Corrigé TP3

1 INF203

Se référer au corrigé du TP2 pour une explication sur `installeTP.sh`.

2 Pour s'échauffer

[a] **Concaténer** signifie mettre bout à bout des chaînes de caractère. Dans l'exemple, la concaténation des chaînes `ploum`, `tra` et `lala` doit donner `ploumtralala`. C'est à dire le tableau de caractère suivant :

'p'	'l'	'o'	'u'	'm'	't'	'r'	'a'	'l'	'a'	'l'	'a'	'\0'	

[b] Pour tester les différentes fonctions que l'on va écrire (`mon_strlen`, `mon_strcpy`, `mon_strcat`, `mon_strcmp`), pas le choix, il faut pour chacune d'entre elles écrire un appel pour chacun des cas possibles. C'est un travail long et fastidieux qu'il est pourtant indispensable de faire pour s'assurer qu'un programme fonctionne correctement.

3 Implémentation possible pour `chaines.c`

```c
#include <stdio.h>
#include <string.h>
#include "generer_entier.c"
#include "bouts_de_phrases.c"

/* longueur de la chaine passee en parametre */
int mon_strlen(char ch[]) {
   int i=0 ;
   while (ch[i] != '\0') {
      i++;
   }
   return i ;
}

void mon_strcpy(char destination[], char source[]) {
   int i=0;
   while (source[i] != '\0') {
      destination[i] = source[i];
      i++;
   }
   destination[i] = '\0';
}

void mon_strcat(char destination[], char source[]) {
   int i=0, j=0;
   while (destination[i] != '\0') {
      i++;
   }
   while (source[j] != '\0') {
      destination[i+j] = source[j];
      j++;
   }
   destination[i+j] = '\0';
}

int mon_strcmp(char chaine1[], char chaine2[]) {
   int resultat = 0;

   int i=0;
   while (chaine1[i] == chaine2[i] && chaine1[i] != '\0') {
      i++;
   }

   if (chaine1[i] == '\0' && chaine2[i] == '\0') {
      resultat = 0;
   } else {
      if (chaine1[i] < chaine2[i]) {
         resultat = -1;
      } else {
         resultat = 1;
      }
   }

   return resultat;
}
```

Avant de parcourir le `main` du fichier `chaines.c`, notons plusieurs choses :

- **ATTENTION !** Toutes les fonctions ci-dessus font l'hypothèse que les pointeurs qui leur sont passés en paramètre sont **non nuls**.
- **ATTENTION !** Toutes celles qui modifient une chaine de caractère considèrent que la mémoire a déjà été allouée. Ex : `mon_strcat` considère que `destination` est un tableau suffisament grand pour stocker le résultat de la concaténation.
- On remarque que la fonction `mon_strcpy` et les suivantes n'utilisent pas `mon_strlen`. Ce n'est pas par masochisme, mais parce que cela permet d'éviter des parcours de chaine inutiles ! Pour connaître la taille, il faut regarder chaque caractère un par un jusqu'à la fin. C'est une opération coûteuse que l'on doit éviter si l'on peut.
- Si vous lisez cette implémation consciencieusement, vous ne manquerez pas de constater que `mon_strcmp`

est écrite un peu légèrement ... En effet, il ne suffit pas qu'un caractère soit situé avant un autre dans la table `ASCII` pour qu'il le précède selon l'ordre lexicographique. Par exemple, selon la table `ASCII`, le caractère `'Z'` est situé avant le caractère `'a'`. Or la lettre z majuscule n'est pas avant la lettre a minuscule dans l'alphabet. Pourtant, est-ce une mauvaise implémentation ? Elle a le mérite d'être très simple. Par ailleurs, elle est peut être suffisante selon le contexte, par exemple si l'on sait que l'on ne traîte que des chaînes contenant des lettres minuscules. Prenez donc garde à ne pas passer trop de temps à perfectionner un programme qui résout déjà 99% des cas !

```c
int main() {
   char chaine[50] ;
   int mon_resultat ;

   printf("un mot (pas plus grand que 50 caractères !) à mesurer ?\n") ;
   scanf("%s", chaine) ;

   mon_resultat=mon_strlen(chaine) ;
   if ( mon_resultat == strlen(chaine) ) {
      printf("longueur du mot ('%s') : %d\n\n", chaine, mon_resultat) ;
   } else {
      printf("non, la longueur de '%s' n'est pas %d\n", chaine, mon_resultat) ;
   }

   char resultat[50];
   mon_strcpy(resultat, "copie simple");
   printf ("mon_strcpy(resultat, \"copie simple\") \t\t resultat = <%s>\n", resultat);
   mon_strcpy(resultat, "");
   printf ("mon_strcpy(resultat, \"\") \t\t\t\t\t resultat = <%s>\n", resultat);

   mon_strcat(resultat, "ajout1");
   printf ("mon_strcat(resultat, \"ajout1\") \t\t\t\t resultat = <%s>\n", resultat);
   mon_strcat(resultat, " ajout2");
   printf ("mon_strcat(resultat, \" ajout2\") \t\t\t resultat = <%s>\n", resultat);
   mon_strcat(resultat, "");
   printf ("mon_strcat(resultat, \"\") \t\t\t\t\t resultat = <%s>\n\n", resultat);

   printf ("mon_strcmp(\"\", \"\") \t\t\t = %d\n", mon_strcmp("", ""));
   printf ("mon_strcmp(\"abc\", \"abc\") \t = %d\n", mon_strcmp("abc", "abc"));
   printf ("mon_strcmp(\"abc\", \"xbc\") \t = %d\n", mon_strcmp("abc", "xbc"));
   printf ("mon_strcmp(\"abc\", \"abC\") \t = %d\n", mon_strcmp("abc", "abC"));
   printf ("mon_strcmp(\"a\", \"abc\") \t\t = %d\n", mon_strcmp("a", "abc"));
   return 0 ;
}
```

Résultat de l'exécution de cette implémentation :

```
corentin@gazelle:~ $ ./chaine
un mot (pas plus grand que 50 caractères !) à mesurer ?
maison
longueur du mot ('maison') : 6

mon_strcpy(resultat, "copie simple")          resultat = <copie simple>
mon_strcpy(resultat, "")                      resultat = <>
mon_strcat(resultat, "ajout1")                resultat = <ajout1>
mon_strcat(resultat, " ajout2")               resultat = <ajout1 ajout2>
mon_strcat(resultat, "")                      resultat = <ajout1 ajout2>

mon_strcmp("", "")            = 0
mon_strcmp("abc", "abc")      = 0
mon_strcmp("abc", "xbc")      = -1
mon_strcmp("abc", "abC")      = 1
mon_strcmp("a", "abc")        = -1
```

4 Fabriquer des phrases

[c] On a bien sûr pensé, lors de l'écriture de `phrases.c`, à ajouter des espaces entre les chaines. Cela donne donc :

```
    char Sujet[50]="la petite souris";
    char Verbe[50]="mange";
    char Compl[50]="le gros chat";
    char Phrase[150];

    mon_strcat(Phrase, Sujet);
    mon_strcat(Phrase, " ");
    mon_strcat(Phrase, Verbe);
    mon_strcat(Phrase, " ");
    mon_strcat(Phrase, Compl);
```

[d] Pour compter les mots dans la phrase, si on l'a construite correctement, il suffit de compter les espaces. Sans oublier de rajouter 1 à la fin, puisqu'il y a une espace entre chaque mot, mais pas à la fin de la phrase. Implémentation complète de `phrases.c` :

```
#include <stdio.h>
#include <string.h>
#include "generer_entier.c"
#include "bouts_de_phrases.c"

int mon_strlen(char ch[]) {
    ...
}

void mon_strcpy(char destination[], char source[]) {
    ...
}

void mon_strcat(char destination[], char source[]) {
    ...
}

int mon_strcmp(char chaine1[], char chaine2[]) {
    ...
}

void faire_phrase(char sujet[], char verbe[], char compl[], char phrase[]) {
    mon_strcat(phrase, sujet);
    mon_strcat(phrase, " ");
    mon_strcat(phrase, verbe);
    mon_strcat(phrase, " ");
    mon_strcat(phrase, compl);
}

int nb_mots(char phrase[]) {
    int i=0, nb=0;
    while (phrase[i] != '\0') {
        if (phrase[i] == ' ') {
            nb++;
        }
        i++;
    }
    return ++nb;
}

int main() {
    long num_sujet = generer_entier(10);
    long num_verbe = generer_entier(10);
    long num_compl = generer_entier(10);

    char Phrase[150]="";

    faire_phrase(sujet[num_sujet], verbe[num_verbe], complement[num_compl], Phrase);

    printf ("phrase (%d) : %s\n", nb_mots(Phrase), Phrase);
    return 0 ;
}
```

5 Sacs de billes

[e] Dans la fonction `lire_joueur`, on veut mettre à jour la variable contenant les informations d'un joueur avec ce que l'utilisateur aura tapé au clavier. On doit donc connaître l'adresse où est effectivement stockée cette variable. Or, dans le prototype de fonction : `lire_joueur(joueur j)`, le paramètre `j` n'est pas un pointeur. C'est donc un passage par **copie**. Si l'on modifie `j` dans cette fonction, c'est la copie fabriquée pour la fonction qui sera modifiée et non pas la variable originale.

[f] Une implémentation possible de billes :

```c
#include <stdio.h>

typedef struct{
   char pseudo[20];
   int nb_billes;
} joueur;

joueur atchoum ={"Atchoum", 42};
joueur dormeur={"Dormeur", 25};
joueur grincheux={"Grincheux", 3};
joueur joyeux={"Joyeux", 100};
joueur prof={"Prof", 2};
joueur simplet={"Simplet", 0};
joueur timide={"Timide", 12};

void afficher_joueur_V1(joueur j) {
   printf("%s a %d billes\n", j.pseudo, j.nb_billes);
}

void afficher_joueur_V2(joueur *pj) {
   printf("%s a %d billes\n", pj->pseudo, pj->nb_billes);
}

void lire_joueur(joueur *pj) {
   printf ("pseudo: ");
   scanf ("%s", pj->pseudo);
   printf ("nb billes: ");
   scanf ("%d", &pj->nb_billes);
}

void transferer_V2(joueur *pj1, joueur *pj2) {
   strcpy(pj2->pseudo, pj1->pseudo);
   pj2->nb_billes = pj1->nb_billes;
}

int main() {
   afficher_joueur_V1(atchoum);
   afficher_joueur_V2(&atchoum);
   lire_joueur(&atchoum);
   afficher_joueur_V2(&atchoum);

   printf("\n\n");
   afficher_joueur_V2(&timide);
   afficher_joueur_V2(&simplet);
   transferer_V2(&simplet, &timide);
   afficher_joueur_V2(&timide);
   afficher_joueur_V2(&simplet);
   return 0;
}
```

6 La command `diff`

On considère les fichiers `d1.c`, `d2.c` et `d3.c` contenant respectivement :

```c
#include <stdio.h>

int main() {
   printf("Hello, world!");
}
```

```c
#include <stdio.h>

int main() {
   printf("Hello, world!");
}
```

```c
#include <stdio.h>

int main() {
   printf("Hello, world!\n");
}
```

À première vue, ils semblent identiques. Voici le résultat de plusieurs commandes `diff`.

```
corentin@gazelle:listing $ diff d1.c d2.c
corentin@gazelle:listing $
```

On voit que le diff de `d1.c` et `d2.c` se termine sans rien imprimer sur l'écran. C'est donc que les deux fichiers sont identiques.

```
corentin@gazelle:listing $ diff d1.c d3.c
4c4
<         printf("Hello, world!");
---
>         printf("Hello, world!\n");
corentin@gazelle:listing $
```

En revanche, le diff de `d1.c` et `d3.c` donne la liste des différences entre les deux fichiers. Plusieurs explications :

gauche/droite Ici, on appelle *fichier de gauche* `d1.c` car c'est le premier paramètre donné à `diff`. Il est donc situé à gauche sur la ligne de commande. Par suite, `d3.c` est le fichier de droite.

4c4
4 → ligne 4 de `d1.c`
c → a changé
4 → correspond désormais à ligne 4 de `d3.c`

`<` Indique que la ligne qui suit se trouve dans le fichier de gauche

`>` Indique que la ligne qui suit se trouve dans le gichier de droite

`---` Sépare les lignes situés dans des fichiers différents