TP5: C, allocation dynamique

David Baelde

14 et 15 octobre 2015

Nous abordons aujourd'hui l'allocation dynamique de mémoire, via les fonctions `malloc()` et `free()`. C'est un outil essentiel du programmeur C…et la source de difficultés supplémentaires liées aux pointeurs, corruption mémoire, comportements non définis, etc.

Exercice - 1 *Bref retour aux automates cellulaires*

Reprenez le code de l'exercice 2 du TP précédent, générant des diagrammes espace-temps d'automates cellulaires élémentaires. Si besoin, repartez du corrigé, disponible sur la page des TPs. Nous avions paramétré ce code par une constante N définie via un `#define` au début du fichier. Il est aisé de changer la valeur de N mais il faut ensuite recompiler pour pouvoir exécuter le programme avec la nouvelle valeur.

1- Adaptez le programme pour qu'il prenne en arguments, sur la ligne de commande :
- Le numéro indiquant la règle de transition, comme avant.
- La taille souhaitée, c'est à dire le paramètre N.

Cela va nécessiter d'ajouter un argument à plusieurs fonctions, pour les informer de la taille des tableaux manipulés. Surtout, puisque la taille des tableaux n'est plus connue statiquement (au moment de la compilation) il va falloir [1] procéder par allocation dynamique, au moyen de la fonction `malloc()`.

Une fois la question précédente réalisée et testée, lancez votre programme sous l'environnement de déboguage `valgrind` :

```
$ valgrind ./monprogramme
```

Cet outil supervise l'exécution de votre programme, et notamment son utilisation de la mémoire. Vous devriez voir un LEAK SUMMARY : il vous informe qu'une partie de la mémoire allouée dynamiquement n'est jamais libérée, c'est une *fuite mémoire*.

Ce n'est pas dramatique ici car la mémoire est bien sûr libérée quand le programme termine, mais c'est une bonne habitude de nettoyer derrière soi : le programme autonome d'aujourd'hui pourrait devenir une fonctionalité, utilisée répétitivement dans un logiciel plus complexe demain, et il serait alors gênant que chaque utilisation alloue plus de mémoire sans jamais la libérer.

2- Libérez la mémoire allouée au moyen de `free()`, vérifiez qu'il n'y a plus de fuite mémoire.

Exercice - 2 *Listes chaînées*

Nous allons définir un type de donnée pour des listes chaînées. C'est comme les listes doublement chaînées du TP 2, mais avec une seule direction de chaînage, et en C.

Ouvre un nouveau fichier `.c`, et commencez par inclure les en-têtes suivant, respectivement pour les fonctions de gestion mémoire, d'affichage, et…`assert()` :

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
```

1. Ceci est vrai en C ANSI (c'est à dire la norme de 1989) mais pas avec les variantes plus récentes. Par défaut, votre compilateur permet très probablement les "variable length arrays", alloués sur la pile bien que leur taille ne soit pas statiquement. Merci de me pardonner ce détail et quand même essayer d'allouer dynamiquement, sur le tas, vos tableaux.

On définit ensuite le type enregistrement `struct _list`, et un alias `list` pour celui-ci :

```
typedef
  struct _list {
    int value;
    struct _list *next;
  }
  list;
```

Chaque cellule de nos listes chaînées sera représentée par un objet de type `list`, qui est une structure comportant deux champs : la valeur contenue dans la cellule, et un pointeur vers la prochaine cellule. Par convention, le pointeur NULL est utilisé en fin de liste, quand il n'y a plus de prochaine cellule. (En réalité, ce qu'on pensera comme une liste est plutôt un objet de type `list*` que `list`. La liste vide est représentée par NULL et une liste non-vide par un pointeur valide sur une cellule.)

1- Définissez une fonction qui "ajoute un élément à une liste". Plus précisemment, elle devra allouer une nouvelle cellule contenant la valeur donnée et pointant ensuite vers la liste pré-existante, qui n'est pas modifée. Votre fonction devra se conformer au prototype suivant :

```
list* cons(int hd, list* tl);
```

2- Créez une fonction pour afficher les éléments d'une liste, avec un nombre maximal `n` d'éléments à afficher. Elle pourra par exemple produire des résultats comme `1:2:3:nil` ou `1:2:3:...`. Quand `n` est négatif, tous les éléments devront être affichés quel que soit leur nombre. La fonction devra se conformer au prototype suivant, et l'on créera aussi une seconde fonction pour ne pas avoir à passer `n=-1` explicitement.

```
void show_max(list* hd, int n);
void show(list* l);
```

3- Testez avec la fonction principale suivante :

```
int main() {
  list* l2 = cons(2,NULL);
  show(l2);
  list* l1 = cons(1,l2);
  show(l1);
  show(l2);
  return 0;
}
```

Cela devrait afficher (modulo détails de présentation) `2:nil`, `1:2:nil` puis `2:nil`.

4- Il y a une fuite mémoire dans le programme précédent. Il nous faut une fonction de libération de la mémoire : celle-ci devra libérer toutes les cellules d'une liste, et se conformer au prototype suivant :

```
void destroy(list* tl);
```

Utilisez cette fonction pour libérer la mémoire une fois qu'on n'a plus besoin des listes `l1` et `l2`.

5- On veut maintenant implémenter une fonction d'accès au $n^{\text{ème}}$ élément d'une liste, l'élément de tête étant considéré comme ayant l'index 0. Faites-le deux fois, d'abord en style récursif avec une fonction `nth_rec` qui se rappelle elle-même, puis en style impératif avec une fonction `nth` qui utilise une boucle (`for` ou `while` selon les goûts) :

```
int nth_rec(int n, list* l);
int nth(int n, list* l);
```

Pour tester, récupérer le fichier `tests.c` sur la page des TPs : il contient une fonction `main()` avec de nombreux tests pré-écrits et commentés, que vous pourrez décommenter au fur et à mesure. Remplacez votre fonction principale par celle-ci.
A la fin du test de la fonction `nth()`, la liste est détruite puis on essaie de nouveau d'accéder à son premier élément : expliquez ce qui se passe alors.

6- Implémenter une fonction de concaténation. Précisemment, étant donné deux listes `l1` et `l2`, elle doit faire pointer la dernière cellule de `l1` sur la première cellule de `l2`. Le prototype :

```
void glue(list* l1, list* l2);
```

Testez en décommentant le bloc suivant dans la fonction `main()` fournie.

7- Implémenter une fonction qui teste si deux listes sont "structurellement" égales, c'est à dire qu'elles ont même contenu. Comme avant, on respectera le prototype suivant et on vérifiera le résultat du test fourni :

```
int equals(list* l1, list* l2);
```

8- Implémenter une fonction qui fasse boucler une liste sur elle même : la dernière cellule de la liste originale doit, après appel de la fonction, pointer sur la première cellule. Par exemple, `1:2:nil` devient, après bouclage, `1:2:1:2:1:2:...` à l'infini.

```
void tie(list* l);
```

9- Pour pouvoir appeler `destroy()` sur une liste circulaire (résultant par exemple d'un appel à `tie()`) il faut briser la circularité. Implémenter pour cela la fonction `untie()` qui prend une liste, supposant que celle ci boucle sur son premier élément, et détruit ce lien. Sur l'exemple précédent, après "débouclage" on devrait récupérer `1:2:nil`.

```
void untie(list* l);
```

Décommenter la suite du test, qui déboucle et détruit la liste. Vérifier qu'il n'y a pas de fuite mémoire.

10- Une liste est dite circulaire s'il y a un cycle dans les liens entre ses noeuds. Il ne s'agit pas forcément d'un cycle qui revient vers la première cellule. Par exemple, la liste `1:2:3:4:3:4:3:4:3:4:...` (qu'on peut obtenir par `glue()` et `tie()`) est circulaire. On pourra noter qu'en fait une liste ne peut être infinie sans être circulaire. Implémenter une fonction testant si une liste est circulaire.

```
int is_circular(list* l);
```

Indice : cherchez un algorithme en espace constant, qui n'utilise aucune structure de donnée auxiliaire ; en particulier, ne cherchez pas à mémoriser tous les noeuds déja rencontrés.

11- Implémentez un test d'égalité entre deux listes, qui fonctionne aussi bien quand les listes sont circulaires :

```
int equals_circular(list* l1, list* l2);
```

Exercice - 3 *Bonus : comptage de références*

Comme vous l'avez peut être remarqué en faisant l'exercice précédent, il peut être difficile de savoir quelles listes peuvent êtres détruites. Si l'on concatène `l1` à `l2`, il ne faut pas détruire `l2` même si on ne l'utilise plus jamais directement. Inversement, détruire `l1` ne doit pas être fait tant qu'une portion de code a encore besoin de `l2`.

Dans certains cas, ces problèmes sont insurmontables, et il devient bon de penser à mettre en place un dispositif de gestion automatique de la mémoire. Un des moyens les plus primitifs pour cela est le *comptage de référence*. L'idée est d'enrichir votre type de données (ici, les noeuds de la liste) avec un compteur indiquant combien d'autres données pointent vers celle-ci. Par exemple, à la création *ex nihilo* d'une liste `l1` non vide, toutes les cellules de la liste auraient un compteur de références à 1. (Pour la première, cela peut se comprendre comme indiquant que la fonction en cours "pointe" sur le premier élément de la liste par le biais de la variable locale qui détient la liste.) Si l'on concatène une liste `l2` en tête de `l1`, alors la première de cellule passe à 2 réferences.

On ne détruit une liste que quand plus personne ne pointe dessus. Sinon, la "destruction" consiste juste à décrémenter le compteur de références du noeud de tête de la liste. Pour clarifier, on distinguera l'opération de *libération* et de *destruction*, la première n'entraînant la seconde que dans le cas où le compteur de références est nul. En reprenant l'exemple précédent, des appels successifs à destroy sur `l1` et `l2` entraîneront la libération de toute la mémoire, quel que soit l'ordre dans lequel on détruit ; de plus, la libération de `l1` n'entraîne pas la destruction de `l2`, qui reste utilisable.

Implémenter un tel mécanisme. Il serait utile de réfléchir à organiser votre code de façon à systématiquement bien mettre à jour les compteurs, typiquement en isolant les opérations ayant trait à ceux-ci.

On pourra aussi critiquer la portée d'un tel système.

Exercice - 4 *Bonus : tables de hachage*

Une fonction de hachage associe un entier $h(x)$ à toute valeur x d'un certain type, avec l'espoir que la répartition de ces entiers soit plutôt uniforme. Etant donné une telle fonction, une table de hachage est une structure de donnée permettant de représenter un ensemble (modifiable) de valeurs indexées par des clés. Nous prendrons ici `string` comme type des clés et `int` comme type des valeurs, une table de hachage permettra ainsi par exemple de stocker l'age d'un certain nombre de personnes désignées par leur nom de famille.

Si une valeur v doit être associée à une clé k, on va la stocker dans la case $h(k)\%n$ d'un tableau de taille n. Comme plusieurs clés peuvent donner le même $h(k)\%n$, le tableau doit en fait contenir une liste d'associations, chaque association indiquant le couple clé-valeur.

1- Déclarer un type de listes d'associations (disons `assoc`) pour des clés de type `char*` et des valeurs de type `int`.

2- Déclarer une structure `table` qui contienne un tableau de listes d'associations, ainsi qu'un entier indiquant la taille du tableau, et enfin le nombre total d'entrées dans la table.

3- Implémenter une fonction pour créer une table vide, prenant en argument la taille du tableau :

```
table* create(int n);
```

4- Implémenter une fonction de hachage de votre choix :

```
int hash(string* s);
```

On pourra par exemple prendre $5381 + \sum_i a_i 33^i$ (une fonction due à Dan Bernstrein, empiriquement efficace) où les a_i sont les caractères successifs.

5- Implémenter la fonction d'insertion :

```
void insert(table* t, char* key, int value);
```

6- Implémenter la fonction de recherche, qui renvoie la cellule de liste d'association trouvée, ou bien NULL :

```
assoc* lookup(table* t, char* key);
```

Tester sur un petit ensemble de chaînes saisies manuellement.

7- Quand il y a plus d'entrées dans la table que la taille du tableau, la probabilité de collisions devient trop grande, et il est raisonnable de redimensionner le tableau. Implémenter cette fonctionalité, en doublant la taille du tableau :

```
void resize(table* t);
```

Modifier ensuite la fonction d'insertion pour automatiquement redimensionner quand on atteint un seuil dans le nombre d'éléments.

8- Générer toutes les chaînes de 3 caractères de $\{a, \ldots, z\}$, les stocker dans une table de taille 23.