écrivez du texte cela ne pose pas de problème chaque `char` a exactement la taille d'un **byte**. Par contre si vous écrivez un **`int`** qui a une longueur de **4 bytes**, il y a 2 manières de transférer le contenu de la mémoire vers le disque, soit en transférant en premier le byte de poids fort (big endian) puis les suivants, soit en transférant en premier le byte de poids faible (Little endian) puis les suivants.



fichier: **AB**



fichier: **1234** ou **4321**

Résumé – 3

- Attention aux caractères de fin de ligne qui sont spécifiques à l'OS. Idem pour les caractères ASCII accentués. Les formats UTF8,etc. gèrent ce problème de compatibilité.
- Pour accéder aux fichiers il y a toujours 3 étapes de base

  `fopen(), fread()/fwrite()/…, fclose()`
- Appel pour les fichiers binaires, le contenu n'est pas interprété

  `fread(), fwrite()`
- Fichier textuel, le contenu peut être interprété (ex. char. fin de ligne)

  `fprintf(),fscanf(), fgets(), fgetc(), etc.`
- Il est possible de tester si la fin d'un ficher est atteinte à l'aide de `feof()`

- Sous windows pour accéder aux fonctions fichier vu dans ce cours, vous devez ajouter la ligne suivante au début de votre programme:

```
#pragma warning(disable : 4996)
```

Quiz - 2

Votre disque contient la suite de bits **0100000101011010**

- Quel est son type ?

a) uchar b) short c) float d) non-défini

- Quel est sa valeur ?

a) 16730_{10} b) AZ_{char} c) $415A_{hex}$ d) non-défini

- Quel sera le contenu de votre mémoire une fois que vous l'aurez lu ?

a) AZ_{char} b) ZA_{char} c) 23105_{10} d) 16730_{10}

Quiz - 3

Votre disque contient le **pixel** suivant **01000010_b**

Si je le lis comme un **char**, dois-je le convertir en **entier** ?

Quiz - 3

Votre disque contient le **pixel** suivant **01000010_b**

Si je le lis comme un **char**, dois-je le convertir en **entier** ?

Comment convertir un **digit** au format **char** en un **entier** ? (Q.17)

Ex. **$'8'_{char}$ -> 8_{10}**

Quiz - 3

Votre disque contient le **pixel** suivant **01000010_b**

Si je le lis comme un **char**, dois-je le convertir en **entier** ?

Comment convertir un **digit** au format **char** en un **entier** ? (Q.17)

Ex. **'8'$_{char}$ -> 8_{10}**

```
int a= '8'- '0’; // a = 8;
                 // '8' =56₁₀   '0' =48₁₀
```

FILE structure in <stdio.h>

```
typedef struct __sFILE {
                unsigned char *_p; /* current position in (some) buffer */
                int _r; /* read space left for getc() */
                int _w; /* write space left for putc() */
                short _flags; /* flags, below; this FILE is free if 0 */
                short _file; /* fileno, if Unix descriptor, else -1 */
                struct __sbuf _bf; /* the buffer (at least 1 byte, if !NULL) */
                int _lbfsize; /* 0 or -_bf._size, for inline putc */

                /* operations */
                void *_cookie; /* cookie passed to io functions */
                int (* _Nullable _close)(void *);
                int (* _Nullable _read) (void *, char *, int);
                fpos_t (* _Nullable _seek) (void *, fpos_t, int);
                int (* _Nullable _write)(void *, const char *, int);

                /* separate buffer for long sequences of ungetc() */
                struct __sbuf _ub; /* ungetc buffer */
                struct __sFILEX *_extra; /* additions to FILE to not break ABI */
                int _ur; /* saved _r when _r is counting ungetc data */

                /* tricks to meet minimum requirements even when malloc() fails */
                unsigned char _ubuf[3]; /* guarantee an ungetc() buffer */
                unsigned char _nbuf[1]; /* guarantee a getc() buffer */

                /* separate buffer for fgetln() when line crosses buffer boundary */
                struct __sbuf _lb; /* buffer for fgetln() */

                /* Unix stdio files get aligned to block boundaries on fseek() */
                int _blksize; /* stat.st_blksize (may be != _bf._size) */
                fpos_t _offset; /* current lseek offset (see WARNING) */
} FILE;
```