Cours de programmation
Pointeurs, listes chaînées

Olivier Aumage
LIP, ENS Lyon (bureau 302),
46, allée d'Italie, 69364 Lyon Cedex 07
email : Olivier.Aumage@ens-lyon.fr

Septembre 2001

1 Introduction

Cette première fiche de *Travaux Dirigés* est consacrée à la maîtrise du concept de pointeur en langage **C**, tout d'abord de manière isolée, puis dans le cadre de l'écriture de programmes utilisant des listes chaînées ou des structures dérivées. Certains exercices seront abordés dans les séances de travaux dirigés et pratiques, d'autres sont donnés uniquement pour vous permettre d'approfondir ou de consolider votre vision du cours et des notions présentées.

2 Pointeurs

2.1 Pour comprendre

2.1.1 Application au passage de paramètres

En **C**, les paramètres passés en arguments aux fonctions sont toujours passés par *valeur*. Une des conséquences de cette propriété est que le bout de code suivant ne produit pas le résultat escompté. Pourquoi ?

```
void echange(int x, int y)
{
  int z;

  z = x;
  x = y;
  y = z;
}

int main()
{
  int a = 1;
  int b = 2;

  printf("a = %d, b = %d\n", a, b);
  echange(a, b);
  printf("a = %d, b = %d\n", a, b);

  return 0;
}
```

Comment peut-on corriger le programme précédent en utilisant des pointeurs pour simuler un passage par référence ?

2.2 Pour approfondir

2.2.1 Gestion d'une ville

Pour devenir experts en programmation par pointeurs, on se propose de les utiliser pour représenter une ville et ses habitants. Pour cela, on se donne le cahier des charges suivant :

- un individu contient un prénom ;
- une famille contient un nom de famille et un ou plusieurs individus ;
- un étage contient un numéro d'étage et une ou plusieurs familles ;
- un immeuble contient un numéro de rue et un ou plusieurs étages ;
- une rue contient un nom et des maisons et des immeubles numérotés (on considèrera qu'une maison est un immeuble à un seul étage) ;
- une ville contient un nom et des rues ;

Questions

1. Définissez vos types de données pour représenter la ville et ses différents éléments.
2. Écrivez les fonctions permettant de construire et de peupler une ville en utilisant les fonctions `malloc` (ou `calloc`), `realloc` et `free` (pour libérer toutes les zones de mémoire convenablement avant de quitter le programme).
3. Écrivez quelques fonctions pour récupérer des informations permettant d'obtenir des informations sur la ville :
   - l'adresse d'un individu ;
   - le nombre d'habitants ;
   - la famille la plus représentée ;
   - ...

Note : l'exercice peut sembler vraiment difficile au premier abord. En réalité, il ne fait appel qu'à des manipulations élémentaires sur les pointeurs. Construisez votre programme pas-à-pas pour que l'exercice ne paraisse pas trop dur. Par exemple, il vaut mieux s'assurer que les individus sont convenablement traités avant de construire des familles !

Si tout s'est bien passé, vous pouvez compliquer un peu les choses. On peut, par exemple, ajouter quelques touches de réalisme, toujours pas-à-pas.

- On peut commencer par ajouter les croisements, en considérant qu'une ville est un ensemble de rues et de croisements entre ces rues. Chaque croisement sera constitué des rues qui le composent et des numéros rues où il se situe.
- Avec le programme dans sa forme actuelle, si on veut savoir si une personne habite ou non dans la ville, on est oblige de parcourir toute la structure. Pour éviter ce problème, modifiez la structure représentant la ville pour quelle permettent un accès immédiat à ses habitants.
- Modifiez la structure représentant un individu pour qu'elle contienne toutes les informations sur son adresse.

3 Listes chaînées

Les différentes versions de listes chaînées sont très employées en programmation en raison de leur souplesse d'utilisation. La section *Pour comprendre* traite les listes simplement et doublement chaînées. La section *Pour approfondir* aborde d'autres points comme les listes circulaires ou la création de listes dont les maillons peuvent contenir des objets quelconques.

3.1 Pour comprendre

3.1.1 Listes à la mode LISP

Les listes chaînées peuvent être présentées selon deux *modes* différentes, la mode **C** et la mode LISP (ou SCHEME). La version mode **C** a été présentée en cours. Au cours de cet exercice, on va voir comment programmer très simplement des listes à la mode LISP et revérifier que la programmation n'est finalement qu'affaire de goût.

1. La déclaration suivante d'une structure de liste chaînée est-elle valide ? Si oui, est-elle équivalente à celle du cours ?

```
1   typedef struct s_maillon *p_maillon_t;
2   typedef p_maillon_t       liste_t;
3   typedef liste_t          *p_liste_t;
4
5   typedef struct s_maillon
6   {
7     int         valeur;
8     p_maillon_t suivant;
9   } maillon_t;
```

2. Lorsqu'on définit une liste à la mode **C**, on dit qu'un maillon contient une *valeur* et un *pointeur sur un maillon*. Lorsqu'on choisit la mode LISP, on dit qu'un maillon contient *valeur* et une liste. Quelle est la mode choisie pour déclarer la liste ci-dessus ? Comment peut-on modifier la déclaration précédente pour adopter l'autre mode ? Les deux déclarations sont-elles équivalentes ?
3. Implémentez `nil` et `cons` ;
4. Implémentez `car` et `cdr` ;
5. Implémentez `concat` et `reverse` ;
6. En utilisant des *pointeurs de fonction* implémentez les fonctions `map` (exécute une fonction sur chaque élément de la liste, ex : affichage des éléments), `apply` (exécute une fonction sur chaque élément de la liste et produit une nouvelle liste avec le résultat de chaque appel, ex : incrémentation, décrémentation) et `reduce` (exécute une fonction sur chaque élément pour produire un unique résultat final, ex : somme, produit).

3.1.2 Listes et structures dérivées

Les listes chaînées sont souvent utilisées pour construire des structures de données plus spécialisées comme les piles et les files.

Piles Les piles, également appelées LIFO (pour *Last-In, First-Out*), se construisent très facilement avec des listes simplement chaînées.

1. Déclarez les types nécessaires à la réalisation d'une pile, suivant le principe utilisé en cours.
2. Écrivez les fonctions `push` et `pop` traditionnelles d'accès à une pile.

Files Les files, également appelées FIFO (pour *First-In, First-Out*), peuvent aussi se construire à l'aide de listes simplement chaînées, mais on préfère souvent à ces dernières les listes doublement chaînées.

1. Pourquoi les listes doublement chaînées sont-elles plus appropriées que les listes simplement chaînées pour la construction d'une file ?
2. Déclarez les types nécessaires à la réalisation d'une file, suivant le principe utilisé en cours. La file sera construite à l'aide d'une liste *doublement* chaînée (à déclarer aussi, évidemment).
3. Écrivez les fonctions usuelles d'accès à la liste doublement chaînée.
4. Écrivez les fonctions `enqueue` et `dequeue` traditionnelles d'accès à une file.

3.2 Pour approfondir

Les exercices proposés dans cette section ne doivent en principe pas poser de problèmes si les précédents ont été bien compris. Utilisez-les pour vérifier vos connaissances.

3.2.1 Listes triées

Une liste triée est une liste dont les éléments sont ordonnés de telle manière que la tête de la liste contienne le plus petit élément et la queue de la liste contienne le plus grand élément. Écrivez les fonctions suivantes (liste non exhaustive) pour une liste triée simplement chaînée.

- `void insere(p_liste_t, int)` qui insère un entier dans la liste, en respectant la propriété d'ordre.
- `int cherche(p_liste_t, int)` qui renvoie `1` si l'entier passé en argument est dans la liste et `0` sinon.
- `p_liste_t fusionne(p_liste_t, p_liste_t)` qui renvoie une troisième liste dont le contenu est l'interclassement des éléments des deux listes passées en argument.

3.2.2 Listes circulaires

Les *listes circulaires* sont des listes chaînées dont le dernier élément est relié au premier élément (ce qui forme un anneau). Les listes circulaires peuvent être simplement ou doublement chaînées.

- Déclarez les types nécessaires à l'implémentation d'une liste circulaire simplement chaînée dont les maillons contiennent un *nom* et un *prénom* au lieu d'une valeur entière.
- Écrivez les fonctions `insere` et `supprime` qui ajoutent et retirent respectivement un étudiant de la liste.
- Écrivez une fonction `suivant` qui fait pointer la tête de la liste vers l'élément suivant, et une fonction `tete` qui donne le nom et le prénom de l'étudiant en tête de la liste.
- Utilisez ces fonctions pour concevoir un programme qui gère de manière équitable l'ordre de passage des étudiants au tableau.
- Juste par esprit de contradiction, modifiez la fonction `suivant` pour que la liste tourne en sens inverse.

3.2.3 Listes avec pointeur générique

L'exemple suivant montre la déclaration d'un maillon de liste simplement chaînée dont les maillons possèdent un pointeur *générique* (`void *`).

```
typedef struct s_maillon
{
  void        *donnee;
  p_maillon_t  suivant;
} maillon_t;
```

Un pointeur générique est un pointeur qui peut pointer sur n'importe quel type. On peut affecter un pointeur générique à un pointeur classique et inversement. Cette propriété permet d'écrire des fonctions de gestion de listes chaînées sans connaître à l'avance le type de valeur à mettre dans les maillons (**note :** la valeur *doit* cependant être un pointeur).

- utilisez cette propriété pour reprendre la gestion de la ville (section 2.2.1, page 2) avec des listes pour chaque niveau.

-=-=-=-