TP 1 – Codage des entiers et flottants

Téléchargez l'archive `TP1_ressources.tar.gz`. Elle contient le fichier `tp1.c` dans lequel vous travaillerez, ainsi qu'un `Makefile`. La fonction `long long int P2(int n)` de `tp1.c` calcule la n-ième puissance de 2.

1 Boutisme

1 – Écrivez la fonction de prototype `void base2(int)` affichant la représentation en base 2 d'un entier en argument (on utilisera les opérateurs de division entière / et %).

2 – Que fait la fonction C :

```
int n_b (char *addr, int i){
  return (char) 0x1 & ((char) *(addr + i/8))>>i%8;
}
```

L'ordre dans lequel les octets de la représentation des entiers sont "rangés" en mémoire varie selon les architectures. Dans la représentation de l'entier `0xDEADBEEF` sur 4 octet, l'octet valant `0xDE` est l'octet de poids le plus fort, et celui valant `EF` est celui de poids le plus faible. Certaines architecture, pour stocker cet entier à partir de l'adresse mémoire `0x00FFFF00`, enregistrent d'abord l'octet de poids le plus fort (ici `DE`) à l'adresse `0x00FFFF00`, puis `AD` à l'adresse `0x00FFFF01`, puis `BE` à l'adresse `0x00FFFF02`, puis `EF` à l'adresse `0x00FFFF03`. Ces architectures sont dîtes gros-boutistes. À l'inverse, les architectures petit boutistes stockent d'abord l'octet de poids le plus faible. Dans l'exemple, une machine petit boutiste stockerait `EF` à l'adresse `0x00FFFF00`, `BE` à l'adresse `0x00FFFF01`, `AD` à `0x00FFFF02`, et enfin `DE` à `0x00FFFF03`.

Pouvez-vous utiliser la fonction `n_b` pour déterminer le boutisme de `turing` ?

2 Entiers

La fonction `char d2c(int i)` (voir ci-dessous) convertit le chiffre `i` compris entre 0 et 36 en le caractère ASCII qui le représente (i si $0 \leq i \leq 9$, la i-ème lettre de l'alphabet sinon).

```
char d2c (int n){
 return ( n<0? '?': (n<10? '0'+n : (n<36 ? 'A' + (n-10) : '?') ));
}
```

3 – Écrivez la fonction `void baseB(int B, int n)` qui affiche la représentation en base `B` ($1 \leq$ `B` ≤ 36) de l'entier `n` passé en argument (en base 10).

3 Flottants

4 – Le but de cet exercice est de décomposer un flottant IEEE simple précision en mantisse, exposant et signe. (On ne tiendra pas compte des flottants spéciaux). Écrivez les fonctions :

- `void mantisse (float f, int result[])` qui remplit le tableau d'entiers `result[]` (auparavant alloué) avec les bits correspondant à la mantisse d'un flottant `f`
- `float mantisseNormalisee(float f)` qui retourne la valeur de la mantisse normalisée du flottant `f`,
- `void exposant (float f, int result[])` qui remplit le tableau d'entiers `result[]` (auparavant alloué) avec les bits correspondant à l'exposant d'un flottant `f`,
- `int exposantSansExces(float f)` qui retourne la valeur de l'exposant sans excès de `f`,
- `int signe(float f)` qui retourne la valeur du bit de signe de `f`.

5 – Ouvrez le fichier "`float.c`".

- Quelle serra selon vous la valeur de la variable "`a`" après la boucle ?

— Exécutez le programme et observez la valeur calculée pour "`a`". D'où peut bien provenir cette différence ? (indice : la valeur affichée est une puissance de 2)

6 – Le code présent en deuxième partie de fichier permet de calculer la somme des entiers compris entre "`plus_petit_carre`" et "`plus_gros_carre`" au carré, $\sum_{x=\text{plus\_petit}}^{\text{plus\_gros}} x \times x$. Cette somme est calculée dans les deux sens, de la plus petite valeur à la plus élevée et inversement.
Modifiez le test présent dans le fichier "`float.c`" affin de permettre au code de s'exécuter, recompilez et observez les résultats.

— Pourquoi la valeur calculée n'est-elle pas exact ?

— D'où proviens la différence entre les deux méthodes de calcul ? (Vous pouvez vous servir de l'aide disponible dans le code source)