TP5 : Implémentation des listes chainées

L'objectif de ce TP est de proposer une réalisation concrète en C de la structure de liste chainée et d'implémenter un certain nombre d'opérations élémentaires. Avant cela nous allons travailler sur les structures en C qui seront nécessaires à cette implémentation.

1 Structures en C

Pour définir un type composé de plusieurs éléments en C on utilisera le type structure. Nous avons rapidement vu les structures en fin du premier TP. Revenons sur l'exemple croisé :

Voici le type stucture date : `struct date {int annee; int mois; int jour;};`

Une date est donc un objet composé de trois valeurs (on dit trois champs) appelés `annee`, `mois` et `jour`. Pour accéder à ces champs pour un objet dont le nom est `ce_jour`, on utilise la commande : `ce_jour.annee`.

Pour déclarer un objet de type `date` nommé `ce_jour`, on utilise la commande suivante : `struct date ce_jour;`. On remarque que le nom du type lors de la déclaration est `struct date` et non pas seulement `date`. Lors du passage par valeur dans une fonction ce sera pareil.

Pour initialiser une structure on utilise la commande suivante : `ce_jour = {1914,09,01};` (j'espère que vous voyez à quoi ça ressemble...)

▷ **Question 1.** Ecrire une fonction qui prend en entrée une date et renvoie la date un an auparavant. ◁

Nous allons maintenant voir d'autres exemples pour nous familiariser avec cette notion.

1. Les complexes :

   ```
   struct complexe{ double reelle; double imaginaire};
   ```

   ▷ **Question 2.** Déclarer et initialiser le complexe $z = 3 + 4i$.

   Ecrire une fonction qui calcule le module d'un nombre complexe. La signature de la fonction sera : `double module (struct complexe z)`

   Ecrire une fonction qui prend en entrée un complexe z et renvoie son conjugué. ◁

2. Tableaux. Pour contourner les problèmes que nous avons pu croiser avec le fait d'obtenir la taille d'un tableau on peut définir une structure qui contient le contenu du tableau mais aussi sa taille de la manière suivante :

   ```
   struct array {
   int* cont;
   int len;
   };
   ```

   ▷ **Question 3.** Déclarer et initialiser le tableau `{1,2,3,4,5,6};` avec le type défini ci-dessus.

   Ecrire une fonction `int somme(struct array t)` qui prend en entrée un tableau de type `array` et qui renvoie la somme de ses éléments.

   Ecrire une fonction `struct array miroir(struct array t)` qui prend en entrée un tableau de type `struct array` et qui renvoie un tableau qui contient le miroir du tableau passé en entrée.

   Ecrire une fonction `struct array append(struct array t,int x)` qui prend en entrée un tableau de type `array` et un entier `x` et renvoie un nouveau tableau contenant tous les éléments du tableau passé en entrée puis `x` en dernière case. ◁

Il peut sembler lourd d'utiliser la commande `struct nom` à chaque fois que l'on déclare un objet ou une fonction. Afin d'alléger le code, on peut utiliser une commande du type : `typedef struct array tableau1D`, ainsi le type `tableau1D` existe et correspond à un raccourci pour `struct array`.

Si l'on souhaite modifier le contenu d'une structure à l'aide d'une fonction (de manière similaire au fait qu'un tableau est modifié par une fonction) alors il faut passer en argument non pas la structure elle même mais un pointeur vers cette structure. En effet tous les objets sont passés par valeur dans une fonction en C c'est-à-dire que C travaille avec une copie de l'objet passé en argument et se contente de lire sa valeur. Quand une fonction reçoit un pointeur alors il travaille avec la valeur de celui-ci c'est-à-dire une adresse mémoire et il peut donc accéder et modifier le contenu de cette adresse. Ceci explique pourquoi les tableaux qui sont en fait des pointeurs sur leur première valeur peuvent être modifiés par les fonctions et ceci sera valable pour tout pointeur vers un objet et en particulier pour tout pointeur vers une structure.

Dans un contexte qui contient ceci :

```
struct array {
 int* cont;
 int len;
 };
```

Que signifient ? `struct array* a;` et `struct array b;` ?

Indiquer, parmi les expressions suivantes, celles qui sont incorrectes, et donnez le type de celles qui sont correctes :

(a) a.cont

(b) *a.cont

(c) (*a).cont

(d) b.cont

(e) *b.cont

(f) (*b).cont

Pour la suite on pourra utiliser la commande `p->membre` qui est strictement équivalente (et moins lourde) à `(*p).membre`

3. Matrices. On considère le type de structure suivant, qui représente un tableau bi-dimensionnel d'entiers :

```
1 struct matrix {
2 int nb_lines, nb_cols;
3 int** lines;
4 };
typedef struct matrix tableau2D;
```

Dans une telle structure, `nblines` et `nb_cols` sont des entiers positifs, et `lines` est un tableau de `nb_lines` lignes, chacune de ces lignes étant un tableau de `nb_cols` entiers.

▷ **Question 4.** Ecrire une fonction `void print_zero_line_index(tableau2D mat)` qui affiche l'indice de la première ligne de `mat` qui ne contient que des 0, s'il y en a une.

Ecrire une fonction `struct matrix* make_matrix(int nb_lines, int nb_cols)` qui crée une matrice de taille $nb_{lines} \times nb_{cols}$, initialisée avec des 0. Si une allocation échoue, la fonction doit renvoyer `NULL` après avoir libéré la mémoire précédemment allouée.

Ici, on commence à prendre garde au fait qu'en pratique un `malloc` peut nous renvoyer une adresse `NULL` et qu'il convient de vérifier systématiquement que ce n'est pas le cas. On va aussi commencer à bien libérer sytématiquement les adresses une fois qu'elles sont inutilisées. ◁

2 Implémentation des listes chainées.

Une liste chainée sera composée de noeuds dont le type sera le suivant :

```
typedef struct node {
    int val;
    struct node* next;
} noeud;
```

Pour comprendre comment initialiser une liste voici un exemple qui crée une liste de longueur 2 comprenant 1 et 2 :

```
noeud* head = NULL;
head = (noeud*) malloc(sizeof(noeud));
head->val = 1;
head->next = (noeud*) malloc(sizeof(noeud));
head->next->val = 2;
head->next->next = NULL;
```

En fait une liste sera un pointeur sur son premier élément. Compléter sa définition : `typedef liste`

▷ **Question 5.** Ecrire une fonction `void print_list(liste l)` qui prend en entrée une liste et qui affiche les éléments de la liste dans l'ordre. On pensera à passer à la ligne entre chaque élément pour plus de lisibilité. ◁

On pourra tester avec la liste définie ci-dessous :

```
int n=15;
liste l;    //l pointera sur le debut de la liste finale
liste courant = (liste) malloc(sizeof(noeud));   //on cree un premier noeud
courant->val=0;
l = courant;   //l prend la premiere valeur de courant et ne changera pas
liste n;

for (int i=1;i<nb;i++){
    n = (liste) malloc(sizeof(noeud));    //on cree un nouveau noeud
    n->val=i;
    courant->next=n;      //ce nouveau noeud succede au precedent sauve dans courant
    courant = n;    //on sauve le nouveau noeud cree car la variable
                            n servira a construire le suivant

}
courant->next=NULL;
```

▷ **Question 6.** Ecrire une fonction `liste ajout_beg(liste l, int elt)` qui ajoute un élément `elt` en tête de la liste et renvoie une nouvelle liste résultat. ◁

▷ **Question 7.** Ecrire une fonction `void push_end(liste l, int elt)` qui prend en entrée une liste et un entier `elt` et ajoute un noeud en fin de liste dont la valeur est `elt`. La fonction modifie la liste passée en entrée. ◁

▷ **Question 8.** Ecrire une fonction `void push_beg(liste* lp, int elt)` qui ajoute un élément `elt` en tête de la liste pointée par `lp`. Puisque l'on veut changer la valeur de la tête on doit alors passer un pointeur vers la liste en paramètre de la fonction et on prendra garde à bien manipuler ici un pointeur de pointeur. ◁

▷ **Question 9.** Ecrire une fonction `int pop(liste* lp)` qui va renvoyer la valeur du premier élément de la liste et supprimer le premier noeud de celle-ci. On pensera comme précédemment à modifier le pointeur correspondant à la tête de la liste et on pensera aussi à libérer (en utilisant `free`) la mémoire associée au noeud supprimé. On n'oubliera pas non plus de traiter le cas particulier où la liste est vide. ◁

▷ **Question 10.** Ecrire une fonction `int pop_last(liste l)` qui va renvoyer la valeur du dernièr élément de la liste et supprimer ce noeud. On libérera l'espace mémoire et on n'oubliera pas de traiter le cas particulier où la liste passée en argument est vide. ◁

▷ **Question 11.** Ecrire une fonction `int remove_by_index(liste* lp, int n)` qui va éliminer le n-ieme élément de la liste et renvoyer sa valeur si n est suffisamment petit et renverra -1 sinon. ◁

▷ **Question 12.** Ecrire une fonction `int remove_by_value(liste* lp, int elt)` qui va éliminer le premier noeud de la liste contenant la valeur `elt` et renvoyer son indice s'il existe et renverra -1 sinon. ◁

▷ **Question 13.** Ecrire une fonction `liste concat(liste l1, liste l2)` qui renvoie la liste résultant de la concaténation des listes `l1` et `l2`. ◁

▷ **Question 14.** Ecrire une fonction `void free_list (liste  lst)` qui va supprimer la liste `lst` c'est-à-dire libérer toute la mémoire qui lui est attribuée. ◁