TD n° 1 : Types de base

Exercice 1 : Codage des entiers naturels

a) Quel est le plus grand entier naturel que l'on puisse exprimer dans une base B si l'on dispose de L positions ? En conséquence, peut-on représenter le nombre d'électrons dans l'univers (soit un nombre de l'ordre de 10^{78}) dans un entier naturel codé sur 32 bits ?

b) Convertissez en binaire le nombre 238_{10} en utilisant le principe des divisions successives. Déduisez-en sa représentation hexadécimale en regroupant les bits par paquets de quatre. Peut-on aussi utiliser une méthode de regroupement pour passer de l'écriture binaire de 238 à son écriture décimale ?

c) Réalisez l'addition binaire de 238 et de 32 après avoir représenté ces nombres en binaire pur sur 8 bits (unsigned char). Que constatez-vous ? Comment éviter ce problème ici ?

d) **Question à faire chez soi :** Convertissez 238 en base 16 en utilisant la méthode des divisions successives (divisions par 16). Vérifiez que vous obtenez bien la même représentation qu'en regroupant les bits du codage binaire par paquets de quatre.

Exercice 2 : Codage des entiers relatifs

a) Quelle est la représentation de -3 et +10 en « binaire - complément à 2 » sur 8 bits (signed char) ?

b) Additionnez à présent les deux nombres (en binaire). Le résultat est-il correct ? Aurait-on obtenu un résultat correct en additionnant les représentations en « signe-valeur absolue » des deux nombres ?

Exercice 3 : Codage des réels

a) Convertir 9 et 5/32 au format IEEE 754 simple précision. Donnez les résultats en écriture binaire et en écriture hexadécimale. Additionnez ces deux nombres.

b) Pour le format IEEE simple précision, quel est l'écart entre deux nombres machines successifs ? En conséquence, combien y a-t-il de chiffres décimaux significatifs dans un nombre codé dans ce format ? A votre avis, écrire `float pi=3.14159265358979` a-t-il un sens ?

Exercice 4 : Opérateurs et conversions implicites en C

Etant données les définitions ci-dessous,

```
char  monChar1 = 'a',   monChar2 = 97;
int   monInt = 1;       long monLong = 1L;
float monFloat = 1.0f;  double monDouble = 1.0;
```

donner le type et la valeur de chacune des expressions suivantes :

```
a) monChar1 + monChar2
b) monChar1 + 1
c) monChar1 + 1.0
d) monChar1 + 1.0f
e) monFloat + monFloat
f) monChar1 + monInt + monLong
g) monChar1 + monInt + monLong + monFloat + monDouble
h) monChar1 = monChar1 + monInt + monLong + monFloat + monDouble
i) monChar1 == monChar2
j) monChar1 = monChar2 + monInt + monLong + monFloat + monDouble, monFloat = 2
```

Exercice 5 : Algorithmique sur des entiers relatifs

On dispose, dans le langage algorithmique, d'un type `booleen` pouvant prendre les valeurs `{vrai, faux}` et sur lequel on peut effectuer des opérations de comparaison (opérateur =), d'affectation (opérateur ←) et de négation (fonction `non(b: booleen): booleen`). On définit alors le type « Entier64 » comme suit :

```
type Entier64 = tableau[1..64] de booleens
```

Un Entier64 permet de coder un entier compris entre -2^{63} et $2^{63} - 1$, de façon identique au codage dans un *signed char*, un *short* ou un *int*, mais sur plus de bits.

a) Ecrire la fonction `addition(e1: Entier 64, e2: Entier64): Entier64` qui retourne la somme de 2 Entier64. On utilisera uniquement les opérations de comparaison, d'affectation et de négation de booléens. On pourra recourir à une variable interne pour la gestion de la retenue. Remarque : on bénéficie toujours du type `entier` « normal », qui permet de gérer des entiers qui restent relativement petits, comme l'indice d'un tableau.

b) Ecrire une procédure `incremente(e: Entier64)` qui ajoute 1 un Entier64, en utilisant la fonction d'addition. On supposera que l'on a déjà écrit la procédure d'affectation entre Entier64 :

```
Procédure affecte(e: Entier64, val: Entier64)
Précondition : val doit appartenir à l'intervalle -2^63 .. 2^63−1
Postcondition : e+ = val
Paramètre en donnée-résultat : e
Paramètre en donnée : val
```

Questions à faire chez soi :

c) Supposons que l'on ait déjà écrit les sous-programmes suivants, en plus des précédents :

```
Procédure initialiseZero(e: Entier64)
Précondition : aucune
Postcondition : e+ = 0
Paramètre en donnée-résultat : e
```

```
Procédure decremente(e: Entier64)
Précondition : l'entier e-1 doit appartenir à l'intervalle -2^63 .. 2^63−1
Postcondition : e+ = e- - 1
Paramètre en donnée-résultat : e

Fonction estSuperieur(e1: Entier64, e2: Entier64) : booleen
Précondition : e1 et e2 appartiennent à l'intervalle -2^63 .. 2^63−1
Résultat : retourne vrai si e1 > e2, faux sinon
Paramètres en mode donnée : e1, e2
```

Proposez alors une fonction de calcul du produit de 2 Entier64 qui ne fasse intervenir que des opérations d'addition et de décrémentation d'Entier64, en plus des opérations de comparaison et d'affectation d'Entier64. On utilisera 4 variables internes de type Entier64 : `zeroE64`, `res`, `nb_ajouts_restants` et `qte_ajoutee`. On écrira l'algorithme pour le cas où e1 et e2 sont positifs, puis on expliquera comment procéder dans le cas plus général (sans écrire l'algorithme).

d) Améliorez l'algorithme précédent en supposant que l'on dispose aussi des sous-programmes suivants :

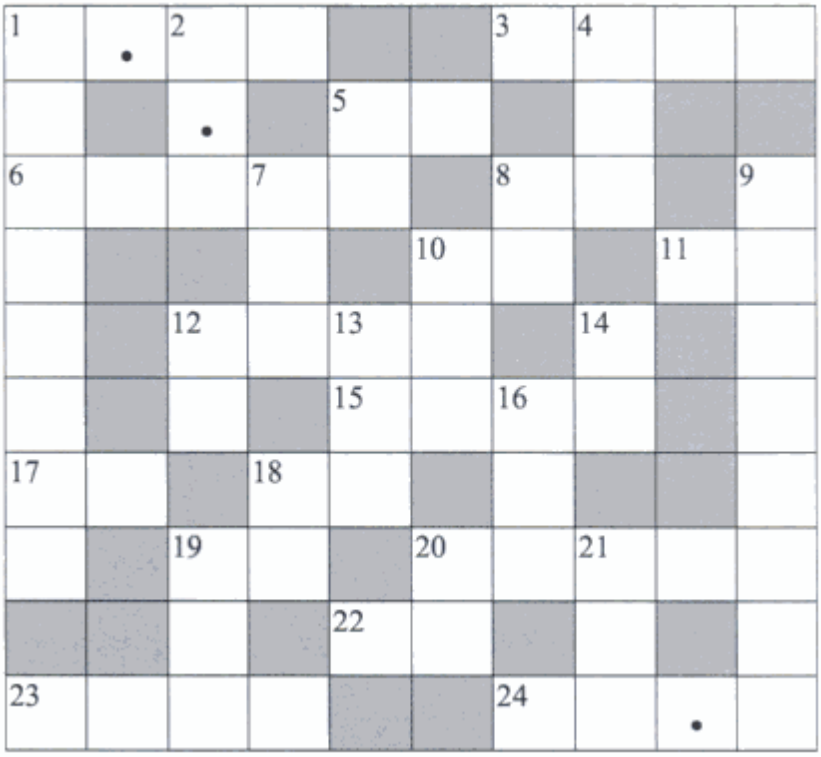
```
Procédure multiplieParDeux(e: Entier64)
Précondition : e et 2*e doivent appartenir à l'intervalle -2^63 .. 2^63−1
Postcondition : e+ = 2*e-
Paramètre en donnée-résultat : e

Procédure diviseParDeux(e: Entier64)
Précondition : e et e/2 doivent appartenir à l'intervalle -2^63 .. 2^63−1
Postcondition : e+ = e-/2 (attention, c'est une division entière)
Paramètre en donnée-résultat : e

Fonction estImpair(e: Entier64) : booleen
Précondition : e appartient à l'intervalle -2^63 .. 2^63−1
Résultat : retourne vrai si e est impair faux sinon
Paramètres en mode donnée : e
```

Indice : posez la multiplication en binaire.

e) Comparez les deux méthodes de multiplication en estimant le nombre d'opérations à effectuer :
   a) nombre de tours de boucle
   b) pour chaque tour, les nombres maximum de comparaisons, d'additions, d'affectations et de décrémentations, de tests de parité, de divisions par 2 et de multiplications par 2.

Quelle méthode vous semble la plus efficace ? Pourquoi ?

Bonus pour vous entraîner !



Chaque case peut contenir un chiffre entre 0 et 9, une lettre entre A et F, un point, ou encore le signe - .

Vous supposerez que les nombres binaires sur 8 bits sont sous la forme « complément à 2 » (donc signed), sauf si autre chose est précisé.

Précision sur le codage BCD (non vu en cours mais simple à comprendre) : En BCD, les nombres sont représentés en chiffres décimaux et chacun de ces chiffres est codé sur quatre bits. Exemple : Pour coder le nombre 127, il suffit de coder chacun des chiffres 1, 2 et 7 ce qui donne 0001, 0010, 0111.

Horizontalement

1) 1.75 en binaire
3) D_{16} en binaire
5) 00101100 en décimal
6) 22 en binaire
8) 01001000 en décimal
10) 00010010 en décimal
11) 01110001_{BCD} en décimal
12) -6 en binaire sur 4 bits
15) pseudo-mantisse de 31 lorsqu'il est stocké dans un float (sans les 0 finaux)
17) pseudo-mantisse de 0,875 lorsqu'il est stocké dans un float (sans les 0 finaux)
18) 00001011 en décimal
19) 11110111 en décimal
20) -3 en binaire sur 5 bits
22) 01010000_{BCD} en décimal
23) 10001111 en décimal
24) 2.5 en binaire

Verticalement

1) -94 en binaire sur 8 bits
2) 1.5 en binaire
4) $7A_{16}$ en décimal
5) 00101000 en décimal
7) 416 en hexadécimal
8) valeur en « Vertic. 4 » - valeur en « Horiz. 5 », en décimal
9) -1 en binaire sur 8 bits
10) 01100001 – 01011100 en binaire
12) 11100110 + 00011101 en binaire
13) 01101111 en décimal
14) 00101001 en décimal
16) l'exposant de $1/2^{122}$ quand ce nombre est stocké dans un float (écrire l'exposant en binaire, en tenant compte du décalage)
18) 00010011 en décimal
19) 10111001 en décimal
20) 00111110 + 11000100
21) 384 en hexadécimal