Cours PRG2

Programmation C

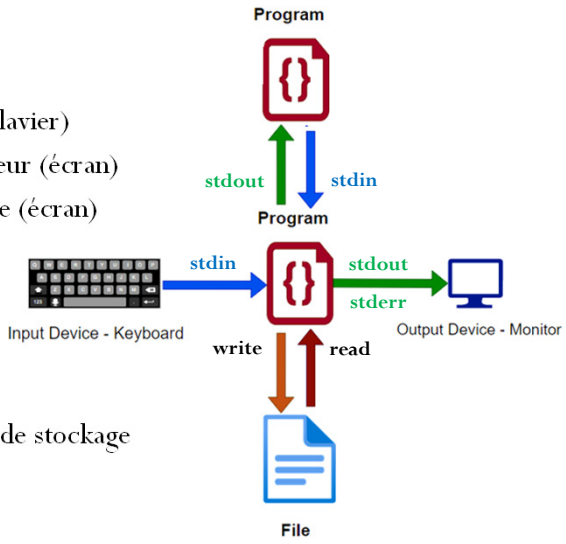
C9 *(2024_3)*

Contenu

- Flux de données et fichiers
- Gestion de fichiers texte
- Gestion de fichiers binaires
- Techniques de *buffering*

Flux de données et fichiers (1/2)

- Flux de données (*streams*)
  - 3 flux standards pour les I/O
    - **`stdin`** Entrée utilisateur (clavier)
    - **`stdout`** Affichage à l'utilisateur (écran)
    - **`stderr`** Affichage en urgence (écran)
  - Fichiers réguliers
    - Stockage persistant
    - Ecriture/lecture sur un support de stockage



Un **flux de données** (stream) désigne un flot séquentiel continu de données en provenance d'une source et à destination d'une cible (un seul sens).

Par défaut, trois flux de texte sont ouverts lors du démarrage d'un programme et sont déclarés dans l'en-tête <*stdio.h*> :

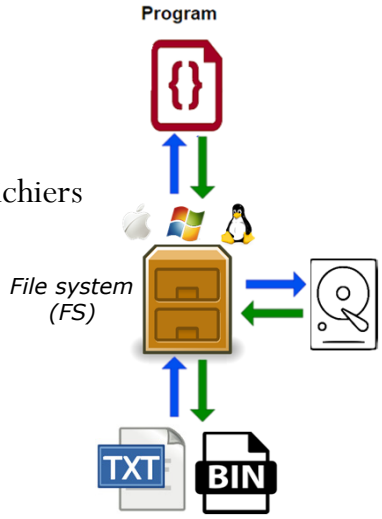
- **stdin** désigne l'**entrée standard**: le flux depuis lequel on récupère les informations fournies par l'utilisateur, en principe avec un clavier.
- **stdout** désigne la **sortie standard**: le flux permettant de transmettre des informations à l'utilisateur, en principe pour affichage à l'écran (dans le "terminal")
- **stderr** désigne la **sortie d'erreur standard**: flux à privilégier pour transmettre des messages d'erreurs ou des avertissements à l'utilisateur. Comme pour `stdout`, ces données sont le plus souvent affichées dans le terminal.

Alternativement, un flux de données peut être lié à la lecture ou l'écriture d'un **fichier** stocké sur le disque ou un autre périphérique de stockage.

Une bonne pratique consiste à afficher systématiquement les messages d'erreur sur `stderr`, aussi pour que ces messages apparaissent à l'écran même lorsque la sortie standard est redirigée (typiquement, dans un fichier texte).

Flux de données et fichiers (2/2)

- Notion de **fichier**
  - Persistance dans un système de stockage
    - Disque dur, SSD, mémoire *flash*, *stick USB*, etc.
  - **Espace d'octets** (bytes) adressables
- Accès et manipulation par le système de fichiers
  - Organisation hiérarchique en dossiers
  - Chemin d'accès
  - Métadonnées (dates, taille, permissions, etc.)
- Fichier **texte**
  - Contenu « lisible » par un humain (ASCII)
- Fichier **binaire**
  - Exécutable, image, audio, etc.



Il y a un lien fort entre les entrées/sorties d'un programme et le système d'exploitation qui les connecte au programme (processus). Pour les fichiers stockés sur disque, c'est le **système de fichiers** (*file system*) qui permet leur traitement par le programme.

Le FS facilite la localisation et la gestion des fichiers. Ces derniers y sont classés et organisés pour permettre à l'utilisateur de les répartir et retrouver dans une arborescence de **dossiers** *(folders)*. Un fichier y est localisé via son **chemin d'accès**.

Le chemin est une chaîne de caractères indiquant une position dans l'arborescence de dossiers. Il donne la suite des dossiers à traverser pour l'atteindre. Les dossiers à traverser sont séparés par des **"/"** sous Unix/Linux et des **"\"** sous Windows. Sous Unix/Linux, le dossier racine est "/" tandis que sous Windows, c'est "\" après un nom de lecteur (par exemple "C:\").

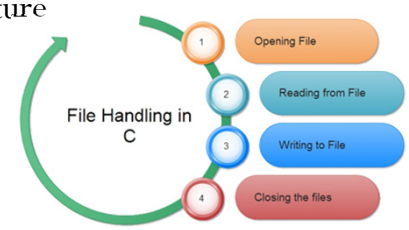
Selon le contenu et la manière dont on veut réaliser les opérations d'E/S sur un fichier, on distingue deux grandes catégories de flux:

- Les **flux de texte** sont visualisables sur un terminal et organisés en **lignes**. Une ligne est une suite de caractères terminée par le **caractère de fin de ligne** (inclus) '`\n`'.
- Les **flux binaires** contiennent des données qui ne sont pas forcément affichables à un utilisateur; les données sont lues ou écrites dans le fichier byte par byte.

Sous Windows, la fin de ligne est codée comme la combinaison des caractères CR et LF. L'écriture de '\n' sur un flux de texte provoque donc automatiquement l'écriture des caractères '\r' et '\n' dans le fichier associé.

Gestion de fichiers texte (1/6)

- API pour Ouverture, traitement, fermeture




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

structure de fichier opaque

```c
FILE *fichier;

FILE *fopen(const char *filename, const char *accessMode);
FILE *freopen(const char *pathname, const char *accessMode,
              FILE *stream)
```

Redirection du flux dans le fichier

```c
fichier = fopen("fichier.txt", "r");

fichier = fopen("./dossier/fichier.txt", "r");

fichier = fopen("C:\\users\\jeb\\fichier.txt", "r");

fclose(fichier);

fichier = freopen("fichier.txt", "w", stdout);
```

Chemin d'accès (relatif/absolu) (Linux/Windows)

Redirection de `stdout` dans le fichier

La fonction **`fopen()`** ouvre un fichier et lui associe un flux de données. La valeur retournée par fopen() est le flux de données, de type **`FILE *`**.

Le premier argument de `fopen()` est le nom du fichier concerné, ou plus exactement le chemin d'accès au fichier sur le système de fichiers sous-jacent. Il est fourni sous la forme d'une chaîne de caractères.

En général, on préfère définir le nom du fichier par une constante symbolique au moyen de la directive `#define` plutôt que d'expliciter le nom de fichier dans le corps du programme.

La fonction **`fclose()`** permet de fermer le flot qui a été associé à un fichier par la fonction `fopen()`.

La fonction **`freopen()`** permet de réutiliser la structure `FILE`, spécifiée en dernier argument ("fichier" sur l'exemple ci-dessus), pour ouvrir un nouveau flux. Le flux précédemment associé à fichier est automatiquement fermé et les éventuelles erreurs précédentes sont ignorées. Les autres arguments ont la même signification que pour la fonction fopen(). L'intérêt principal de cette fonction est de pouvoir réaffecter les flux standards (`stdin`, `stdout` et `stderr`) du programme, comme ci-dessus.

Gestion de fichiers texte (2/6)

- Le mode d'accès (ouverture) indique le type d'action possible sur le fichier.
- Pour un fichier texte, **`\n`** signifie une fin de ligne.

Mode	Opération(s)	[1]	[2]	[3]
`r / rb`	**Lecture**			
`r+ / r+b`	**Lecture et écriture**			
`w / wb`	**Écrasement**	✓	✓	
`w+ / w+b`	Lecture et écrasement	✓	✓	
`a / ab`	Ajout	✓		✓
`a+ / a+b`	Lecture et ajout	✓		✓

[1] Crée le fichier si besoin
[2] Efface le contenu
[3] Ajoute à la fin du fichier

'b' indique un **fichier binaire**

Le **mode d'accès** indique au système le type de buffer à préparer pour traiter le flux de données ainsi que les permissions à obtenir sur le fichier pour effectuer ce traitement.

Les principaux modes d'accès sont ceux représentés sur la table illustrée ci-dessus. Ils sont caractérisés ainsi :

- avec la lettre **'r'**, le fichier doit exister
- avec la lettre **'w'**, le fichier peut ne pas exister et sera créé ; s'il existe déjà, son contenu sera perdu
- avec la lettre **'a'**, le fichier peut ne pas exister et sera créé ; s'il existe déjà, les nouvelles données seront ajoutées à la fin du contenu existant.

Quand on veut exploiter un fichier sous forme binaire, il faut rajouter la lettre **'b'** à la fin de la chaîne décrivant le mode (Exemple : « `rb` », « `w+b` », etc.), sinon le fichier doit être traité comme un fichier texte.

Avec certains OS, l'option 'b' n'est pas nécessaire ; on ouvre le fichier et on applique les fonctions correspondant au format sous lequel on souhaite traiter le fichier (format binaire ou format texte). Attention, toutefois à la portabilité du code source si le programme est également destiné à d'autres plateformes.

Dans les faits, les modes le plus couramment utilisés sont "r"/"rb", "w"/"wb" et "r+"/"r+b". Il est conseillé, par sécurité, de préférer "r/rb" si l'on a juste l'intention de lire. Le mode "r+" fonctionne aussi, mais avec le risque de modifier le fichier par erreur.

Gestion de fichiers texte (3/6)

- Test de l'existence d'un fichier et fermeture

```c
#include <stdio.h>

int main() {
    FILE *fichier;

    fichier = fopen("fichier.txt", "r+");

    if (fichier == NULL)
        printf("impossible d'ouvrir fichier.txt\n");
    else
        fclose(fichier);

    // Création d'un fichier vide

    fichier = fopen("/home/jeb/fichier.txt", "w");

    if (fichier == NULL)
        printf("impossible d'ouvrir /home/jeb/fichier.txt\n");
    else
        fclose(fichier);

}
```

Ouverture du fichier en lecture et écriture, s'il existe

Fermeture requise après utilisation

Ouverture en écriture

Les répertoires doivent exister !

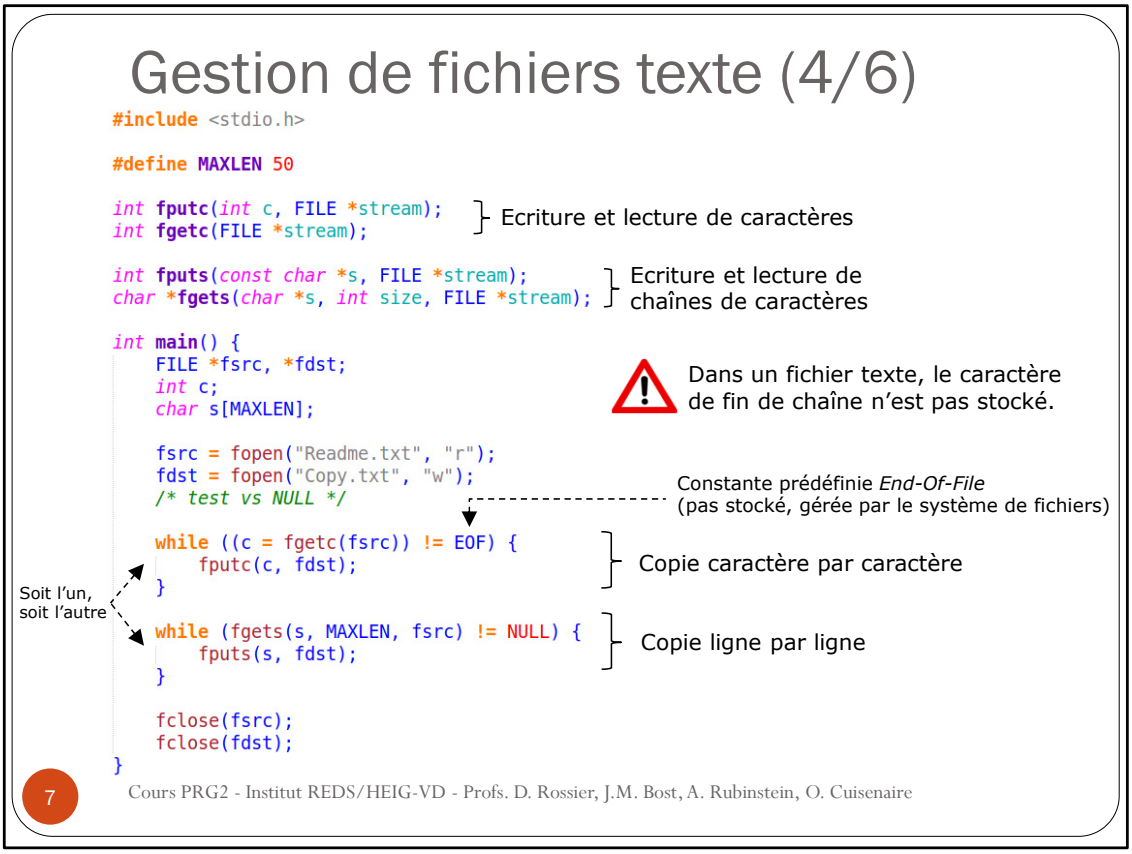
6

Cours PRG2 - Institut REDS/HEIG-VD - Profs. D. Rossier, J.M. Bost, A. Rubinstein, O. Cuisenaire

Si l'exécution de `fopen()` ne se déroule pas normalement (par exemple, si une permission nécessaire est refusée par le système), la valeur retournée est le pointeur `NULL`.

Il est recommandé de **toujours tester** si la valeur renvoyée par la fonction `fopen()` est égale à NULL afin de détecter les erreurs (lecture d'un fichier inexistant, permission refusée, ...).

La fonction `fclose()` retourne un entier qui vaut zéro si l'opération s'est déroulée normalement et une valeur non nulle en cas d'erreur. Les erreurs peuvent être importantes à tester car l'invocation de `fclose()`, pour un fichier ouvert en écriture, peut provoquer l'écriture des données en attente dans le buffer et cette écriture peut échouer. Toutefois, la pratique montre que ce test est rarement effectué, la probabilité d'un échec reste très faible.



Similaires aux fonctions `getchar()` et `putchar()`, les fonctions **`fgetc()`** et **`fputc()`** permettent respectivement de lire et d´écrire un caractère dans un flux au sens général (i.e. pas seulement dans `stdin`/`stdout`) et donc, dans un fichier.

La fonction `fgetc()`, de type `int`, retourne le caractère lu dans le fichier (ou un code d'erreur, raison du type entier). Elle retourne la **constante `EOF`** lorsqu'elle détecte la fin du fichier ou une erreur. Comme pour la fonction `getchar()`, il est conseillé de déclarer de type `int` la variable destinée à recevoir la valeur de retour de `fgetc()` pour pouvoir détecter correctement la fin de fichier.

La fonction `fputc()` écrit un caractère (de type entier, comme déjà vu avec `putchar()`) dans le flux de données. Elle retourne l'entier correspondant au caractère lu ou la constante `EOF` en cas d'erreur.

Il existe également deux versions optimisées des fonctions `fgetc()` et `fputc()` qui sont implémentées par des macros. Il s'agit respectivement de `getc()` et `putc()`. Leur syntaxe est similaire à celle de `fgetc()` et `fputc()`.

Gestion de fichiers texte (5/6)

- Implémentation de `fgets()` avec `fgetc()`

```c
int feof(FILE *stream);   <---------- Teste la fin de fichier, sans lecture

char *fgets_prg2(char s[], int maxL, FILE *f) {
    int val;
    int i = 0;

    while (i < maxL - 1) {
        val = fgetc(f);
        if (val == EOF) {   <---------- Fin de fichier ou erreur

            if ((i == 0) || !feof(f)) {
                // Nothing to read and EOF encountered
                return NULL;
            }
            break;

        } else if (val == '\n') {
            s[i] = val;
            i++;
            break;

        } else {
            s[i] = val;
            i++;
        }
    }

    s[i] = '\0';

    return s;
}
```


Dans le cas de la fonction **`fgets()`**, il est possible de sécuriser la copie des caractères de la chaîne en spécifiant une taille maximale à lire (*`maxL`* dans l'exemple ci-dessus). Ainsi, si la chaîne à lire est plus longue que le buffer de réception, on peut contrôler le risque de dépassement de capacité. Le nombre de caractères pouvant être lu sera au maximum de (*`maxL`* `- 1`) ou moins si un caractère de fin de flux (`EOF`) ou un caractère de fin de ligne est rencontré. Dans tous les cas, un caractère '`\0`' sera systématiquement ajouté en fin de chaîne après le dernier caractère lu.

La fonction `fgets_prg2()` ci-dessus est une implémentation *custom* de `fgets()` à partir de `fgetc()`. Elle illustre plus explicitement les différents cas de troncation de la chaîne lue en fonction de la situation rencontrée.

Gestion de fichiers texte (6/6)

- Lecture & écriture de chaînes formattées

#include <stdio.h>

```c
#define MAXLEN  50

int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);

int main(int argc, char **argv) {

    FILE *fin = fopen(argv[1], "r");
    FILE *fout = fopen("moyennes.txt", "w");

    // Title row
    char row[MAXLEN], col1[MAXLEN], col2[MAXLEN];
    // data rows
    float mrk1, mrk2, mrk3;

    if (fscanf(fin, "%s %s %*s %*s %*s", col1, col2) != 2) {
        fprintf(stderr, "[e] could not get title row\n");
        ferror(fin);
    }

    fprintf(fout, "%s,%s,MOYENNE\n", col1, col2);

    while (fscanf(fin, "%s %s %f %f %f", col1, col2, &mrk1, &mrk2, &mrk3) == 5)
        fprintf(fout, "%s, %s, %f\n", col1, col2, (mrk1 + mrk2 + mrk3) / 3);
}
```

Similaire à `printf()` et `scanf()`

----- Indique une chaîne à ignorer

input.txt:

NOM	PRENOM	NOTE1	NOTE2	NOTE3
Becile	Alain	1.8	1.9	2.0
Hochon	Paul	3.5	3.7	3.1
Trichet	Ella	6.0	6.0	6.0

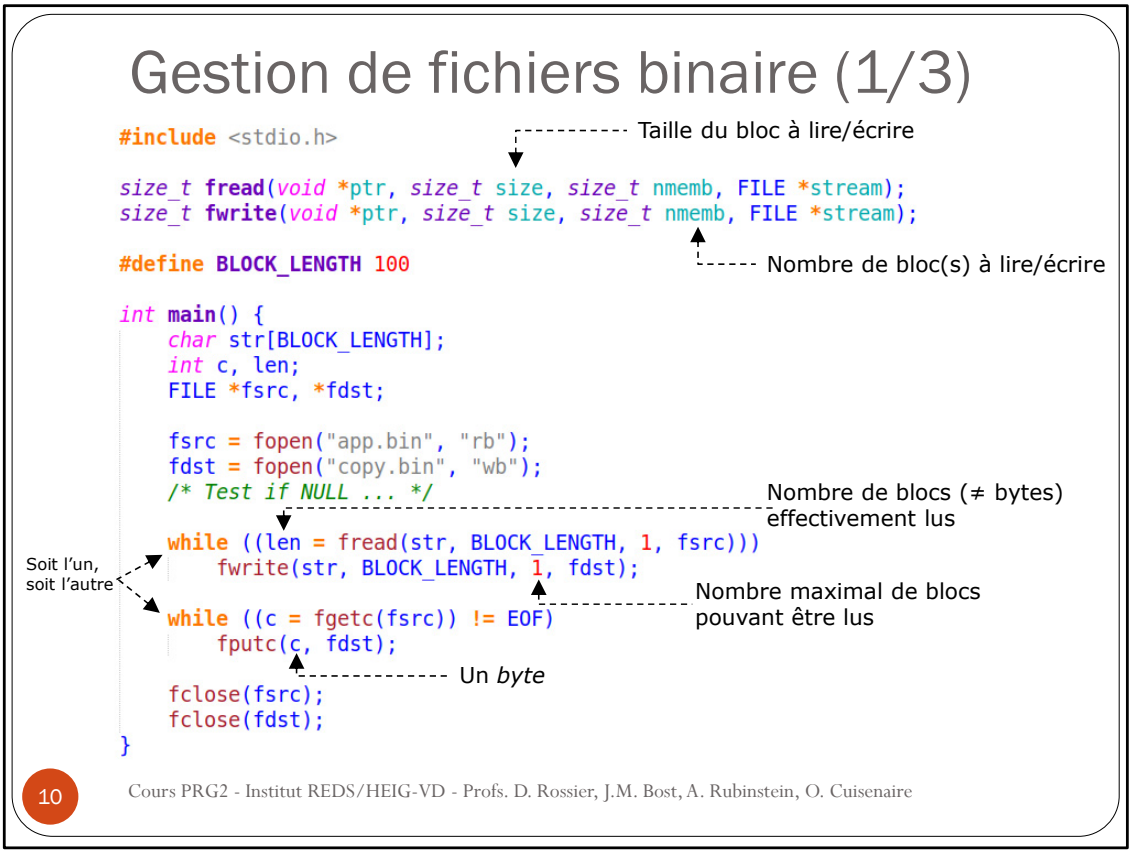
moyennes.txt:

```
NOM,PRENOM,MOYENNE
Becile, Alain, 1.900000
Hochon, Paul, 3.433333
Trichet, Ella, 6.000000
```


`fscanf()` et **`fprintf()`** fonctionnent exactement comme `scanf()`/`sscanf()` et `printf()`/`sprintf()`, excepté que l'information y est respectivement lue et écrite dans un flux précédemment ouvert avec le bon mode plutôt que sur les entrées/sorties standards ou une chaîne de caractères.

Comme `scanf()`, la fonction `fscanf()` exécute chaque instruction de la chaîne de formats dans l'ordre spécifié. Si une directive échoue, la fonction s'arrête. Les échecs sont décrits comme des échecs d'entrée (dus à l'indisponibilité d'octets d'entrée) ou des échecs de **correspondance** (dus à une entrée inappropriée). La fonction retourne le nombre de directives exécutées avec succès.

Comme `printf()`, la fonction `fprintf()` produit une chaîne de caractères en exécutant les instructions de la chaîne de formats et en appliquant celles-ci aux arguments qui suivent dans leur ordre de succession.



On peut lire ou écrire un fichier binaire byte par byte avec les fonctions `fgetc()` et `fputc()` que nous avons déjà vues.

La bibliothèque standard stdio.h offre également des fonctions permettant de réaliser des opérations d'entrées/sorties par blocs de bytes. Il s'agit des fonctions **fread()** et **fwrite()**.

L'utilisation de `fread()` et `fwrite()` est relativement simple ; la première lit et la seconde écrit à la position courante dans le fichier. Si elles réussissent, la première lit, tandis que la seconde écrit, le nombre indiqué de blocs de bytes de la taille spécifiée à partir de cette position, puis elles mettent à jour la position du fichier avec la nouvelle valeur. La valeur de retour des deux fonctions est le nombre de blocs qui ont été effectivement lus ou écrits.

Gestion de fichiers binaire (2/3)

- Ecriture & lecture d'une structure

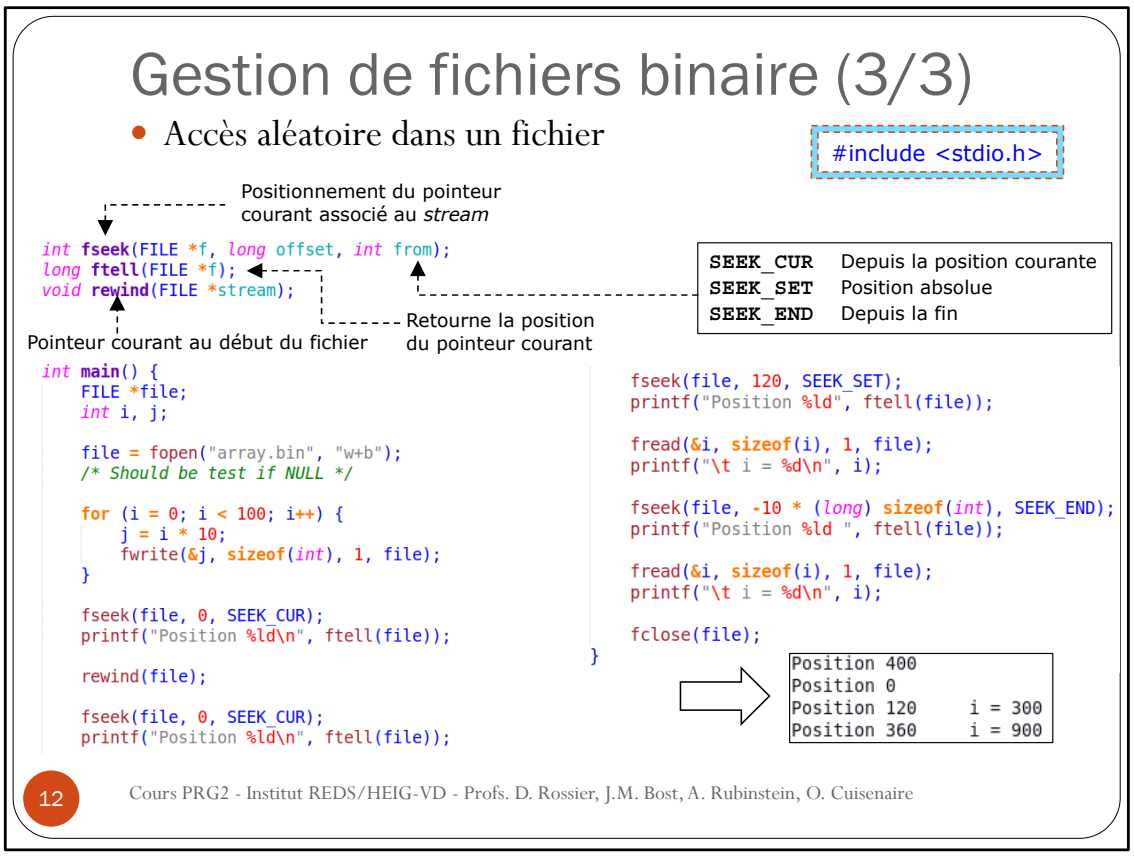
```c
typedef struct {
    char nom[20];
    char prenom[20];
    float note1;
    float note2;
    float note3;
} eval_t;

int main(int argc, char **argv) {

    eval_t te1[3] = {
            "Becile", "Alain", 1.8, 1.9, 2.0,
            "Hochon", "Paul", 3.5, 3.7, 3.1,
            "Trichet", "Ella", 6.0, 6.0, 6.0
            };

    FILE *fp;

    fp  = fopen("notes.bin", "wb");
    /* Should test if NULL... */

    fwrite(te1, sizeof(eval_t), 3, fp);
    /* Should test the return and make sure it's 3 */

    fclose(fp);
}
```

```c
memset(te1, 0, sizeof(eval_t) * 3);

fp = fopen("notes.bin", "rb");
/* Should test if NULL... */

if (fread(te1, sizeof te1[0], 3, fp) == 3) {
    for (int i = 0; i < 3; i++) {
        printf("te1[%d] = %s, %s", i,
               te1[i].nom, te1[i].prenom);
        printf(", %.1f, %.1f, %.1f\n",
               te1[i].note1, te1[i].note2, te1[i].note3);
    }
}

fclose(fp);
```

Ecriture de 3 blocs de taille de la structure

Le code ci-dessus illustre l'intérêt du format binaire, à savoir la sauvegarde et la restauration de données structurées du programme sans avoir à les convertir en format texte.

On voit bien sur cet exemple que `fread()` et `fwrite()` retournent le nombre de blocs traités (3 dans ce cas).



la fonction ***`fseek()`*** permet de modifier la position courante dans un fichier. La nouvelle position est précisée avec 2 arguments : ***l'offset*** qui représente le déplacement et l'origine à partir de laquelle le déplacement est calculé.

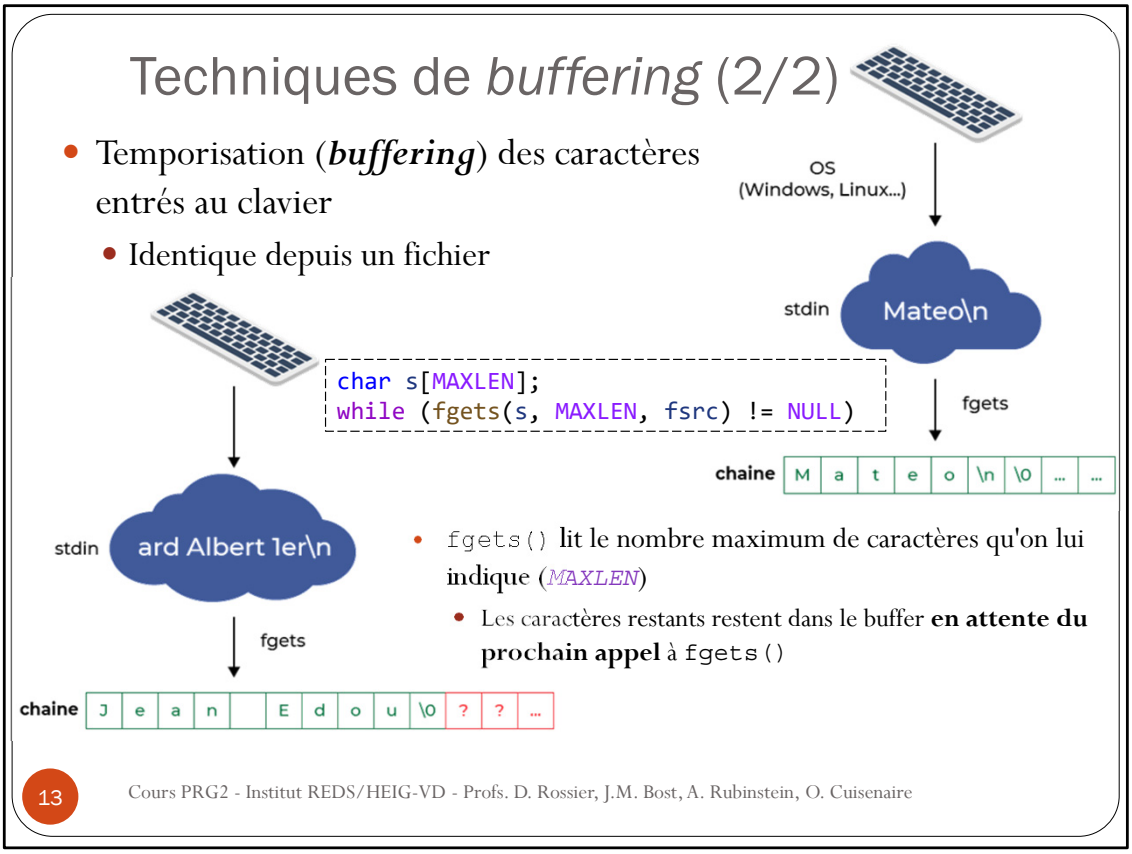
Les trois valeurs possibles sont définies ci-dessus.

Bien entendu, l'*offset* peut être **positif** ou **négatif**. Si la fonction réussit, 0 est retourné (une valeur différente de zéro indique donc une erreur).

La fonction **`rewind()`** est un "raccourcit" sur la fonction `fseek()` avec un *offset* de 0 et une origine pointant sur le début du fichier.

L'utilisation de `fseek()` avec un flux de texte peut donner des résultats inattendus à cause du '`\n`' qui peut être codé comme plusieurs caractères. Pour aller au n-ième caractère d'un fichier texte, il est préférable de se repositionner au début avec `fseek()` ou `rewind()` puis utiliser `fgetc()` pour compter les caractères un à un jusqu'à la position désirée.

La fonction **`ftell()`** retourne la position courante dans le fichier en cours de lecture ou écriture. Dans le cas où le fichier est associé à un flux binaire, il s'agit du **nombre de bytes** entre le début du fichier et la position courante. Si le fichier est associé à un flux de texte, la valeur retournée par cette fonction est tout simplement une information représentant la position actuelle dans le fichier et on ne peut rien dire de plus.

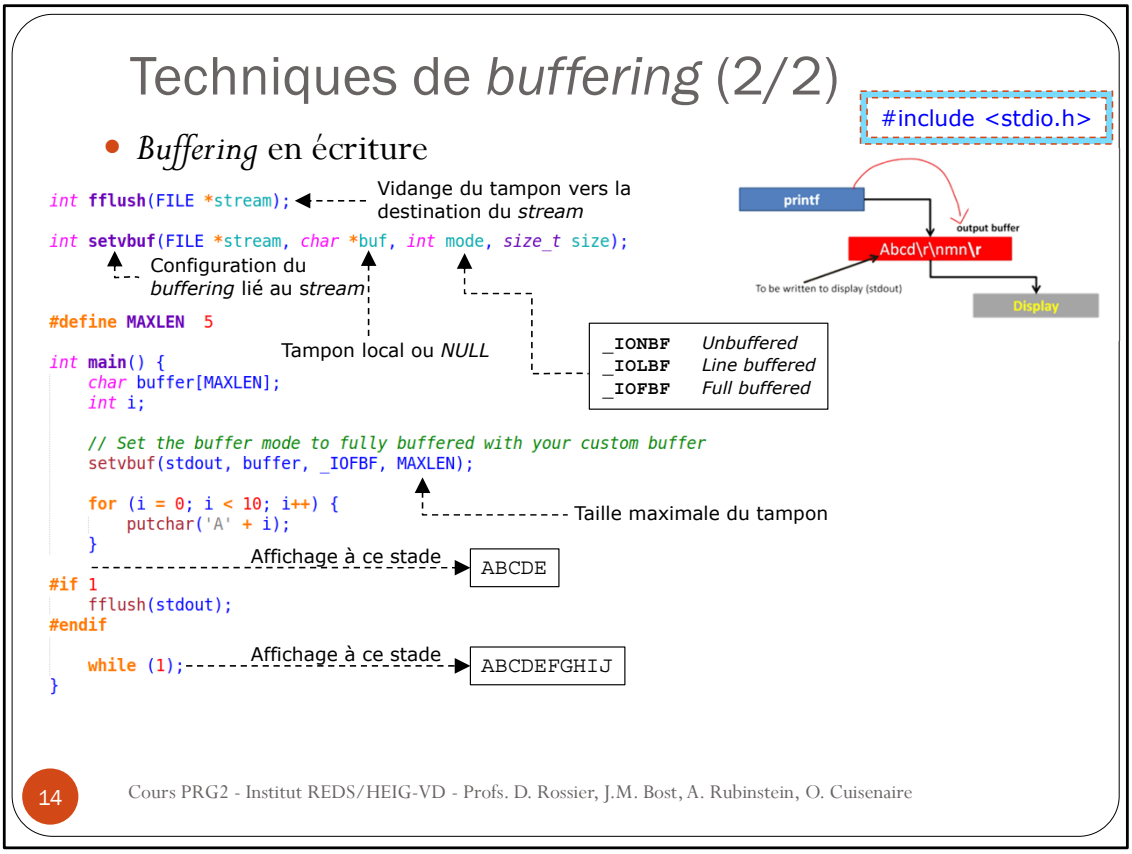


Le **buffer** (ou tampon) **d'un flux ouvert en lecture** (`stdin` ou un fichier) est une mémoire temporaire qui stocke les données en provenance de la source (le clavier ou le système de fichiers) en attente de leur traitement par le programme. Ainsi, lorsque l'utilisateur du programme tape du texte au clavier, le système d'exploitation copie directement le texte tapé dans le buffer `stdin` du programme.

La fonction `fgets()` est un exemple de traitement possible du contenu du buffer par le programme ; elle va extraire du *buffer* les caractères qui s'y trouvent pour les copier dans la zone mémoire qu'on lui indique (tableau pour chaine de caractères).

Après avoir effectué son travail, `fgets()` a enlevé du *buffer* tout ce qu'elle a pu copier dans le tableau cible. Si tout s'est bien passé, `fgets()` a pu copier tout le *buffer* dans la chaîne. Dans ce cas, le *buffer* se retrouve vide à la fin de l'exécution de la fonction.

Mais si l'utilisateur entre trop de caractères, et que la fonction `fgets()` ne peut copier qu'une partie d'entre eux, seuls les caractères lus seront supprimés du buffer. Tous ceux qui n'auront pas été lus y restent en attente jusqu'à la prochaine invocation de `fgets()`.



Le **buffer d'un flux ouvert en écriture** est une mémoire temporaire qui stocke les données avant de les envoyer vers la destination principale: un fichier, *stdout*, *stderr*, etc.

Le *buffer* se vide en fonction de la "stratégie de *buffering*". Celle-ci change selon la plateforme cible mais elle peut être modifiée via la fonction **`setvbuf()`**. On peut, par exemple, activer la temporisaiton (*buffering*) avec une taille fixe pour le *buffer* en lui passant l'argument **`_IOFBF`** ainsi que la taille du *buffer* en bytes. Cette stratégie aura pour effet de vider automatiquement le buffer vers sa destination chaque fois qu'il est plein.

La fonction **`fflush()`** peut, quant à elle, être utilisée pour vider le *buffer* sans attendre que le *buffer* soit plein. `fflush()` en C est définie dans `stdio.h`. L'invocation de `fflush()` provoque l'envoi de l'intégralité du *buffer* vers le flux destination.