Série 8 :
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` comprennant 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`).

Prototypez 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 initialiser 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 8 de la **série 2**), 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.
Définisez le type `Examen` comme un ensemble dynamique (pointeur) de `QMC`. 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 `detruire_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 `demander_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 `ListeChainee`, 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. Prototypez 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)

[proposé par T. Coppey, 2010]

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 différentes 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 protoype 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>       lit les adresses du fichier
save <file>       enregistre 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)

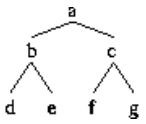
[adapté par T. Coppey (2010) sur la base d'un ancien sujet d'examen.]

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 (c.-à-d. sans sous-arbre).

1. En vous inspirant de votre connaissance des listes chaînées, définissez, une structure de données pour représenter les arbres binaires.

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

2. Écrire ensuite la fonction permettant de créer un arbre binaire à partir d'une valeur et de deux sous-arbres.
3. 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. Dans le `main()`, testez que votre code fonctionne correctement.

   Optionnel: assurez-vous qu'un même arbre puisse partager un ou plusieurs sous-arbre(s). 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)))`

   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. Écrivez ensuite 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.

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

Série 9 :
Programmation *C* - Pointeurs (5/5) - arithmétique des pointeurs ; révisions

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.

Exercice 1 : retour sur l'explorateur de mémoire

Reprenez **l'exercice 3 de la série 5**, mais en l'écrivant en utilisant l'arithmétique des pointeurs, en affichant par exemple directement toutes les adresses (sans passer par un index) :

```
0x7fffb7ed88ac : 01010000  80 ('P')
0x7fffb7ed88ad : 00000000   0
0x7fffb7ed88ae : 00000000   0
0x7fffb7ed88af : 00000000   0
```

Dans le même esprit, vous pouvez reprendre d'autres anciens exercices et écrire des solutions utilisant cette fois l'arithmétique des pointeurs. Par exemple, **l'exercice 2 de la série 7 (segmentation en mots)**.

Exercice 2 : piles et parenthèses (pointeurs, niveau 3)

Buts

Utiliser des tableaux dynamiques (**exercice 1 de la série 6**) ou des listes chaînées (**exercice 3 de la série 8**), ou votre propre structure (à inventer), pour implémenter une structure de pile (empiler, dépiler, valeur du sommet, test de pile vide) et résoudre le problème du parenthésage à deux parenthèses (« **langage de Dyck** ») : est-ce qu'une expressions utilisant 2 sortes de parenthèses, comme par exemple « `(a + [ b * c * ( d + e ) - 3 ] ) * ( x - [ 7 * b ] + 3)` » est correctement parenthèsée ou non ?

Indications

Les indications sont progressives, aussi, dès que vous pensez en savoir assez, essayez de faire l'exercice par vous-même.

1. On veut implémenter une pile. Qu'est-ce qui définit une pile ? Son sommet (le seul élément auquel on peut accéder) et les 4 fonctions : empile, dépile, lecture du sommet et test si la pile est vide.

   On vous demande d'implémenter la pile soit à l'aide de tableaux dynamiques (**exercice 1 de la série 6**), soit de listes chaînées (**exercice 3 de la série 8**) [ou autre chose si vous trouvez mieux]. Il vous suffit simplement de définir où mettre le sommet : en tête ou en queue de la structure de données ?

2. Que se passe-t-il quand on empile un nouvelle élément ?
   Il devient le nouveau sommet.

   Si on choisit de représenter le sommet de la pile par la premier élément du vecteur (élément 0) alors quand on empile un nouvel élément il faudra le mettre en 0 et déplacer tout le reste.
   Ce qui n'est pas une bonne solution !

   Par contre si on choisit de représenter le sommet par le dernier élément du vecteur, empiler consiste alors juste à ajouter un élément au vecteur.

3. Une pile sera donc représentée par un tableau dynamique/une liste chaînée, le sommet de la pile étant le dernier élément pour un tableau dynamique, le premier pour une liste chaînée.

   Avec cette implémentation, empiler un élément revient à l'ajouter à la fin/au début, dépiler revient à enlever le dernier/premier élément, lire le sommet revient à lire le dernier/premier élément et tester si la pile est vide à tester si la structure de données est vide ou non.

4. Concernant l'application « test de parenthésage », l'algorithme est le suivant :
   - **Tant qu'**il y a un caractère à lire
     - **Si** le caractère lu est `(` ou `[`, l'empiler
     - **Sinon, Si** c'est `)` ou `]`
       - **Si** la pile est vide, **Retourner :** *faux*
       - **Sinon** lire le sommet de la pile
         - **Si** le caractère lu et le sommet de la pile « correspondent » (se ferment l'un l'autre, de la même famille)
           - dépiler
         - **Sinon**
           - **Retourner :** *faux*
   - **Retourner :** la pile est vide ?

Exemple d'application

```
Entrez une expresssion parenthésée : ([])
  -> Erreur
Entrez une expression parenthésée : ([([()[]])][[()]]())
  -> OK
Entrez une expression parenthésée :
```

Note : Le programme se termine lorsque la chaîne saisie est vide.

Exercice 3 : piles et notation polonaise inverse (niveau 3)

Description

On veut faire un programme qui interprète les opérations arithmétiques en notation polonaise inverse (c'est-à-dire notation « postfixée »).

Cette notation consiste à donner **tous** les arguments avant l'opération.

Par exemple : `4+3` s'écrit `4 3 +`. De même : `(4 * 5) + 3` s'écrit `4 5 * 3 +`

Notez qu'avec ce système de notation il n'y a pas besoin de parenthèse. Il suffit d'écrire les opérations dans le bon sens.

Par exemple : `(4+3)*(3+2)` s'écrit `4 3 + 3 2 + *` alors que 4 + (3*3) + 2 s'écrit `4 3 3 * + 2 +`

Méthode

L'algorithme pour effectuer des opérations données en notation postfixée est le suivant, où `P` est une pile :

```
Tant que lire caractère c
  Si c est un opérande
     empiler c sur P
  Sinon
     Si c est un opérateur
        y <- sommet(P)
        dépiler P
        x <- sommet(P)
        dépiler P
        empiler le resultat de "x c y" sur  P
```

À la fin de cet algorithme, le résultat est au sommet de `P`.

À faire

Dans cette série nous allons simplement nous intéresser aux opérations sur les chiffres : les seuls opérandes seront les chiffres de `0` à `9`.
(Par exemple : `14+` veut dire `1 + 4` [on pourra bien entendu aussi noter avec des espaces : `1 4 +`])

1. Reprenez vos piles de l'exercice 3 ci-dessus et changez le type des éléments de `char` à `int`.
2. Prototypez puis définissez la fonction `eval` qui prend en entrée une chaîne de caractères et renvoie un entier, résultat de l'expression postfixée contenue dans la chaîne.

   Cette fonction devra implémenter l'algorithme ci-dessus, et utilisera donc une pile d'entiers.

3. Dans la fonction `main`, lisez une chaîne de caractères au clavier (correspondant à une opération arithmétique en notation postfixée) et évaluez-là à l'aide de la fonction précédente, puis affichez le résultat.

   La fonction `main` bouclera tant que la chaîne entrée n'est pas vide (voir l'exemple ci-dessous).

Indications

a. Pour tester sur un caractère (`char`) `c` est un chiffre, faire :

```
if ((c >= '0') && (c <= '9'))
```

Note : on peut aussi utiliser la fonction standard `isdigit((int) c)` définie dans `ctype.h`.

b. Pour convertir un caractère `c` représentant un chiffre en sa valeur entière (par exemple convertir `'3'` en `3`), faire :

```
(int)(c - '0')
```

Exemple de déroulement

```
Entrez une expression à évaluer : 8 6 2 - / 3 +
 -> résultat : 5
Entrez une expression à évaluer : 4 3 + 3 2 + *
 -> résultat : 35
Entrez une expression à évaluer : 4 3 3 * + 2 +
 -> résultat : 15
Entrez une expression à évaluer :
```

Exercice 4 : algorithmes de Needleman-Wunsch (algorithme de Viterbi sur le graphe d'édition) et LCS (niveau 2)

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

La comparaison et l'alignement de séquences de caractères est une tâche fondamentale dans plusieurs domaines d'applications, notamment en reconnaissance vocale, traitement d'images et bio-informatique. Le but de cet exercice est d'implémenter l'algorithme de Needleman-Wunsch (qui est un algorithme de Viterbi), utilisé par exemple dans le BioWall que vous avez pu observer toutes les semaines en entrant dans les salles d'exercices IN.

La première partie de l'algorithme consiste à évaluer la comparaison entre deux chaînes de caractères de tailles différentes, en minimisant le nombre d'insertions de « trous » entre deux caractères d'une même chaîne. En pratique, une matrice `M` de taille `(m + 1) × (n + 1)` est construite, où m et n sont les nombres de caractères des deux chaînes à comparer. Un « trou » est caractérisé par un déplacement horizontal ou vertical dans la matrice. On assigne alors à chaque cellule `M_i,j` de la matrice un score défini par les équations suivantes :

$$M_{i,0} = M_{0,j} = 0 \quad \text{pour } 0 \leq i \leq m \text{ et } 0 \leq j \leq n$$

$$M_{i,j} = \max\{ M_{i-1,j-1} + S_{i,j},\ M_{i,j-1} - 2,\ M_{i-1,j} - 2 \} \quad \text{pour } 1 \leq i \leq m \text{ et } 1 \leq j \leq n$$

avec

$$S_{i,j} = \begin{cases} 2 & \text{si } M_i = M_j \\ -1 & \text{si } M_i \neq M_j \end{cases}$$

où M_i désigne le $i^{ème}$ caractère de la première chaîne et M_j le $j^{ème}$ caractère de la 2è chaîne.

Le but est d'implémenter en C une fonction permettant de construire une telle matrice à partir de deux chaînes de caractères introduites par l'utilisateur. Chaque cellule devra, en plus de son score, conserver la/une des2 cellule(s) parente(s), pour laquelle le maximum de l'équation (1) a été atteint (voir figure 1).

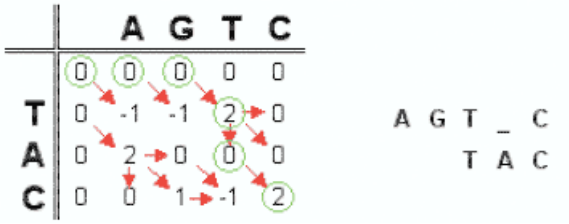


Fig. 1 – Exemple de matrice. Les flèches représentent les cellules parentes.

La seconde partie de l'algorithme permet de retrouver un alignement correct. Pour cela, il faut rechercher un chemin en partant de la cellule $M_{m,n}$ et arrivant à la cellule $M_{0,0}$, en utilisant à chaque fois la cellule parente stockée.

Pour implémenter cet algorithme, vous procèderez par étapes:

1. Dans le fichier `needle.h` décrivez les structures de données:
   - Pour comparer 2 chaînes `s1` et `s2` de longueurs respectives `l1` et `l2`, vous aurez besoin de créer un tableau de `(l2 + 1) x (l1 + 1)` cellules (en plaçant s1 horizontalement et s2 verticalement).
   - Chaque cellule contient une valeur (numérique) et une référence vers la cellule prédécesseur (il n'y a que 3 possibilités: immédiatement à gauche, en haut ou en diagonale).

2. Ecrivez ensuite dans `needle.c` une fonction `computeTable` qui prend en entrée les deux chaînes de caractères et qui construit la table en affectant à chaque cellule un prédécesseur et une valeur selon les équations précédentes :
   - Pour la première ligne, le prédécesseur est à gauche et la valeur 0.
   - Pour la première colonne, le prédécesseur est en haut et la valeur 0.
   - Remplir ensuite le tableau selon les équations précédentes, en mémorisant le prédécesseur

3. Ecrivez une fonction `extractSolution` qui prend en entrée la table et calcule le chemin de $M_{0,0}$ à $M_{m,n}$ en utilisant le prédécesseur de chaque cellule (Indication: vous devrez commencer à calculer le chemin par la fin puis l'inverser).

4. Vous pourrez ensuite tester votre programme à l'aide de la fonction

```
void afficheSolution(Solution* sol, const char* s1, const char* s2) {
 int i,j;
 for (i=0,j=0; i<sol->size; ++i) {
  switch (sol->dirs[i]) {
   case DirDiag: /*continued*/
   case DirHorz: printf("%c",s1[j]); ++j; break;
   case DirVert: printf("_");
  }
 }
 printf("\n");
 for (i=0,j=0; i<sol->size; ++i) {
  switch (sol->dirs[i]) {
   case DirDiag: /*continued*/
   case DirVert: printf("%c",s2[j]); ++j; break;
   case DirHorz: printf("_");
  }
 }
 printf("\n");
}
```

   Qui vous donnera pour les valeurs:

```
s1 = "Bonjour monsieur, quelle heure est-il à votre montre ?"
s2 = "Bonne journée madame, que l'heureuse fillette vous montre le chemin"
```

   le résultat suivant:

```
Bon___jour____ monsieur, quelle heure est-_il à__ votre montre ?________
Bonne journée madame__, que l'_heureu_se fillette vous_ montre le chemin
```

5. **Optionnel**: vous pouvez à présent aisément remplacer l'algorithme de Needleman-Wunsch par l'algorithme LCS (Longest Common Subsequence) si vous le souhaitez, remplacez simplement la formule par:

$$M_{i,j} = \begin{cases} M_{i-1,j-1} + 1 & \text{si } x_i = y_j \\ \max(M_{i-1,j}, M_{i,j-1}) & \text{si } x_i \neq y_j \end{cases}$$

Dernière mise à jour le 15 avril 2019
Last modified: Mon Apr 15, 2019