Programmation C avancée
TP 2

Pour ce second TP, les exercices restent encore de l'ordre de la mise en jambes. Les exercices sont toujours indépendants. Vous pouvez finir la première fiche ou passer directement à celle-ci si vous avez à peu près terminé la première. Histoire de se remettre à flot, veuillez à bien diversifier les thèmes des exercices que vous traitez (entrées/sorties, allocation, argument du `main`, types entiers, manipulation de chaîne).

Exercice 1 Manipulation d'entiers

Parmi les nombres entiers compris entre 0 et 500, certains réalisent une propriété rigolote : ils sont la somme des cubes de leurs chiffres écrits en base 10. Soit un entier $0 \leqslant n \leqslant 500$, on peut écrire $n = \overline{xyz}^{10}$ en base 10 (ce qui veut juste dire que x est le nombre des centaines, y des dizaines et z des unités). Par exemple, si $n = 381$, alors $x = 3, y = 8, z = 1$. On voit que n réalise la propriété si et seulement si

$$n = x^3 + y^3 + z^3$$

Trouver tous ces nombres !

Exercice 2 Manipulation de flottants

Vos enseignants en mathématiques vous ont martelés que la série harmonique diverge vers $+\infty$. Cette suite $(U_n)_{n\in\mathbb{N}}$ est définie comme il suit

$$U_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$$

Malheureusement, en informatique, avec les types flottants, c'est pas si facile de le voir. Les types flottants étant non exacts, ils sont limités pour faire ce genre de calculs.

- Écrire un programme C qui calcule pas à pas les termes U_n de cette suite et affiche la valeur de la suite toutes les milles itérations en utilisant le type `float`. Ceci devrait vous donner pour les premières valeurs.

```
1000 : 7.485478
2000 : 8.178369
3000 : 8.583756
4000 : 8.871394
5000 : 9.094514
6000 : 9.276819
7000 : 9.430960
8000 : 9.564480
9000 : 9.682266
10000 : 9.787613
```

  Cette suite finit par stationner (la valeur ne grimpe plus...). Notez cette valeur stationnaire.

- Recommencez le même procédé avec des `double`. Pour apprécier la différence de précision (plus long à calculer même si les machines sont performantes), diminuez la fréquence d'affichage et utilisez des entiers plus long.

Exercice 3 Miroir sur les chaînes

Écrire une fonction `miroir_string` qui prend en paramètre une chaîne de caractères et qui inverse ses caractères (la chaîne de caractères passée en paramètre est modifiée). Pour tester votre fonction, écrivez un programme affichant l'argument du programme à l'envers. Par exemple :

```
$>miroir trace
ecart
```

Attention, on ne peut pas modifier `argv` directement !

Exercice 4 Humour d'informaticien

Ne rigolez pas! Le développement du noyau `Linux` et de ses différentes distributions est très sérieux. Les propositions d'amélioration et les correctifs sont systématiquement soumis au contrôle d'autres contributeurs sérieux avant intégration. Toutefois, on retrouve quelques perles dans les dépôts officiels.

Au milieu des packages officiels et fondamentaux se trouve le package `cowsay` qui, couplé avec le package `fortune`, peut donner

```
nborie@perceval:~/ > fortune | cowsay -n
 ______________________________________________________________________
/ You are destined to become the commandant of the fighting men of the \
\ department of transportation.                                        /
 ----------------------------------------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
nborie@perceval:~/ >
```

De manière plus élémentaire, `cowsay` prend en argument une chaine et affiche la chaîne dans une bulle:

```
nborie@perceval:~/ > cowsay "Non mais c'est quoi ce TP sérieux ?"
 ____________________________________
< Non mais c'est quoi ce TP sérieux ? >
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
nborie@perceval:~/ >
```

- Écrire un programme `blasay` qui prend en argument une chaîne de caractères et affiche le dessin ascii art de votre choix avec une bulle de taille adaptée au message et le

message à l'intérieur. Dans un soucis de simplicité, on supposera que le message tient sur une ligne. Toutefois, la bulle ne doit être plus grande que le message. Remarquer aussi que le dessin est centré par rapport à la bulle.

Exercice 5 Organisation de la mémoire (pour aller plus loin...)

Pour comprendre de manière profonde le langage C, le programmeur curieux passe son temps à écrire des petits programmes à la limite du langage. Le but étant d'observer le comportement du compilateur et des exécutables. L'exercice vous propose ici de faire des tests sur la machine pour voir où place-t-elle la mémoire...

Lorsque deux variables `a` et `b` de type `char` sont accessibles (dans une fonction), l'expression `(&a-&b)` est un entier qui mesure ici la distance en nombre d'octets à l'intérieur de la mémoire entre les deux données (on reverra plus tard en cours l'arithmétique sur les pointeurs).

- Écrire un programme (une fonction `main` doit suffire) avec une ou plusieurs variables `static char`, un ou plusieurs caractères globaux, un ou plusieurs caractère locaux (à votre fonction `main`), un ou plusieurs tableaux de `char` alloués dynamiquement (dans votre `main`). Afficher la distance en nombre d'octets de toutes ces variables.

Par exemple avec une distribution Linux 64 bits exploitant un processeur Intel Core i7 et deux tableaux alloués dynamiquement de 103 `char`:

```
static <--> globale : 4
static <--> locale1 : -904909386
static <--> locale2 : -904909387
static <--> allouée1 : 15982524
static <--> allouée2 : 15982636
globale <--> locale1 : 904909382
globale <--> locale2 : 904909382
globale <--> allouée1 : 15982520
globale <--> allouée2 : 15982632
locale1 <--> allouée1 : -888926862
locale1 <--> allouée2 : -888926750
locale1 <--> locale2 : 1
alloué1 <--> allouée2 : 112
```

Dans le même état d'esprit, regardez la taille des structures suivantes (en utilisant la commande `sizeof`).

```
1  typedef struct{
2    char c1;
3    int i1;
4    char c2;
5    int i2;
6  }Test1;
7
8  typedef struct{
9    char c1;
10   char c2;
```

```
11      int i1;
12      int i2
13  }Test2;
```

- Écrivez un programme utilisant deux variables locales de type `Test1` et `Test2` (un petit `main` doit suffire). Regardez la taille mémoire de ces deux structures et réfléchissez. Tentez de tester les options `-O{1,2,3}` de `gcc` pour voir si le compilateur optimise (ou pas...).