TP1

Langage C (LC4)

semaine du 6 février

1 Premiers pas en C

1.1 Bonjour le monde

Écrivez le programme `hello.c` suivant :

```
#include <stdio.h>

int main() {
  fputs("pouet\n", stdout);
  return 0;
```

Question 3. Trouvez dans la page `man ascii` les codes ascii décimaux pour les lettres 'a' et 'z'.

Question 4. Expliquez ligne par ligne ce que fait le programme.

Question 5. Améliorez le programme pour qu'il transforme aussi les majuscules.

Question 6. Que se passe-t-il si on entre plus de 64 caractères ? Améliorez le programme pour qu'il marche sur de long textes.

Question 7. Produisez une version chiffrée de la « GNU General Public License » (`http://www.gnu.org/licenses/gpl.txt`), qui au passage est la licence d'utilisation du compilateur `gcc`.

2 Un langage de bas niveau

Deux exercices pour se rendre compte qu'on peut facilement faire des bêtises lorsqu'on programme en C, et que ça peut être très difficile de s'en rendre compte puis de trouver d'où vient l'erreur...

2.1 La mémoire

Le programme suivant déclare un tableau de 128 valeurs de type **`int`**, et avant toute initialisation, il affiche les valeurs que le tableau contient.

```
#include <stdio.h>

int main() {
  int i;
  int t[128];
  for(i = 0; i < 128; i++) printf("valeur %d: %d\n", i, t[i]);
  return 0;
}
```

Question 8. Qu'est ce que ce programme affiche ? Affiche t-il toujours la même chose quand on l'exécute à nouveau ? À quoi correspondent les valeurs présentes dans le tableau ?

Question 9. Ajoutez une instruction ou une suite d'instructions pour initialiser toutes les cases du tableau à la valeur 1, immédiatement après sa déclaration.

Question 10. Qu'est ce qui se passe si on modifie notre programme pour afficher les valeurs contenues dans le tableau pour des indices supérieurs à la limite 128 (ou inférieurs à 0) ? Essayez aussi avec de très grands indices.

2.2 Les types de données

On a utilisé des tableaux d'entiers pour représenter des chaînes de caractères. En fait le type `string` n'existe pas en C, on se débrouille avec les opérations qu'on sait faire sur les tableaux. Cela permet de rester proche de la représentation mémoire et de contrôler exactement ce qu'il se passe, mais on verra plus tard que ce n'est pas toujours facile de faire ce qu'on veut.

Pour le type booléen c'est la même chose, il n'existe pas. Les processeurs ne savent pas travailler avec des quantités de données aussi petites, et c'est la raison (bête direz vous ?) pour laquelle on ne l'a pas introduit dans le langage C. On utilise des nombres (de type **`char`**, **`int`**, **`long`**... ou **`float`**, **`double`**) pour représenter le vrai et le faux.

Question 11. Quel effet a le code suivant :

```
if (2) printf("oui\n");
else printf("non\n");
```

Question 12. Expérimentez et déterminez parmi les entiers et les flottants quelles valeurs représentent le 'vrai' et quelles valeurs représentent le 'faux' dans les instructions conditionnelles.

En particulier, les opérations de comparaison sont juste des opérations sur des nombres, et il est tout à fait permis (même si c'est assez mal vu) d'écrire du code comme `7 + (5 > 3)`.

Question 13. Combien vaut, en C, l'expression `7 + (5 > 3)` ?

3 Une exécution rapide

3.1 Compilation optimisée

Essayez votre programme `rot13.c` amélioré sur un grand fichier texte, par exemple la liste des mots de la langue française que l'on peut obtenir avec :

```
aspell -d fr dump master > mots.txt
```

Si le dictionnaire français n'est pas présent, 10 copies du dictionnaire anglais feront l'affaire :

```
for i in $(seq 1 10); do aspell -d en dump master; done > mots.txt
```

Question 14. Quel est le temps d'exécution du programme ? (utilisez la commande `time`)

Le compilateur `gcc` possède une option `-O` qui contrôle le niveau d'optimisation utilisé. (À ne pas confondre avec l'option `-o` qui permet de désigner le nom de l'exécutable produit.)

Question 15. Essayez de compiler avec `-O3` et comparez les temps de calcul.

Malgré un haut niveau d'optimisation, il y a certaines choses que le compilateur ne sait pas deviner. L'utilisation de la fonction `strlen` n'est pas vraiment nécessaire, or elle parcourt

Tout comme le format binaire, le format non assemblé dépend du type de machine sur lequel on travaille, et on ne peut pas le réutiliser sur une machine différente.

Question 17. Comparez les exécutables non assemblés que l'on obtient avec et sans optimisation.

4 Boucles et tableaux

S'il vous reste encore du temps, c'est maintenant à votre tour de programmer...

4.1 Permutations

Les permutations sont fréquemment croisées en mathématiques, mais aussi en informatique où elles sont utilisées par exemple en cryptographie ou pour analyser l'efficacité d'algorithmes. Nous modélisons des permutations de $[0 \dots n-1]$ dans $[0 \dots n-1]$ à l'aide d'un tableau. La valeur de la case i du tableau correspond à l'image de i par la permutation. Ainsi la permutation :

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ 1 & 3 & 0 & 2 \end{array} !$$

est dénotée par le tableau C :

```
int f[4] = { 1, 3, 0, 2 } ;
```

Question 18. Écrire une fonction `void affiche_tableau(int n, int f[])` qui affiche le contenu du tableau `f` de taille `n` de manière lisible.

Question 19. Écrire une fonction `int est_identite(int n, int f[])` qui teste si une permutation `f` de $[0...n-1]$ est l'identité. Cette fonction renverra un entier valant `1` en cas de réponse positive et `0` autrement.

Question 20. Une inversion d'une permutation σ est un couple (i, j) d'entiers tels que $i < j$ et $\sigma(i) > \sigma(j)$. Écrire une fonction `int nombre_inversions(int n, int f[])` qui calcule le nombre d'inversions d'une permutation.

Question 21. Écrire la fonction `void compose_permutations(int n, int perm1[], int perm2[], int perm_composee[])` qui calcule la composée σ, de σ_1 et σ_2, telle que $\sigma = \sigma_1 \circ \sigma_2$ (et donc $\sigma(i) = \sigma_1(\sigma_2(i))$).

Question 22. Écrire un fonction `void inverse_directe(int n, int perm[], int perm_inverse[])` qui calcule l'inverse σ^{-1} de la permutation σ , tel que $\sigma^{-1} \circ \sigma = Id$.

Question 23. Écrire une fonction `int rang(int n, int perm[])` qui calcule le nombre de fois qu'il faut composer la permutation `perm` avec elle-même pour obtenir l'identité.

Question 24. Écrire une fonction `void inverse_ordre(int n, int perm[], int inverse[])` qui calcule l'inverse σ^{-1} de la permutation σ, tel que $\sigma^{-1} \circ \sigma = Id$, en utilisant la fonction précédente.