

TP11 C Listes chaînées

Published on 2022-01-13 by **Marc de Falco**

Développement en C (Rappels)

Environnement

Pour ce retour au C, je vous propose d'utiliser :

- soit votre ordinateur directement avec un éditeur de texte et gcc ou clang dans le terminal pour compiler.
- soit dans la machine virtuelle
- soit, et je pense que c'est une bonne solution, directement depuis info.classeprepa.net :
 - Dans le menu en haut à droite New vous choisissez Terminal
 - Il suffit de créer un répertoire pour votre tp et un fichier c :
 - `mkdir tp10`
 - `cd tp10`
 - `touch main.c`
 - Ensuite, naviguez vers le fichier `main.c` vide dans le Jupyter, cliquez dessus et vous avez un éditeur, il suffit de faire Sauvergarder ou Ctrl-S puis de compiler.

Compilation

Pour compiler, on va directement lancer le compilateur dans le terminal. Pour cela, en tenant compte du fait que votre fichier est `main.c`. On pourra écrire `gcc -o main main.c` puis `./main` pour le lancer.

Vous risquez d'avoir des programmes qui font plein d'erreurs de mémoire ou de crash. Je vous recommande donc de compiler avec la commande suivante :

```
gcc -Wall -Werror -g -fsanitize=address -o main main.c
```

Découpons cette commande :

- gcc est le compilateur, on peut le remplacer par clang qui est plus récent
- -Wall signifie qu'on active tous les avertissements warning.
- -Werror signifie qu'on traite les avertissements comme des erreurs de compilations.
- -g qu'on compile avec les informations de débogage, c'est-à-dire le lien entre votre code source et l'endroit où une erreur se produit.
- -fsanitize=address signifie qu'on active le vérificateur d'erreurs de gestion mémoire.

Comme c'est un peu long à taper, vous pouvez tirer partie de l'historique du shell, c'est-à-dire qu'en utilisant la touche fléchée vers le haut, on retrouve la commande précédente, en appuyant deux fois, la commande qui précède la précédente et ainsi de suite. Il suffit alors d'appuyer sur la touche Entrée pour exécuter la bonne commande ainsi retrouvée.

Rendu et tests

Pas de rendu pour ce TP et pas de tests automatiques. Je vais vous demander de faire vos propres tests progressivement. La démarche sera alors d'écrire les fonctions dans votre fichier `main.c` et de rajouter des tests dans votre fonction `main`, éventuellement, ces tests seront groupés par thème dans des fonctions de tests comme dans les TP précédents.

Listes chaînées

On va définir des fonctions sur les listes chaînées en utilisant le type suivant :

```
struct maillon {
    int valeur;
    struct maillon *suivant;
};
typedef struct maillon maillon;

struct liste {
    maillon *premier;
    maillon *dernier;
};
typedef struct liste liste;
```

Invariants à maintenir

- Dans un maillon, `suivant` est soit `NULL` si c'est le dernier maillon de la chaîne, soit un pointeur vers le maillon qui le suit.
- Si la liste est vide alors `premier` et `dernier` valent `NULL`. Sinon, ils pointent respectivement sur le premier et le dernier maillon de la chaîne.

Première partie

On va construire une bibliothèque qui pourra nous servir dans de futurs TP.

On vous donne les prototypes des fonctions et une brève description de ce qu'elles doivent faire. Certaines de ces fonctions ont été vues en cours.

La plupart des fonctions passent et renvoient des listes par copie, mais pour quelques fonctions il faut modifier les pointeurs et on passe donc par un `liste *`.

Rappel pour accéder au champ `x` d'une struct `s` passée par pointeur, on écrit `s->x`. En fait, c'est un raccourci sur `(*s).x`.

TESTEZ VOS FONCTIONS

```
// renvoie une valeur correspondant à la liste vide
liste liste_vide();

// Affiche la liste sous la forme [1,2,3,4]
void liste_affiche(liste l);

// alloue et initialise un nouveau maillon
maillon *maillon_creer(int valeur, maillon *suivant);

// libere un maillon
void maillon_detruire(maillon *m);

// libere une liste
void liste_detruire(liste l);

// valeur en tete de la liste
int liste_tete(liste l);

// renvoie la queue de la liste
// sous forme d'une liste attention !
liste liste_queue(liste l);

// renvoie la longueur de la liste, c'est-a-dire le
// nombre de maillons
int liste_longueur(liste l);

// ajout en tete en modifiant directement les pointeurs dans l
// O(1)
void liste_ajout_en_tete(liste *l, int valeur);

// supprime un maillon en tête de la liste
// O(1)
void liste_suppr_en_tete(liste *l);

// ajoute un maillon à la fin de la liste
// O(1)
void liste_ajout_en_fin(liste *l, int valeur);

// renvoie un pointeur sur le premier maillon de la
// liste de valeur valeur, renvoie NULL s'il n'y en a pas
maillon *liste_cherche_maillon(liste l, int valeur);
```

```
// insertion en O(1) après un maillon
// Attention, doit fonctionner si m est le dernier maillon
void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur);

// ajoute un maillon de valeur valeur avant le maillon m
// Attention, doit fonctionner si m est le premier ou le dernier
// maillon aussi et en O(1)
void liste_ajout_maillon_avant(liste *l, maillon *m, int valeur);

// supprime le maillon m. Doit fonctionner si c'est le premier
// maillon.
void liste_suppr_maillon(liste *l, maillon *m);
```

Listes doublements chaînées

On va définir des fonctions sur les listes chaînées en utilisant le type suivant :

```
struct maillon {
    int valeur;
    struct maillon *precedent;
    struct maillon *suivant;
};
typedef struct maillon maillon;

struct liste {
    maillon *premier;
    maillon *dernier;
};
typedef struct liste liste;
```

Invariants à maintenir

- Dans un maillon, suivant est soit NULL si c'est le dernier maillon de la chaîne, soit un pointeur vers le maillon qui le suit.
- Dans un maillon, precedent est soit NULL si c'est le premier maillon de la chaîne, soit un pointeur vers le maillon qui le précède.
- Si la liste est vide alors premier et dernier valent NULL. Sinon, ils pointent respectivement sur le premier et le dernier maillon de la chaîne.

Deuxième partie

- Reprenez les questions précédentes avec ce nouveau type de listes.
- Rajoutez les fonctions suivantes :

```
// supprime le dernier maillon en O(1)
void liste_suppr_en_fin(liste *l);

// renverse la liste en place !
void liste_renverse(liste *l);

// insertion en O(1) avant un maillon
void list_insertion_avant(liste &l, maillon *m, int valeur);

// suppression en O(1) d'un maillon
void list_supprime(liste *l, maillon *m);
```
