Algorithmique (en C)

Julien Alexandre dit Sandretto

Department U2IS
ENSTA ParisTech
Version 2019-2020

ENSTA

IP PARIS

Listes chaînées

Structure de liste

- Une **liste** c'est :
  - une **liste vide**
  - **ou** un élément suivi d'une **liste**.
- ⇒ Type inductif : list : := `Nil` | elem ; list
- En C, pas de types inductifs ⇒ encodage à base de pointeurs :
  - On forme une «*chaîne* ».
  - Chaque élément est un **maillon** de la chaîne.
  - On passe d'un élément au suivant (précédent) en suivant un **pointeur**.

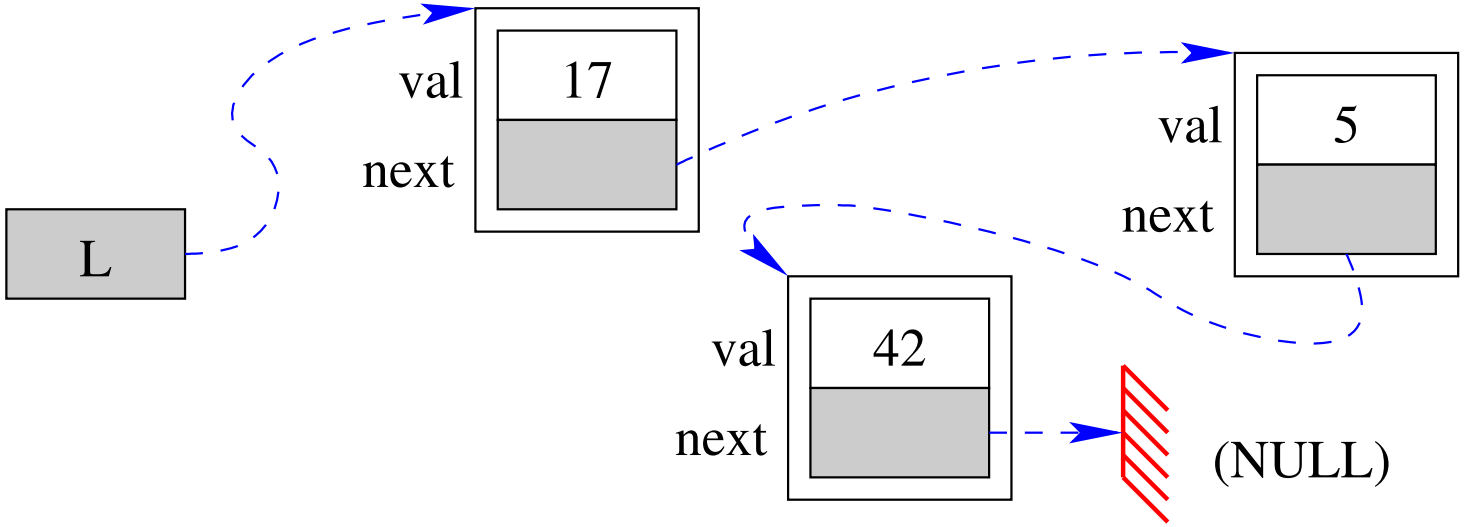
```
struct list {
  int data ;      // Le contenu n'est pas forcément un int.
  struct list *next ; // Pointeur vers l'élément suivant (ou
      NULL).
};
```

- On ne sait pas **efficacement** accéder au $i^{\text{ème}}$ élément.
- On «*tient* » une liste par le **pointeur vers son premier élément**.

Le champ `next` du dernier élément pointe vers `NULL`.

Organisation mémoire

- En mémoire, éléments **dispersés** $\Rightarrow \neq$ tableau où éléments contigus.
- Impossible d'accéder directement au $i^{\text{ème}}$ élément.
- Nécessité de suivre toute la chaîne (parcourir la liste).
- $\Rightarrow$ Toujours garder accès à la tête de la liste.



Intérêts de listes (1)

- Liste [1 ; 5 ; 6]
  - **tête** (« *head* ») = 1 de type *elem*.
  - **queue** (« *tail* ») = [5 ; 6] de type *elem list*.
- Nombreuses opérations en temps constant ($\Theta(1)$).
  - Insérer un élément au **début** (« *cons* »).
  - Supprimer le **premier** élément (la « *tête* »).
  - Avancer à l'élément **suivant**.
  - Insérer un élément **après** un élément **dont on connaît l'adresse.**
  - Supprimer l'élément **suivant un élément dont on connaît l'adresse**.

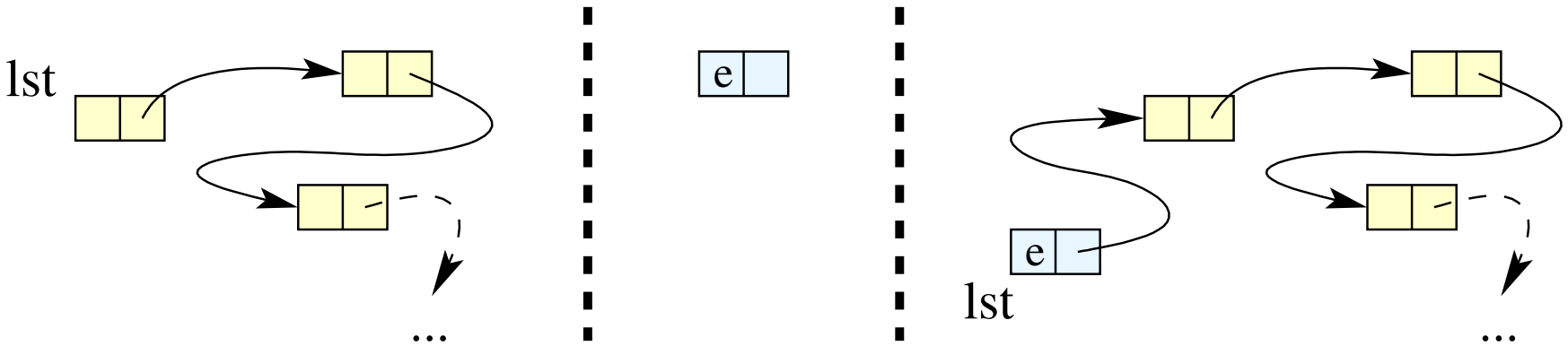
Intérêts de listes (2)

- D'autres opérations pour pas cher, juste 1 ou 2 changements de pointeurs :
  - Concaténer 2 listes (« *append* »).
  - Construire 2 listes partageant la même « *queue* ».
    - **Attention** à ne pas faire de double-libération de mémoire !
  - Échanger les « *queues* » de 2 listes...
- Structure facilement **re-dimensionnable** (extension, réduction).
- Accès « direct » plus cher que pour les tableaux...
- Mais dimensionnement **dynamique** plus facile !

Implémentation en C : ajout en tête (« *cons* »)

- `cons : elem → list → list`

```
struct list* cons (int elem, struct list *lst)
{
  struct list *tmp = malloc (sizeof (struct list)) ;
  if (tmp == NULL) return (NULL) ;
  tmp->data = elem ;
  tmp->next = lst ;
  return (tmp) ;
}
```



Ajout en tête : les trucs à ne **pas** faire !

```
struct list* cons (int elem, struct list *lst)
{
  struct list *tmp ; /* Là il faut faire un malloc ! */
  tmp->data = elem ;
  tmp->next = lst ;
  return (tmp) ;
}
```

- Avoir un pointeur **ne signifie pas** avoir de la mémoire !
- Il faut **l'initialiser** avec une adresse de zone **allouée**.

```
struct list* cons (int elem, struct list *lst)
{
  struct list tmp ; /* Attention, c'est une variable locale ! */
  tmp.data = elem ;
  tmp.next = lst ;
  return (&tmp) ;
}
```

- Retourne l'adresse d'une variable **locale** qui meurt à la fin de la fonction !

Ajout en tête : les trucs à ne **pas** faire également !

```
struct list cons (int elem, struct list* lst)
{
  ...
}
```

ou

```
struct list cons (int elem, struct list lst)
{
  ...
}
```

- . . . ne marchent **pas** non plus !
- Écrasement mémoire, adresse de paramètre (qui meurt à la fin de la fonction), et plus si affinités . . .

Implémentation en C : impression (1)

- print : list → void

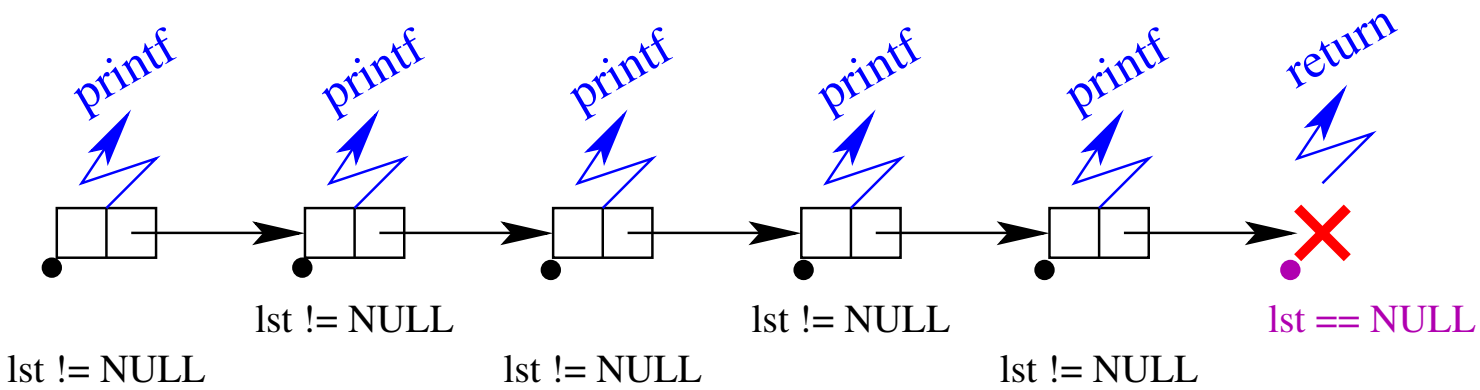
```
void print (struct list *lst)
{
  printf ("[") ;
  while (lst != NULL) {
    printf (" %d", lst->data) ;
    lst = lst->next ;
  }
  printf (" ]") ;
}
```

- Écriture (plus simple, mais plus coûteuse en pile si compilé naïvement) en récursif :

```
void rec_print (struct list *lst)
{
  if (lst != NULL) {
    printf (" %d", lst->data) ;
    rec_print (lst->next) ;
  }
}
```

Implémentation en C : impression (2)

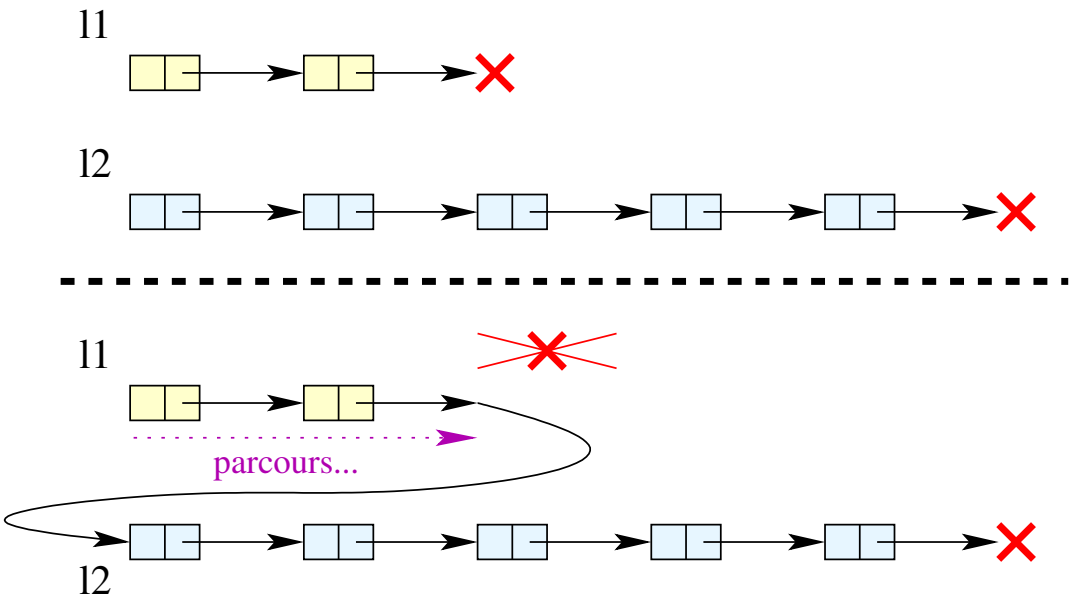
```
void print (struct list *lst)
{
  printf ("[") ;
  while (lst != NULL) {
    printf (" %d", lst->data) ;
    lst = lst->next ;
  }
  printf (" ]") ;
}
```



Implémentation en C : concaténation (« *append* »)

- append : `list` → `list` → `list`

```
struct list* append (struct list *l1, struct list *l2)
{
  struct list *tmp = l1 ;
  if (tmp == NULL) return (l2) ;
  while (tmp->next != NULL) tmp = tmp->next ;
  tmp->next = l2 ;
  return (l1) ;
}
```



Implémentation en C : libération

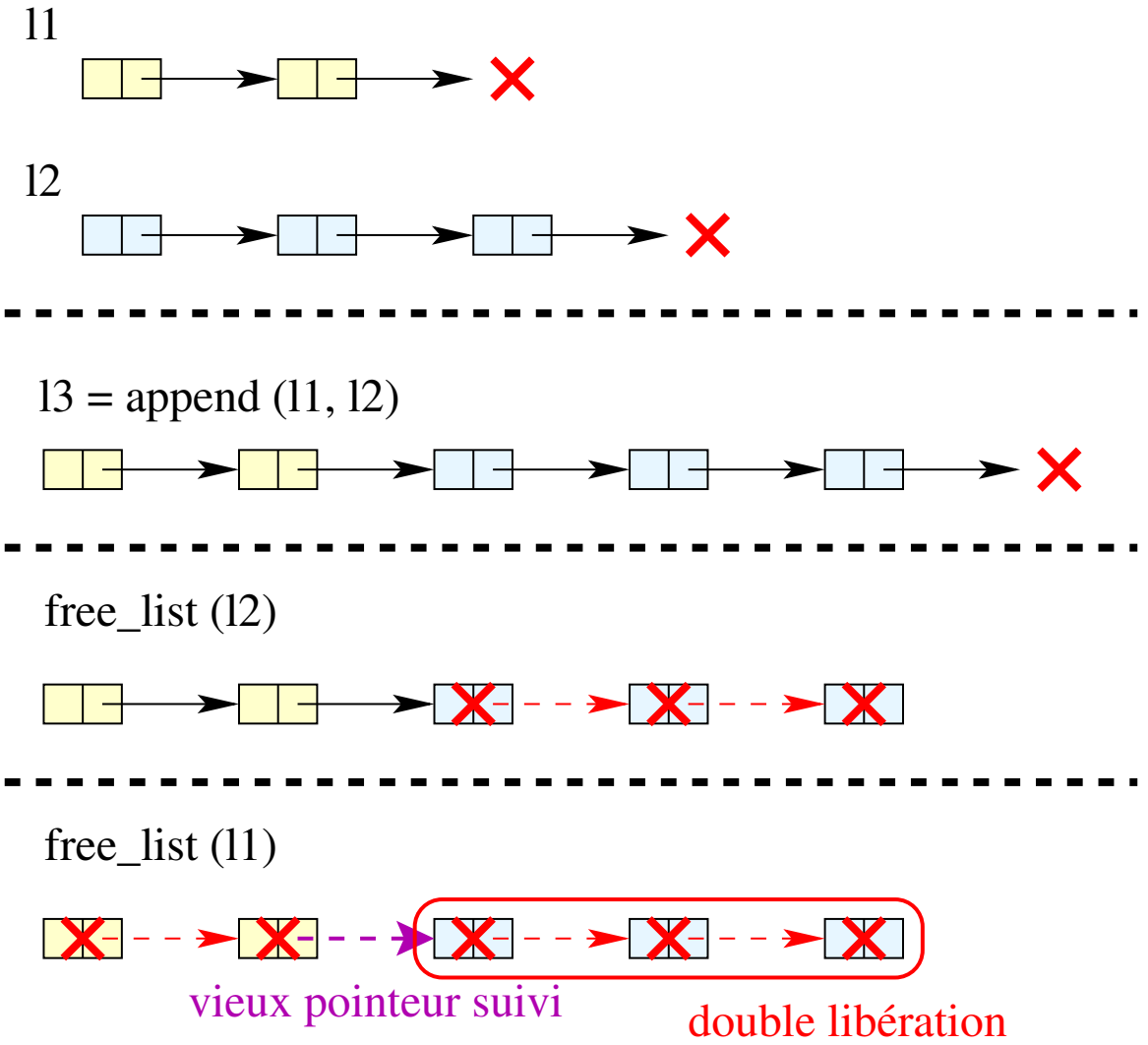
- `free_list` : `list` → `void`

```
void free_list (struct list *lst)
{
  struct list *tmp ;
  while (lst != NULL) {
    tmp = lst->next ;
    free (lst) ;
    lst = tmp ;
  }
}
```

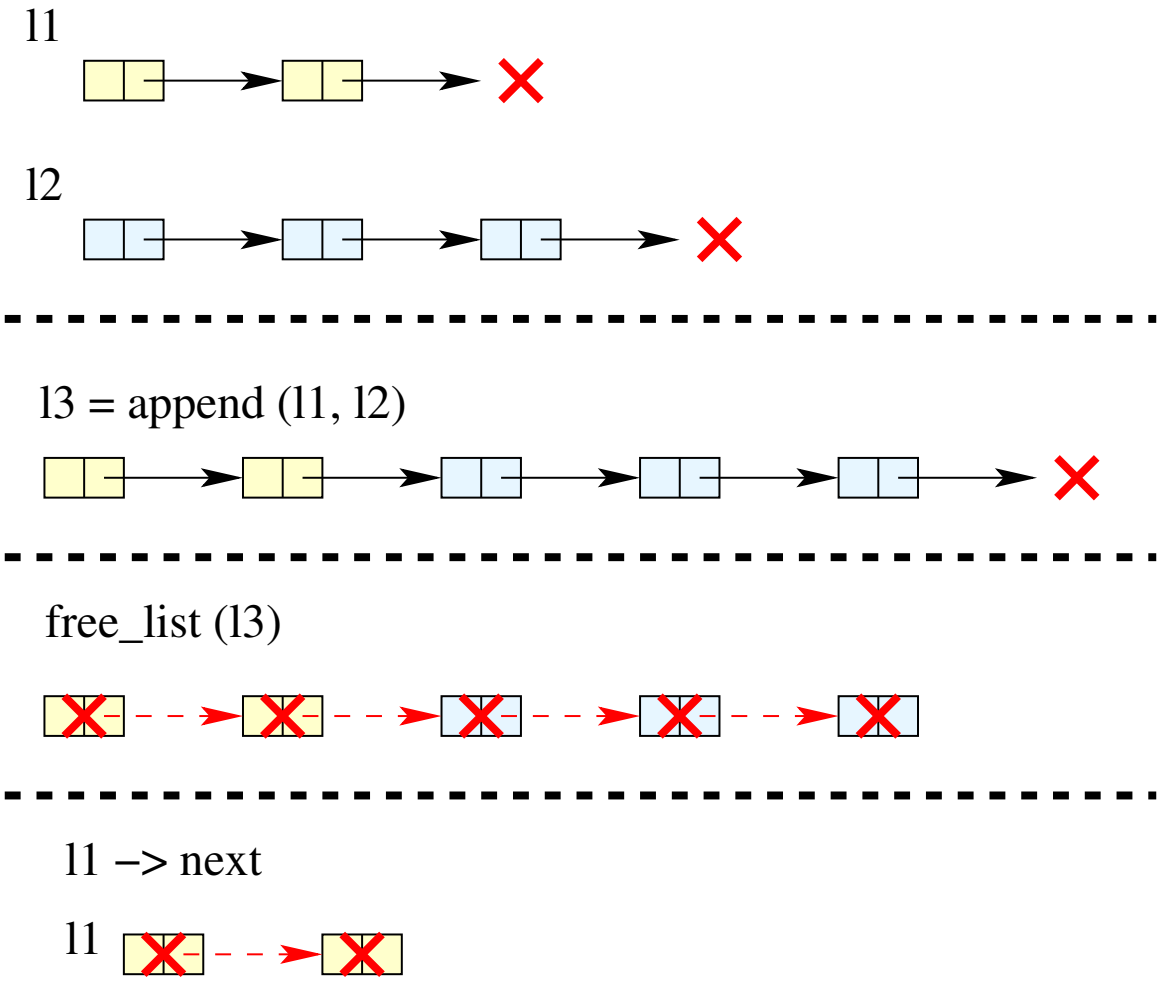
- Libération (plus simple, mais plus coûteuse en pile) en récursif :

```
void rec_free_list (struct list *lst)
{
  if (lst != NULL) {
    rec_free_list (lst->next) ;
    free (lst) ;
  }
}
```

Implémentation en C : double libération



Implémentation en C : libération anticipée



Variations sur les listes

- Beaucoup de variantes :
  - Doublement chaînées (pointeur `prev`) pour reculer.
  - Listes circulaires : dernier élément pointe sur le premier.
  - Listes circulaires doublement chaînées ...
- La liste peut être plus que le pointeur vers son premier élément :
  - Peut contenir le nombre d'éléments.
  - Peut contenir le pointeur vers le dernier élément.
  - ...

```
struct list {
  int nb_elems ;
  struct cell *head ;
  struct cell *last ;
};
```

```
struct cell {
 int data1 ;
 int data2 ;
 struct cell *next ;
};
```

- En fait, listes = cas «*simple*» d'**arbres**.

Quelques utilitaires en C

Les fichiers

- Fichier : stockage **persistant** d'informations.
- Sous Unix, notion plus étendue : vue **utilisateur** de n'importe quel *« périphérique »* :
  - disques (dur, USB, réseaux),
  - souris, clavier, terminaux virtuels, ...
  - systèmes de communication (pipe, socket),
  - espace mémoire de chaque processus, etc...
- Opérations :
  - Charger le contenu (tout ou partie) en mémoire.
  - Écrire des données de la mémoire dans un fichier.
  - Effacer un fichier.
  - Demander la taille du fichier.
  - Se positionner dans le fichier, etc...
- Comme toute ressource, soumis à une **demande** puis une **restitution**.
- Nécessite `#include <stdio.h>`
- Manipulé via un **pointeur** sur **descripteur** `FILE` : `FILE*`.

Ouverture

- **Ouverture** : `FILE* fopen (char *filename, char *mode);`
  - **filename** : chaîne représentant le chemin du fichier.
  - **mode** : chaîne spécifiant si ouvert en lecture et/ou écriture.
    - "r" : (read) ouverture en lecture (fichier déjà existant)
    - "w" : (write) ouverture en écriture (destruction si existant)
    - "a" : (append) écriture, rajoute à la fin, (création si non existant)
    - "r+", "w+", "a+" : variantes (c.f. `man 3 fopen`).
    - Suffixe "b" : (binary) lire/écrire les données sans transformation.
  - Succès si le pointeur retourné $\neq$ `NULL`.

```
#include <stdio.h>

int main ()
{
  FILE *in ;
  in = fopen ("/tmp/input", "rb") ; /* Lecture, données brutes. */
  if (in == NULL) return (-1) ;      /* Ouverture échouée ? */
  ...
```

Fermeture

- `int fclose(FILE *stream);`
  - **stream** : pointeur sur le descripteur du fichier.
- Succès si la valeur retournée = 0.
- Fermeture importante : garantit l'**écriture effective** des données.

```c
#include <stdio.h>

int main ()
{
  FILE *in ;
  in = fopen ("/tmp/input", "rb") ; /* Lecture, données brutes. */
  if (in == NULL) return (-1) ;     /* Ouverture échouée ? */
  fclose (in) ;                     /* Fermeture. */
  return (0) ;
}
```

Lecture

- Lire fait avancer un « *repère* » implicite dans le fichier.
- Lecture **formatée** :
  `int fscanf (FILE *stream, char *format, ...);`
  - **stream** : pointeur sur le descripteur du fichier.
  - **format** : format « à la » `scanf/printf`.
  - **...** : pointeurs vers les variables où stocker ce qui est lu.
- Retourne le **nombre d'items** lus.
- Lecture **par « blocs »** :
  `size_t fread (void *ptr, size_t size, size_t nitems, FILE *stream);`
  - **ptr** : pointeur vers la zone mémoire où stocker ce qui est lu.
  - **size** : taille d'un « bloc » à lire.
  - **nitems** : nombre de « blocs » à lire.
  - **stream** : pointeur sur le descripteur du fichier.
- Retourne le **nombre de « blocs »** lus.

Exemple de lectures

```c
#include <stdio.h>
int main () {
  FILE *in ;
  int i1, i2 ;
  char buffer [256] ;
  in = fopen ("/tmp/myfile", "rb") ;
  if (in == NULL) return (-1) ;
  fscanf (in, "%d %s %d", &i1, buffer, &i2) ;
  printf ("Read: %d %s %d\n", i1, buffer, i2) ;
  fread (buffer,sizeof (int), 1, in) ;
  printf ("Read: %x\n", ((int*) buffer)[0]) ;
  fclose (in) ;
  return (0) ;
}
```

Fichier `/tmp/myfile`

45␣Glop␣16↵
abcd

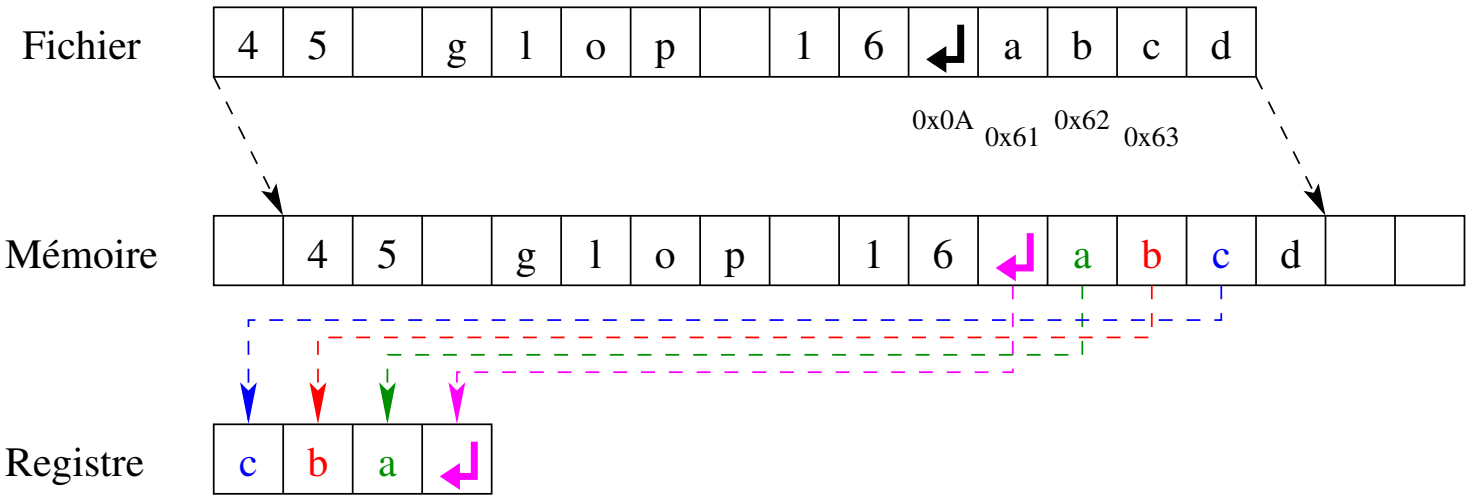
```
mymac:/tmp/$
./a.out
Read: 45 Glop 16
Read: 6362610a
```

- 0x63 → 'c', 0x62 → 'b', 0x61 → 'a', 0x0A → '↵'
- Montre l'ordre des **octets** d'un `int` **en mémoire**.

Ordre des octets : « endianness », « boutisme »

```
mymac:/tmp/$
./a.out
Read: 45 Glop 16
Read: 6362610a
```

- 0x63 → 'c', 0x62 → 'b', 0x61 → 'a', 0x0A → '↵'



- Processeur Intel Core i7 : « little endian ».
- Octet de poids faible → adresse de poids faible en mémoire.
- Autres processeurs autres mœurs (« big/middle/bi-endian »).

Écriture (dual de la lecture)

- Écrire fait avancer le (même) « *repère* » implicite dans le fichier.
- Écriture **formatée** :
  `int fprintf(FILE *stream, char *format, ...);`
  - **stream** : pointeur sur le descripteur du fichier.
  - **format** : format « à la » `scanf/printf`.
  - **...** : variables desquelles imprimer la valeur.
- Retourne le **nombre de caractères** écrits.
- Écriture **par « blocs »** :
  `size_t fwrite (void *ptr, size_t size, size_t nitems, FILE *stream);`
  - **ptr** : pointeur vers la zone mémoire où trouver ce qui est à écrire.
  - **size** : taille d'un « bloc » à écrire.
  - **nitems** : nombre de « blocs » à écrire.
  - **stream** : pointeur sur le descripteur du fichier.
- Retourne le **nombre de « blocs »** écrits.

Exemple d'écritures

```c
#include <stdio.h>

int main ()
{
  FILE *out ;
  int i = 0x01020304 ;
  out = fopen ("/tmp/myfile", "wb") ;
  if (out == NULL) return (-1) ;
  fprintf (out, "%d %s %d", 23, "Glop", 57) ;
  fwrite (&i,sizeof (int), 1, out) ;
  fclose (out) ;
  return (0) ;
}
```

```
mymac:/tmp$ hexdump -C myfile
00000000 32 33 20 47 6c 6f 70 20 35 37 04 03 02 01        |23 Glop 57....|
0000000e
```

- 0x01020304 ≠ 04 03 02 01

Montre (encore) l'ordre des **octets** d'un `int` **en mémoire**.

Fin de fichier

- Lire c'est bien, mais quand s'**arrêter** ?

⇒ Besoin d'une fonction testant l'atteinte de **fin de fichier**.

- `int feof (FILE *stream);`
- Retour : $\neq 0$ si fin de fichier atteinte **suite** aux lectures **précédentes**.

⇒ ⚠ Toujours avoir **lu avant de tester** avec `feof`.

```c
#include <stdio.h>
int main () { /* INCORRECT */
  FILE *in ;
  int i ;
  in = fopen ("myfile", "rb") ;
  if (in == NULL) return (-1) ;
  while (!feof (in)) {
    fscanf (in, "%d", &i) ;
    printf ("%d\n", i) ;
  }
  fclose (in) ;
  ...
```

```c
#include <stdio.h>
int main () {/ * CORRECT */
  FILE *in ;
  int i ;
  in = fopen ("myfile", "rb") ;
  if (in == NULL) return (-1) ;
  fscanf (in, "%d", &i) ;
  while (!feof (in)) {
    printf ("%d\n", i) ;
    fscanf (in, "%d", &i) ;
  }
  fclose (in) ;
  ...
```

Position actuelle dans un fichier

- Connaître la position actuelle (lect/écr) dans un fichier :
  `long ftell (FILE *stream);`
- Se déplacer (**sans lire**) dans un fichier :
  `int fseek (FILE *stream, long offset, int whence);`
  - **offset** : déplacement en nombre d'**octets**.
  - **whence** : repère de déplacement :
    - `SEEK_SET` : depuis le **début** du fichier,
    - `SEEK_CUR` : depuis la **position courante** dans le fichier,
    - `SEEK_END` : depuis la **fin** du fichier.
- Utilisation naturelle : trouver la **longueur** d'un fichier :
  - Aller à la fin et demander la position courante.

  ⇒ `fseek (myfile 0, SEEK_END)` puis `ftell (myfile)`.

Les trucs à ne **pas** faire !

- Inutile de connaître la taille d'un fichier pour lire dedans.
- Lire à reculons (typique d'une utilisation de `fseek` sans retour au début du fichier).
- Lire après la fin du fichier.
- Lire dans un fichier ouvert en écriture (et inversement).