TP1 : Les valeurs flottantes et les union

1 Quelques calculs sur les valeurs flottantes

Afin de mieux comprendre les limites des calculs avec les valeurs flottantes, on se propose de réaliser certains calculs sur des réelles et de constater que le résultat n'est pas celui auquel on s'attendait.

1. Voici un code à compiler et exécuter :

```
float y;
y=0.1;
printf("%f", y);
return 0;
```

   Normalement, vous constaterez qu'à l'affichage, la variable **y** possède la valeur attendue... Que se passe-t-il si vous demandez à la fonction printf d'afficher 20 décimales après la virgule, en faisant :

```
printf("%.20f", y);
```

2. Voici un code à compiler et exécuter :

```
double a = 12345.0;
double b = 1e-16;

if(b==0.0)
{
	printf("b est nul\n");
}
else
{
	printf("b n'est pas nul\n");
}

if(a+b==a)
{
	printf("a+b est egal a a\n");
}
return 0;
```

   Le résultat est-il celui auquel vous vous attendiez ? Qu'en déduisez-vous à propos de l'addition d'une grande valeur flottante avec une petite valeur flottante ?

3. Voici un code à compiler et exécuter :

```
float f = 0;
int i;

for(i=0; i<1000; i++)
{
	f+=0.1;
}
printf("%f\n", f);
return 0;
```

   A la fin, la valeur contenue dans la variable **f** est-elle conforme avec ce à quoi vous vous attendiez ? Si non, pourquoi ?

4. Voici un code à compiler et exécuter :

```
float f = 1.0;

while(f!=0)
{
        f=f-0.00001;
        printf("%f\n", f);
}

printf("Fini\n");
return 0;
```

   Ce code boucle-t-il à l'infini ? Et si oui, pourquoi ? Qu'en déduisez-vous à propos de tester l'égalité entre deux flottants ? Comment modifier ce code afin d'éviter la boucle ?

5. Voici un code à compiler et exécuter :

```
double a=100000000000000000;
double b = 1.0;

printf("%f\n", a+b);
return 0;
```

   Les problèmes de précision des flottants sont-il réservés à l'utilisation de valeurs décimales, ou a-t-on un problème similaire lorsque les valeurs sont entières ?

2 Calculer le nombre de flottants entre deux entiers

Nous allons tenter de calculer le nombre de double existant dans les intervalles $[0; 1[, [1, 2[, ..., [199, 200[$. Pour ce faire, considérez la structure suivante :

```
union u
{
       uint32_t x;
       float y;
};
```

Cette structure permettra de déclarer une union de deux variables. Quand **x** parcourra le spectre complet des entiers 32 bits, alors **y** parcourra le spectre entier des flottants (32 bits).

Ecrivez la suite de la fonction : vous devez faire parcourir à **x** tous les entiers possibles et, pour chaque valeur, tester le flottant **y** associé. Si **y** est dans l'un des intervalles que l'on souhaite ($[0; 1[, [1, 2[, ..., [199, 200[$), alors on compte un flottant de plus dans cet intervalle.

A la fin, affichez un récapitulatif du nombre de valeurs flottantes trouvées dans chaque intervalle, et tracez-les sur Excel. Que constatez-vous ? Y a-t-il autant de flottants dans $[0; 1]$ que dans $[199; 200[$?

3 Une fonction générique de lecture de valeur

Dans cet exercice, vous apprendrez à utiliser l'union pour stocker, dans un même espace mémoire, une variable pouvant avoir différents types.

1. Dans les fichiers joints au TP, ouvrez le fichier *lecture_valeur.c*. Regardez le code : la fonction *demanderValeur* demande à l'utilisateur d'entrer une valeur numérique, et la transforme, selon le cas de figure, en double, uint64_t ou int64_t.

   Le problème est que cette fonction ne peut pas renvoyer une valeur classique. En effet, il faudrait qu'elle renvoie, selon les différents cas possibles, un double, un uint64_t ou un int64_t. Proposez une solution, à l'aide de la structure suivante, pour que la fonction puisse retourner une valeur qui prendra, en fonction de la situation, le type double, uint64_t ou int64_t.

```
typedef enum{DOUBLE, LONG, ULONG} TypeGenerique;

typedef struct
{
```

```
    TypeGenerique type;

    union
    {
        double d;
        uint64_t u;
        int64_t i;
    }data;
} generique;
```

2. Proposez une fonction *afficheGenerique*, qui prend en paramètre un *generique*, et qui affiche ce dernier. Cette fonction utilisera des *switch/case* afin de déterminer le type du *generique* et afficher ce dernier dans le bon format.

   Votre fonction devra aussi afficher, entre parenthèses, le type du *generique*. Par exemple, si le *generique* est un double qui vaut 2.7, elle devra afficher :

```
2.700000 (double)
```

   Dans le main, affichez le *generique* obtenu après l'appel de la fonction *demanderValeur*.

3. Proposez une fonction *creerGenerique* prenant en paramètre une chaîne de caractères représentant un nombre et renvoyant un *generique*. Dans le cas d'une chaîne de caractères mal formée, on arrêtera le programme avec assert. Votre fonction devra reprendre une grande partie du code de *demanderValeur*.

   Une fois la fonction écrite,modifiez la fonction *demanderValeur* afin qu'elle appelle la fonction *creerGenerique*. Vous devriez maintenant pouvoir, dans le main, créer des *generique* de deux façons :

```
generique g = demanderValeur();
generique j = creerGenerique("234");
```

4. Faîtes une fonction *addGenerique* prenant en paramètre deux *generique* et permettant de les additionner. Le résultat de la somme de deux *generique* produit un *generique* dont le type dépend du type des deux *generique*. Voici le type de *generique* que votre fonction *add(generique g, generique j* devrait produire en fonction du type de g et de j :

		type de g		
		double	**int64_t**	**uint64_t**
type de j	**double**	double	double	double
	int64_t	double	int64_t	int64_t
	uint64_t	double	int64_t	uint64_t

   Conseil : Utilisez des tests conditionnels pour décider du type de *generique* à contruire et retourner, puis d'autres tests pour savoir quels champs de g et j additionner dans le résultat.

5. Dans le fichier *tp1_add_macro.c*, vous trouverez une fonction d'addition de deux *generique* qui utilise des macros (définies en début de fichier) pour "effacer" le test du type de g et j du code, qui est donc beaucoup plus court.

   Inspirez-vous de cette fonction pour proposez une fonction *multGenerique* réalisant le produit de deux *generique*.

6. Proposez une fonction permettant de soustraire deux *generique*.

   Indice : soustraire deux *generique* revient à multiplier l'un d'eux par l'entier -1, puis à additionner les deux *generique*...

7. Proposez une fonction permettant de diviser deux *generique*. Cette fonction devra toujours retourner un *generique* de type double. Vérifiez bien que 2 divisé par 3 vous donne un double d'environ 0.6666.