# Remarques sur le TD / TP 3

Il est fondamental de bien comprendre la différence entre les deux approches suivantes des structures de données :

→ approche fonctionnelle – celle qui va généralement avec les listes
  ⇝ les fonctions ne font aucune modification mémoire et renvoient leurs résultats

→ approche "à effet de bord"
  ⇝ les fonctions modifient directement les structures et ne renvoient rien
  ⇝ ces approches reposent donc sur du passage par référence (ou de pointeurs) en argument

Ces deux types de vision co-existent et il est important de voir et comprendre la différence.

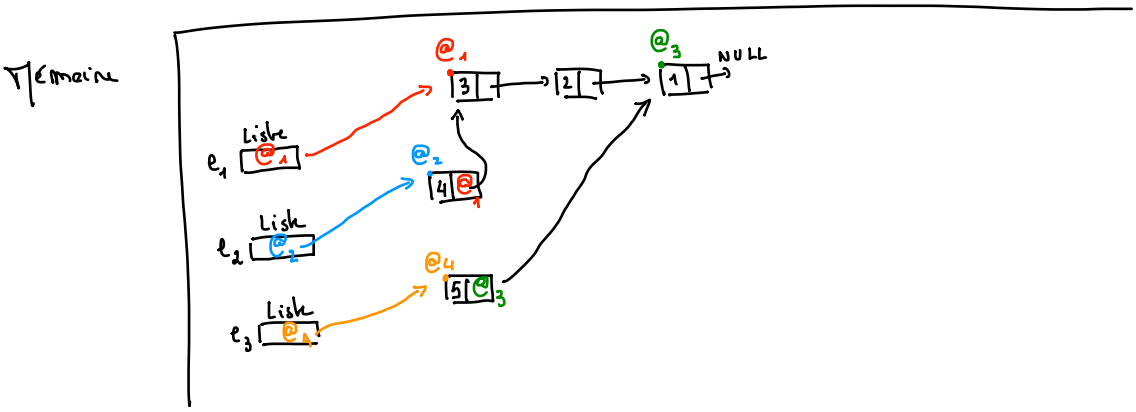
⚠ Elles ne posent pas de problèmes tant que l'on reste au niveau de l'implémentation du TAD.
L'implémentation fonctionnelle des listes par exemple, est sûre et efficace dans ce cadre (créant naturellement du partage mémoire).

Exemple :

```
Liste l1, l2, l3 ;
l1 = ajouter(3, ajouter(2, ajouter(1, creer_liste_vide())));

l2 = ajouter(4, l1);
l3 = ajouter(5, queue(queue(l1)));
```

intérêt de la vision fonctionnelle
↓
les fonctions renvoient des (valeurs)
↓
on peut les (composer)



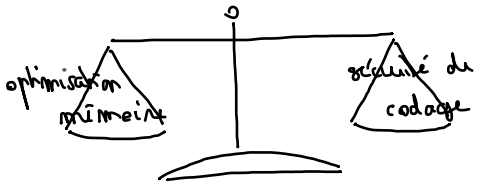

Comme on le voit, on crée du partage mémoire (qui est gérable tant qu'on manipule le TAD : ajouter / tête / queue ...) mais évidemment délicat pour libérer la mémoire ...

```
free_liste(l1);
free_liste(l2);
free_liste(l3);
```

plante évidemment !
↳ il faut conserver les "branchements" pour pouvoir libérer

Donc → soit on évite de tels partages (en créant des copies fraîches au besoin)
→ soit on gère finement la libération mémoire

On est face à un équilibre :




Il y a alors 2 grandes classes d'approches :

① Langages simples de type C :

→ on peut plus ou moins tout faire (et n'importe quoi aussi)
→ le programmeur est totalement responsable de la correction
→ vision 100% "optimisation mémoire"

② Langages objet

→ chaque objet a des constructeurs et destructeurs
→ c'est à la conception des classes qu'on reporte l'attention à la correction et qu'on limite les possibilités de n'importe quoi
→ vision intégrant une forte dose de "sécurité de codage"
→ C++ 50% optim. mém / 50% sécurité code
→ Java gestion auto. de la mémoire / 100% "

La gestion automatique de la mémoire est utilisée dans les langages fonctionnels. En fait dans notre exemple avec partage mémoire : des maillons peuvent être libérés quand il n'y a plus de pointeurs sur eux (plus personne ne connaît leur adresse).

→ c'est le garbage collector (GC) qui gère cette opération
(Java, Python, OCAML ...)