

Série 7 : Programmation C - Pointeurs (suite)

Buts

On continue notre pratique des pointeurs...

Exercice 1 : Questionnaire QCM (structures + chaînes de caractères + allocation dynamique, niveau 2)

On cherche ici à faire un programme d'examen sous forme de questionnaire à choix multiple (QCM) où une question est posée et la réponse est à choisir parmi un ensemble de réponses proposées (une seule bonne réponse possible par question).

Dans un programme `qcm.c`, définissez une structure `QCM` comprenant 3 champs :

1. un champ `question`, chaîne de caractères, qui contiendra la question à poser
2. un champ `reponses` qui sera un tableau d'au plus 10 chaînes de caractères contenant les réponses proposées ;
3. Un champ `nb_rep`, de type entier non signé, contenant le nombre de réponses possibles
4. un champ entier `solution` (entier positif) qui contient le numéro de la bonne réponse (dans le champ `reponses`).

Prototypiez puis définissez une fonction `affiche` qui prend un `QCM` en argument et l'affiche. Par exemple :

```
Combien de dents possède un éléphant adulte ?
1- 32
2- de 6 à 10
3- beaucoup
4- 24
5- 2
```

Dans le `main`, créez et initialisez le `QCM` ci-dessus, puis affichez le. Compilez et vérifiez que tout fonctionne correctement jusqu'ici.

Reprennez la fonction `demander_nombre` (avec 2 arguments, point 4 de l'exercice 1 de la [série 4](#)), et créez une fonction `poser_question` qui prend un `QCM` en argument, appelle successivement `affiche` et `demander_nombre` et retourne la réponse de l'utilisateur.

Avant de continuer, testez votre programme (affichage de la question et saisie de la réponse).

On cherche maintenant à faire un examen de plusieurs questions.

Définissez le type `Examen` comme un ensemble dynamique (pointeur) de `QCM`. Créez un `Examen` dans le `main`, puis remplissez-le (dans une fonction `creer_examen` c'est mieux !) avec les questions suivantes (code partiel à compléter) où `retour` est un pointeur sur un `Examen`:

```
strcpy((*retour)[0].question,
      "Combien de dents possède un éléphant adulte");
(*retour)[0].nb_rep = 5;
strcpy((*retour)[0].reponses[0], "32");
strcpy((*retour)[0].reponses[1], "de 6 à 10");
strcpy((*retour)[0].reponses[2], "beaucoup");
strcpy((*retour)[0].reponses[3], "24");
strcpy((*retour)[0].reponses[4], "2");
(*retour)[0].solution = 2;

strcpy((*retour)[1].question,
      "Laquelle des instructions suivantes est un prototype de fonction");
(*retour)[1].nb_rep = 4;
strcpy((*retour)[1].reponses[0], "int f(0);");
strcpy((*retour)[1].reponses[1], "int f(int 0);");
strcpy((*retour)[1].reponses[2], "int f(int i);");
strcpy((*retour)[1].reponses[3], "int f(i);");
```

```

(*retour)[1].solution = 3;

strcpy((*retour)[2].question,
      "Qui pose des questions stupides");
(*retour)[2].nb_rep = 7;
strcpy((*retour)[2].reponses[0], "le prof. de math");
strcpy((*retour)[2].reponses[1], "mon copain/ma copine");
strcpy((*retour)[2].reponses[2], "le prof. de physique");
strcpy((*retour)[2].reponses[3], "moi");
strcpy((*retour)[2].reponses[4], "le prof. d'info");
strcpy((*retour)[2].reponses[5], "personne, il n'y a pas de question stupide");
strcpy((*retour)[2].reponses[6], "les sondages");
(*retour)[2].solution = 6;

```

Important: dans cette étape, vous allez être amenés à faire des allocations dynamiques. Pensez à désallouer la mémoire une fois le travail terminé (par exemple avec une fonction `detruiure_examen`).

Pour terminer :

1. Posez les questions une à une ;
2. Comptez le nombre de bonnes réponses ;
3. Donnez le score à la fin.

Exemple d'exécution

```

Combien de dents possède un éléphant adulte ?
1- 32
2- de 6 à 10
3- beaucoup
4- 24
5- 2
Entrez un nombre entier compris entre 1 et 5 : 2
Laquelle des instructions suivantes est un prototype de fonction ?
1- int f(0);
2- int f(int 0);
3- int f(int i);
4- int f(i);
Entrez un nombre entier compris entre 1 et 4 : 3
Qui pose des questions stupides ?
1- le prof. de math
2- mon copain/ma copine
3- le prof. de physique
4- moi
5- le prof. de programmation
6- personne, il n'y a pas de question stupide
7- les sondages
Entrez un nombre entier compris entre 1 et 7 : 5
Vous avez trouvé 2 bonnes réponses sur 3.

```

Exercice 2 : QCM revisités (niveau 2)

Cet exercice reprend l'exercice précédent. On veut maintenant que ce programme `qcm` soit indépendant d'un QCM particulier (séparation de l'algorithme et des données).

Pour cela on va donner la possibilité à ce programme de lire les QCM dans un fichier.

Recopier le programme `qcm.c` dans un nouveau fichier puis éditez-le.

Commencez par y recopier la fonction `demandeur_fichier` de l'**exercice 4 de la série 4** (`stat.c`).

Puis changez la fonction `creer_examen` de l'exercice précédent (celle qui créait le QCM) de sorte qu'elle crée le contenu du QCM à partir d'un fichier passé en argument :

```
Examen creer_examen(FILE* fichier);
```

Le format du fichier à lire est le suivant :

```
Q: question
reponse1
```

```
->reponse2
reponse3
```

```
Q: question2
...
```

Le signe `->` en début de ligne indique la bonne réponse.
Le signe `#` en début de ligne indique une ligne de commentaire.

Pour indiquer une réponse commençant par `"->"` ou par `"#"`, il suffit de ne pas mettre ce signe juste au début de ligne mais de mettre un espace. La procédure de lecture devra éliminer ces espaces initiaux.

Note : l'avantage de ce format est que l'ordre des réponses peut facilement être changé dans le fichier de configuration sans avoir à modifier l'indication de la bonne réponse.

Exemple de fichier :

```
Q: Combien de dents possède un éléphant adulte
32
-> de 6 à 10
beaucoup
24
2

Q: Quel signe est le plus étrange
#
->
->->##<-
#b
a
```

La dernière question correspond à

```
Quel signe est le plus étrange ?
1- #
2- ->
3- ->##<-
4- a
```

et la réponse est 3.

Créez, dans `exam1.txt` le fichier correspondant au questionnaire de la fois passée et testez votre programme.

Exercice 3 : liste chaînées (structures de données, pointeurs, niveau 2)

On veut écrire le programme `listeschainees.c` qui implémente une version des listes chaînées.

1. Décidons d'implémenter une liste chaînée dynamique d'éléments de type `type_el`.

Pour rendre la chose facilement modifiable, définissez ce type, par exemple comme type entier.

2. Définissez ensuite le type `ListeChaine`, représentant une « liste chaînée », en s'appuyant sur la constatation : une liste chaînée est une suite d'éléments contenant chacun une valeur et la liste chaînée des valeurs suivantes.
3. Prototypiez et définissez les fonctions de manipulation associées aux listes : insertion d'un élément dans la liste après un élément déjà connu (donné), insertion d'un élément en tête de liste, suppression du premier élément de la liste, suppression d'un élément donné de la liste (il ne faut pas oublier de **libérer** la mémoire demandée lors de l'insertion pour l'élément qui est supprimé) et le calcul de la longueur de la liste (nombre d'éléments) (pour changer du cours essayez de le coder avec une boucle `for`).

Amélioration (niveau 3) : pour la calcul de la longueur de la liste, faire attention aux listes cycliques (telles que le suivant d'un élément est un élément qui précédait).

Exercice 4 : Carnet d'adresses (niveau 3)

Le but de cet exercice est de réaliser une mini-application complète qui comprend une structure de données (arbres binaires non balancés), des fichiers (lecture/écriture) et un petit interpréteur de commandes.

Un arbre binaire permet de stocker différents objets ordonnés par une clé. Dans notre carnet d'adresses, nous utiliserons comme clé le nom de la personne et comme valeurs le numéro de téléphone (mais vous pouvez bien sûr rajouter d'autres champs).

1. Créez un fichier `abook.c` dans lequel vous pouvez copier les prototypes ci-dessous. Vous complèterez également la structure `addr__` qui représente un noeud de l'arbre afin qu'elle contienne (en de la structure d'arbre): le nom, le numéro de téléphone, et éventuellement d'autres champs (adresse, e-mail, ...).

```
/* un noeud de l'arbre */
typedef struct addr__ addr_t;
struct addr__ {
    /* à compléter */
};

/* l'arbre, défini par sa racine */
typedef struct book__ {
    addr_t* root;
} book_t;

/* créer un nouveau carnet d'adresses vide */
book_t* book_create(void);

/* libérer les ressources associées */
void book_free(book_t* b);

/* lister dans l'ordre tous les noms du carnet d'adresses */
void book_list(book_t* b);

/* afficher une entrée du carnet d'adresses */
void book_view(book_t* b, const char* name);

/* ajouter ou modifier une entrée du carnet d'adresse */
void book_add(book_t* b, const char* name, const char* num);

/* supprimer une entrée du carnet d'adresses */
void book_remove(book_t* b, const char* name);

/* remplacer le contenu du carnet d'adresses par celui du fichier */
void book_load(book_t* b, const char* file);

/* sauver le contenu du carnet dans un fichier au format CSV */
void book_save(book_t* b, const char* file);
```

2. Implémentez ensuite les fonctions qui ont été prototypées (vous êtes libres de modifier le prototype si cela vous arrange). Les fonctions sont classées un peu près par ordre croissant de difficulté.

Note : le format CSV est : un enregistrement nom-numéro par ligne, où les champs sont séparés par des ';'. Par exemple:

```
Lucien;012 345 67 89;
Stéphane;021 879 51 32;
Julien;079 523 12 45;
Antoine;076 125 08 78;
Damien;022 329 08 85;
```

3. Pour pouvoir manipuler votre carnet d'adresse vous aurez besoin d'un interpréteur de commande. Le mode de fonctionnement est très simple, il faut
 - lire la ligne qui contient la commande
 - découper la chaîne reçue en arguments
 - en fonction du premier argument, appeler la fonction correspondante avec les bons arguments
 - recommencer

Pour découper la chaîne reçue en arguments, vous pouvez utiliser le code suivant:

```
/* on découpe la commande en arguments, voir man strsep */
int an=0; /* nombre d'arguments */
char *a[10]; /* tableau d'arguments */
char *ptr=cmd; /* cmd est le buffer qui contient la ligne */
```

```

while (an<10 && (a[an]=strsep(&ptr," \t\n"))!=NULL) {
    if (a[an][0]!='\0') ++an;
}
/* les arguments sont dans a[0] .. a[an-1] */

```

Les interprétations des commandes sont les suivantes. Elles correspondent directement à une fonction que vous avez précédemment implémentée.

```

add <name> <num>    ajouter un numéro
del <name>           supprimer un numéro
view <name>          afficher les informations
list                lister les noms
load <file>          lire les adresses du fichier
save <file>          enregistrer les adresses dans le fichier
quit               quitter le programme

```

Les plus courageux d'entre vous iront encore plus loin en implémentant l'interpréteur de commandes sous la forme d'un tableau de pointeur sur des fonctions génériques.

Exercice 5 : Dérivation formelle (niveau 3)

[adapté par T. Coppey (2010) sur la base d'un ancien sujet d'examen.]

Le but est ici de faire un programme capable de dériver formellement des expressions.

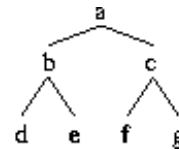
On utilisera pour cela une structure d'**arbre binaire**. Un arbre binaire est une structure de donnée définie de façon récursive par :
un arbre est

- soit une valeur ;
- soit une valeur et deux sous-arbres (binaires).

Par exemple, si l'on note un arbre entre parenthèses, en premier son sous-arbre droit, puis sa valeur et enfin son sous-arbre gauche, alors

((d) b (e)) a ((f) c (g))

est un arbre binaire : la valeur est a et les deux sous-arbres ((d) b (e)) et ((f) c (g)).



De même ((f)) est un arbre réduit à une valeur (i.e. sans sous-arbre).

1. Dans un fichier `arbre.h`, définissez, en vous inspirant de votre connaissance des listes chaînées, une structure de données pour représenter les arbres binaires.

Dans cette structure de données, la valeur sera représentée par un `char`.

2. Dans le fichier `arbre.c`, écrire la fonction permettant de créer un arbre binaire à partir d'une valeur et de deux sous-arbres.
3. Toujours dans le fichier `arbre.c`, écrire une fonction permettant d'afficher un arbre binaire en format parenthésé comme illustré plus haut.

On affichera récursivement le sous-arbre de gauche, puis la valeur puis (récursivement) le sous arbre de droite.

4. Écrire un petit programme `test_arbres.c` illustrant que votre code fonctionne correctement.

Écrire également le `Makefile` qui effectue la compilation et l'édition de liens entre les différents éléments.

Optionnel: assurez-vous que l'arbre puisse partager un ou plusieurs sous-arbres. Par exemple, l'arbre ((x) + (x)) peut être construit à partir d'un seul arbre (x). Pour ce faire, vous utiliserez soit la copie profonde, soit à l'aide d'un compteur de références.

5. Passons maintenant à la représentation des expressions arithmétiques.

On utilisera pour cela les arbres binaires (nous ne considérerons ici que les opérateurs binaires + - / * et ^ (puissance)) :

par exemple $a + b$ sera représenté par l'arbre

((a) + (b))

où '+' est la valeur du premier arbre et "(a)" et "(b)" ses sous-arbres.

De même l'expression arithmétique

$(a + b) * (c + d)$

sera représentée par l'arbre

$((a) + (b)) * ((c) + (d))$

Dans un fichier `expression.c`, très similaire à votre programme de test précédent, construisez les arbres représentant les expressions suivantes (4 expressions) :

$x + a$

$(x + a) * (x + b)$

$((x * x) * x) + (a * x)$

$(x ^ a) / ((x * x) + (b * x))$

Indication : commencez par créer les arbres binaires représentant les expressions a , b et x (seuls), puis ceux représentant les expressions $x+a$, $x+b$ et $x*x$.

6. Dans un fichier `derive.c`, écrivez une fonction `derive` qui prend un arbre binaire en argument, supposé être une représentation d'une expression arithmétique valide et un caractère représentant la variable par rapport à laquelle il faut dériver, et qui retourne l'arbre binaire correspondant à l'expression arithmétique dérivée.

On procédera pour cela de façon récursive en se rappelant les règles usuelles de dérivation :

$da/dx = 0$ où a est une constante (ici : caractère différent de x)

$dx/dx = 1$

$d(f+g)/dx = df/dx + dg/dx$

$d(f-g)/dx = df/dx - dg/dx$

$d(f*g)/dx = (df/dx * g) + (f * dg/dx)$

$d(f/g)/dx = ((df/dx * g) - (f * dg/dx)) / (g * g)$

$d(f^a)/dx = a * df/dx * (f ^ (a - 1))$

(où a ne dépend pas de x , ce que l'on supposera ici)

Note importante : on ne veut pas l'arbre minimal de dérivation ! En clair, on ne cherchera pas à simplifier les expressions obtenues.

7. Testez votre fonction de dérivation sur les 4 expressions précédentes.