TP n°12 - Implémentation des listes en C.

On rappelle que pour pouvoir exécuter les programmes compilés sur les machines du lycée, votre répertoire de travail doit être situé en dehors du répertoire `Documents`

Exercice 1 (Implémentation par maillons chaînés).

Toutes les fonctions ci-dessous seront implémentées dans un fichier C `NOM_listes_maillons.c`

Pour cet exercice, encore plus que d'habitude, vous devez sortir une feuille et un crayon afin de dessiner ! Gardez une trace écrite de ce TP au propre dans votre classeur.

Je vous invite à ne pas reprendre ce qui a été fait pendant le dernier cours et à essayer de retrouver ces éléments par vous-même, de zéro.

1. Représenter par un schéma sur votre feuille une liste chaînée contenant les éléments 1.5, 3.1, 2.25, 1.33, 2.17 et 5.0. Représenter la variable `l1` permettant de stocker cette liste, puis représenter une autre variable `l2` représentant la liste des éléments de l_1 à partir du deuxième.
2. Rappelez la définition récursive (en fait inductive) d'une structure de liste.
3. Définir dans votre fichier C une nouvelle structure de données permettant de stocker une liste de réels sous la forme de maillons chaînés. On nommera `addr_next` le champ contenant l'adresse du maillon suivant (où une adresse vide si c'est la fin de la liste). Appelez le professeur pour valider votre nouvelle structure.
4. Revenons à votre schéma. Repérer sur le schéma précédent ce que représente `(*l1)`, `(*l1).val`, `(*l1).addr_next` et `*((*l1).addr_next)` .
5. Écrire une fonction `push` qui ajoute un réel **en tête** de la liste.
6. Écrire une fonction `print_list` qui affiche tous les éléments d'une liste implémentée par maillons chaînés. Réfléchissez d'abord sur papier à votre algorithme.
7. Tester ces premières fonctions en ajoutant un à un quelques éléments dans une liste, **en le codant « dur » dans la fonction principale**
8. Écrire une fonction `get_ith` qui renvoie la i-ème valeur d'une liste. Pensez à appliquer les principes de la programmation défensive pour vérifier que la demande de l'utilisateur a un sens !
9. Écrire une fonction `delete_ith` qui supprime le i-ème élément de la liste. Pensez à réfléchir d'abord rigoureusement sur papier et pensez également à appliquer les principes de la programmation défensive. Tester abondamment.
10. Écrire une fonction `length` qui calcule la taille d'une liste. Quelle est la complexité de cette fonction ? Quelle amélioration de notre structure de données peut-on envisager pour éviter ce coût ?
11. Adapter votre structure de données en conséquence en créant une nouvelle structure encapsulante nommée `list`. Appelez le professeur pour valider votre nouvelle structure.
12. Créer une fonction `create_list` permettant de créer un objet de type `list`
13. Adaptez toutes les fonctions précédemment codées pour qu'elles manipulent des objets de type `list` et testez-les dans le `main`
14. Écrire une fonction `insert_ith` qui insère un élément en i-ème position de la liste. Pensez à réfléchir d'abord rigoureusement sur papier et pensez également à appliquer les principes de la programmation défensive. Tester abondamment.
15. Écrire une fonction `free_list` qui libère tout l'espace mémoire utilisé par une liste.
16. Écrire une fonction `concatenate` qui concatène deux listes et renvoie un pointeur vers la nouvelle liste. Cette fonction ne doit pas allouer de nouveaux maillons. Que pensez-vous de cette fonction en terme de sûreté ? Quelle différence faites-vous avec ce qui se passe lors de la concaténation de listes OCaml ?

La structure de liste codée à l'exercice 1 ne permet de représenter que des listes de réels.

Pour conserver une seule implémentation et gagner en modularité, nous allons utiliser une nouvelle directive de pré-traitement :

```
#define identificateurA identificateurB
```

qui a pour effet de remplacer dans le code source toutes les occurrences de *identificateurA* par *identificateurB*.

Exercice 2 (Vers plus de modularité).
On travaillera à nouveau dans le fichier `NOM_listes_maillons.c`

1. Comment modifier le code du fichier `NOM_listes_maillons.c` pour pouvoir changer le type des éléments des listes ?
2. Adapter toutes les fonctions de votre code (sauf la fonction `print_list`) en conséquence.
3. Comment faire pour gérer le cas de la fonction `print_list` ? Proposer une implémentation.

Exercice 3 (Implémentation avec un grand tableau).
Dans un fichier C `NOM_listes_tableau.c`, implémenter toutes les fonctions précédemment codées, mais en utilisant cette fois une implémentation utilisant un grand tableau.

La taille du grand tableau pourra être fixée lors de la création de la liste, dans la fonction `create_list`.

A l'exception de cette fonction, toutes les autres fonctions devront avoir les mêmes prototypes que les fonctions implémentées dans le fichier `NOM_listes_maillons.c`

Pour aller plus loin...

Il reste encore un problème : imaginons que j'utilise mon implémentation `NOM_listes_maillons.c` dans un code. Une fois défini `elt_type` grâce à la directive de pré-traitement comme expliqué ci-dessus, par exemple :

```
#define elt_type char
```

et après compilation, mon code ne pourra manipuler QUE des listes de caractères...

Cela pose problème si je souhaite manipuler dans le même code des listes d'entiers et des listes de caractères... je devrais finalement dupliquer le code pour l'adapter à chaque type de liste, et je n'aurai rien gagné en terme de modularité.

Une autre technique utilise le `transtypage` sur des pointeurs de type `void *`. Elle consiste à définir une liste générique de la manière suivante :

```
struct s_cell
{
  void          *addr_elt; // pointeur vers l'élément de type void (non def)
  struct s_cell *addr_next; // pointeur contenant l'adresse
                            // du maillon suivant
};

typedef struct s_cell cell;

struct s_list
{
  unsigned int  siz;
  cell         *addr_first;
};

typedef struct s_list list;
```

Il est impossible de déréférencer directement un pointeur `pv` de type `void *`. En effet, ce type de pointeur peut contenir une adresse mais ne permet pas de savoir combien de bits il faut lire à partir de cette adresse lors du déréférencement ni comment les interpréter (format).

Il faut donc d'abord effectuer un transtypage pour créer un nouveau pointeur `p` contenant la même adresse que `pv` mais ayant un type explicite, par exemple ici le type `int *` :

```
int *p = (int *)pv
```

Ce type explicite permet au compilateur de comprendre comme interpréter et lire la valeur pointée. Par exemple, ici, le type `int *` indique que la valeur pointée est un entier, et donc que le déréférencement consiste à lire 32 bits (taille mémoire d'un entier) à partir de l'adresse stockée dans le pointeur.

Exercice 4 (Polymorphisme en C grâce au transtypage sur `void *`).
Copier votre fichier `NOM_listes_maillons.c` dans un nouveau fichier `NOM_listes_polymorphes.c`.

Dans ce fichier, utilisez les remarques précédentes pour adapter votre implémentation et être capable de manipuler dans le même code des listes dont le type des éléments diffère. On écrira des tests en dur dans le `main` (pas de récupération d'argument depuis la ligne de commande).