

Langage C

fichiers texte

Wieslaw Zielonka
zielonka@irif.fr

fichiers

De point de vu du C un fichier c'est juste une suite d'octets stockée sur une mémoire externe :

un disque dure, une mémoire SSD, clef USB

Cette suite d'octets n'a aucune structure interne.

De point de vu de programme C chaque fichier est soit une fichiers texte soit un fichier binaire mais les deux types de fichiers c'est juste une suite d'octets et dans le fichier lui-même il n'y a aucune information sur le type : texte ou binaire.

fichiers

Exemple : un fichier avec 4 octets (chaque octets en notation hexadécimale, deux chiffres hexa pour un octet)

61 62 63 64

si on regarde ce fichier comme un fichier texte alors le fichier contient 4 caractères :

a b c d

(61 etc ce sont les codes ASCII pour ces 4 lettres)

Si on regarde le même fichier et si on pense qu'on stocke ici une valeur int (sur une machine avec `sizeof(int)==4`) alors on pourra lire cette valeur et la mettre dans une variable int qui prendra la valeur 0x61626364

C'est à l'utilisateur de savoir si le fichier est un fichier texte ou un fichier binaire et d'utiliser les fonctions appropriés pour la lecture et l'écriture.

fichiers

- ouvrir un fichier `fopen()`
- lire le fichier et/ou écrire dans le fichier
- fermer le fichier `fclose()`

Le fichier en-tête pour les prototypes de toutes les fonctions sur les fichiers :

`stdio.h`

ouverture de fichier

```
FILE *fopen(const char *nom_fichier,  
            const char *mode)
```

l'ouverture de fichier ../file.txt en lecture

```
FILE *flot = fopen( "../file.txt", "r");
```

Notez que le dernier paramètre est char * et non char

```
FILE *f
```

lot = fopen("../file.txt", 'r');

le dernier paramètre incorrect

flot = l'objet FILE * retourné par open()














ouverture d'un fichier

```
FILE *fopen(const char *nom_fichier, const char *mode)
```

les modes possibles:

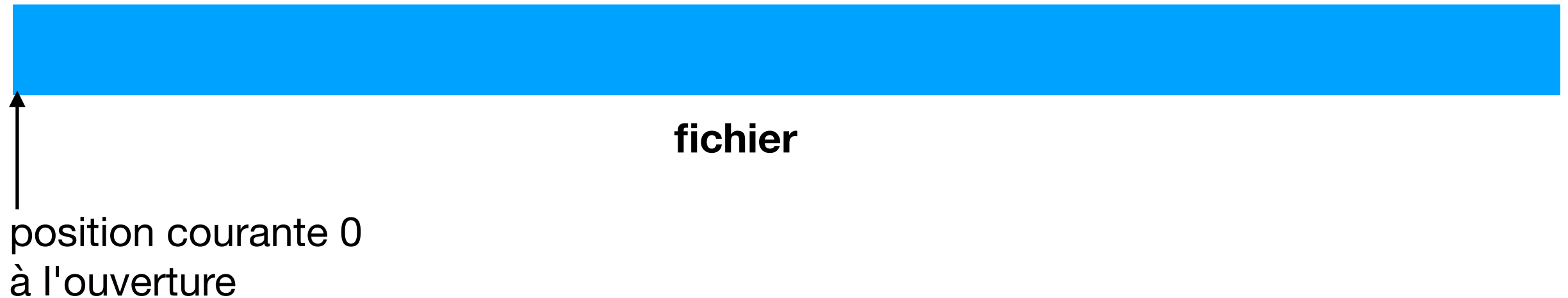
- "r" – ouvre le fichier en lecture, position initiale au début du fichier, erreur si le fichier n'existe pas
- "w" -- ouvre le fichier en mode écriture, si le fichier existait le contenu est écrasé à l'ouverture, si le fichier n'existait pas il est créé
- "a" -- ouvre le fichier en mode "append" (ajout), chaque écriture se fait à la fin de fichier, si le fichier n'existe pas il est créé
- "r+" - ouvre le fichier en mode lecture/écriture, la position initiale au début de fichier, si le fichier n'existe pas alors erreur
- "w+" - ouvre le fichier en mode lecture/écriture, si le fichier est non vide le contenu est écrasé, si le fichier n'existe pas alors erreur
- "a+" – ouvre le fichier en lecture/écriture, si le fichier n'existe pas alors il est créé, l'écriture se fait à la fin de fichier

résumé de six modes d'ouverture

	r	w	a	r+	w+	a+
fichier doit déjà existé ?						
le contenu du fichier est écrasé à l'ouverture ?						
lecture du flot						
écriture dans le flot 0 partir de la position courante						
écriture uniquement à la fin du flot (mode append)						

mode d'ouverture d'un fichier et la position courante

Supposant que le fichier existant est ouvert en lecture et écriture et **sans mode append et sans écrasement à l'ouverture.**



Chaque lecture et chaque écriture change la position courante dans le fichier.

mode d'ouverture d'un fichier et la position courante

lecture : transfert de données depuis le fichier vers la mémoire vive

fichier

position courante 0
à l'ouverture

lecture de n octets

n octets

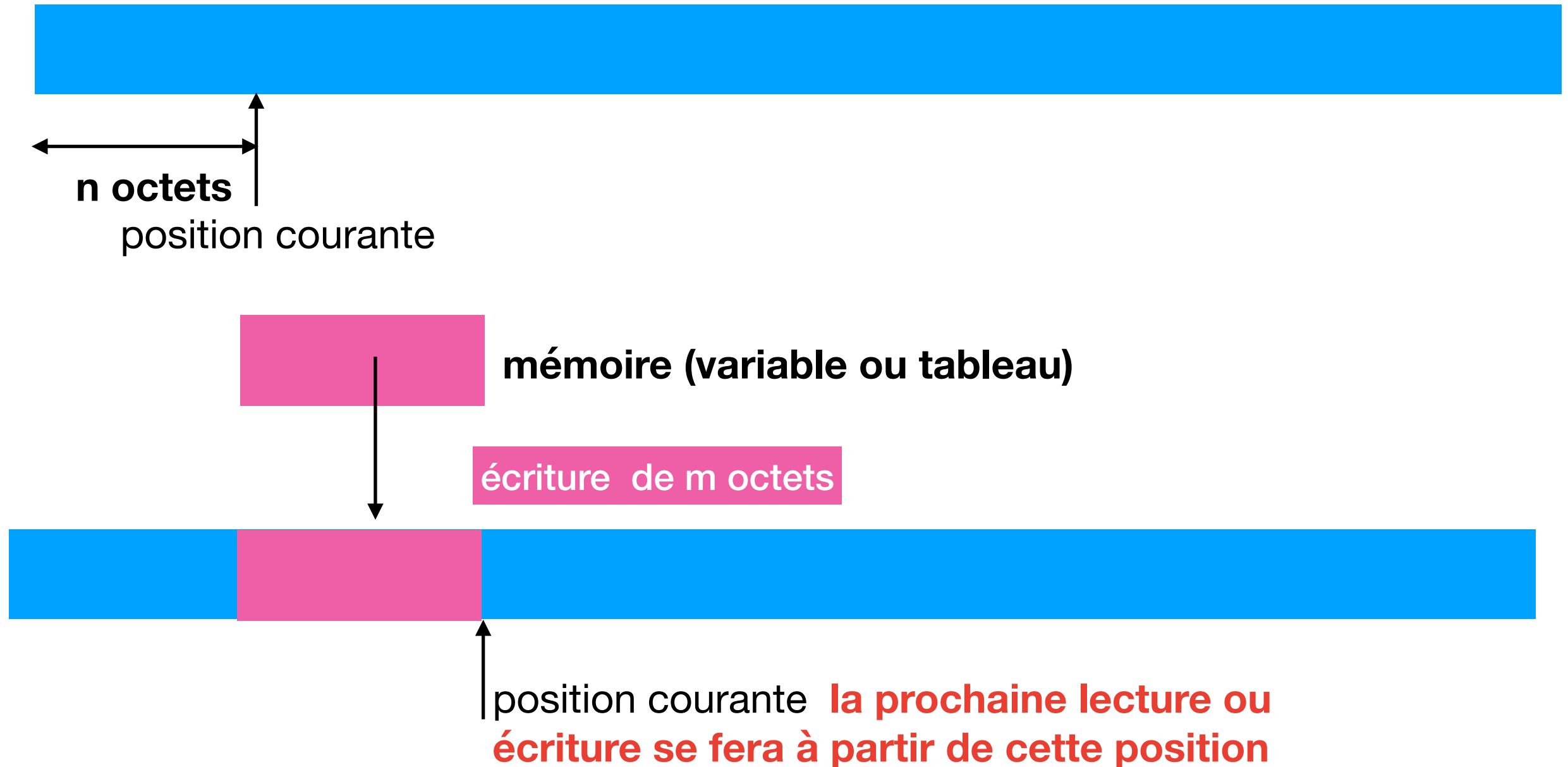
position courante

mémoire interne (variable ou tableau)

les n premiers octets sont copiés depuis le fichier vers la mémoire interne
(dans un tableau, dans une variable, ou dans une mémoire allouée par malloc)

mode d'ouverture d'un fichier et la position courante

écriture : transfert de données depuis la mémoire vive vers le fichier

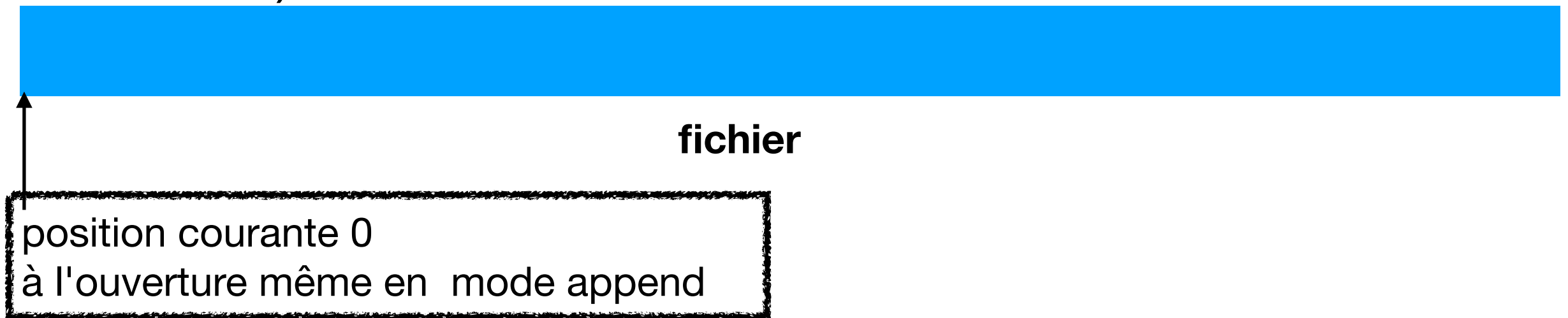


m octets copiés de la mémoire (d'un tableau, d'une variable, ou d'une mémoire allouée par malloc()) vers le fichier

Il y a écrasement de m octets dans le fichier.

mode d'ouverture d'un fichier en mode append

Supposant que le fichier existant est ouvert en lecture et écriture et **en mode append** (mais sans écrasement à l'ouverture) : mode "a+"

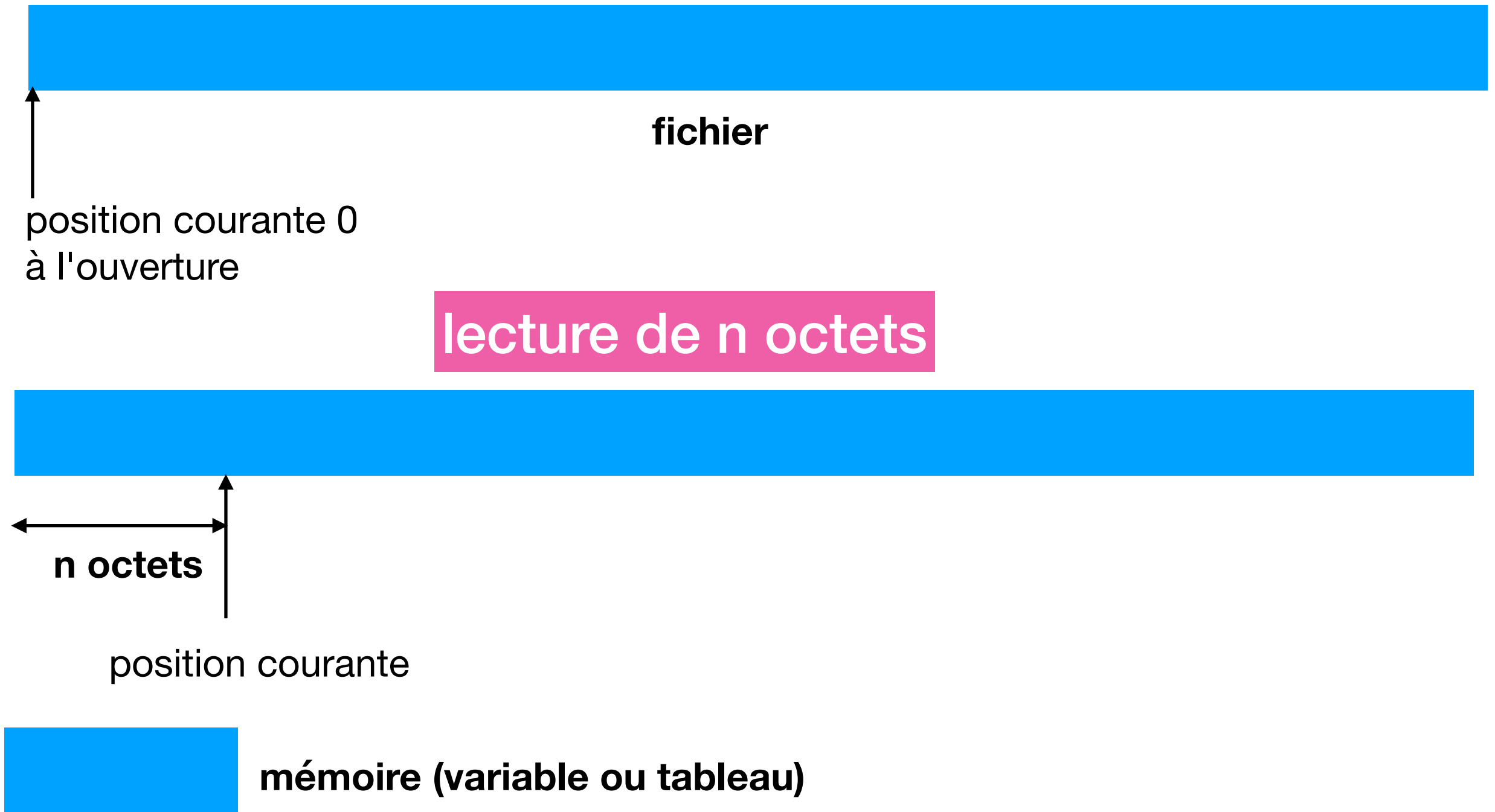


En mode append :

Chaque lecture et écriture change la position courante.

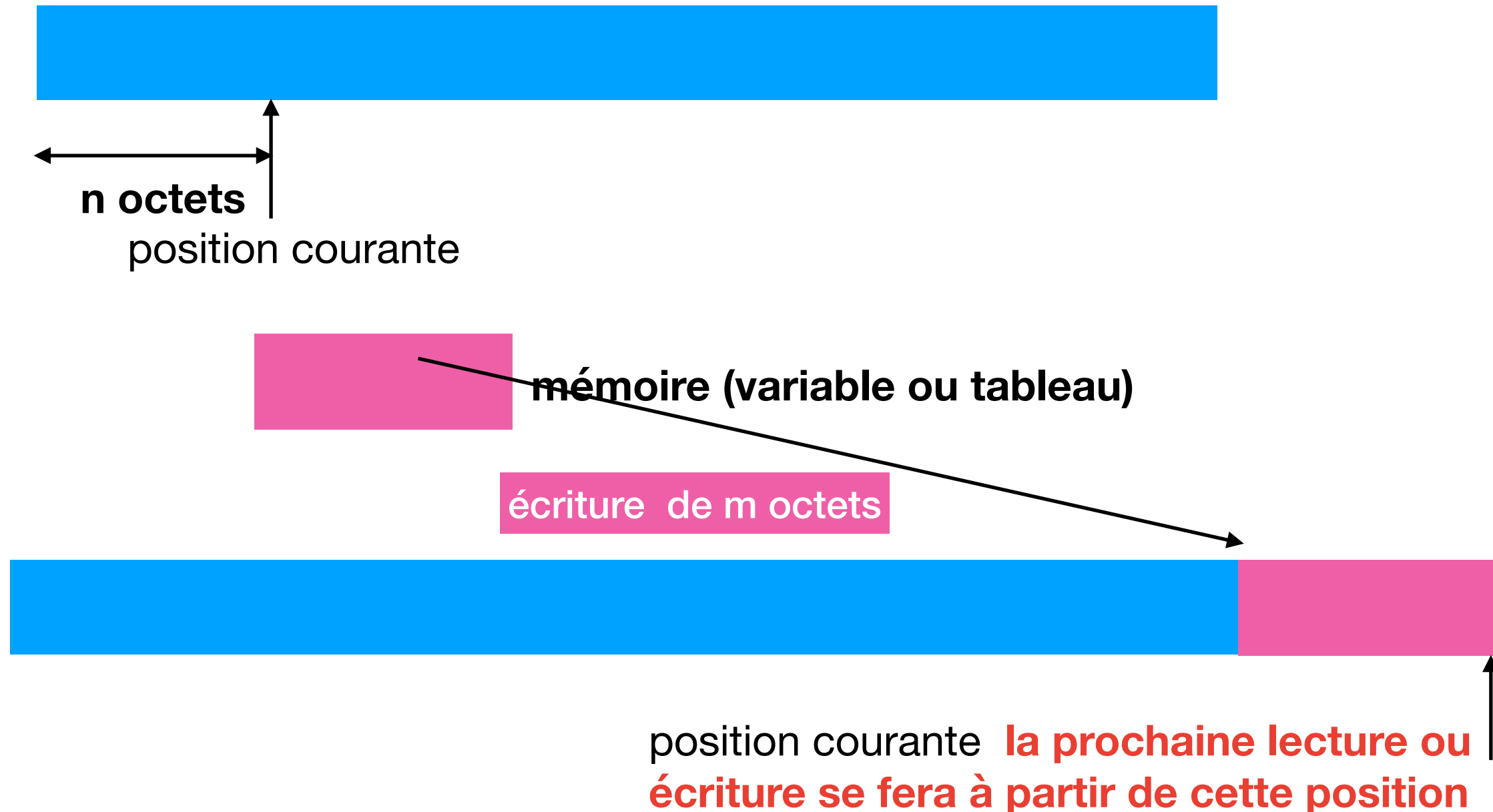
Avant chaque écriture (et uniquement avant l'écriture) la position courante se déplace automatiquement à la fin du fichier.

lecture d'un fichier ouvert en mode "a+"



les n premiers octets sont copiés depuis le fichier dans la mémoire (dans un tableau, dans une variable, ou dans une mémoire allouée par malloc). **Même chose qu'en mode non-append.**

dans un fichier ouvert en mode "a+"



en mode append m octets copiés de la mémoire (d'un tableau, d'une variable, ou d'une mémoire allouée par malloc()) vers le fichier.

La copie s'effectue à la fin de fichier (la taille du fichier augmente de m octets);
avant l'écriture la position courante se déplace à la fin de fichier.

Après l'écriture la position courante reste à la fin du fichier.

ouverture d'un fichier avec écrasement

Quand le fichier est ouvert en écriture (en mode append ou non) **avec l'écrasement initiale** le contenu du fichier est effacé à l'ouverture : immédiatement après l'ouverture le fichier aura 0 octets.

changement de la position courante

Il existe des fonctions qui permettent de changer la position courante sans lecture ni écriture.

fermeture de flot

```
int fclose(FILE *f) {
```

`fclose()` retourne 0 si OK et EOF en cas d'erreur.

trois flots standard

Trois flots

- `stdin` – flot d'entrée standard
- `stdout` – flot sortie standard
- `stderr` – flot sortie d'erreurs standard

sont déjà ouverts au début de l'exécution du programme. (Il est possible de les fermer au lancement de programme).

`stdin`, `stdout`, `stderr` sont des objets de type `FILE *` donc les opérations lecture/écriture qu'on applique aux fichiers s'appliquent aussi à ces trois flots.

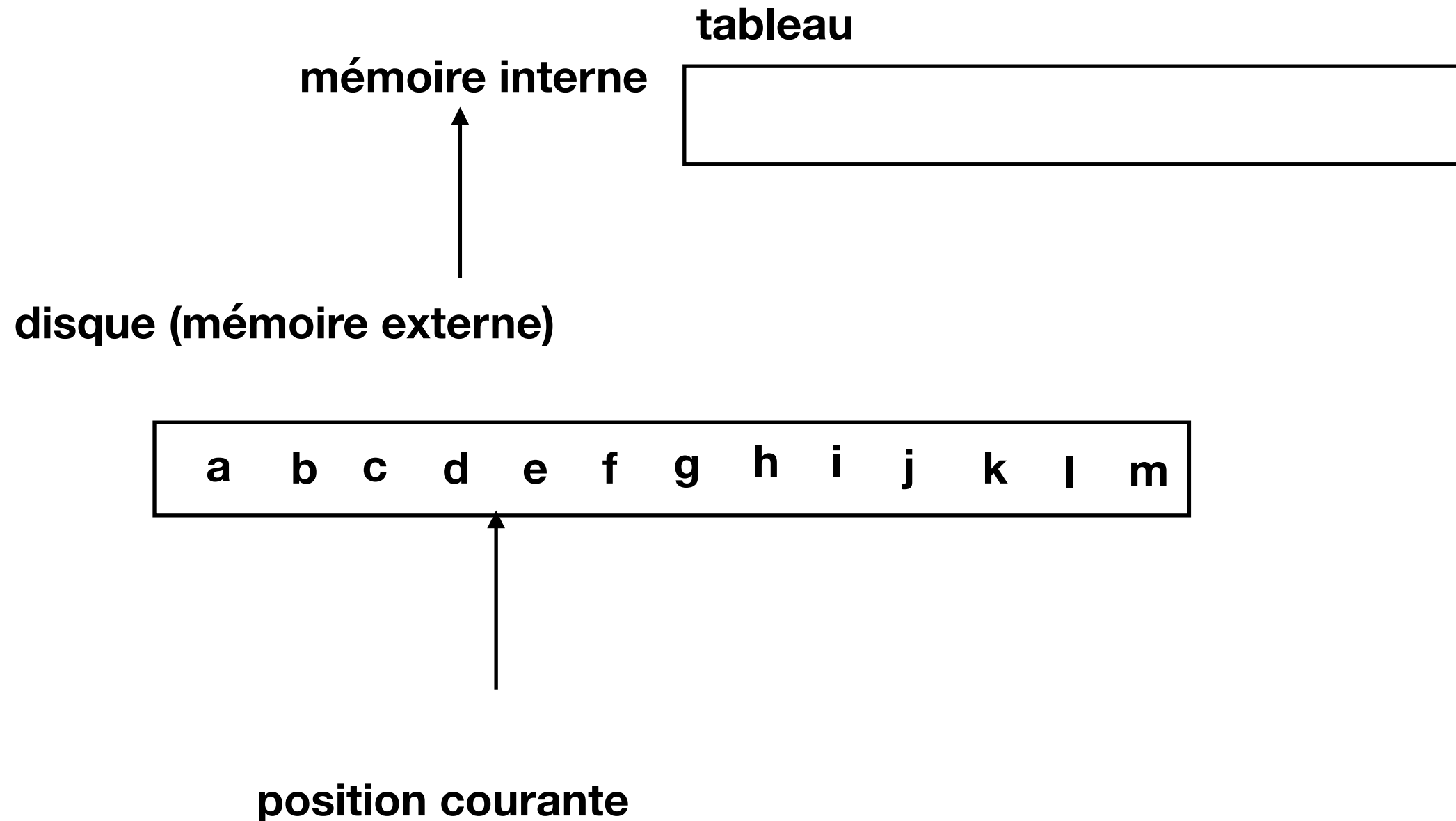
FILE *

FILE est une structure opaque pour l'utilisateur qui contient toutes les informations nécessaires pour gérer les entrées / sorties par les fonctions de la bibliothèque standard :

- le descripteur de fichier utilisé pour effectuer les opérations entrée/sortie (voir le cours de systèmes)
- le pointers vers le tampon du flot
- la taille du tampon,
- le nombre de caractères actuellement dans le tampon,
- indicateur d'erreur,
- la **position courante** dans le flot
- pour les flots mémorisés, *la position courante* dans le tampon,
- indicateur de fin de flot.

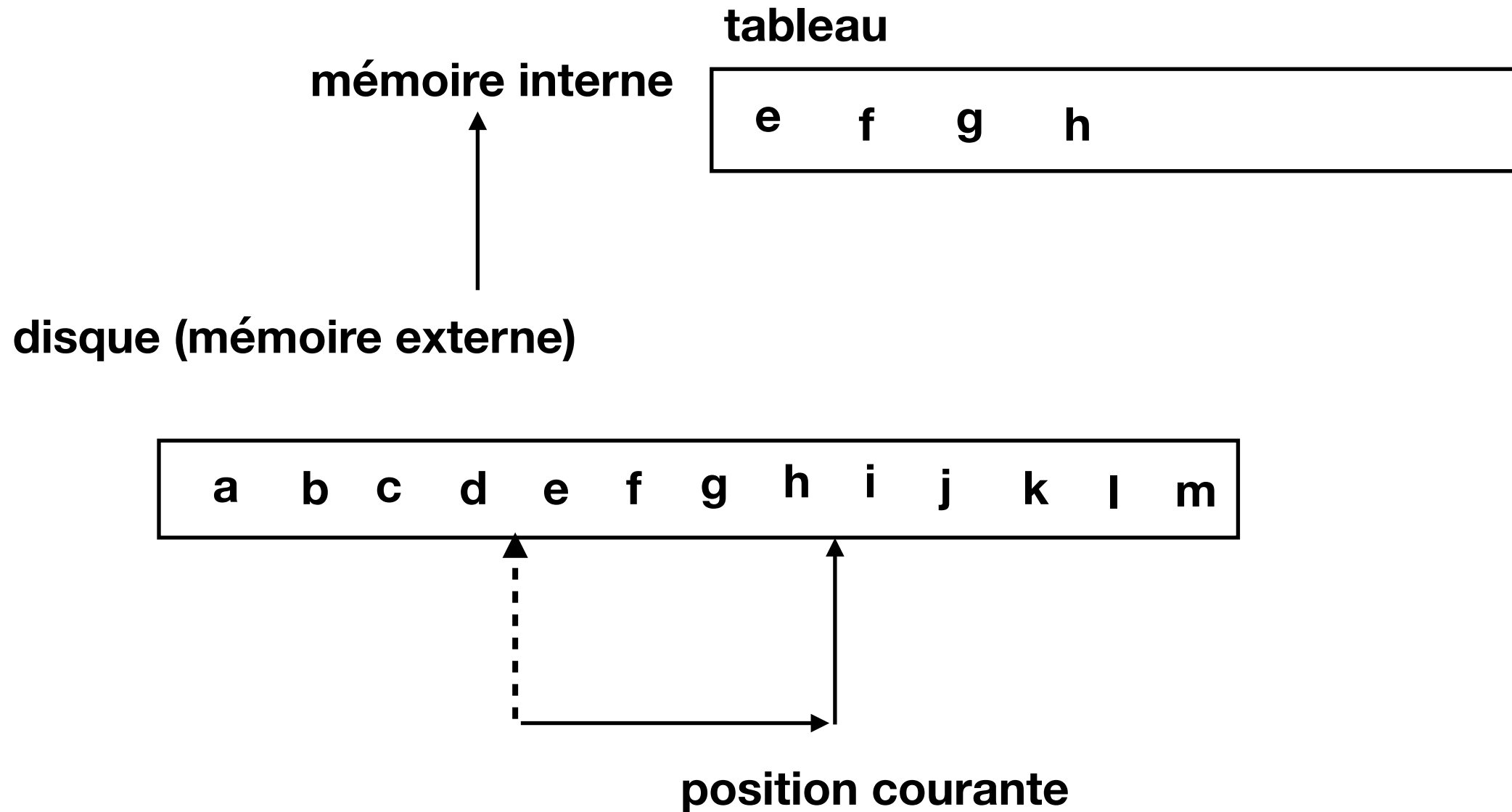
FILE *

Le fichier en C est juste une suite d'octets sans structure complémentaire. L'opération de **lecture** :



opération de lecture de 4 octets

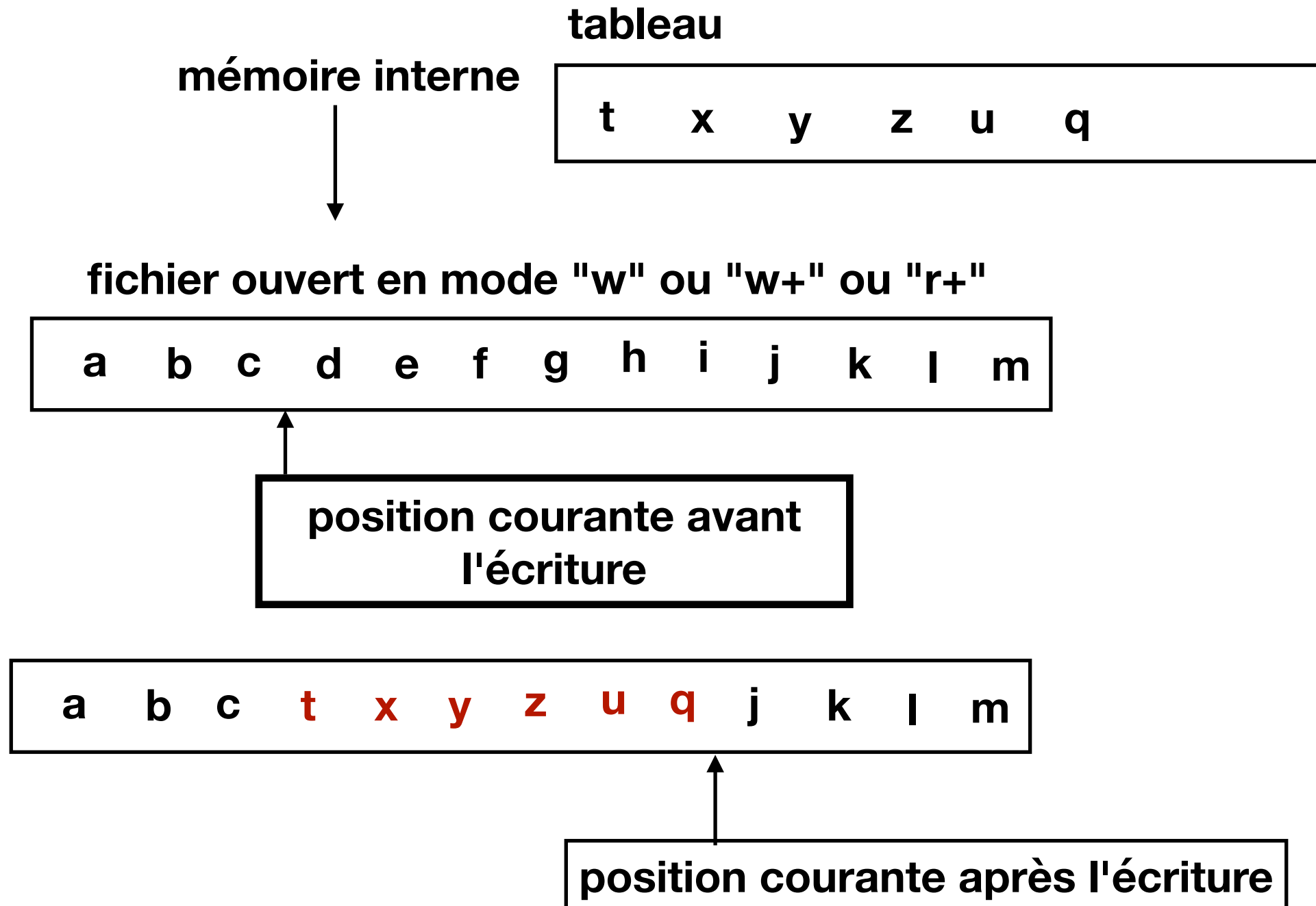
L'opération de lecture de 4 octets :



Chaque lecture déplace la position courante dans le fichier.

opération d'écriture dans le fichier

L'opération d'**écriture** de 6 octets :



position courante

Chaque opération de lecture ou d'écriture déplace la position courante dans le fichier.

Si le fichier est ouvert en mode "a" ou "a+" avant chaque l'opération d'écriture la position courante se déplace à la fin du fichier.

lecture caractère par caractère

`int fgetc(FILE *f)`

retourne le caractère lu. La fonction retourne EOF si la fin de fichier ou en cas d'erreur.

`int getc(FILE *f)` une macro-fonction, l'effet identique à `fgetc()`

`int getchar(void)` équivalent à `getc(stdin)`

Pour le traitement correcte de **EOF** et d'erreurs il faut déclarer comme `int` les variables qui reçoivent le résultats de ces fonctions.

```
int i;
```

```
while( ( i = fgetc( file ) ) != EOF ){
```

```
    /* traiter le caractère lu */
```

```
}
```

écriture caractère par caractère

```
int fputc(int c, FILE *f) ;
```

écrit le caractère c dans le flot, retourne c si OK et EOF en cas d'erreur.

```
int putc(int c, FILE *f) ;
```

 même chose mais implémentée comme une macro-fonction

```
int putchar(int c) ;
```

 équivalent à `putc(c, stdout)`

lecture d'une ligne

une ligne : une suite de caractères qui termine par le caractère '\n'

`char *fgets(char *s, int n, FILE *f, int *p)`

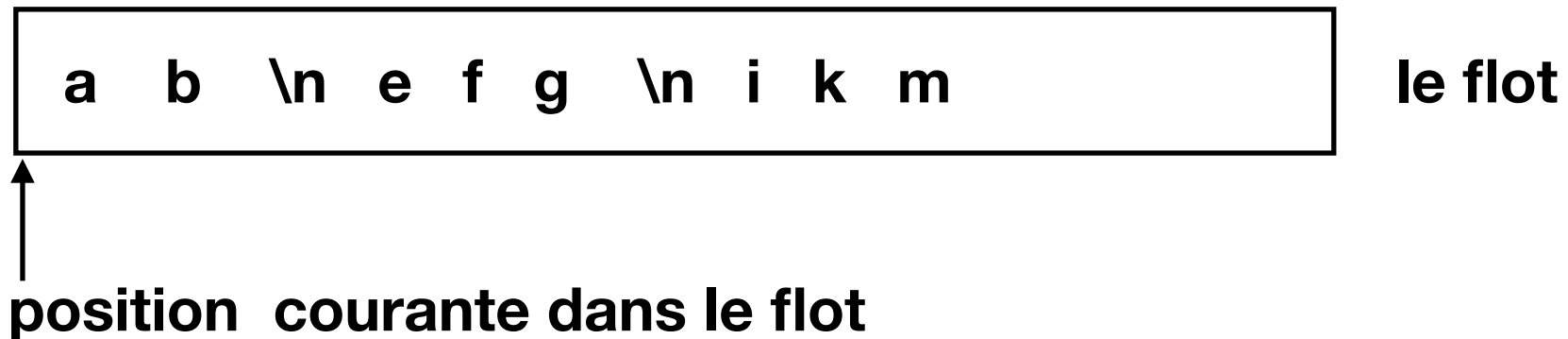
lit au plus $n-1$ caractères et les place à l'adresse `s`. La lecture s'arrête si la fonction rencontre le caractère '\n' qui sera aussi recopié dans `s`. La fonction place '\0' à la fin de la suite de caractères lus.

`fgets()` retourne `s` si tout est OK ou `NULL` si la fin de fichier ou en cas d'erreur.

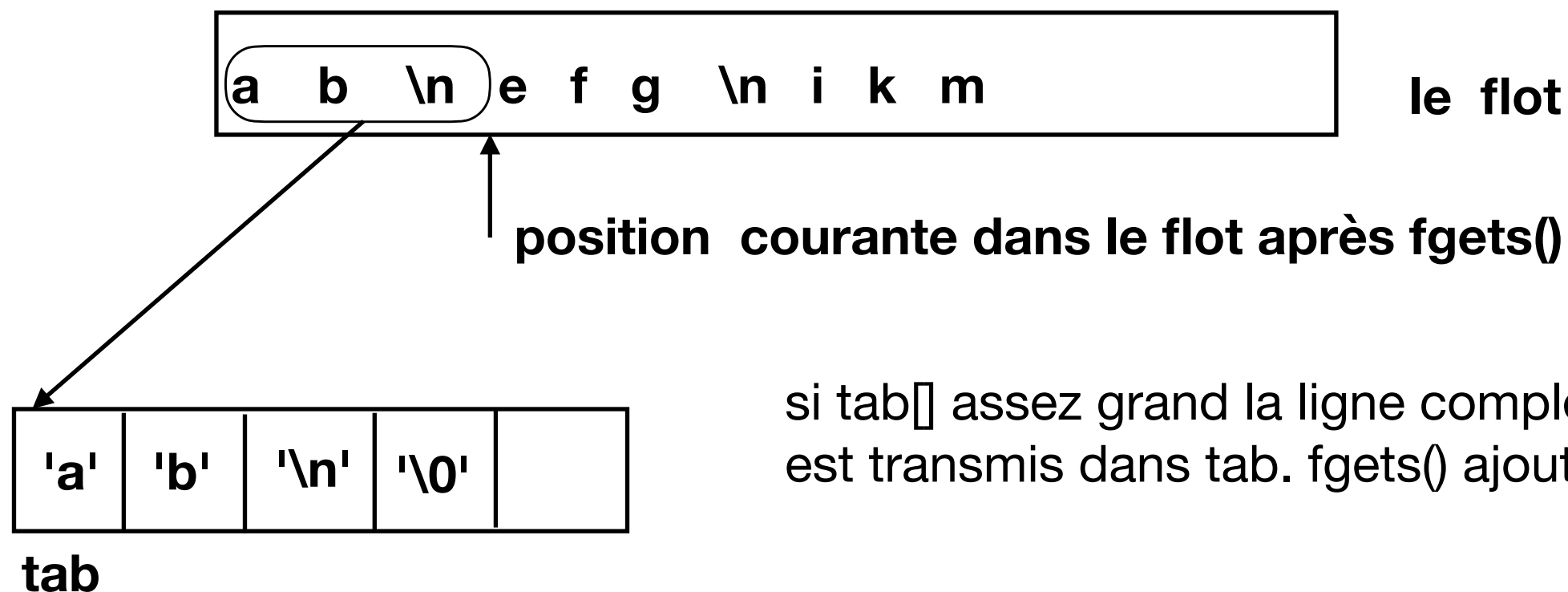
lecture d'une ligne

```
#define LEN 5
```

```
char tab[LEN];
```



```
fgets( tab, LEN, flot);
```



si `tab` assez grand la ligne complète, avec `'\n'` est transmis dans `tab`. `fgets()` ajoute `'\0'` à la fin

lecture d'une ligne

```
#define LEN 81
char tab[LEN];
FILE *fplot=fopen("fic.txt", "r");
if( fplot == NULL ){ perror("fopen"); return ...}
char *ligne = fgets( tab, LEN, fplot );
if(ligne == NULL){ // traiter erreur ou la fin de fichier
    }

size_t i = strlen(tab);

if( tab[i] == '\n' ){ //toute la ligne est lue
    //parce que le dernier caractère '\n'
}
else{ //tampon trop petit pour une ligne?
}
```

lecture d'une ligne

```
#define LEN 5
```

```
char tab[LEN];
```

a b c e f g \n i k m



position courante dans le flot

```
fgets( tab, LEN, flot);
```

a b c e f g \n i k m



position courante dans le flot après fgets()

tab

'a'	'b'	'c'	'e'	'\0'
-----	-----	-----	-----	------

tab trop petit, seulement 4 caractères lus dans tab[], caractère '\0' ajouté dans tab[]

fichier texte et le caractère nul

Le fichier texte ne contient jamais le caractère '\0'

écriture de chaînes de caractères dans un fichier

```
int fputs(const char *s, FILE *fput)
```

s pointeur vers une chaîne de caractère (qui termine avec '\0').

fputs () écrit les caractères de la chaîne pointée par s dans le flot (**sans jamais écrire le caractère '\0'**).

fputs () retourne un nombre non-négatif si OK et EOF en cas d'erreur.

fputs () n'est pas adapté pour écrire dans des fichiers binaires (qui peuvent contenir le caractère '\0'). fputs () est à utiliser uniquement avec les fichiers texte.

```
int puts( const char *str )
```

put écrit la chaîne pointé par str dans le flot stdout et écrit le caractère '\n' à la suite de str (contrairement à fputs() qui n'ajoute pas '\n' à suite de caractères écrits).

Sorties formatées

sorties formatées

```
int fprintf(FILE *fplot, const char *format, ...)
```

```
int printf(const char *format, ...)
```

écriture dans un fichier texte ou sur stdout, les deux fonctions retournent le nombre de caractères écrits

```
FILE * fplot = fopen("valeurs.txt", "r+");
```

```
int tab[ ]={ 3, -38 };
```

```
fprintf( fplot, "%5d_ %6d\n", tab[0], tab[1]);
```

écrira dans le flot la suite de caractères :

3_ -38\n

5 char 6 char

**notez que les nombres sont formatés en texte
fprintf() sert à écrire dans un fichier texte**

sorties formatées vers la mémoire

```
int  
snprintf( char *str, size_t size, char *format, ...)
```

écrire à l'adresse `str` (mémoire de taille `size`), des valeurs formatées selon le format. Au plus `size-1` caractères sont écrits et à la fin `snprint()` met `'\0'`

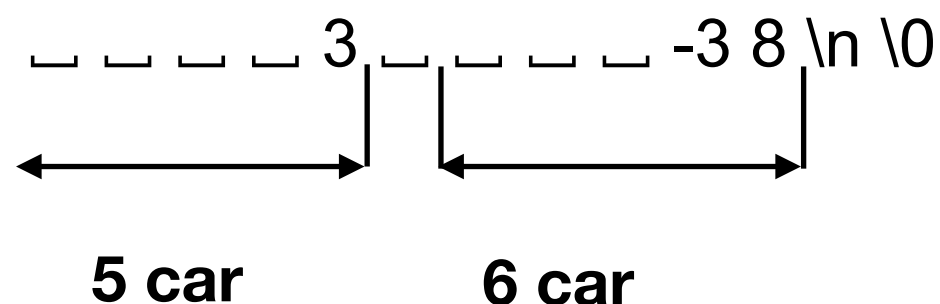
```
#define LEN 20
```

```
char tab[ LEN ];
```

```
int i = 3; int j = -38;
```

```
fprintf( tab, LEN, "%5d_ %6d\n", i, j );
```

écrira dans `tab[]` la suite de caractères :



sorties formatées - exemples

`%4d` un entier signé sur 4 caractères

```
int i=-21; printf("%4d", i);
```

_ -21

`%-6u` un entier non-signé sur 6 caractères aligné à gauche

```
int i=21; printf("%-6u", i);
```

21_ _ _ _

`%05d` un entier signé sur 5 caractères, complété à gauche par des zéros

```
int i = 24; printf( "%05d", i);
```

00024

sorties formatées avec largeur variable

```
int i=5;
```

```
int j = -21;
```

```
printf("j=%*d", i , j);
```

largeur du champ

valeur à écrire

j= -21

*** à la place de largeur du champ indique que la variable correspondante dans la liste donne la largeur du champs**

formats de printf

%c	int (char)
%d %l	int
%o %u %x	unsigned int
%ld %li	long int
%lo %lu %lx	unsigned long int
%hd %hi	short
%ho %hu %hx	unsigned short
%e %f %g	double
%s	char *
%p	void *
%lld	long long int
%zd	size_t
%td	ptrdiff_t
%hhd	signed char
%hhu	unsigned char

%o écrit en octal (base 8)

%x écrit en hexadécimal (base 16)

entrées formatées

entrées formatées

```
int fscanf(FILE *fplot, const char *format,...)
```

```
int scanf(const char *format, ...)
```

les fonctions lisent depuis un fichier texte, transforment le texte lu en une suite de valeurs selon le format.

`scanf(....)` est équivalent à `fscanf(stdin,)`

les fonctions retournent le nombres de "match" (les nombres de variables sur la liste ... dont la valeur est lue) ou **EOF** si fin de flot ou erreur.

entrées formatées

Comment fermer le flot `stdin` ouvert sur le terminal ?

Ctrl - D

entrées formatées

```
int fscanf(FILE *f, const char *format, ...)
```

```
FILE *f = fopen("toto.txt", "r");
```

```
int a, b, c;
```

```
int n = fscanf(f, "%d\n\n%d%d", &a, &b, &c);
```

Règles :

- les caractères blancs au début du flux sont ignorés,
- un caractère blanc dans le format (espace \t \n) correspond à une suite quelconque (peut-être vide) de caractères blancs dans le fichier
- tout autre caractère correspond à lui même

entrées formatées

```
FILE *fic = fopen("toto.txt" , "r");
```

```
int a, b, c;
```

```
int n = fscanf( fic, "%d_\n\n%d_%d", &a, &b, &c);
```

le fichiers toto.txt contient les caractères suivants :

```
_\n_-13_\n_\n3_\n_\nabc_\n_-56
```

a <-- -13 b <-- 3 mais la valeur de c ne change pas,

fscanf() retourne 2 (deux valeurs lues)

la position courante dans le fichier après ce fscanf est juste avant la lettre a

on essaie lire la troisième valeur :

```
n = fscanf( fic, "%d" , &c ); /* retourne 0, la position courante ne change pas */
```

```
n = fscanf( fic, "abc %d", &c); /* OK maintenant on réussit lire -56 dans la variable c */
```

entrées formatées

```
FILE *fic = fopen("toto.txt" , "r");
```

```
int a, b, c;
```

```
int n = fscanf( fic, "%d%d%*3s%d", &a, &b, &c);
```

le fichiers toto.txt contient

```
___\n_-13_____3_____abc____-56
```

le format `%*3s` : lire 3 caractères non blancs et `*` signifie qu'il faut les ignorer (on ne donne pas d'adresse pour les stocker)

format %s

%s -> lire jusqu'à une suite de caractères non blancs.

La lecture s'arrête quand on a rencontré un caractère non blanc.
Problème possible : débordement de tampon si on n'a pas assez places pour stocker ces caractères et le caractère '\0' ajouté à la fin

%20s -> lire jusqu'à 20 caractères non plans, la lecture s'arrête soit parce que on rencontre un caractère blanc soit parce que on a lu 20 caractères non blanc.

les formats pour scanf

%c

char *

%d %i

int *

%o %u %x

unsigned int *

%ld %li

long int *

%lo %lu %lx

unsigned long *

%hd %hl

short int *

%ho %hu %hx

unsigned short int *

%e %f %g

float *

%le %lf %lg

double *

%s

char *

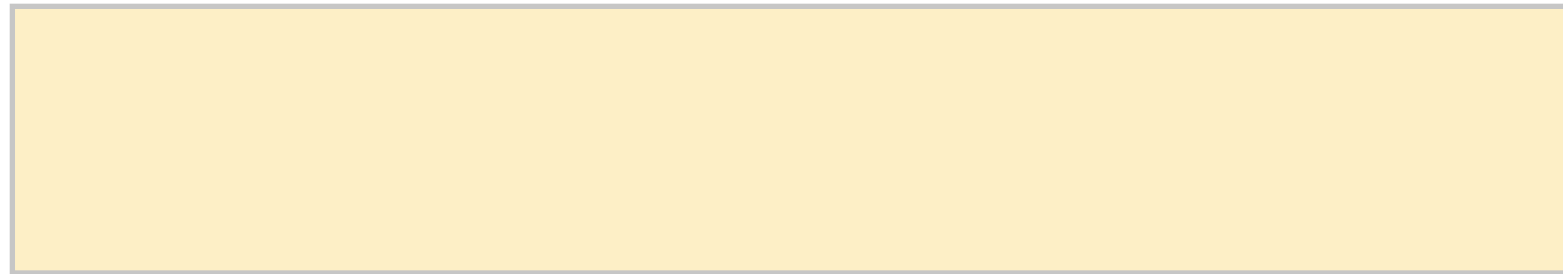
le tampon de flot
minimiser le nombre d'accès au disque dur

le tampon de flot

Les accès au disque dur prennent beaucoup de temps : l'exécution du programme est interrompue jusqu'à ce que les données soient écrites sur le disque ou lues depuis le disque. Pour minimiser le nombre d'accès au disque FILE contient un tampon (invisible pour l'utilisateur).

le tampon de FILE

le tampon de flot (dans l'objet FILE)



```
FILE * flot = fopen("toto.txt", "r");
```

```
char tab[5];
```

```
fgets( tab, 5, flot);
```

le fichier sur le disque

```
a b c d e q f g h \n 0 1 3 j \n
```

le tampon de FILE

le tampon de flot (dans la mémoire de processus)

a b c d e q f g h \n

fgets() met dans
le tampon plus de caractères
que ce que demande fgets()

a b c d \0

tab[]

```
FILE * flot = fopen("toto.txt", "r");
```

```
char tab[5];
```

```
fgets( tab, 5, flot);
```

le fichier sur le disque

a b c d e q f g h \n 0 1 3 j \n

fgets() met dans tab[] 5 caractères (si on compte '\0') mais le même fgets() met plus de caractères dans le tampon. La lecture suivante va s'effectuer depuis le tampon. Cela est transparent pour l'utilisateur.

le tampon de flot

De la même façon quand une fonction demande d'écrire dans un fichier l'écriture peut se faire dans le tampon de FILE et ensuite a un moment non-déterminé, par exemple, quand le tampon est plein le contenu du tampon sera écrit dans le fichier.

```
int fflush(FILE *fplot)
```

force l'écriture du tampon dans le fichier sur le disque.

`fflush(NULL)` provoque flush de tous les fichiers.

le tampon d'un flot

Chaque flot a un des trois modes de fonctionnement :

- non mémorisé (*unbuffered*) -- les octets sont transmis le plus tôt possible après chaque read/write. Le flot n'utilise pas de tampon
- pleinement mémorisé (*buffered*) -- les octets sont transmis par le bloc de la taille du tampon.
- mémorisé par ligne (*line buffered*) -- les octets stockés dans les tampon sont transmis vers le disque quand les caractère '`\n`' est envoyé dans le flot (ou quand le tampon est plein).

les modes de fonctionnement des flots

Dans le flot `stderr` est `unbuffered`.

Tous les autres flots sont

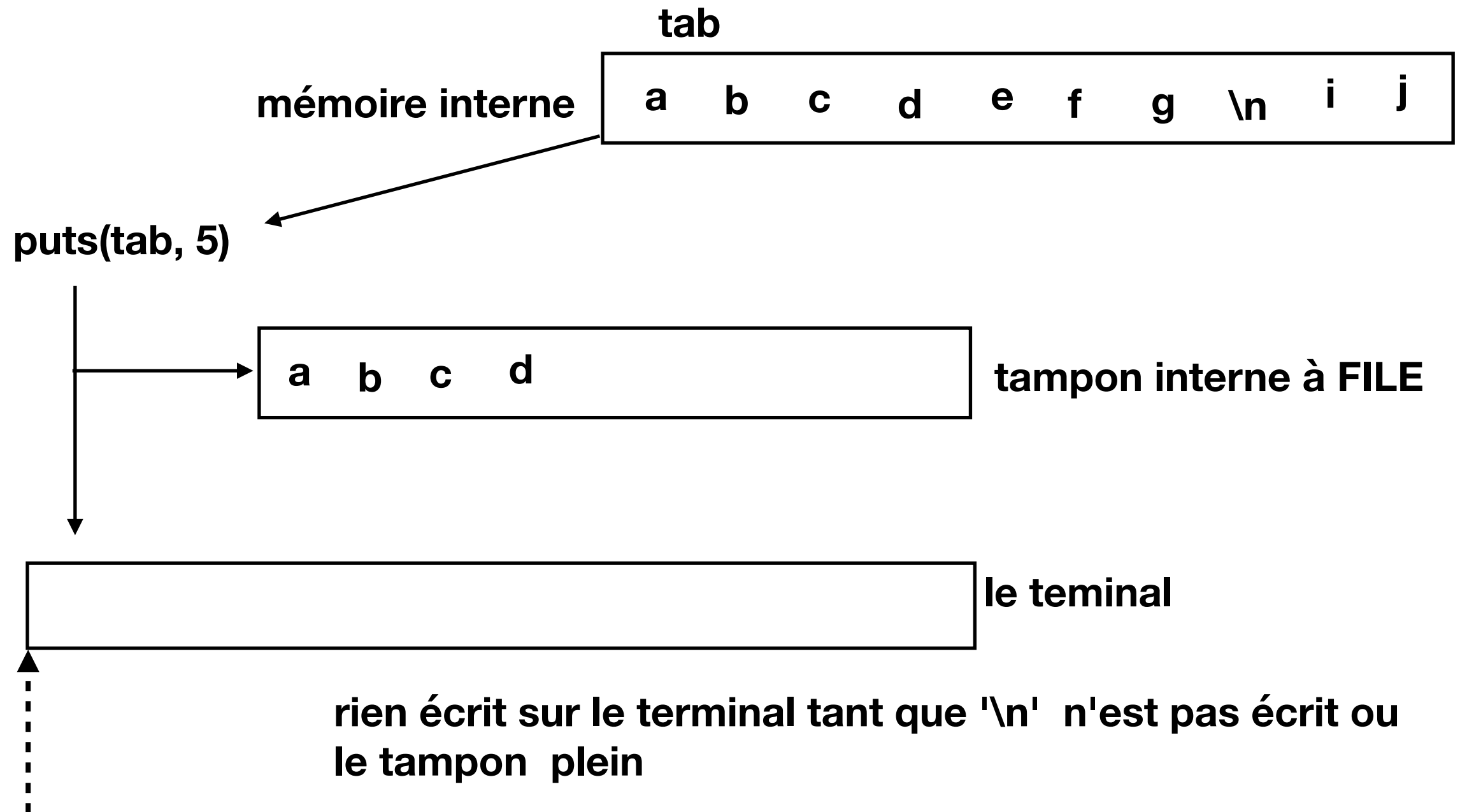
- " `line buffered` " pour les flots qui correspondent à un terminal et
- " `fully buffered` " pour le flots qui correspondent à un fichier.

Pour les flots ouvert en écriture le tampon de `FILE` est écrit sur le disque quand :

- `fclose(flot)` ferme le flot,
- le programme termine.

exemple fputs() de 4 octets

L'opération d'écriture de 4 octets, mode line buffered :



ancienne position

position courante

contrôle de tampon d'un flot

Un flot qui n'est pas associé à une entrée/sortie interactive (terminal/clavier) s'ouvre par défaut en mode pleinement mémorisé (*fully buffered*).

Pour changer le mode :

```
int setvbuf(FILE *flot, char *buf, int mode, size_t t)
```

buf – si non NULL le tampon à utiliser, t – la taille du tampon

mode :

- `_IOFBF` fully buffered
- `_IOLBF` line buffered
- `_IONBF` unbuffered

`void setbuf(FILE *flot, char *buf)` forme simplifiée :

`setbuf(flot, NULL)` met le flot en mode unbuffered, par exemple

`setbuf(stdout, NULL)` met le flot de sortie standard en mode non mémorisé

tout ce qui est écrit dans stdout est envoyé immédiatement sur l'écran

Détecter erreur pour une
opération lecture/écriture

Détecter fin de flot en lecture

l'indicateur de fin de flot et indicateur d'erreur

Souvent les fonctions d'entrée/sortie retournent la même valeur pour deux raisons différentes :

- à la fin du flot (par exemple pour signaler la fin de fichier en lecture) et
- en cas d'erreur de lecture/écriture.

Pour distinguer la fin de flot et l'erreur le flot possède deux indicateurs : indicateur de fin de fichier et indicateur d'erreur.

```
int feof(FILE *fplot)
```

```
int ferror(FILE *fplot)
```

retournent une valeur différente de 0 si l'indicateur correspondant est positionné.

Une fois activé l'indicateur reste dans l'état activé.

Pour mettre les deux indicateurs à 0 on utilise :

```
void clearerr(FILE *fplot)
```

l'indicateur de fin de flot et indicateur d'erreur

```
FILE *flot;
flot = fopen(nom_fichier, "r");
#define T 124
char in[ T ];
while( fgets(in, T, flot) != NULL ){
    /* faire le traitement de caractères lus */
}
// vérifier si fin de flot ou erreur
if( ferror( flot ) ){
    /* traiter l'erreur de la lecture */
}
if( feof( flot ) ){
    // traiter fin du fichier
}
clearerr( flot );
```


Fichiers temporaires

fichiers temporaires

`FILE *tmpfile(void)`

ouvre un flot temporaire en mode "w+". Le flot sera supprimé quand on le ferme avec `fclose()` ou quand le programme termine avec `exit()` ou `return` de `main`.

(Si le programme est tué par un signal il se peut que le fichier temporaire obtenu avec `tmpfile()` ne soit pas supprimé automatiquement.)

`char *tmpnam(char *s)`

s pointeur vers un tableau d'au moins `L_tmpnam` caractères.

`tmpnam()` retourne un pointeur vers un nom unique de fichier.

Le nom retourné par `tmpnam()` sera utilisé comme un nom de fichier, le système garantie que le fichier qui porte ce nom n'existe pas au moment de l'appel à `tmpnam()`