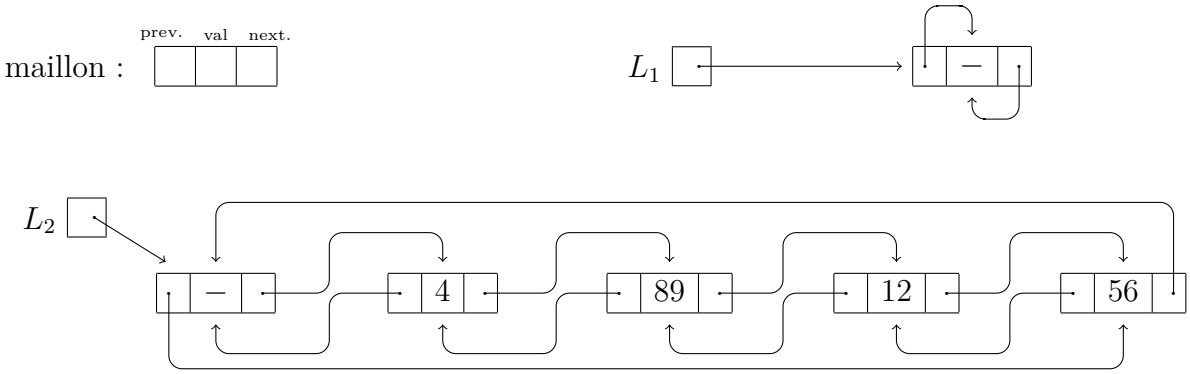
Programmation C
TP n° 8 : Listes doublement chaînées circulaires

On considère ici des listes pour stocker des entiers. Un maillon (ou élément) de la liste est définie de la manière suivante :

```
typedef struct element element;
struct element {
  int val;
  element *next;
  element *previous;
};
```

Comme d'habitude, le champ `val` sert à stocker les valeurs entières de la liste. Le champ `next` sert à indiquer l'adresse du maillon *suivant* de la liste, et le champ `previous` contient l'adresse du maillon précédent. On parle donc ici de liste *doublement chaînées*.
Pour simplifier les algorithmes, on supposera que le premier maillon d'une liste ne contient pas de valeur. Dans l'implémentation, ce maillon aura toujours un champ `val`, mais nous ferons comme s'il n'existait pas. Une liste vide sera donc réduite à cet unique maillon [1] que l'on nommera la *tête de liste* (ou maillon de tête). Dans la suite, on appelle liste un pointeur sur la tête de liste.
Pour finir, on supposera que notre liste est *circulaire*, c'est à dire que le champ `next` du dernier maillon pointe sur le maillon de tête et que le champ `previous` du maillon de tête pointe sur le dernier maillon de la liste.
Dans la figure, ci-dessous, on représente la liste vide L_1 et la liste $L_2 = [4; 89; 12; 56]$:



On peut remarquer que si `p` contient l'adresse d'un maillon quelconque d'une telle liste, alors on a toujours : `p->next->previous == p->previous->next == p`.
Pensez à tester chacune de vos fonctions.

Exercice 1 : Constructeur de listes

1. Écrire une fonction `element *cons_list()` qui renvoie une liste vide, c'est-à-dire une adresse vers un maillon de tête (à allouer avec `malloc` !) qui code une liste vide.
2. Écrire une fonction `void add_fst_list(element * L,int v)` qui ajoute la valeur `v` au début de la liste `L`. On suppose que `L` pointe vers le maillon de tête d'une liste correctement formée et on souhaite donc ajouter un maillon contenant la valeur `v` après le maillon de tête.

1. La liste vide n'est donc pas le pointeur `NULL` !

3. Écrire une fonction `void add_lst_list(element * L,int v)` qui ajoute à la liste `L` un maillon contenant la valeur `v` à la dernière position.

Est-ce que ces opérations sont efficaces ? Comparer avec les mêmes opérations sur une liste simplement chaînée.

Exercice 2 : Manipulation de listes

1. Écrire une fonction `int isempty_list(element * L)` qui renvoie 1 si la liste `L` est vide et 0 sinon.
2. Écrire une fonction `int len_list(element * L)` qui renvoie la longueur de la liste `L` (c'est-à-dire le nombre de valeurs entières stockées ; `len_list(cons_list())` renvera donc `0`).
3. Écrire une fonction `void print_list(element * L)` qui affiche les entiers stockés dans la liste `L`.
4. Écrire un `int main(int argc, char *argv[])` qui permet de passer des entiers en argument depuis le shell et insère ces entiers dans une liste. On rappelle que `argc` représente le nombre d'arguments passés, incluant toujours `argv[0]` qui est le nom utilisé pour exécuter le programme (les autres arguments sont donc `argv[1]`... `argv[argc-1]`). Enfin, on rappelle que `atoi` permet de convertir une chaîne de caractères en un entier de valeur correspondante.

Exercice 3 : Destruction et copie de listes

1. Écrire une fonction `int del_fst_list(element * L)` qui supprime le premier élément de la liste `L` (et affiche un message d'erreur si la liste est vide) et renvoie la valeur entière qui y était stocké (on n'oubliera pas de désallouer la zone mémoire devenues inutile !).
2. Écrire une fonction `int del_lst_list(element * L)` qui supprime le dernier élément de la liste `L` (et affiche un message d'erreur si la liste est vide) et renvoie la valeur entière qui y était stockée.
3. Écrire une fonction `void free_list(element * L)` qui libère toute la mémoire utilisée par les maillons de la liste `L`.
4. Écrire une fonction `element * copy_list(element * L)` qui fait une copie complète de la liste `L` et renvoie un pointeur vers le maillon de tête de cette copie.

On peut observer que nos listes munies des opérations `add_fst_list`, `del_fst_list`, `isempty_list` et `cons_list` correspondent à des structures LIFO (pile), et lorsqu'on les utilise avec `add_fst_list`, `del_lst_list`, `isempty_list` et `cons_list`, nous obtenons des structure FIFO (file).

Exercice 4 : Modifications locales

1. Écrire une fonction `void insert_prev_list(element * p, int v)` qui insert un maillon contenant la valeur `v` avant le maillon pointé par `p`. On ne suppose plus ici que `p` pointe vers un maillon de tête. Avez-vous déjà rencontré cette fonction ?
2. Idem pour `void insert_next_list(element * p, int v)` qui insert `v` après `p`.
3. Idem pour `void del_list(element * p)` qui détruit le maillon pointé par `p`.

NB : c'est grâce au double chaînage que ces opérations peuvent toutes se réaliser facilement !

Exercice 5 : Toujours plus

1. Écrire une fonction `element* get_elet_list(element* L, int v)` qui cherche si la valeur `v` est contenue dans la liste `L`. Si c'est le cas, la fonction retourne un pointeur vers un maillon contenant la valeur `v` (et `NULL` sinon).
2. Écrire une fonction `void concat_list(element * l1, element * l2)` qui insère la liste `l2` en tête de la liste `l1` (sans réallouer de la mémoire : on garde les maillons de `l2`). Est-ce que cette opération est efficace ?
3. Écrire une fonction `int get_val_list(element * L,int i)` qui retourne l'entier stocké dans le `i`-ème élément de la liste pointée par `L` (ici on suppose que `L` désigne bien le maillon de tête). Est-ce que cette opération est efficace ?
4. Écrire une fonction `void reverse_list(element * L)` qui inverse la liste pointée par `L` (le dernier élément devient le premier, de l'avant-dernier le second, *etc.*)

Exercice 6 : Fusion de liste (Optionnel)

1. Écrire une fonction `element* fusion_list(element* l1, element* l2)` qui prends deux listes `l1`, `l2` que l'on suppose triées et qui renvoie une nouvelle liste triée contenant les éléments de `l1` et `l2`, répétitions comprises.