TRAVAUX PRATIQUES XI

# Piles par listes chainées

On rappelle que les options suivantes doivent être utilisées à la compilation :

```
-Wall -Wextra -fsanitize=undefined,address
```

Il est fortement recommandé pour ce TP de *faire des dessins !*.

## A Compilation séparée

### A.1 Fichiers objets et édition de liens

La compilation est en réalité composée de plusieurs étapes :

— Précompilation : cela applique des transformations mineures au code. (Exemple hors-programme que nous croiserons peut-être : remplacer les `#define` par leur valeur.)

— Analyses (lexicale, syntaxique, sémantique) : le compilateur « lit » le code et en tire sa structure. C'est là qu'il vérifie que le code écrit respecte bien les règles du langage, et que le typage est bon.

La notion de grammaires est *extrêmement* importante lors de cette étape : rendez-vous en MPI !

— Génération de code : le compilateur transforme le code source en code (presque entièrement) compilé. Cela crée un fichier objet `.o` . Il contient le code compilé du fichier… mais uniquement celui du fichier. Toute fonction appelée depuis un autre fichier/librairie/module n'est pas présente dans le `.o` . Il manque également de « l'administratif » requis pour le lancement.

— Édition de liens : le compilateur « lie » différents `.o` afin de créer un exécutable qui peut accéder à toutes les fonctions. Cette étape a uniquement besoin des `.o` : il s'agit simplement [1] de piocher les fonctions dans les `.o` et de les mettre ensemble dans un exécutable. En particulier, le compilateur recherche un et un seul `main` , et fait un peu d'administratif pour que ce soit cette fonction qui soit lancée à l'exécution.

Jusqu'à présent, en compilant nous avons fait les 4 étapes. Il est possible de ne faire que les 3 premières et de créer le `.o` : en effet, pour une librairie, on ne veut pas la recompiler intégralement à chaque fois qu'elle est utilisée. On veut créer son fichier objet une seule fois, qui sera ensuite utilisé lors de l'édition de liens.

En C, pour compiler sans éditer les liens et donc créer un fichier objet, on utilise : `gcc -o fichier.o -c fichier.c` . On peut [2] ajouter des options à la fin.

Enfin, pour éditer les liens, il faudra faire `gcc -o nom_prgm fichier0.o fichier1.o fichier2.o` . Là encore, on peut [3] ajouter des options à la fin.

### A.2 Headers/interface en C

Une librairie (telle que la précédente sur les tableaux dynamiques) peut se décrire comme une implémentation concrète d'un type abstrait. La description de ce que contient une librairie, i.e. de son type abstrait, est appelée une **interface**. Un code qui utilise une librairie est appelé un **code client** de cette librairie.

En C, les fichiers d'interface s'appellent des *headers*. Leur fonctionnement est le suivant :

— Si l'implémentation s'appelle `code.c` , le header associé s'appelle `code.h` .

— L'implémentation demande à inclure *uniquement* son header. C'est le header qui demande les autres inclusions : l'idée est que le header centralise toutes les informations dont le programmeur peut avoir besoin, dont les dépendances.

— Le code client appelera `code.h` avec `#include "code.h"` (notez les guillemets au lieu des chevrons : c'est pour indiquer « où » aller le chercher).

1. Ce n'est pas (aussi) simple.
2. On doit.
3. Bis Repetita Placent : « On doit.»

— Toute fonction et tout type utilisable par un code client doit être déclaré dans le header. Pour les types on peut donner leur entière définition dans le header (l'implémentation appelle son header donc elle y aura accès !), pour les fonctions on se contente de déclarer leur prototype [4].

— **On documente !**

— Ne mettez pas de `main` dans une librairie ! Le seul main doit être celui du code source : l'édition de lien finale aura besoin d'un et d'un seul main !

Vous trouverez par mail et/ou sur les machines et/ou en ligne un header pour ce TP. Vous pourrez être amenés à le modifier pour l'enrichir ou améliorer sa documentation (elle est assez minimaliste).

# B Listes simplements chainées

L'objectif de ce TP est de recoder les listes OCaml en C (c'est à dire des piles).

## B.1 Cours

Le fonctionnement de ces listes correspond à ce que l'on appelle des **listes simplement chainées**. On veut pouvoir :

— Créer la liste vide ( `[]` en OCaml).

— Tester si une liste est vide.

— À partir d'une liste `l` non-vide, accéder au premier élément (filtrer en `x::l` et renvoyer `x` ).

— À partir d'une liste `l` non-vide, accéder à la liste composée de tous les éléments sauf le premier (filtrer en `x::l` et renvoyer `l` ).

— À partir d'une liste `l` et d'un élément `x`, créer une nouvelle liste qui contient `x` *puis* les éléments de `l` ( `x::l` en OCaml).

Le fonctionnement de l'implémentation OCaml est le suivant :

— La liste est découpée en « chainon ».

— Chaque chainon stocke un élément, ainsi qu'un pointeur vers l'élément suivant. Cf Figure XI.1.

Cette implémentation est la première que nous voyons qui n'est *pas basée sur des tableaux* ! Cela permet de facilement ajouter en enlever des éléments de tête, puisqu'il n'y a pas de tableau à agrandir/contracter [5]. Cela permet également de limiter les redondances lors « d'ajouts » d'éléments de tête, comme nous le verrons bientôt.

Chaque maillon est alloué en mémoire. Une liste est alors un pointeur vers un maillon.

**1.** Sur la Figure XI.1, ajouter le pointeur qui correspond à la liste `[12; 99; 37]`

La liste vide correspond au pointeur `NULL`. Sur la Figure, il est représenté comme un pointeur vers une case barrée [6] . Pour parcourir une liste, il faut donc « suivre les flèches » et lire les éléments au fur et à mesure, jusqu'à tomber sur la flèche `NULL` qui marque la fin.

Les opérations décrites précédemment fonctionnent ainsi :

— Pour créer une liste vide, il n'y a rien à faire : une liste vide est le pointeur vers aucun maillon, donc le pointeur `NULL`.

— Pour tester si un pointeur vers un maillon correspond à la liste vide, il faut tester si le pointeur pointe bien vers un maillon, çad tester s'il est non-`NULL`.

— Accéder à la tête ou à la queue de la liste demande de suivre le pointeur vers le maillon de tête, puis de renvoyer soit la valeur stockée dans le maillon soit le pointeur vers la suite.

— Créer une nouvelle liste en ajoutant un élément en tête est l'opération la plus intéressante. Pour cela… on va créer un nouveau maillon, le faire pointer vers l'ancienne liste, et renvoyer l'adresse de ce maillon. En particulier, on ne duplique pas l'ancienne liste !

**2.** Sur la Figure, ajouter :

**a.** Un pointeur correspondant à la lsite `[37]` (i.e. la liste du schéma dont on a passé les deux premiers éléments de tête).

**b.** Un pointeur correspondant à la liste `0::[99; 37]` (il faut créer un nouveau maillon).

4. En particulier cela permet de faire en sorte que l'implémentation ait vu une déclaration des fonctions avant leur implémentation : on peut donc les implémenter dans n'importe quel ordre !

5. Cela a le désinconvénient, hors-programme, de ne pas être très bien optimsé par le cache.

6. C'est une représentation standard, qui a ses intérêts pour des variantes de la structure de données, mais que je n'aime pas pour les listes simplement chainées telles que nous les faisons

**c.** Un pointeur correspondant à la liste `6::[12; 99; 37]` (idem).
**d.** `0::[12; 99; 37]` (idem).
**e.** Un pointeur correspondant à la liste `20::6::[12; 99; 37]` (idem).

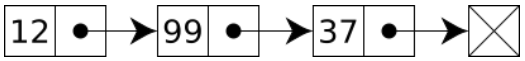


FIGURE XI.1 – Des cellules chainées. Source : Wikipédia.

**3.** Comptenez le nombre total de maillons. Comparez avec la somme des longueurs des différentes listes crées. Commentez.

Cette façon d'implémenter des listes est appélée listes simplement chainée. Telle que décrite ci-dessus, il s'agit d'une structure de donnée fonctionnelle (les transformateurs renvoient une nouvelle structure de donnée au lieu de modifier celle passée en argument).

## B.2 Type C

Une cellule sera implémentée par le type suivant :

C : list.h
```c
struct cellule_s {
  int elem;
  struct cellule_s* next;
};
typedef struct cellule_s cellule;
```

Une liste sera donc un pointeur vers une cellule, c'est à dire un élément de type `cellule*` .

**4.** Déplacez le header `list.h` dans votre dossier à vous. Dans le même dossier, créez un fichier `list.c` qui pour l'instant ne doit contenir que l'inclusion de son header (*include*), et un autre fichier `test.c` qui contient un `main` (qui ne fait rien pour l'instant à part renvoyer `ECIT_SUCCESS`) et inclut `list.h` .

**5.** Compilez (en utilisant la compilation séparée) ![7] *Des fichiers [...].o doivent avoir été créés dans votre dossier.*

## B.3 C'est parti

Maintenant, c'est à vous : la question suivante vous laisse en autonomie sur le TP.

**6.** Implémentez les fonctions présentes dans le header, et testez-les depuis `test.c` . Vous n'êtes pas obligé·es de les implémenter dans l'ordre.
*Rappel poru la fonction de taille :* `size_t` *est un type d'entiers non signés qui est garanti assez grand pour pouvoir stocker la taille de n'importe quel objet mémoire (aka assez grand pour parler de toute votre RAM).*

À titre informatif, voici l'ordre dans lequel je pense corriger les fonctions au tableau. C'est plus ou moins un ordre croissant de difficulté.

— `list_create` et `list_hd` et `list_tl`.
— Puis `print_list`.

7. Certes le code ne fera rien, mais il faut prendre la main !

— Puis `cellule_free` puis `list_free`.
— Puis `list_cons`.
— Puis le reste (je n'aurai pas le temps, soyons réalistes).

# C Pour les plus rapides

7. Déclarez et créez une fonction permettant d'accéder au ième élément d'une liste.
8. Déclarez et créez une fonction permettant de transformer un tableau (et/ou une zone allouée) en liste. De même pour la réciproque (on prendra la taille de la liste en argument).
9. (Difficile) Déclarez et créez une fonction permettant de trier une liste.
   *On devra renvoyer une copie de la liste triée, et non trier la liste donnée en argument.*
   *Indice : Le tri insertion est particulièrement adapté à cette structure de donnée.*

## Et ensuite ?

10. Proposez une autre façon d'implémenter le type abstrait des listes OCaml en utilisant des tableaux et non des cellules.