Langage C
TP n° 6 : Chaînes de caractères

Important : Une chaîne de caractères est une suite de caractères se terminant avec le caractère nul ('\0'). Cette suite de caractères peut être stockée dans un tableau ou plus généralement dans une zone mémoire allouée par un `malloc` et dont l'adresse est stockée dans un pointeur sur `char` (donc un `char *`). **Il y a en C plusieurs fonctions de manipulation de chaînes, mais pour ce TP, nous n'autorisons, sauf mention explicite, que la fonction `strlen`.** L'allocation de zones mémoire se fera avec `malloc` et les recopies se feront avec `memmove`.

Exercice 1 : Opérations de base

1. Écrire une fonction `char * dupliquer(const char * s)` qui duplique une chaîne de caractères pointée par `s` et renvoie l'adresse de la nouvelle zone mémoire où une copie de `s` a été stockée.

2. Écrire une fonction `int ordrealpha(const char * s1, const char * s2)` qui prend deux pointeurs sur des chaînes ne contenant que des lettres en argument et retourne 1 si la chaîne d'adresse `s1` est supérieure dans l'ordre alphabétique à celle de `s2`, -1 si c'est l'inverse et 0 si les deux chaînes sont identiques.

3. Écrire une fonction `char * multiplier(const char *s, unsigned int n)` qui retourne un pointeur sur une nouvelle chaîne correspondant à la concaténation de `n` fois la chaîne pointée par `s`.

Exercice 2 : Passer des arguments à la fonction `main`

1. Modifier le code de l'exercice précédent pour invoquer la fonction `ordrealpha` sur deux chaînes passées en argument de la fonction `main`.

2. Modifier le code de l'exercice précédent pour invoquer la fonction `multiplier` sur une chaîne et un entier passés en argument de la fonction `main`. Pour convertir une chaîne en un entier (lorsque c'est possible !), on pourra utiliser la fonction `int atoi(const char *str)`.

Exercice 3 : Algorithmique du texte

Dans cet exercice, on travaille sur une représentation de l'ADN sous forme de chaînes de caractères composées des caractères a, c, g, t. On nomme *mutation* une différence d'une chaîne par rapport à une autre de même taille. Par exemple, dans `"acca"`, `"cc"` à l'indice 1 est une mutation de longueur 2 par rapport à la chaîne `"aaaa"`. Une mutation est représentée par la structure suivante :

```c
typedef struct {
  size_t indice;
  size_t len;
} mutation;
```

1. Écrire une fonction `int nboc(const char *s, const char *sub)` qui renvoie le nombre d'occurrences de la chaîne d'adresse `sub` dans celle d'adresse `s`. Par exemple `"aa"` a trois occurrences dans `"aaacaa"` (aux positions `0`, `1` et `4`).
2. Écrire une fonction `mutation diff(const char *s, const char *t)` qui renvoie la première mutation de `t` par rapport à `s` (on vérifiera que les deux chaînes pointées sont de même longueur). Par exemple, si `m = diff("acca", "aaaa")`, alors `m.indice = 1` et `m.len = 2`. Si les chaînes sont identiques, la mutation renvoyée est de longueur nulle et d'indice quelconque.
3. En se servant de cette fonction, écrire `mutation longest(const char *s, const char *t)` qui renvoie la première plus longue mutation de `t` par rapport à `s`. Par exemple, si `m = longest("atcgatatt", "aaagccata")`, alors `m.indice = 1` et `m.len = 2`. Pour cela, on utilisera `diff` sur `s` et `t` puis à nouveau sur les suffixes de ces chaînes commençant après la première mutation pour trouver la deuxième, et ainsi de suite.
4. Écrire une fonction `char *longest_string(const char *s, const char *t)` qui renvoie l'adresse d'une chaîne qui est une copie de la plus longue mutation de `t` par rapport à `s`.

Exercice 4 :

Dans les questions qui suivent, on travaille sur des chaînes composées de caractères alphabétiques et d'espaces. Un *mot* d'une chaîne est une suite non vide maximale de caractères adjacents dans la chaîne et différents du caractère espace (`' '`).
Par exemple, la suite des mots de `" a aa ba a bbbb "` est `"a"`, `"aa"`, `"ba"`, `"a"` et `"bbbb"`.
Les remarques suivantes sont à garder en tête :

1. Il peut y avoir plusieurs espaces entre deux mots.
2. Il peut y avoir des espaces avant le premier ou après le dernier.
3. Même non vide, une chaîne peut ne contenir aucun mot – si elle ne contient que des espaces.

1. Écrire une fonction `int nbr_words(const char *s)` renvoyant le nombre de mots de la chaîne d'adresse `s`. Par exemple, si cette chaîne est `" a aa ba a bbbb "`, la fonction doit renvoyer `5`.

2. Écrire une fonction `int word_len(const char *w)`. Cette fonction suppose que `w` est l'adresse d'une lettre dans une chaîne de caractères (à vérifier avec un `assert`) et elle doit renvoyer la longueur du mot commençant par cette lettre. Par exemple, si la chaîne `" abc d"` est à l'adresse `s`, l'adresse `s + 1` est celle du caractère `'a'` dans la chaîne, et `word_len(s + 1)` renvoie `3`.

3. Écrire une fonction `char *extract_word(const char *w,int * pl)`. Cette fonction suppose que `w` est l'adresse d'une lettre dans une chaîne de caractères (à vérifier !) et doit copier le mot commençant avec elle dans une nouvelle chaîne de caractères dont on renverra l'adresse (on allouera la zone mémoire avec `malloc` et on utilisera `memmove`...). Et on stockera à l'adresse `pl` la longueur de la chaîne extraite.

4. Écrire une fonction `char *next_word(const char *w)` qui cherche, à partir de l'adresse `w` (inclue), la première adresse de caractère qui n'est ni un espace, ni `'\0'` si cette adresse existe (et renvoie NULL sinon).

Remarque : Plutôt que de considérer que deux mots sont séparés par un simple espace, on pourrait aussi prendre en compte les tabulations (`'\t'`), les retours à la ligne (`'\n'`), *etc.* Pour cela on peut facilement adapter les fonctions précédentes en utilisant la fonction `int isspace(int c)` de la librairie standard.

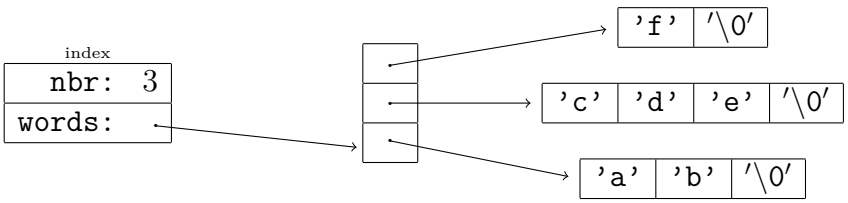
Exercice 5 : Des caractères très très star

Lorsqu'on veut manipuler plusieurs chaînes de caractères, on peut utiliser des tableaux. Nous obtenons alors des tableaux de *pointeurs vers des chaînes de caractères*, donc des tableaux de `char *`. L'adresse d'une telle zone mémoire est donc de type `char * *`.
Dans cet exercice, on va définir un type spécial pour cela :

```
1  typedef struct {
2    int nbr;
3    char **words;
4  } w_index;
```

Cela nous permettra, dans la suite de l'exercice précédent, de construire l'ensemble des mots contenus dans une chaîne de caractères.
Nous appellerons *index* toute valeur de type `w_index` respectant les contraintes suivantes :

1. La structure a été allouée par `malloc`.
2. La valeur de son champ `words` est l'adresse d'une zone mémoire allouée par `malloc` contenant une suite de `nbr` pointeurs de type `char *`.
3. Chacun de ces pointeurs est l'adresse d'une chaîne non vide sans espacement, c'est-à-dire réduite à un mot, toujours allouée par `malloc`.



La Figure ci-dessus est un exemple d'index contenant trois mots : le premier mot est `"ab"`, le second `"cde"` et le troisième `"f"`.

1. Écrire une fonction `void free_index(w_index *pi)` qui libère (par des `free`) *tout* l'espace-mémoire alloué pour l'index d'adresse `pi`.

2. Écrire une fonction `void print_index(w_index *pi)` qui affiche joliment l'index d'adresse `pi`.

3. Écrire une fonction `int size_words(w_index *pi)` qui renvoie le nombre total de caractères des mots de l'index d'adresse `pi`.

4. Écrire une fonction `char *concat_words(w_index *pi)` qui retourne la concaténation des mots de l'index d'adresse `pi`. Cette fonction doit allouer un espace mémoire suffisant par `malloc`, construire et renvoyer l'adresse d'une nouvelle chaîne formée de la concaténation de tous les mots de l'index, en séparant chaque couple de mots successifs par un unique espace. Utiliser `size_words` et tenir compte des espaces et du caractère nul dans le calcul de la taille d'allocation.

Remarque. Rien n'interdit dans la définition d'un index que son ensemble de mots soit vide (`nbr` vaut `0`). Dans ce cas particulier, la fonction doit renvoyer une chaîne vide.

5. Écrire une fonction `w_index *cons_index(const char *s)`. Cette fonction suppose que `s` est l'adresse d'une chaîne de caractères. Elle doit allouer par `malloc` une structure de type `w_index`, l'initialiser pour former l'index des mots de la chaîne, puis renvoyer l'adresse de l'index.

Il faudra donc initialiser la valeur du champ `words` de la structure par un second `malloc` en allouant une zone-mémoire permettant de stocker `nbr_words(s)` pointeurs vers `char`.

Soit `pi` l'adresse de la structure allouée. Le nombre de mots contenus dans la chaîne d'adresse `s` sera copié dans `pi -> nbr`. Pour chaque mot de la chaîne de rang `i` dans la suite des mots de la chaîne d'adresse `s`, la fonction doit initialiser `pi -> words[i]` à l'adresse d'une copie de ce mot. On utilisera la fonction `extract_word` de l'exercice précédent, et la fonction `next_word` afin de rechercher le début du mot suivant dans la chaîne.
Exemple. Considérons le code suivant :

```
1  w_index *pi = cons_index(" ab   cde  f ");
2  char *s = concat_words (pi);
3  printf ("%s\n", s);
4  free (s);
5  free_index (pi);
```

Après l'appel de `cons_index`, le pointeur `pi` vaut l'adresse d'une nouvelle structure de type `w_index`, et `pi -> words` contient l'adresse d'une zone-mémoire contenant trois adresses de chaînes :

1. `p -> nbr` vaut `3`,
2. `p -> words[0]` est l'adresse d'une nouvelle chaîne `"ab"`,
3. `p -> words[1]` est l'adresse d'une nouvelle chaîne `"cde"`,
4. `p -> words[2]` est l'adresse d'une nouvelle chaîne `"f"`.

En appliquant `concat_words` à `pi`, on obtient une chaîne contenant la même suite de mots que la chaîne initiale - sans espaces avant le premier mot et après le dernier, et avec un seul espace entre chaque couple de mots adjacents.

6. Modifier la fonction main pour pouvoir donner une chaîne de caractères (avec des espaces), appeler la fonction `cons_index` et afficher le résultat.