Série 7 :
Programmation *C* - Pointeurs (3/5) - chaînes de caractères, pointeurs sur fonctions

Buts

Le but de cette série d'exercices est de vous permettre de continuer à pratiquer les aspects de la programmation en C utilisant les pointeurs, en particulier les « chaînes de caractères » et les pointeurs sur fonctions.

Exercice 1 : Générateur automatique de lettres (fonctions et chaînes de caractères, niveau 1)

Le but de cet exercice est d'écrire un programme nommé `lettre.c`, qui constituera un générateur automatique (très simpliste) de lettres.

1. Écrivez tout d'abord une fonction `genereLettre` (sans argument, et sans valeur de retour). Cette fonction devra simplement produire la sortie suivante à l'écran : (ne vous occupez pas de la mise en évidence (gras et soulignés))

   ```
   Bonjour chère Mireille,
   Je vous écris à propos de votre cours.
   Il faudrait que nous nous voyons le 18/12 pour en discuter.
   Donnez-moi vite de vos nouvelles !
   Amicalement, John.
   ```

   Invoquez (appelez) simplement la fonction `genereLettre` depuis la fonction principale `main`, compilez votre programme et assurez vous de son fonctionnement correct.

2. Modifiez maintenant la fonction `genereLettre`, de manière à rendre paramétrables les parties mise en évidence dans le texte précédant. Il vous faudra pour cela passer un certain nombre d'arguments à la fonction, de manière à spécifier :
   - La formulation adaptée pour l'entrée en matière (`"chère"` pour un destinataire féminin, et `"cher"` pour un destinataire masculin). Ce choix devra être fait en fonction de la valeur d'un argument énuméré (p.ex. `genre`).
   - Le nom du destinataire, nommé par exemple `destinataire`
   - Le sujet de la lettre (paramètre nommé `sujet`)
   - La date du rendez-vous sous forme de deux paramètres entiers, l'un pour le jour et l'autre pour le mois
   - La formule de politesse (paramètre `politesse`)
   - et le nom de l'auteur (`auteur`).

   La fonction aura donc 7 paramètres.

   Invoquez la fonction au moins deux fois de suite depuis le `main`, en paramétrant les appels de sorte à produire la lettre précédente, et la réponse ci-dessous :

   ```
   Bonjour cher John,
   Je vous écris à propos de votre demande de rendez-vous.
   Il faudrait que nous nous voyons le 16/12 pour en discuter.
   Donnez-moi vite de vos nouvelles !
   Sincèrement, Mireille.
   ```

Exercice 2 : Segmentation en mots (chaînes de caractères, niveau 2)

Dans le fichier `token.c`, prototypez puis définissez la fonction :

```
int nextToken(char const * str, size_t* from, size_t* len)
```

dont le rôle sera d'identifier (position et longueur) dans la chaîne `str` le premier mot à partir de la position `from`.

On considèrera que les séparateurs de mots sont les séquences d'au moins un caractère *espace* (c.-à-d. `' '`). La fonction positionnera les arguments `from` et `len` de sorte qu'ils déterminent l'index du premier caractère du mot et sa longueur, pour autant qu'un tel mot existe. Dans ce cas la fonction devra retourner une valeur strictement positive.
Dans le cas contraire, les valeurs retournées dans `from` et `len` ne sont pas significatives, et le résultat retourné par la fonction doit être `0`.

Depuis le `main` du programme, vous demanderez à l'utilisateur d'entrer une chaîne de caractères au clavier, et afficherez (en faisant des appels successifs à la fonction `nextToken`) l'ensemble des mots de la chaîne entrée, à raison de un mot par ligne, placés entre apostrophes.

Utilisez l'exemple de fonctionnement ci-après pour vérifier la validité de votre programme ; faites en particulier attention à ce que les apostrophes entourent les mots **sans qu'il y ait d'espace entre les deux** .

Vérifiez également que le programme se comporte correctement même lorsque la chaîne entrée se termine par une suite d'espaces.

Pour lire une ligne entière, utilisez :

```
fgets(ligne_a_lire, taille, stdin);
```

où `ligne_a_lire` est un `char [taille]`.

Vous ne pouvez pas connaître à l'avance la valeur que prendra *`taille`*, puisque c'est l'utilisateur qui entre une phrase. Dans ce cas, prédéfinissez une taille `TAILLE_MAX` suffisamment large pour que l'utilisateur puisse entrer une phrase de longueur raisonnable.

Exemple de fonctionnement :

```
Entrez une chaîne :  heuu bonjour, voici ma chaîne !
Les mots de " heuu bonjour, voici ma chaîne !  " sont:
'heuu'
'bonjour,'
'voici'
'ma'
```

```
'chaîne'
'!'
```

Exercice 3 : Intégrales revisitées (tableaux, typedef, pointeurs sur fonctions, niveau 3)

Pointeurs sur fonctions

On veut ici approfondir l'exercice 9 de la **série 2** dans le sens suggéré par la dernière remarque (et vu en cours).

Copier votre programme `integrale.c` (ou le nom que vous lui avez donné) dans un nouveau fichier, puis éditez-le pour le modifier.

Essayez (évidemment !) de faire cet exercice **sans** copier les transparents correspondants du cours...

Définissez trois fonctions de votre choix qui combinent des fonctions mathématique définies dans `math.h` (`man sin`, `man exp`, `man sqrt`).

Toutes ces fonctions doivent être compatibles avec le prototype `double f(double);`

Definisez un type `Fonction` qui est un **pointeur** sur une fonction de prototype semblable à celui indiqué ci-dessus.

Écrivez une fonction `demander_fonction` ne prennant pas d'arguments et retournant une `Fonction`. Cette fonction devra demander à l'utilisateur un nombre entre 1 et 5. S'il répond 1, 2 ou 3, la fonction `demander_fonction` renvoie le pointeur sur celle de vos fonction qui correspond. Si l'utilisateur répond 4, elle retourne un pointeur sur la fonction sinus (`sin`) et si il repond 5 elle retourne un pointeur sur la fonction exponentielle (`exp`).

Calculez ensuite l'intégrale de la fonction à l'aide de la fonction `integre()` (définie dans **l'exercice 9 de la série 2**) que vous aurez modifiée de façon à ce qu'elle prenne en argument supplémentaire une `Fonction`.

Tableaux de pointeurs sur fonctions

Si vous ne l'avez pas déjà fait comme cela, nous allons maintenant encore raffiner le choix en implémentant un tableau de `Fonction`s représentant tous les choix possibles.

Vous pouvez choisir de l'implémenter en tableau statique (définisez la taille comme une constante) ou dynamique (c.-à-d. pointeur ici [on aura donc un pointeur de pointeur de fonction !]).

Définissez un tableau de `Fonction`s nommé `choix`.

L'idée de base est de chosir la fonction directement en indexant le tableau des choix possibles. Si la réponse de l'utilisateur est stokée par exemple dans la variable `rep`, la bonne fonction sera alors `choix[rep]`. (La fonction `demander_fonction` renvoie maintenant un entier).

Initialisez ce tableau pour correspondre au choix précédent (`f1, f2, f3, sin, exp`).

Testez votre programme.

Pour finir si vous avez encore le courage et voulez encore rafiner votre programme vous pouvez créer une structure qui contient une chaîne de caractères décrivant la fonction et une `Fonction`.

Lors de la demande à l'utilisateur, affichez la description de la fonction pour aider l'utilisateur dans son choix.

Exemple de déroulement

```
Vous pouvez choisir parmi les fonctions suivantes :
   1- x carré
   2- racine d'exponentielle
   3- log(1+sin(x))
   4- sinus
   5- exponentielle
De quelle fonction voulez vous calculer l'intégrale [1-5] ? 4
Entrez un nombre réel : -3.141592653589
Entrez un nombre réel : 0
Integrale de sinus entre -3.14159265359 et 0 :
-2.00001781364
```

Exercice 4 : mini VM (pointeurs sur fonctions, niveau 1)

Revenons un peu sur les pointeurs sur fonctions en simulant un très simple interpréteur de commandes.

Un interpréteur de commande est simplement une boucle infinie (ou presque) qui fait correspondre une action (appel de fonction) à une chaîne de caractères saisie par l'utilisateur.

Nous allons ici faire une version très simple, pour illustrer les pointeurs sur fonctions, et simplement supposer que l'on travail avec une machine à 2 registres et 5 commandes :

- `quit`, qui affiche simplement « Bye! » ;
- `pop`, qui écrit la valeur du 2e registre dans le 1er ;
- `push`, qui écrit la valeur du 1er registre dans le 2e, demande une valeur (`double`) à l'utilisateur et la mets dans le 1er registre ;/p>
- `add`, qui additionne la valeur du 2e registre à celle du 1er ;
- `print` qui affiche la valeur du 1er registre.

Si une commande inconnue est entrée, on considérera que c'est `quit`.

Exemple de déroulement :

```
Entrez une commande (print, add, push, pop, quit) : print
-> 0
```

```
Entrez une commande (print, add, push, pop, quit) : push
Valeur ? 1.2
Entrez une commande (print, add, push, pop, quit) : print
-> 1.2
Entrez une commande (print, add, push, pop, quit) : push
Valeur ? 2.3
Entrez une commande (print, add, push, pop, quit) : print
-> 2.3
Entrez une commande (print, add, push, pop, quit) : add
Entrez une commande (print, add, push, pop, quit) : print
-> 3.5
Entrez une commande (print, add, push, pop, quit) : quit
Bye!
```

Pour travailler de façon générique (on pourrait vouloir généraliser encore un peu plus), toutes ces commandes auront comme prototype :

```
void commande(void* data);
```

et interpréteront `data` comme nécessaire : soit un, soient deux `double` consécutifs.

Ecrivez également une fonction `interprete` qui prend un `const char*` et retourne un pointeur sur une commande. Cette fonction doit bien entendu traduire le nom d'une commande en la fonction correspondante.

Terminez enfin votre programme en déclarant dans le `main` un tableau de deux `double` initialisés à 0 (qui servira de représentation de la mémoire à 2 registres de notre machine) et en faisant une boucle qui demande une commande à l'utilisateur puis exécute cette commande. La boucle s'arrêtera dès que la commande reçu est `quit` (cf exemple de déroulement ci-dessus)

Dernière mise à jour : $Date: 2013/01/11 13:26:44 $ (`$Revision: 1.18 $`)