



Maîtrisez les nombres à virgule en C

Par Maëlan



www.siteduzero.com

Dernière mise à jour le 18/08/2011

Sommaire

Sommaire	1
Informations sur le tutoriel	0
Maîtrisez les nombres à virgule en C	2
Informations sur le tutoriel	2
Les bases de l'utilisation des nombres à virgule en C	2
La base de la base	3
Des constantes inexactes	4
Quelques informations pratiques	4
IEEE 754 : À quoi ressemble un nombre flottant en mémoire ?	7
Principes généraux	8
Les différents types de nombres représentables	9
Nombres dénormalisés	11
Autour de la représentation en mémoire : un peu de mathématiques... ..	12
Retrouver la valeur du nombre flottant	12
Intervalle entre les nombres en fonction de l'exposant	14
Précision et chiffres significatifs	14
IEEE 754 : Exceptions & arrondis	15
Les exceptions	16
Les arrondis	16
L'environnement des flottants : <fenv.h>	18
Comparer des nombres flottants	18
L'infâme trahison de l'opérateur ==	18
L'écart absolu & l'écart relatif	19
La représentation en mémoire convertie en entier	20
Code complet	25
Mais qu'en dit la norme C ?	29
IEEE 754 et la norme C	29
En pratique : savoir si l'implémentation utilise IEEE 754	31
Q.C.M.	31
Lire aussi	33

Maîtrisez les nombres à virgule en C


Vous souhaitez manipuler dans vos programmes en C de très grands nombres et/ou des nombres à virgule ? Ou alors vous avez déjà essayé mais vous rencontrez des problèmes incompréhensibles ? Ce cours vous est destiné !

Vous y apprendrez tout ce qu'il faut savoir sur les nombres à virgule en C.

Au programme :

- quelques rappels (ou pas) sur les nombres à virgule en C (types, syntaxe...);
- la représentation en mémoire d'un nombre à virgule dite "flottante" (selon la norme IEEE 754), et les propriétés qui en découlent (valeurs possibles, etc.);
- les inconvénients des nombres à virgule flottante en C ;
- effectuer une comparaison de nombres flottants ;
- le point de vue de la norme C, et les implémentations.

Prérequis :

- connaître un minimum le langage C ! Au moins jusqu'au chapitre sur les pointeurs (chapitre 2 de la partie II) du [tutoriel de M@teo21](#) si vous le suivez ;
- avoir quelques notions mathématiques (rien de bien méchant) sur les puissances (de 10 et de 2, dans notre cas), et idéalement les bases numériques (mais ce n'est pas indispensable) ;
- maîtriser les notions de bit, d'octet, de binaire, etc., ainsi que les deux façons principales de représenter un nombre relatif en mémoire (le bit de signe ou le complément à 2) ; si ce n'est pas le cas, vous pouvez lire [ce tutoriel sur le "vrai visage des variables" en mémoire](#) ;
 - [ce tutoriel-ci](#), à la fois clair et complet, vous introduit toutes les notions listées ci-dessus, et je vous invite à le lire s'il vous manque quelque chose (arrêtez-vous après avoir lu la partie sur les nombres entiers, la suite gâcherait mon suspense) ;
- être motivé et curieux ; 
- *that's all!*

Sommaire du tutoriel :




- [Les bases de l'utilisation des nombres à virgule en C](#)
- [IEEE 754 : À quoi ressemble un nombre flottant en mémoire ?](#)
- [Autour de la représentation en mémoire : un peu de mathématiques...](#)
- [IEEE 754 : Exceptions & arrondis](#)
- [Comparer des nombres flottants](#)
- [Mais qu'en dit la norme C ?](#)
- [Q.C.M.](#)

Informations sur le tutoriel

Auteur :

- [Maëlan](#)

Difficulté : 


Temps d'étude estimé : 3 heures, 7 minutes

Licence :



Les bases de l'utilisation des nombres à virgule en C

Tout d'abord, une présentation des **nombres à virgule flottante** !

Bon, je ne vais pas vous expliquer ce qu'est un nombre à virgule. Si vous ne savez pas ce que c'est, ce tutoriel ne vous sera d'aucune utilité. 

On parle de virgule *flottante* car on peut faire varier la place de la virgule en variant la puissance de 10 (puisque tout nombre peut être écrit avec une puissance de 10). Par exemple : $42,1337 = 42,1337 \times 10^0 = 0,421337 \times 10^2 = 42133,7 \times 10^{-3}$.

En informatique, on parle de virgule flottante par opposition à virgule *fixe*, qui indique une méthode de représentation en mémoire d'un nombre avec un nombre fixe de chiffres après la virgule. En C, nous avons des nombres à virgule flottante.

Remarque : On parle de la *partie entière* pour désigner la partie qui se trouve avant la virgule, et de *partie décimale* (ou *fractionnaire*) pour celle qui se trouve après ; attention à ne pas faire de confusion avec respectivement les représentations entières (qu'on verra plus tard) et la base décimale, c'est-à-dire la base 10 (je parlerai de préférence de "base 10" pour éviter les confusions).

Je pourrais également parler de la *partie significative* d'un nombre pour désigner ce nombre sans la puissance de 10 qui va derrière (c'est-à-dire l'ensemble de ses parties entière et fractionnaire) ; par exemple, 3,1416 sera la partie significative de $3,1416 \times 10^{\text{exposant}}$. Attention, la partie significative varie selon l'exposant, c'est pourquoi je préciserais la notation à laquelle je me réfère (la plupart du temps, ce sera la notation scientifique).

La base de la base

Les types pour représenter un nombre à virgule

En C, un nombre à virgule flottante peut être représenté par les types `float` et `double` (il existe aussi `long double`). Comme pour les types entiers, leur taille en mémoire dépend de l'architecture de l'ordinateur, mais les valeurs sont très fréquemment **32 bits (4 octets) pour un float** et **64 bits (8 octets) pour un double** (nous verrons pourquoi par la suite). Pour vous assurer de la taille chez vous, vous pouvez faire :

Code : C

```
printf("taille d'un float : %u\n", sizeof(float));  
printf("taille d'un double : %u\n", sizeof(double));
```

Intérêt des nombres flottants

Vous pouvez utiliser les types flottants dans vos programmes si vous souhaitez manipuler des **nombres à virgule**, mais également pour stocker de **très grands nombres**. En effet, un `float` permet d'atteindre 10^{38} , et un `double` 10^{308} ! Toutefois, vous ne pourrez pas stocker tous les 38 ou 308 chiffres significatifs jusqu'au chiffre des unités (on détaillera la précision plus tard).

Écrire une constante

La syntaxe pour écrire un nombre à virgule est : `-3141.59e7` (ou `-3141.59E7`, avec un 'E' majuscule), ce qui signifie $-3141,59 \times 10^7$. Attention, on met un point et non une virgule ! La lettre 'E' collée au nombre signifie "exposant". Celui-ci peut très bien être négatif comme dans `945.68e-3` (ce qui signifie $945,68 \times 10^{-3}$).

De plus, les parties du nombre (c'est-à-dire la partie à gauche ou celle à droite de la virgule, mais pas les deux, et l'exposant) qui valent 0 peuvent être omises (mais il doit y avoir au moins un point ou un exposant d'écrit, pour signifier qu'il s'agit d'un nombre à virgule). Par exemple, les constantes suivantes sont strictement identiques :

Code : C

```
0.0e0 ;      0.0 ;  
0.e0 ;       0. ;  
.0e0 ;       .0 ;  
0e0 ;
```

En revanche, écrire 0 tout court produira un entier. En effet, le compilateur ne voyant ni point ni 'e', il ne peut pas savoir qu'il s'agit d'un nombre à virgule flottante.

Le type par défaut d'une constante à virgule flottante est `double`, mais on peut le changer avec un suffixe collé après la constante :

- la lettre 'F' minuscule ou majuscule demande un `float` ;
- la lettre 'L' minuscule ou majuscule demande un `long double`.

Par exemple, `.1e-3f` sera de type `float`, `42.1337e-3l` de type `long double`.

Enfin, il peut être utile de savoir que le C99 permet aussi d'écrire ses constantes flottantes en hexadécimal ! 🐼 Pour cela :

- ajoutez le préfixe `0x` (ou `0X`) avant votre constante (mais après le signe) ;
- écrivez la partie significative en base 16 (c'est-à-dire avec les chiffres de 0 à 9 et les lettres de A à F) ;
- l'exposant est obligatoire (à la limite, vous n'avez qu'à mettre 0) ; il doit être écrit en base 10 comme d'habitude (et non en hexadécimal), mais précédé de la lettre `p` (ou `P`) au lieu de `e`. De plus, il se réfère à une puissance de 2 et non de 10.

Par exemple, `-0xC45.8p3` signifie `-0x C45,8 × 23` (soit `- 3141,5 × 23 = - 25 132`).

L'intérêt à ~~part faire mal au crâne~~ ? Obtenir un nombre exact ! Les explications suivent...

Des constantes inexactes

Attention : **tous les nombres décimaux (c'est-à-dire écrits en base 10) ne sont pas forcément représentables de façon finie en binaire, et certains sont donc arrondis** ! Ainsi, 0,1 en base 10 s'écrit en binaire `0b0,00011001100110011...` avec une infinité de 0011 (de même que $1/3 = 0,3333...$ en base 10).

Une partie significative écrite en base 10 ainsi qu'une puissance de 10 négative (c'est-à-dire qu'on divise par une puissance de 10) peuvent donc mener à des nombres inexacts.

En fait, seuls les nombres dont la partie décimale est le résultat d'une division par une puissance de 2 (comme $.5 = 1/2$, $.75 = 3/4$ ou encore $.625 = 5/8$) sont représentables en binaire avec un nombre fini de décimales.

Spécifier la partie significative en hexadécimal (base 16) et l'exposant en termes de puissance de 2 permet donc un nombre exact (dans la limite de la capacité du type, bien sûr 🐼).

Quelques informations pratiques

Les fonctions d'entrée et de sortie formatées : `printf` & `scanf`

Vous voudrez sûrement savoir quels sont les formateurs pour lire ou écrire des nombres flottants. Voici un petit tableau résumé. Pour des infos complètes (plus d'options) sur l'utilisation de `printf` et `scanf` : RTFM, **bien sûr** !

Formateurs de `printf` pour les flottants

Formateurs	Utilisation
<code>"%f",</code> <code>"%F"</code>	Affiche le <code>double</code> "simplement", comme vous avez sans doute l'habitude de l'écrire : l'affichage est du type <code>[-] XXXX.XXX</code> , où les X sont des chiffres de 0 à 9 et <code>[-]</code> symbolise le "moins" éventuel.
<code>"%e",</code> <code>"%E"</code>	Affiche le <code>double</code> en écriture scientifique, c'est-à-dire avec un seul chiffre avant la virgule et un exposant introduit par la lettre 'e' (ou 'E' pour <code>"%E"</code>) ; l'affichage est donc du type <code>[-]X.XXXXXXeYY</code> (ou <code>[-]X.XXXXXXEYY</code>).
<code>"%g",</code> <code>"%G"</code>	Une sorte de "combinaison" des formateurs précédents : utilise le premier style si le nombre n'est pas trop grand ou trop petit, le deuxième style sinon.

Remarquez que tous ces formateurs attendent des flottants de type `double`. Il n'existe pas de formateurs pour `float`, ce qui n'est pas dramatique car vous pouvez convertir vos nombres de `float` vers `double` pour les afficher. Le C99 vous permet d'ajouter la lettre 'L' majuscule entre le '%' et le formateur afin de correspondre à un `long double`.

Formateurs de `scanf` pour les flottants

Formateurs	Utilisation
<code>"%e", "%E", "%f", "%F", "%g", "%G"</code>	<p>Lit un nombre à virgule flottante et l'écrit dans la variable de type <code>float</code> indiquée. Le nombre lu doit être écrit de la même manière que vous écrivez une constante dans votre code source, c'est-à-dire (vous avez déjà oublié ?) :</p> <ul style="list-style-type: none"> un signe éventuel, puis un nombre à virgule en base 10 (chiffres de 0 à 9), puis un exposant de 10 éventuel introduit par 'e' (ou 'E') ; ou alors un signe éventuel, puis le préfixe '0x' (ou '0X'), puis un nombre à virgule en base 16 (chiffres de 0 à F), puis un exposant de 2 éventuel introduit par 'p' (ou 'P'). <p>Contrairement à <code>printf</code>, tous ces formateurs sont équivalents.</p>
<p>Pour lire un <code>double</code>, il faut ajouter la lettre 'l' minuscule entre le '%' et le formateur ; pour un <code>long double</code>, il faut ajouter 'L' (ou 'll').</p>	

Les fonctions mathématiques avec `<math.h>`

Matheux, vous serez comblés : la bibliothèque standard du C met à votre disposition toute une gamme de fonctions mathématiques. Ces fonctions sont définies dans le header `<math.h>`, qu'il vous faudra donc inclure dans vos sources.

À titre indicatif, voici une liste des fonctions que vous pourrez trouver dans `<math.h>` :

Secret (cliquez pour afficher)

Code : C - Liste des fonctions de `<math.h>`

```
#include <math.h>

/** fonctions courantes */

double pow(double x, double n);    // x à la puissance n

double sqrt(double);              // racine carrée
double cbrt(double);              // [C99] racine cubique

double fabs(double);              // valeur absolue
double fdim(double, double);      // [C99] différence positive
double fmin(double, double);      // [C99] minimum
double fmax(double, double);      // [C99] maximum

/** fonctions trigonométriques (les angles sont en radians) */

double cos(double);               // cosinus
double sin(double);               // sinus
double tan(double);               // tangente
double acos(double);              // arc-cosinus
double asin(double);              // arc-sinus
double atan(double);              // arc-tangente
double atan2(double y, double x); /* arc-tangente avec 2
arguments : angle entre l'axe
des abscisses et le point de coordonnées (x ; y) ;
attention, y est le 1er argument ! */
double cosh(double);              // cosinus hyperbolique
double sinh(double);              // sinus hyperbolique
double tanh(double);              // tangente hyperbolique
```

```

double acosh(double); // [C99] arc-cosinus hyperbolique
double asinh(double); // [C99] arc-sinus hyperbolique
double atanh(double); // [C99] arc-tangente hyperbolique

/** arrondis, parties entière et fractionnaire */

double round(double); // [C99] arrondi à l'entier, tel
que round(.5)==1
long int lround(double); // [C99] idem
long long int llround(double); // [C99] idem

double rint(double); // [C99] arrondi à l'entier
suivant le mode d'arrondi [1]
long int lrint(double); // [C99] idem
long long int llrint(double); // [C99] idem

double nearbyint(double) /* [C99] arrondi à l'entier suivant le
mode d'arrondi,
sans lever d'exception "inexact" [1] */

double ceil(double); // arrondi à l'entier supérieur
double floor(double); // arrondi à l'entier inférieur

double trunc(double); // [C99] partie entière (troncature)
double modf(double, double*); /* retourne la partie
fractionnaire et écrit la partie
entièrè dans le nombre pointé */

/** exponentielles, logarithmes & co. */

double exp(double); // fonction exponentielle
double expm1(double x); // [C99] équivalent à exp(x)-1, mais
plus précis
double exp2(double x); // [C99] 2 à la puissance x

double log(double); // logarithme naturel (ou "népérien")
double log1p(double x); // [C99] équivalent à log(1+x), mais
plus précis
double log10(double); // logarithme en base 10
double log2(double); // [C99] logarithme en base 2

double logb(double x); /* [C99] logarithme entier en base 2
(équivalent à
floor(log2(x)), mais plus rapide) [2] */
int ilogb(double); // [C99] idem

double scalbn(double x, int n); // [C99] x * 2^n [2]
double scalbln(double, long int); // [C99] idem
double ldexp(double x, int n); // x * 2^n
double frexp(double x, int*); /* fraction normalisée : écrit un
exposant EXP dans l'entier
pointé et renvoie une valeur A, tels que x == A / (2^EXP) ;
A est compris entre 0.5 inclus et 1 exclus */

/** plus de fonctions */

double fmod(double, double); // reste de la division
double remainder(double, double); // [C99] idem (avec une
subtile nuance)

double hypot(double, double); /* [C99] hypoténuse du triangle
rectangle dont les deux
autres côtés sont donnés */
double erf(double); // [C99] fonction d'erreur de Gauss
double erfc(double); // [C99] fonction d'erreur complémentaire
double tgamma(double); // [C99] fonction gamma
double lgamma(double); // [C99] logarithme naturel de la valeur

```

```

absolue de la fonction gamma

double copysign(double x, double y);    // [C99] valeur de x avec
le signe de y
double fma(double x, double y, double z) // [C99] x*y + z

double nextafter(double x, double y);    /*      [C99]   prochaine
valeur flottante représentable
après x (vers y) */
double nexttoward(double, long double);  // [C99] idem

/** NOTES :
[1] Modes d'arrondis, exceptions : on verra ça plus tard.
[2] En fait, j'ai mis 2 pour simplifier, mais en réalité il s'agit
de FLT_RADIX
(une constante définie dans le header <float.h> et dont vous
n'avez pas encore
les connaissances requises pour en comprendre le sens) ; FLT_RADIX
vaut probablement 2.
**/

```

Cette liste n'est absolument pas à connaître par cœur ! Elle n'est là que pour vous donner une idée de ce que vous pouvez faire. D'ailleurs, les explications sont très succinctes et je vous invite à consulter le manuel qui est fait pour ça si vous souhaitez plus d'informations sur une fonction donnée (tout ça pour ne pas dire encore une fois : RTFM...).

De plus, chacune des fonctions listées ci-dessus se décline en fait en trois versions :

- une qui travaille avec des `double` : celle présentée ;
- une qui travaille avec des `float` : il faut ajouter la lettre 'f' à la fin du nom de la fonction ;
- une qui travaille avec des `long double` : il faut ajouter la lettre 'l' à la fin du nom de la fonction.

Vous devez néanmoins retenir une chose. Ces fonctions ne sont pas incluses dans l'exécutable avec le reste de la bibliothèque standard, lors de l'édition des liens (la phase suivant la compilation). Il faut les lier manuellement.

Sous GCC, cela se fait en passant le paramètre `-lm` lors de cette phase.

Si vous travaillez avec Code::Blocks, alors il est probable que ce dernier ajoute automatiquement cette liaison. Si ce n'est pas le cas, vous pouvez suivre la démarche suivante :

- Project > Build options... > onglet *Linker settings* ;
- dans le champ *Link libraries* (à gauche), cliquez sur Add et entrez le nom de la bibliothèque à lier, soit « m », puis validez ;
- autre possibilité pour cette dernière étape : taper simplement « -lm » dans le champ *Other linker options* (à gauche).
- Voilà, les fonctions de `<math.h>` seront désormais incluses dans vos programmes !

Voilà, on a fini avec les fondamentaux. Vous avez maintenant des connaissances suffisantes pour manier les nombres à virgule en C, ~~vous pouvez aller jouer dans le jardin~~ mais ce serait dommage de s'arrêter en si bon chemin... Après cette brève (*hum*) introduction en la matière, nous allons nous plonger au cœur des nombres flottants pour en étudier le moindre détail. Pas trop fatigué ? Il vaudrait mieux, parce qu'on vient à peine de commencer. Yêêêhaaa !

IEEE 754 : À quoi ressemble un nombre flottant en mémoire ?

On va maintenant s'intéresser à la manière dont sont **représentés** en mémoire les nombres à virgule flottante ! Attention, on va faire une grosse partie théorique, alors préparez vos barres céréales en cas d'hypoglycémie. 😊

Ce que vais vous raconter dans cette partie **n'est pas dans la norme C**. La suite de ce cours se base sur la **norme IEEE 754** (ou plus précisément ANSI/IEEE Std 754-1985). Celle-ci spécifie :

- la **manière de représenter les nombres flottants en mémoire** avec plusieurs formats ;

- cinq **opérations associées** : l'addition (+), la soustraction (-), la multiplication (*), la division (/) et la racine carrée (`sqrt()`);
- des **modes d'arrondi** ;
- des **exceptions**.

Ne vous inquiétez pas pour les deux derniers points, on en reparlera.

Elle est proposée par l'IEEE (*Institute of Electrical and Electronics Engineers*, l'Institut des Ingénieurs Électriciens et Électroniciens), une organisation américaine qui est devenue une référence en matière d'informatique.


Cette norme est très suivie (en fait, sur la très grande majorité des ordinateurs aujourd'hui), mais vous pouvez très bien tomber sur une implémentation qui utilise un autre mode de représentation. 😞 Un point sur la norme C sera fait à la fin de ce chapitre (ainsi qu'une manière de déterminer si IEEE 754 est bien employé sur votre implémentation).

Ce format aura son importance plus loin dans ce tutoriel.

Principes généraux

Tout nombre à virgule peut être écrit en **notation scientifique**, c'est-à-dire avec un seul chiffre avant la virgule et multiplié par une puissance de b (b étant la base dans laquelle on écrit le nombre : 10 pour la base décimale, 2 pour le binaire, 16 pour l'hexa, etc.). Par exemple, le nombre -3141,5 est égal à $-3,1415 \times 10^3$. L'écriture binaire de ce nombre est `0b -110001000101,1`, soit `0b -1,100010001011 $\times 2^{11}$` .

Pour stocker un nombre à virgule flottante, on se base sur sa notation binaire scientifique. En mémoire, un nombre flottant se décompose donc en **3 parties**. À partir du bit de poids fort, on a :

- le **signe**, codé sur **un bit** : ce bit vaut 0 si le nombre est positif, 1 s'il est négatif.
Remarquez ici que la représentation du signe des nombres flottants se fait selon le bit de poids fort et non selon la règle du complément à 2. En conséquence, il existe deux représentations pour zéro, un positif et un négatif (c'est néanmoins un défaut mineur, les implémentations gérant cette dualité ; par exemple, `+0.0 == -0.0` renverra vrai), et les valeurs possibles sont totalement "symétriques" par rapport à zéro (c'est la raison pour laquelle j'utiliserai souvent le symbole \pm , plus ou moins, par la suite) ;
- l'**exposant**, c'est-à-dire la puissance à laquelle il faut élever 2 (et non 10 !).
Le nombre de bits qu'il occupe dépend de la taille du type.
Ce nombre est un entier et peut être négatif. Pour le représenter, on n'utilise ni la règle du bit de signe, ni celle du complément à 2 (car cela rendrait la comparaison de nombres plus difficile, comme on le verra ensuite) ; on opère un **décalage de l'exposant** réel en lui ajoutant $2^e - 1$, où e représente le nombre de bits occupés par l'exposant ; en gros, ce nombre représente la valeur où tous les bits sauf celui de poids fort sont à 1 (c'est-à-dire $127 = 0b11111111$ si l'exposant est codé sur 8 bits, ou $1023 = 0b11111111111$ s'il est codé sur 11 bits) ;
 **Remarque** : Par la suite, j'emploierai les termes d'exposant "décalé" et "non-décalé" pour différencier l'exposant tel qu'il est représenté en mémoire de l'exposant ainsi représenté.
- la **mantisse**, c'est-à-dire la valeur de la **partie décimale en notation binaire scientifique** (avec un seul chiffre avant la virgule). On ne garde que la partie décimale car, du fait de la notation binaire, la partie entière est forcément égale à 1 (sauf dans les cas particuliers de zéro et de certains nombres très petits dont on reparlera). On parle de **bit implicite** (un bit "caché" représentant la partie entière, qui ici vaudrait 1).
On économise ainsi un bit, ce qui permet d'avoir une précision plus grande.
La mantisse occupe les bits de poids faibles restants.

Les deux types du C pour les nombres à virgule flottante répondent aux deux formats spécifiés par IEEE 754.

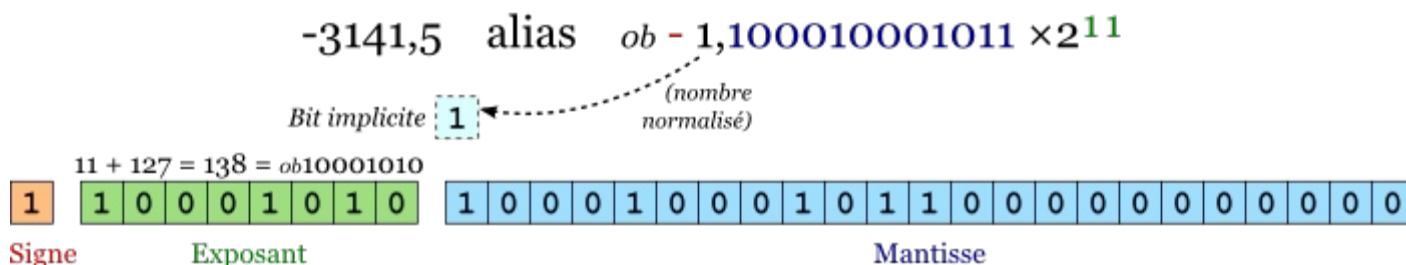
Tableau résumé des deux formats de la norme IEEE 754

Nom dans la norme IEE 754	Type en C	Taille (en bits)				Chiffres significatifs (en base 10)	Valeurs absolues possibles	
		Total	s	e	m		minimum	maximum

simple précision	float	32 bits	1	8	23	7	$1,2 \times 10^{-38}$	$3,4 \times 10^{+38}$
double précision	double	64 bits	1	11	52	16	$2,2 \times 10^{-308}$	$1,8 \times 10^{+308}$

Nous détaillerons plus tard les deux dernières colonnes.

En résumé (pour un `float` de 32 bits), la représentation de -3141,5 est :



Les différents types de nombres représentables

Il y a cependant des cas particuliers, car un nombre à virgule flottante peut représenter autre chose que des nombres "normaux". Il peut aussi valoir :

- **l'infini positif ou négatif** (noté ∞ par les mathématiciens). C'est par exemple le résultat de la division d'un nombre non nul par zéro (le signe dépendant alors du signe du numérateur et du zéro en dénominateur) ;
- **NaN** (*not a number*, pas un nombre). **NaN est une valeur spéciale qui est utilisée pour signaler une erreur** ($0 \div 0$ ou $\sqrt{-1}$ par exemples). N'importe quel calcul avec un NaN doit renvoyer NaN (sauf quelques exceptions), et n'importe quelle comparaison avec un NaN doit renvoyer faux (sauf $! =$) ; un NaN n'est même pas égal à lui-même, c'est pourquoi $x == x$ peut renvoyer faux ;
- enfin, on n'a pas encore réglé le problème de zéro.

Quelques calculs avec ces valeurs particulières :

Secret (cliquez pour afficher)

Le symbole \pm dans le résultat d'une division signifie que le signe dépend de ceux du numérateur et du dénominateur selon la règle suivante :

- deux signes identiques : signe +
- deux signes différents : signe -

$$x \div 0 = \pm \infty$$

$$0 \div 0 = NaN$$

$$x \div \infty = \pm 0$$

$$\infty \div \infty = NaN$$

$$0 \times \infty = NaN$$

$$(+\infty) + (+\infty) = +\infty$$

$$(+\infty) - (+\infty) = NaN$$

Ça ne vous autorise pas à écrire des choses comme `var / 0` dans votre programme ! En effet, la division par zéro



avec des nombres entiers a un comportement indéterminé, c'est-à-dire non prévu par la norme du langage C. Ça signifie qu'il peut se passer n'importe quoi lorsque vous faites ceci, selon le bon plaisir de votre compilateur, de votre système d'exploitation... Généralement, cela provoque une erreur fatale et un beau plantage en règle de votre programme.

D'ailleurs, le comportement de la division par zéro de nombres flottants est également indéterminé. Les règles de calcul ci-dessus ne sont en fait déterminées que par la norme IEEE 754, et non par la norme C elle-même.

Pour représenter tout ce petit monde, on utilise des valeurs spéciales de l'exposant (qui ne peuvent donc pas être utilisées pour des nombres normaux) :

- Si l'exposant vaut sa valeur maximale (soit 2^e-1) et que la mantisse est nulle, alors c'est **l'infini** (positif ou négatif, selon le bit de signe).
- Si l'exposant vaut sa valeur maximale (soit 2^e-1) et que la mantisse n'est pas nulle, alors c'est **NaN**.
- Si l'exposant décalé vaut 0 et que la mantisse est nulle, alors c'est **zéro** (positif ou négatif, selon le bit de signe).
- Si l'exposant décalé vaut 0 et que la mantisse n'est pas nulle, alors c'est un **nombre dénormalisé** : on considère que le bit implicite (la partie entière en notation scientifique) vaut 0.
- Dans tous les autres cas, c'est un **nombre normalisé** : on considère que le bit implicite vaut 1.

Un petit tableau pour récapituler tout ça de manière visuelle (pour un **float** de 32 bits)...

Tableau récapitulatif des différents types de valeurs d'un nombre à virgule flottante

Type	Représentation mémoire		Valeur
	binaire	hexadécimal	
Not a Number	[0/1] — 11111111 — 111111111111111111111111	[7/F]F FF FF FF	NaN

	[0/1] — 11111111 — 000000000000000000000000	[7/F]F 80 00 01	NaN
Infini	[0/1] — 11111111 — 000000000000000000000000	[7/F]F 80 00 00	$\pm \infty$
Nombre normalisé	[0/1] — 11111110 — 111111111111111111111111	[7/F]F 7F FF FF	$\pm 3,4028235 \times 10^{+38}$

	[0/1] — 00000001 — 000000000000000000000000	[0/8]0 80 00 00	$\pm 1,1754944 \times 10^{-38}$
Nombre dénormalisé	[0/1] — 00000000 — 111111111111111111111111	[0/8]0 7F FF FF	$\pm 1,1754942 \times 10^{-38}$

	[0/1] — 00000000 — 000000000000000000000001	[0/8]0 00 00 01	$\pm 1,4012985 \times 10^{-45}$
Zéro	[0/1] — 00000000 — 000000000000000000000000	[0/8]0 00 00 00	± 0

À titre informatif, je vous mets aussi le tableau équivalent pour un **double** de 64 bits (sans le binaire, ça prend trop de place).

idem pour un double

Type	Représentation mémoire	Valeur
Not a Number	[7/F]F FF FF FF FF FF FF FF	NaN

	[7/F]F F0 00 00 00 00 00 01	NaN
Infini	[7/F]F F0 00 00 00 00 00 00	$\pm \infty$
Nombre normalisé	[7/F]F EF FF FF FF FF FF FF	$\pm 1,7976931348623157 \times 10^{+308}$

	[0/8]0 10 00 00 00 00 00 00	$\pm 2,2250738585072014 \times 10^{-308}$
Nombre dénormalisé	[0/8]0 0F FF FF FF FF FF FF	$\pm 2,2250738585072010 \times 10^{-308}$

	[0/8]0 00 00 00 00 00 00 00	± 0

Notez que pour passer d'un nombre positif à son équivalent négatif, il suffit d'additionner 0x 8000 0000 à sa représentation (ou 0x 8000 0000 0000 0000 s'il s'agit d'un **double**). Ceci nous servira plus tard. 😊

Nombres dénormalisés

On ruse donc en écrivant le nombre avec le plus petit exposant possible. *Question* : quel est cet exposant (pour un `float`) ? Si vous avez répondu -127 (et je suis sûr que vous l'avez fait, si vous suivez encore ce cours), vous êtes tombés dans le piège. Mouahaha. 😏



```
0b 0,000000000000000000000000 x2-126 // zéro  
0b 0,00000000000000000000000001 x2-126 // 1er nombre dénormalisé  
0b 0,00000000000000000000000010 x2-126 // nombre dénormalisé suivant  
...  
0b 0,1111111111111111111111110 x2-126 // avant-dernier nombre dénormalisé  
0b 0,1111111111111111111111111 x2-126 // dernier nombre dénormalisé  
0b 1,000000000000000000000000 x2-126 // 1er nombre normalisé  
0b 1,00000000000000000000000001 x2-126 // nombre normalisé suivant  
...  
0b 1,1111111111111111111111110 x2-126 // avant-dernier nombre normalisé de cet exposant  
0b 1,1111111111111111111111111 x2-126 // dernier nombre normalisé de cet exposant
```

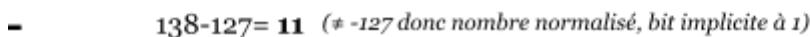
On traduit ensuite ceci de façon similaire à ce qu'on a fait pour un nombre normalisé : suivez les couleurs !

- ***flottant*** la valeur du flottant représenté ;
- ***m*** et ***e*** le nombre de bits occupés par la mantisse et l'exposant (respectivement) ;
- ***mantisse*** la mantisse (plus précisément, sa représentation entière) ;
- ***exposant*** l'exposant non-décalté (en termes de puissances de 2) ; le décalage de l'exposant est ***décalage*** = $2^{e-1} - 1$, on en déduit les valeurs limites de l'exposant :
 - la valeur minimale de l'exposant non-décalté, réservée aux zéros et nombres dénormalisés, est ***exposantMin*** = $0 - \text{d\`ecalage} = 1 - 2^{e-1}$ (mais l'exposant "réel" des dénormalisés est ***exposantMin*** + 1) ;
 - la valeur maximale de l'exposant non-décalté, réservée aux infinis et NaNs, est ***exposantMax*** = $(2^e - 1) - \text{d\`ecalage} = 2^{e-1}$.

- 1 ou 0 représente la partie entière du flottant (c'est-à-dire le bit implicite).
- La fraction qui suit représente la partie décimale (on a $0 \leq \frac{\text{mantisse}}{2^m} < 1$).

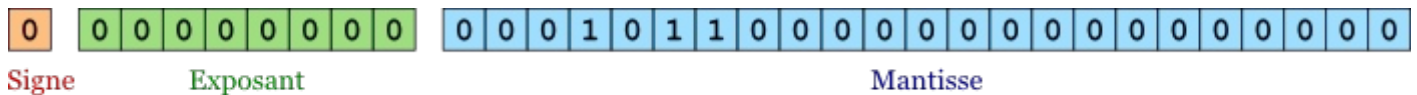
- $flottant = \pm \left(1 + \frac{mantisse}{2^m}\right) \times 2^{exposant} \text{ (normalisé) ;}$
- $flottant = \pm \left(0 + \frac{mantisse}{2^m}\right) \times 2^{exposantMin+1} \text{ (dénormalisé).}$

Exemple 1 : Nombre normalisé



On retrouve bien -3141,5. 😊

www.siteduzero.com



+ 0-127 = -127 (donc nombre dénormalisé, bit implicite à 0 et exposant réel de -126 et non -127)

$$\begin{aligned}
 \text{flottant} &= + \left(0 + \frac{0b\ 000101100000000000000000}{2^{23}} \right) \times 2^{-127+1} \\
 &= + \left(0 + \frac{720896}{2^{23}} \right) \times 2^{-126} \approx + 1,0101905 \times 10^{-39}
 \end{aligned}$$

Là aussi, on retrouve bien le nombre de départ.

Intervalle entre les nombres en fonction de l'exposant

Selon la norme IEEE 754, les nombres consécutifs de même exposant (qu'ils soient normalisés ou pas) sont "placés" à **intervalle régulier**. En effet, pour passer d'un nombre au nombre suivant, on ajoute toujours

$$\delta = 0.000000000000000000000001 \times 2^{\text{exposant}} = \frac{1}{2^m} \times 2^{\text{exposant}} = 2^{\text{exposant}-m}.$$

Cet intervalle double quand on passe d'un exposant à l'exposant supérieur (logique, d'après la formule). Quelques valeurs remarquables pour un **float** (ne les apprenez pas !):

- $2^{-126} - 2^{-23} = 2^{-149} \approx 1,40 \times 10^{-45}$: écart entre les nombres dénormalisés et les premiers nombres normalisés (et valeur du tout premier nombre dénormalisé) ;
- $2^0 - 2^{-23} \approx 1,19 \times 10^{-7}$: écart entre les nombres normalisés compris entre 1 et 2 ;
- 1 : écart entre les nombres normalisés d'exposant non-décalé 23 ;
- $2^{127} - 2^{-23} \approx 2,03 \times 10^{31}$: écart entre les plus grands nombres normalisés.

L'écart séparant le dernier nombre d'un exposant donné du premier nombre de l'exposant suivant est le même que celui séparant les nombres de l'exposant donné (c'est encore logique, si vous réfléchissez).

Précision et chiffres significatifs

Vous aurez remarqué une colonne "chiffres significatifs" dans le premier tableau. Vous-êtes vous demandé ce que cela signifiait ? C'est ce dont on va parler ici.

Mais qu'est-ce qu'un **chiffre significatif** ? Le nombre de chiffres significatifs d'un nombre, c'est tout simplement le nombre de chiffres utilisés pour écrire ce nombre (dans une base numérique donnée). Par exemple, 43,1337 a 6 chiffres significatifs.

En physique et en chimie, on y voue une attention particulière. En effet, les mesures n'étant jamais exactes, on s'en sert pour **indiquer le degré de précision** de la mesure (qui varie selon les instruments). En conséquence, pour un physicien, 42,1337 est différent de 42,13370 ; le second nombre est plus précis, car il indique quel est le 5^e chiffre après la virgule tandis que le premier nombre s'arrête au 4^e (le 5^e chiffre pourrait être 0, 1, 2, 3, 4, mais pas 5 ou plus car sinon on aurait arrondi au supérieur et le 4^e chiffre serait 8).

On compte les chiffres significatifs à partir du premier chiffre de gauche différent de 0, ce qui signifie que 3,1416 et 003,1416 sont équivalents (ils ont tous deux 5 chiffres significatifs).

Vous pouvez maintenant comprendre la colonne du tableau : elle indique le nombre de chiffres significatifs en base 10 que

nous permet chaque format :

- le format 32 bits garantit **7** chiffres significatifs en base 10 ;
- le format 64 bits garantit **16** chiffres significatifs en base 10.

Maintenant, *contrôle surprise* : combien de chiffres significatifs en base 2 permettent ces formats ? 🐼 Mais si, vous pouvez tout à fait répondre, réfléchissez un peu.

C'est tout simple : la mantisse représentant la partie significative du nombre, un nombre flottant a autant de chiffres significatifs en base 2 que sa mantisse occupe de bits.

Hmm hmm. Vous êtes certain ? N'oubliez pas le bit implicite ! Il faut donc ajouter 1 à ce nombre. En vérité, **un nombre normalisé a donc $m + 1$ chiffres significatifs en binaire**, soit $23+1=24$ pour le format 32 bits, ou $52+1=53$ pour le format 64 bits.

On appelle souvent le nombre de chiffres significatifs en base 2 la **précision** tout court.

Mais ce n'est pas aussi simple ! Cette "formule" pour la précision n'est valable que pour les nombre normalisés. Voyez-vous pourquoi ?

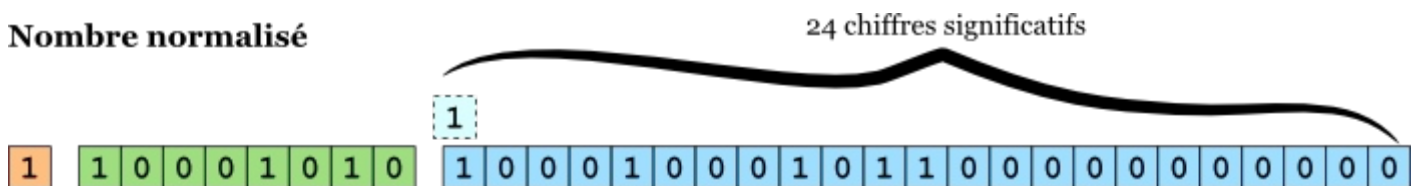
Rappelez-vous que pour les nombres dénormalisés, le bit implicite, c'est-à-dire la partie entière du nombre, est 0. Comme c'est le premier chiffre, on ne le compte pas dans les chiffres significatifs.



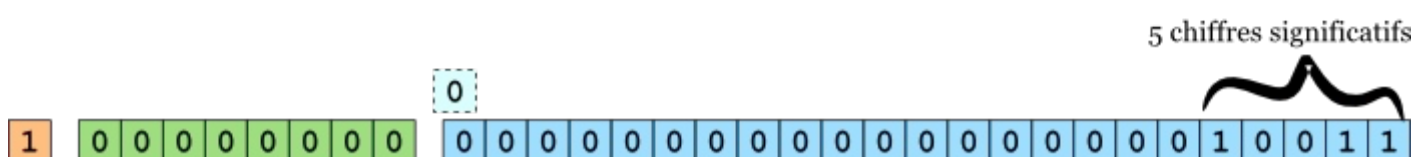
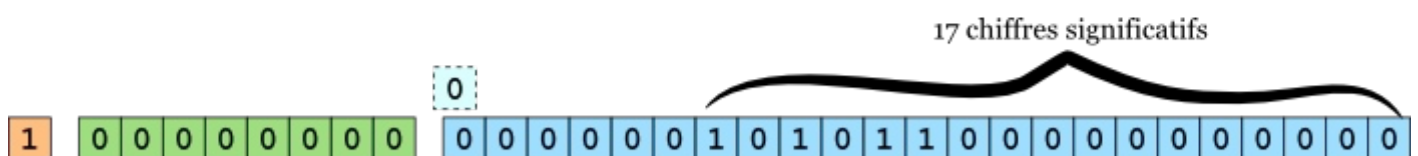
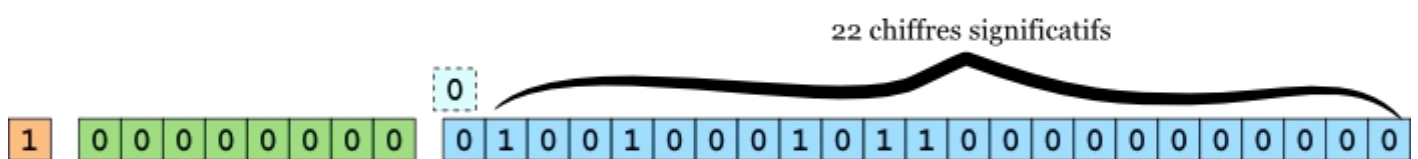
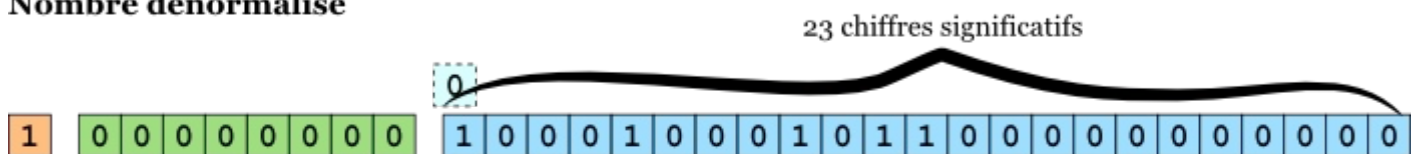
Ah bah alors, pour un nombre dénormalisé, la précision c'est juste le nombre de bits de la mantisse ?

Que nenni, jeune padawan. Un court schéma vaut mieux que de longues explications.

Nombre normalisé



Nombre dénormalisé



Comme vous le voyez, **plus un nombre dénormalisé est proche de zéro, moins il est précis.**

IEEE 754 : Exceptions & arrondis

Bien. On vient de se farcir deux chapitres de théorie pure sur la représentation en mémoire d'un nombre à virgule flottante. Vous en avez marre ? Tant mieux, on change de sujet ! Bon, ça va rester théorique, mais un peu moins quand même. La norme IEEE 754 ne se limite pas à la représentation des flottants ; elle définit également des exceptions et des modes d'arrondis. Décortiquons tout ça.

Les exceptions

Citation : Zér0 impatient (et avec une bonne mémoire)

Ah, on va enfin savoir ce que c'est que les "exceptions" dont tu nous as parlé !

Oui oui, on va voir ça tout de suite. 😊

Une **exception** est une sorte de "signal" qui est envoyé dans certains cas, afin de traiter ~~les erreurs~~ les cas particuliers qui, s'ils étaient ignorés, seraient des erreurs. Cette définition est en fait générale (le terme vous est peut-être familier si vous faites du C++).

C'est encore flou pour vous ? Ça ne fait rien, vous allez mieux comprendre avec cette liste. IEEE définit cinq exceptions :

- **invalid operation (opération invalide)** : se produit lorsqu'une opération interdite est effectuée ; le résultat du calcul en question sera NaN ;
- **division by zero (division par zéro)** : se produit lorsqu'on tente de diviser un nombre (non nul) par zéro ; le résultat sera $\pm \infty$;
- **overflow** : se produit lorsque le résultat d'un calcul est trop grand (en valeur absolue) pour être stocké ; on arrondit à $\pm \infty$;
- **underflow** : se produit lorsque le résultat d'un calcul est trop petit (en valeur absolue) pour être stocké ; on arrondit à zéro ;
- **inexact** : se produit lorsqu'on effectue un autre arrondi pour pouvoir stocker un nombre flottant (parce que le résultat a plus de chiffres significatifs qu'on peut en stocker).

Et là, j'ai une transition de malââââde !...

Les arrondis

Citation : Naïf que vous êtes...

C'est super-chouette, c'est merveilleux, je peux manipuler des nombres à virgule en C avec une précision extrême !

Oui, **MAIS** :

Les nombres stockés en mémoire ayant un nombre fini de chiffres significatifs (essayez de stocker un nombre infini de chiffres...), **les calculs peuvent mener à des arrondis**. IEEE 754 définit quatre **modes d'arrondis** :

- **au nombre le plus proche** : c'est le comportement par défaut (si le nombre à arrondir tombe pile-poile entre deux, alors on arrondit à celui dont le dernier bit vaut 0) ;
- **vers zéro** ;
- **vers $+\infty$** ;
- **vers $-\infty$** .

C'est là que ça devient intéressant (enfin ça l'était déjà avant, c'est une façon de parler, n'est-ce-pas) . Les imprécisions s'accumulent au fil des

calculs, et au final vous pouvez obtenir quelque chose d'incohérent !

Ces imprécisions se manifestent par exemple lorsqu'on manipule deux nombres dont les exposants sont très éloignés. Par exemple, $4.2e17 + 13.37$ devrait donner 420000000000000001337 soit $4.20000000000000001337e17$, mais il y a plus de chiffres significatifs qu'on ne peut en stocker ; le nombre sera donc arrondi, et finalement il vaudra $4.2e17$ comme si l'opération n'avait pas eu lieu !

Certains opérateurs entraînent beaucoup d'arrondis, comme l'addition ou la soustraction. Au contraire, d'autres, tels la multiplication ou la division, sont beaucoup plus sûrs.

L'exemple précédent devrait vous aider à comprendre pourquoi. 😊

Autre source de problèmes : comme vu précédemment, tous les nombres décimaux ne sont pas représentables de façon finie en binaire, et certains (comme $1.0/10.0 = .1$) sont donc arrondis.

Enfin, et en conséquence de ce qui vient d'être dit :

Citation : newtow

Les opérations avec les flottants ne sont pas associatives : l'ordre dans lequel on fait un calcul change le résultat. Par exemple, $1 - (.2 + .2 + .2 + .2 + .2)$ aura un résultat différent de $((((1 - .2) - .2) - .2) - .2) - .2$.

Pas convaincus ? Vérifions ça !

Code : C

```
#include <stdio.h>
int main(void) {
    printf("%g\n%g\n", (1.0 - .2 - .2 - .2 - .2 - .2),
                    (1.0 - (.2 + .2 + .2 + .2 + .2)) );
    return 0;
}
```

Compilé chez moi, j'obtiens ceci :

Code : Console

```
5.55112e-017
0
```

Ça parle tout seul.

Remarque hors-sujet (mais importante)

De façon plus générale, il faut rester vigilant lorsqu'on écrit des expressions impliquant des flottants, car des expressions *a priori* équivalentes, du point de vue d'un humain, se révèlent en fait différentes en pratique. 😊 L'exemple précédent l'illustre bien. D'autres cas parlants sont imaginables.

- Il y a pas mal de cas où ce qui change d'une expression à l'autre est le signe du zéro obtenu. Par exemple, $x - y$ et $-(y - x)$ ne sont pas équivalents car si les deux nombres sont égaux, alors la première expression donne $+0.0$ mais la deuxième -0.0 .
- Il y a aussi des cas où les expressions ne sont pas équivalentes si on considère les valeurs "spéciales" des nombres, à savoir zéro, l'infini ou NaN :
 - $x - x$ ne vaut pas 0.0 si x vaut l'infini ou NaN ;
 - $0 * x$ ne vaut pas 0.0 si x vaut -0.0 , l'infini ou NaN ;
 - x / x ne vaut pas 1.0 si x vaut zéro, l'infini ou NaN ;
 - $x == x$ est faux si x vaut NaN ; 😊
 - $x != x$ est vrai si x vaut NaN.

Cette liste est tirée de la norme C99 (ISO/IEC 9899:TC3), que je vous invite à consulter si vous en voulez une plus complète (localisation dans le [draft PDF](#) de la norme : *Annex F — F.8.2 Expression transformations & F.8.3 Relational operators* (p. 464-466)).

L'environnement des flottants : `<fenv.h>`

Cet ensemble de paramètres (les exceptions et le mode d'arrondi) forme ce qu'on appelle en informatique (c'est une définition générale) un **environnement**. C'est le cadre dans lequel on travaille.

Cet environnement pour les flottants est accessible avec le header standard (C99) `<fenv.h>`. Il permet de manipuler les exceptions (les surveiller, en lever, etc.) et les modes d'arrondis (savoir quel est le mode utilisé et en changer). Je ne détaillerais cependant pas son utilisation dans ce tutoriel déjà bien assez long. Si vous voulez en savoir plus, consultez (par exemple) [cette page Wikipédia](#) ou [cette page du site d'Open Group](#).

Comparer des nombres flottants

L'infâme trahison de l'opérateur `==`

Bon. On a donc vu que l'utilisation des flottants en C est semée d'embûches : on risque des arrondis et des expressions "normalement" équivalentes ne le sont pas.

Mais ce n'est pas tout ! Les comparaisons vont aussi vous donner du fil à retordre. 😞 En effet, **l'opérateur de comparaison `==` renvoie vrai si ses deux opérandes sont EXACTEMENT égales**, ou s'il compare un zéro positif et un zéro négatif. Or, avec les problèmes d'arrondis, ce n'est pas forcément vrai, car une différence minuscule peut s'être glissée entre les deux nombres testés. Ainsi, vous risquez de vous retrouver avec des égalités qui devraient être vraies mais qui sont fausses, ou le contraire !

Cela peut faire planter lamentablement un programme (boucles infinies, instructions dans un bloc conditionnel jamais exécutées...) avec un problème dont la source est difficilement identifiable si l'on est pas averti.

Un petit exemple ?

`4.2e17 == 4.2e17 + 13.37` renverra vrai (d'après l'exemple précédent) !

Autre exemple :

Code : C

```
int main(void) {
    float f=0;
    int i;

    for(i=0; i<100; ++i)
        f+= .01;

    if(f==1)    printf("f==1\n");
    else       printf("f!=1, f==%f", f);

    return 0;
}
```

La sortie sera : `f!=1, f==0.999999`.

Évidemment, ces exemples sont stupides, mais ils permettent de mieux saisir dans quels cas se pose ce problème.

Les utilisateurs de GCC seront intéressés de savoir qu'il existe une option `-Wfloat-equal` qui déclenche un *warning* dès que l'on utilise l'opérateur `==` sur des nombres flottants.

Citation : Vous voilà déniaisés

Argghh...! mais c'est abominable !

Eh oui, c'est horrible. Il est encore temps d'abandonner définitivement l'informatique et d'apprendre le patchwork.

Non attendez, revenez ! Bien sûr, il existe des astuces ! Et heureusement !



La suite est fortement inspirée de [cette page Internet](#) (en anglais). Elle est complète et détaillée et je vous invite à la lire, car je n'aborderai pas forcément tout ce qu'elle dit (mais y ajouterai quelques brouilles de mon cru).

Alors, récapitulons : on cherche à comparer deux nombres à virgule flottante (**float** ou **double**) en tenant compte d'une marge d'erreur due aux arrondis ; on va pour cela écrire une fonction qui renverra un booléen (vrai ou faux, 1 ou 0), pour remplacer l'opérateur ==.

L'écart absolu & l'écart relatif

La première méthode consiste tout simplement à mesurer l'écart entre les deux nombres. On parle d'**écart absolu** (par opposition à l'écart relatif que nous verrons par la suite). Si cet écart est inférieur à une certaine valeur (souvent appelée *epsilon*, ce qui pour un mathématicien signifie « valeur très petite »), alors on considère que les deux nombres sont égaux.

Code : C

```
#include <math.h> // pour la fonction fabs, renvoyant la valeur absolue d'un flottant

#define EPSILON 1e-8

/* ici pour des doubles, mais on fait la même chose pour des floats */
int doublesAreEqual(double a, double b) {
    return fabs(a-b) <= EPSILON;
}

/* La fonction fabs renvoie la valeur absolue (c'est-à-dire positive) du nombre de type double passé en argument ; ses équivalents (C99) sont fabsf pour les float, et fabsl pour les long double (pensez donc à adapter votre code en conséquence pour comparer les autres types flottants). */
```

Ici, `fabs` nous renvoie l'écart absolu entre `a` et `b`. Le reste est facile à comprendre. Simple, non ?

Oui, mais c'est encore loin d'être parfait : en effet, un écart de $1e-8$ (soit 0.00000001) peut s'avérer judicieux pour comparer des nombres compris entre 0.1 et 1 (par exemple, en fonction de la précision que vous souhaitez), mais trop petit pour des nombres entre 10 et 100, ou trop grand pour des nombres entre 0.00001 et 0.0001 ; si vous comparez des nombres compris entre 0.00000001 et 0.0000001, vous avez même une marge d'erreur de 100% !

N'employez donc cette méthode que si vous êtes sûr de l'ordre de grandeur des nombres à comparer, et choisissez un *epsilon* adapté.

Pour pallier à ce problème et faire une fonction plus générique, une solution serait de passer l'écart maximal en argument à la fonction et non de se baser sur une constante de préprocesseur ; ainsi, l'utilisateur pourrait fournir un écart adapté à l'ordre de grandeur des nombres qu'il veut comparer.

Allons plus loin. On va employer l'**écart relatif**. Celui-ci ramène l'écart absolu dans les proportions des nombres comparés :

Code : C

```
#include <math.h>

#define EPSILON 1e-8

int doublesAreEqual(double a, double b) {
    if(a==b) return 1; // si a et b valent zéro (explications
    plus bas)

    double absError= fabs(a-b); // écart absolu
    a= fabs(a); // on ne garde que les valeurs absolues pour la
    suite ...
    b= fabs(b); // ... pour pouvoir calculer le nombre le plus grand
    en valeur absolue

    return ( absError / (a>b? a:b) ) <= EPSILON;
}
```

On divise l'écart absolu par le plus grand des deux nombres en valeur absolue (signification du ternaire), pour avoir quelque chose d'adapté à l'ordre de grandeur des deux nombres.

La ligne commençant par **if** (a==b) vous paraît sans doute bizarre. Elle est là pour gérer le cas où a et b sont égaux à zéro. En effet, sans elle, on aurait une division par zéro qui vaudrait NaN, et la comparaison serait donc toujours fausse. On compare donc les deux nombres pour que la fonction renvoie vrai s'ils sont identiques et égaux à zéro (positif ou négatif).



Remarque : Dans ce tutoriel, j'utilise le C99, et non le C90 enseigné dans le cours de M@teo21. Celui-ci autorise de déclarer des variables après des instructions. Si c'était juste pour ce détail, ce serait superflu, mais plus loin on en aura réellement besoin.

Cependant, il subsiste un problème : dans le cas de deux nombres très proches de zéro, l'écart relatif sera très important (car on divise par un tout petit nombre) alors que ces deux nombres seront très proches... Pour y remédier, on réintroduit l'écart absolu : la fonction retournerait vrai si l'écart absolu OU l'écart relatif (au moins l'un des deux) est inférieur à une valeur donnée (qu'on passe en argument à la fonction). D'où le code définitif :

Code : C

```
#include <math.h>

#define EPSILON 1e-8

int doublesAreEqual(double a, double b, double maxAbs, double
maxRel) {
    if(a==b) return 1;

    double absError= fabs(a-b);
    a= fabs(a);
    b= fabs(b);

    return absError <= maxAbs || ( absError / (a>b? a:b) ) <=
maxRel;
}
```

La représentation en mémoire convertie en entier

Maintenant, vous avez du code à peu près potable et fonctionnel. Toutefois, il existe une autre manière de faire, plus pratique mais un peu plus *hard*. Accrochez-vous, ça va secouer. 🤖

Imaginez qu'au lieu de se baser sur l'écart entre les deux nombres, on cherche à déterminer combien de nombres possibles les séparent ? Ainsi, on aimerait placer une marge d'erreur, non sur la valeur elle-même des nombres flottants, mais sur leur "éloignement", pour pouvoir dire par exemple : « *J'accepte les 5 nombres en dessous et les 5 nombres au dessus de la valeur machin* ».

Eh bien, grâce au format de l'IEEE, c'est possible !



Ici, on a donc impérativement besoin du format IEEE 754 ! L'astuce présentée est basée dessus. Si jamais ce n'est pas ce format que vous avez chez vous, vous ne pourrez sans doute pas la mettre en œuvre.

Toutefois, je vous invite à lire quand même ce qui suit, cela pourra peut-être vous intéresser (~~et vous convaincre de changer d'ordinateur~~) ; et en tous cas, lisez la dernière sous-partie (intitulée "Code complet"), car bien que je me base sur cette astuce, je vous présente des idées qui pourront vous intéresser même si vous utilisez l'écart absolu/relatif.

En effet, ce format garantit que « *si deux nombres du même type à virgule flottante sont consécutifs, alors leurs représentations entières le sont aussi (selon le bit de poids fort pour déterminer le signe, et non la règle du complément à 2).* »

Que signifie ce charabia ? Eh bien, prenons 2 nombres de type `float` codés sur 32 bits selon le format IEEE 754 (évidemment, cela s'applique aussi aux `double`) :

Code : Console

Valeur du float	binaire	représentation en mémoire	hexadécimal	en
+1.9999998	0 0111111 1	1111111 11111111 11111110	3F FF FF FE	10
+1.9999999	0 0111111 1	1111111 11111111 11111111	3F FF FF FF	10

Il apparaît que ces 2 nombres sont "consécutifs", il ne peut pas y avoir d'autre nombre du même type dont la valeur serait comprise entre les 2. Or, que constate-t-on ? **Leurs représentations en mémoire, si on les lit comme des nombres entiers**, sont également consécutives !

Pour savoir si les deux nombres à comparer sont "voisins", il suffit donc de comparer leur représentation en mémoire convertie en nombre entier.



Par la suite, je dirais (abusivement, certes) "représentation entière" plutôt que représentation en mémoire lue comme un entier" (c'est quand même plus court).



Mais comment accéder à cette représentation entière ?

~~Ben, c'est simple, il suffit de faire `(int)monFloat` ...~~ Surtout pas ! En faisant ça, on convertit le nombre à virgule en nombre entier, et on obtient donc un nombre arrondi à l'unité. Ça n'a rien à voir avec ce que l'on veut. La bonne formule est donc, tenez-vous bien :

`* (int*) &monFloat`

Je vous laisse méditer là-dessus. 🤔 Ce n'est pas vraiment compliqué, quand on y pense. Quelques explications si vraiment vous bloquez :

Secret (cliquez pour afficher)

pour comprendre cette expression, il faut en fait la lire de droite à gauche :

- `&monFloat` renvoie l'adresse de la variable de type float, donc un pointeur sur float (un `float*`) ;
- `(int*)` convertit ce pointeur sur float en un pointeur sur int ; ainsi, l'ordinateur considérera la variable pointée comme un int et non plus un float ;
- et hop ! le tour est joué, il ne nous reste plus qu'à déréférencer ce pointeur avec `*` pour accéder à la variable pointée, cette fois lue comme un entier (int) et non comme un flottant.

Maintenant, du code avec ce que je viens de vous dire :

Code : C

```
#include <stdlib.h> // pour la fonction abs, renvoyant la valeur
absolue d'un entier
#include <stdint.h> /* header du C99, qui fournit des types entiers
de taille fixe :
- int32_t, int64_t : entier signé de 32 bits ou de 64 bits ;
- uint32_t, uint64_t : entier non-signé de 32 bits ou de 64 bits. */

#define INTREPOFFLOAT(f) ( *(int32_t*)&(f) ) // représentation
entière d'un float de 32 bits
#define INTREPOFDOUBLE(d) ( *(int64_t*)&(d) ) // représentation
entière d'un double de 64 bits

#define MAXULPS 5

/* ici pour des floats, mais on fait exactement pareil pour des
doubles */
int floatsAreEqual(float a, float b) {
    if(a==b) return 1;

    return abs( INTREPOFFLOAT(a) - INTREPOFFLOAT(b) ) <= MAXULPS;
}
/* attention à la fonction abs, qui prend un int en argument ; un
int fait 16
ou 32 bits : il peut donc être trop petit pour contenir les 32 bits
de la
représentation entière d'un float, et sera de toutes façons
insuffisant
pour les 64 bits de celle d'un double. Voyez la fonction labs qui
prend un
long int, ou llabs (C99) qui prend un long long int, ou mieux,
écrivez vos
propres fonctions de valeur absolue, pour 32 et 64 bits (avec les
types de
<stdint.h>) ; ainsi, vous n'aurez plus de problèmes de taille des
types
pouvant varier. Ici, je garde les fonctions standards par souci de
clarté. */
}
```

La constante MAXULPS (de ULP, "Unit of Least Precision", c'est-à-dire la valeur qui sépare deux flottants consécutifs) nous fournit notre marge d'erreur. Si l'écart entre les représentations entières est inférieur à MAXULPS, alors on considère que les nombres sont égaux. 😊

Ici, la ligne commençant par **if** (a==b) est là pour gérer le cas où les deux nombres seraient +0.0 et -0.0. En effet, +0.0 et -0.0 ont des représentations entières très différentes, ce qui fait que la fonction retournerait faux sans ce test préalable (avant, nous comparions des flottants avec l'opérateur == qui renvoie vrai si les deux nombres sont des zéros de signes différents, ce qui explique pourquoi nous n'avons pas eu de problèmes jusqu'à présent).

En outre, remarquez qu'on utilise les types entiers définis dans le header standard <stdint.h> au lieu des types habituels (int, long int...). En effet, la taille de ces derniers dépend de l'implémentation et n'est donc pas connue, il serait donc dangereux (non portable) de s'appuyer dessus ; au contraire, la taille de int32_t et int64_t est connue et fixe. Ce header a été introduit avec C99, c'est pourquoi je vous ai dit qu'on allait devoir se baser sur cette version du langage C.

Un peu de maths pour vous aider à choisir votre marge d'erreur (rien de méchant, rassurez-vous) !

Une variation de Δ_{mantisse} dans la représentation entière de la mantisse codée sur m bits correspond à une variation de la valeur du flottant donnée par la formule suivante :

$\Delta_{\text{flottant}} = \frac{\Delta_{\text{mantisse}}}{2^m} \times 2^{\text{exposant}} = \Delta_{\text{mantisse}} \times 2^{\text{exposant}-m}$ (où *exposant* est l'exposant réel). Cette formule provient directement de celle donnant l'intervalle entre les nombres consécutifs en fonction de l'exposant.

Pour un nombre flottant compris entre 1 et 2, une variation d'un ULP correspond donc à une variation de valeur du nombre flottant de $\frac{1}{2^{23}} \approx 1,192 \times 10^{-7}$ pour un `float` et $\frac{1}{2^{52}} \approx 2,220 \times 10^{-16}$ pour un `double`.

Si vous choisissez comme moi une marge de 5 ULPs, alors votre marge de valeur (pour un nombre flottant compris entre 1 et 2) sera $\frac{5}{2^{23}} \approx 5,960 \times 10^{-7}$ pour un `float` et $\frac{5}{2^{52}} \approx 1,110 \times 10^{-15}$ pour un `double`.

Mais (eh oui, encore un mais) il reste encore un détail à régler, et à ce stade j'aimerais que vous leviez tous la main pour me le dire. Allez, un indice : ça concerne la parenthèse de la phrase en italique de tout à l'heure... 🤔 Ben oui, le signe ! Les flottants, selon la norme IEEE 754, sont signés selon le principe du bit de signe et non du complément à 2. Or, les entiers (du moins sur la grande majorité des ordinateurs aujourd'hui) sont stockés... selon la règle du complément à 2.

Un exemple pour bien voir (je ne vous met plus le binaire, vous êtes grands maintenant) :

Code : Console

Valeur du float	représentation en mémoire	
	hexadécimal	en base 10 selon le complément à 2
+4.2038954 e-45	00 00 00 03	3
+2.8025969 e-45	00 00 00 02	2
+1.4012985 e-45	00 00 00 01	1
+0.0000000	00 00 00 00	0
-0.0000000	80 00 00 00	-2147483648
-1.4012985 e-45	80 00 00 01	-2147483647
-2.8025969 e-45	80 00 00 02	-2147483646
-4.2038954 e-45	80 00 00 03	-2147483645

Comme vous le voyez, le dernier nombre est inférieur à l'avant-dernier, et pourtant sa représentation entière (en `signed` comme en `unsigned`) est supérieure !

En vérité, cela ne porte pas à conséquence si l'on compare deux nombres négatifs, car on ne s'intéresse qu'à l'écart entre les représentations entières, qui lui ne change pas ; le problème se pose lorsque l'on compare deux nombres de signes opposés.

Heureusement, il existe une solution. Il suffit de convertir les nombres négatifs selon la règle du bit de signe, en nombres négatifs selon la règle du complément à 2. Je vous laisse chercher ; aidez-vous de l'exemple ci-dessus...

Trouvé ? Il suffit de garder la valeur telle quelle si le flottant est positif, ou s'il est négatif de soustraire `0x 80 00 00 00` à la représentation entière puis d'inverser le signe de cette représentation. Cela revient à faire l'opération suivante :

```
représentation = 0x 8000 0000 - représentation ;
```

Similairement, pour un `double` de 64 bits, on fera `représentation = 0x 8000 0000 0000 0000 - représentation`.

On obtient alors ceci :

Code : Console

Valeur du float	représentation "transformée"		
	hexadécimal	en base 10 selon le complément à 2	
+4.2038954 e-45	00 00 00 03	3	
+2.8025969 e-45	00 00 00 02	2	
+1.4012985 e-45	00 00 00 01	1	
+0.0000000	00 00 00 00	0	
-0.0000000	00 00 00 00	0	*
-1.4012985 e-45	FF FF FF FF	-1	*
-2.8025969 e-45	FF FF FF FE	-2	*
-4.2038954 e-45	FF FF FF FD	-3	*

(* = a été transformé)

Comme vous le voyez, les représentations entières sont maintenant cohérentes, on peut les comparer sans problème. Et même les deux zéros (positif/négatif) sont égaux !

Du code, du code !

Code : C

```
#include <stdlib.h>
#include <stdint.h>

#define INTREPOFFLOAT(f) ( *(int32_t*)&(f) )
#define INTREPOFDOUBLE(d) ( *(int64_t*)&(d) )

#define MAXULPS 5

int floatsAreEqual(float a, float b) {
    int32_t aInt= INTREPOFFLOAT(a);    // représentations entières
    int32_t bInt= INTREPOFFLOAT(b);

    if(aInt<0)    aInt= 0x80000000 - aInt;    // ou 0x8000000000000000
    pour des doubles
    if(bInt<0)    bInt= 0x80000000 - bInt;
    /* NOTE: on teste (aInt<0) et non (a<0). En effet, si a== -0.0 (zéro
    négatif),
    alors le test (a<0) renverrait faux, et on garderait la
    représentation de
    -0.0, à savoir 0x80000000. La règle du complément à 2 garde
    l'avantage du
    bit de signe : si le bit de poids fort est à 1, alors le nombre
    entier est
    négatif, et réciproquement ; on peut donc utiliser le test sur
    l'entier et
    non sur le flottant pour savoir s'il faut "transformer" la
    représentation. */

    return abs( aInt - bInt ) <= MAXULPS;
}
```

Attention à bien utiliser les types signés (`int32_t` et `int64_t`) et non les non signés (`uint32_t` et `uint64_t`).

Bon, ce n'est pas encore parfait, mais c'est très convenable. Quelques points améliorables :

- les infinis : comme vu précédemment, les infinis sont adjacents aux plus grands nombres en valeur absolue. Notre fonction pourrait par exemple nous dire qu'un nombre positif très très très grand et $+\infty$ sont égaux, alors que ce n'est pas vrai (ne discutez pas, d'un point de vue mathématique c'est faux 😊);
- les NaN : de même, certains NaN pourraient être comparés comme égaux à l'infini ou à un nombre extrêmement grand en valeur absolue, voire à un autre NaN. Or, normalement, n'importe quelle comparaison avec un NaN (hormis `!=`) devrait valoir faux.

Ces "détails" peuvent être corrigés avec des vérifications supplémentaires. À ce sujet, les [macros de test de nombres flottants \(C99\)](#) peuvent servir. En résumé (je vous invite à consulter le manuel avec le lien précédent) :

Citation : Le manuel : `fpclassify`, `isfinite`, `isnormal`, `isnan`, `isinf`

Depuis le C99, le header `<math.h>` définit les macros `isfinite`, `isnormal`, `isnan` et `isinf` ; elles prennent toutes un nombre flottant en argument (peu importe son type), et renvoient un booléen indiquant respectivement si le nombre est fini, normalisé, NaN ou infini (le retour de `isinf` n'est pas forcément 1 ou 0).

La macro `fpclassify` est également définie. On l'utilise comme les autres, et sa valeur de retour indique le type du nombre flottant (`FP_ZERO`, `FP_SUBNORMAL`, `FP_NORMAL`, `FP_INFINITY`, `FP_NAN`).

Code complet

Je vous propose finalement un code complet.

J'y ai introduit une fonction `cmpFloats` (ou `cmpDoubles`) de mon cru qui permet une comparaison plus générale : en effet, à la manière de `strcmp`, elle renvoie -1 si $a > b$, 0 si $a == b$ ou 1 si $a < b$; elle renvoie par ailleurs -2 si l'un des deux nombres au moins est NaN.

Ainsi, il devient plus facile de tester les deux nombres (j'ai de plus écrit des macros simples pour faciliter les comparaisons). En effet, avant, pour tester par exemple $a < b$ (strictement inférieur), il fallait faire `if (a < b && !floatsAreEqual(a, b))`, ce qui était plus lourd à écrire.

Header

Secret (cliquez pour afficher)

Code : C - `cmpfloats.h` (header)

```
#ifndef INCLUDE_CMPFLOATS_H
#define INCLUDE_CMPFLOATS_H

#include <stdint.h>
#include <math.h> /* pour les macros de test des nombres
flottants ( isnan() et isinf() ) */

/* représentation entière du nombre en virgule flottante */
#define INTREPOFFLOAT(f) ( *(int32_t*)&(f) )
#define INTREPOFDOUBLE(d) ( *(int64_t*)&(d) )

/* marge maximale séparant deux nombres flottants considérés comme
égaux,
en termes d'ULPs ("Unit of Least Precision") */
#define MAXULPSFLOAT 5
#define MAXULPSDOUBLE 5

/* renvoie vrai (1) si les 2 nombres sont égaux, faux (0) sinon */
int floatsAreEqual(float a, float b);
int doublesAreEqual(double a, double b);

/* renvoie -1 si a>b, 0 si a==b, 1 si a<b, ou -2 si a ou b est NaN
*/
int cmpFloats(float a, float b);
int cmpDoubles(double a, double b);

/* macros booléennes (à utiliser dans des tests simples) */
#define CMPFLOATS_EQUAL(a,b) (cmpFloats((a),(b))==0) // => a==b
#define CMPFLOATS_UNEQUAL(a,b) (cmpFloats((a),(b))!=0) // => a!=b
#define CMPFLOATS_GT(a,b) (cmpFloats((a),(b))==-1) // => a>b
#define CMPFLOATS_LT(a,b) (cmpFloats((a),(b))==1) // => a<b
// #define CMPFLOATS_GTEQUAL(a,b) (cmpFloats((a),(b))!=1) // =>
a>=b
// #define CMPFLOATS_LTEQUAL(a,b) (cmpFloats((a),(b))!=-1) // =>
a<=b
#define CMPFLOATS_GTEQUAL(a,b) ((cmpFloats((a),(b))-1)&2) // =>
a>=b
#define CMPFLOATS_LTEQUAL(a,b) (cmpFloats((a),(b))>=0) // => a<=b
#define CMPFLOATS_NAN(a,b) (cmpFloats((a),(b))==-2) // => a==NaN
// b==NaN

#define CMPDOUBLES_EQUAL(a,b) (cmpDoubles((a),(b))==0) // => a==b
```

```
#define CMPDOUBLES_UNEQUAL(a,b) (cmpDoubles((a),(b))!=0) // =>
a!=b
#define CMPDOUBLES_GT(a,b) (cmpDoubles((a),(b))==-1) // => a>b
#define CMPDOUBLES_LT(a,b) (cmpDoubles((a),(b))==1) // => a<b
// #define CMPDOUBLES_GTEQUAL(a,b) (cmpDoubles((a),(b))!=1) // =>
a>=b
// #define CMPDOUBLES_LTEQUAL(a,b) (cmpDoubles((a),(b))!=-1) // =>
a<=b
#define CMPDOUBLES_GTEQUAL(a,b) ((cmpDoubles((a),(b))-1)&2) // =>
a>=b
#define CMPDOUBLES_LTEQUAL(a,b) (cmpDoubles((a),(b))>=0) // =>
a<=b
#define CMPDOUBLES_NAN(a,b) (cmpDoubles((a),(b))==-2) // => a==NaN
// b==NaN

#endif //INCLUDE_CMPFLOATS_H
```

Remarquez que j'ai mis des macros en commentaires (correspondant à \leq et \geq), qui ont été remplacées par d'autres. En effet, ces macros ne sont plus valables si `cmp...` renvoie -2 (qui est le code pour NaN).



Les macros `CMP..._GTEQUAL` de remplacement sont très bizarres. En fait, c'est une astuce que j'ai trouvée pour éviter quelque chose du type `(cmpFloats((a),(b))==-1 || cmpFloats((a),(b))==0)`, ce qui est dangereux car `a` et `b` sont potentiellement évalués 2 fois, et non optimisé car on appelle 2 fois la fonction `cmp...`. En fait, les seules macros qui renverront vrai en cas de NaN seront `CMP..._NAN` et `CMP..._UNEQUAL`.

L'utilisation de ces macros est conseillée dans le cas d'un test "simple", c'est-à-dire avec un seul test ; par exemple :

Code : C

```
instructions1;
if(CMPFLOATS_GT(a,b)) { // a>b
    instructions2;
}
instructions3;
```

En revanche, il vaut mieux éviter de les enchaîner pour traiter différentes possibilités (si $a < b$, faire machin, si $a > b$, faire truc...) le **else** est aussi à éviter (car il engloberait aussi le cas de NaN, ce qui dans la plupart des cas n'est pas voulu) :

Code : C

```
instructions1;
if(CMPFLOATS_GT(a,b)) { // a>b
    instructions2;
}
else if(CMPFLOATS_LT(a,b)) { // a<b
    /* /\ on appelle la fonction cmpFloats 2 fois */
    instructions2b;
}
else // a==b
    instructions2t; /* /\ ce code est aussi exécuté dans le cas
de NaN ! */
}
instructions3;
```

Il vaut mieux utiliser un **switch** dans ce cas, qui n'appelle la fonction qu'une seule fois tout en permettant un contrôle précis :

Code : C

```
instructions1;
switch(cmpFloats(a,b)) {
    case -1: // a>b
        instructions2;
        break;
    case 0: // a==b
        instructions2t;
        break;
    case 1: // a<b
        instructions2b;
        break;
    default: break; // NaN
}
instructions3;
```

Autre exemple pour bien saisir l'utilisation du **switch** :

Code : C

```
instructions1;
switch(cmpFloats(a,b)) {
    case -1: // a>b
    case 0: // a==b
        instructions2; // ce code est donc exécuté si a>=b
        break;
    case 1: // a<b
        instructions2b;
        break;
    default: break; // NaN
}
instructions3;
```

Fichier source

Secret (cliquez pour afficher)

Code : C - cmpfloats.c (fichier source)

```
#include "cmpfloats.h"

/* valeur absolue d'un entier de 32 ou 64 bits */
uint32_t abs32(int32_t x) { return x<0? -x : x; }
uint64_t abs64(int64_t x) { return x<0? -x : x; }

int floatsAreEqual(float a, float b) {

    /* vérification pour NaN : si l'un des deux nombres est NaN,
    alors on
    retourne toujours faux */
    if(isnan(a) || isnan(b))
        return 0;

    /* vérification pour les infinis : si l'un des deux nombres est
    infini,
    alors on ne retourne vrai que si les deux nombres sont strictement
    égaux (tous les deux +inf ou -inf) */
```

```

    if(isinf(a) || isinf(b))
        return a==b;

    int32_t aInt= INTREPOFFLOAT(a);
    int32_t bInt= INTREPOFFLOAT(b);
    if(aInt<0)    aInt= 0x80000000 - aInt;
    if(bInt<0)    bInt= 0x80000000 - bInt;
    return abs32( aInt - bInt )  <=  MAXULPSFLOAT;
}

int doublesAreEqual(double a, double b) {
    if(isnan(a) || isnan(b))
        return 0;

    if(isinf(a) || isinf(b))
        return a==b;

    int64_t aInt= INTREPOFDOUBLE(a);
    int64_t bInt= INTREPOFDOUBLE(b);
    if(aInt<0)    aInt= 0x8000000000000000LL - aInt;
    if(bInt<0)    bInt= 0x8000000000000000LL - bInt;
    return abs64( aInt - bInt )  <=  MAXULPSDOUBLE;
}

int cmpFloats(float a, float b) {
    if(isnan(a) || isnan(b))
        return -2;    // -2 si on a au moins un NaN

    if(isinf(a) || isinf(b))
        return (a<b)? -1 : (a>b)? 1 : 0;    // gestion des infinis
    similaire à ci-dessous

    int32_t aInt= INTREPOFFLOAT(a);
    int32_t bInt= INTREPOFFLOAT(b);
    if(aInt<0)    aInt= 0x80000000 - aInt;
    if(bInt<0)    bInt= 0x80000000 - bInt;
    return (abs32(aInt-bInt) <= MAXULPSFLOAT)? 0    // 0 si les
    nombres sont égaux
                                   : (aInt<bInt)? 1    // 1 si a<b
                                   : -1;    // -1 si a>b
}

int cmpDoubles(double a, double b) {
    if(isnan(a) || isnan(b))
        return -2;

    if(isinf(a) || isinf(b))
        return (a<b)? -1 : (a>b)? 1 : 0;

    int64_t aInt= INTREPOFDOUBLE(a);
    int64_t bInt= INTREPOFDOUBLE(b);
    if(aInt<0)    aInt= 0x8000000000000000LL - aInt;
    if(bInt<0)    bInt= 0x8000000000000000LL - bInt;
    return (abs64(aInt-bInt) <= MAXULPSDOUBLE)? 0
                                   : (aInt<bInt)? 1
                                   : -1;
}

```

Ce code est à compiler en C99 (utilisation du header `<stdint.h>`, des macros `isnan` et `isinf`, déclaration de variables après des instructions, utilisation du suffixe `LL` après `0x8000000000000000` pour indiquer que c'est un `long long int...`).

Si vous programmez en C++, pourquoi ne pas surcharger les opérateurs de comparaison ? 🤖 Cela vous simplifiera la vie (toutefois, vous risquerez alors, à la longue, d'oublier que vous avez fait quelque chose pour pouvoir comparer convenablement des flottants, et un jour ça ne marchera plus car vous n'aurez plus inclus votre petit header magique.)

Mais qu'en dit la norme C ?

Citation : Le Zéro harassé

Oh non ! Encore de la théorie !

Rassurez-vous, si vous êtes fatigués, vous pouvez passer cette partie. Elle se destine aux petits curieux qui voudraient aller plus loin pour savoir plus précisément quelle relation entretient IEEE 754 vis à vis de la norme C, et comment déterminer si le compilateur suit bien les formats IEEE 754 ou pas.

IEEE 754 et la norme C

La norme C90 était très floue sur ce sujet, et n'imposait ni ne privilégiait aucun format pour les nombres à virgule flottante. Le C99 a changé cela. En effet, la norme C99 (alias ISO/IEC 9899:TC3, téléchargeable [ici](#) en PDF) introduit le support de la norme IEEE 754.

Voici un extrait le montrant (issu de la liste des changements majeurs depuis le C90) :

Citation : ISO/IEC 9899:TC3 — Foreword (§5, p. xi-xii)

This second edition cancels and replaces the first edition, ISO/IEC 9899:1990 [...]. Major changes from the previous edition include:

[...]

— IEC 60559 (also known as IEC 559 or IEEE arithmetic) support

[...]



Quoi ? C'est quoi IEC 60559 ? Encore une nouvelle norme au nom tordu !

Oulala, pas de panique ! IEC 60559, c'est juste un autre nom de IEEE 754.

Cette norme est donc **supportée par le langage C depuis sa version C99**. Attention ! Supporté ne veut pas dire imposé. Les compilateurs ne sont pas obligés d'adopter les formats IEEE 754. C'est juste que s'ils le font, ils doivent suivre les règles de support spécifiées par la norme C99.

Ce passage le montre clairement :

Citation : ISO/IEC 9899:TC3 — 6.2.6: Representation of types — General (§1, p.37)

The representations of all types are unspecified except as stated in this subclause. [...]

Cela signifie que la représentation de n'importe quel type (et pas seulement les flottants) est inconnue ; elle reste aux choix du compilateur.

Certains compilateurs peuvent supporter plusieurs formats ; dans ce cas, vous devrez spécifier lequel utiliser avec des arguments (sauf si vous voulez garder le format par défaut). GCC, pour sa part, implémente IEEE 754 par défaut, ses utilisateurs peuvent donc dormir sur leurs deux oreilles. 😊



Mais quelles sont les règles de support de IEEE 754 selon le C99 ?

La norme C99 comporte une annexe (l'annexe F) dédiée aux nombres flottants, où se trouve la réponse à cette question. En voici le début :

Citation : ISO/IEC 9899:TC3 — Annex F (normative): IEC 60559 floating-point arithmetic (p.444)

F.1 Introduction

This annex specifies C language support for the IEC 60559 floating-point standard. The *IEC 60559 floating-point standard* is specifically *Binary floating-point arithmetic for microprocessor systems, second edition* (IEC 60559:1989), previously designated IEC 559:1989 and as *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE 754–1985). *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE 854–1987) generalizes the binary standard to remove dependencies on radix and word length. *IEC 60559* generally refers to the floating-point standard, as in IEC 60559 operation, IEC 60559 format, etc. An implementation that defines `__STDC_IEC_559__` shall conform to the specifications in this annex. Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise.

F.2 Types

The C floating types match the IEC 60559 formats as follows:

- The `float` type matches the IEC 60559 single format.
- The `double` type matches the IEC 60559 double format.
- The `long double` type matches an IEC 60559 extended format, else a non-IEC 60559 extended format, else the IEC 60559 double format.

Any non-IEC 60559 extended format used for the `long double` type shall have more precision than IEC 60559 double and at least the range of IEC 60559 `double`.

Recommended practice

The `long double` type should match an IEC 60559 extended format.

[...]

Bon, l'introduction, ce n'est que du blabla administratif ; on passe pour l'instant.

La suite est plus intéressante. Elle indique que le type `float` du langage C doit correspondre au format simple précision (32 bits) de IEC 60559 (alias IEEE 754), etc., etc.

Il est aussi question du type `long double`, dont j'ai peu parlé dans ce tutoriel ; sachez qu'en C, son format est moins bien défini, mais qu'il correspond souvent au format de double précision étendue de IEEE 754 (dont je n'ai pas parlé non plus), ou alors au format de double précision tout court (comme un `double`).

Je ne parlerai pas de la suite de cette annexe, vous pouvez la lire si vous voulez (vous êtes grands). Elle décrit notamment le comportement des opérations sur les flottants.

Enfin, sachez que :

Citation : Taurre

un système peut visiblement encoder les nombres flottants suivant le format défini par la norme IEEE 754, sans pour autant remplir toutes les conditions de l'annexe F de la norme C99 ([cf ce sujet](#)).

Dans le cadre de ce tutoriel, qui s'est surtout focalisé sur les formats de représentation des flottants, cela ne devrait pas poser trop de problème. En effet, n'importe quelle implémentation de IEEE 754, même partielle, devrait au moins respecter les formats spécifiés par cette norme et les correspondances avec les deux types principaux du C (`float` pour 32 bits, `double` pour 64 bits).

Le non-respect partiel de la norme IEEE 754 peut aussi être le fait d'options du compilateur. Par exemple, l'option d'optimisation `-ffast-math` de GCC améliore les performances en accélérant les calculs sur les nombres flottants, mais enfreint certaines règles de IEEE 754.

Je reviens sur l'introduction, elle contient quelque chose d'intéressant. Il est dit que si la macro `__STDC_IEC_559__` est définie, alors c'est le format IEEE 754 qui est utilisé. Cela peut vous être utile pour faire des tests ou adapter votre code. 😊

Cependant, le contraire n'est pas vrai ! Vous pouvez parfaitement avoir une implémentation qui suit IEEE 754 mais qui ne définit pas cette constante. Cela semble être le cas de GCC sous Windows (portage MinGW par exemple), car GCC laisse cette définition aux headers du système ; or, ceux de Windows ne définissent pas `__STDC_IEC_559__`, en partie parce qu'il y aurait un risque d'incompatibilité entre GCC et la bibliothèque C de Windows.

En pratique : savoir si l'implémentation utilise IEEE 754

Puisqu'on ne peut pas compter sur la constante `__STDC_IEC_559__` pour nous renseigner, il faut trouver un autre moyen de déterminer si oui ou non on travaille avec IEEE 754.

Pour cela, le meilleur moyen reste de se renseigner auprès de votre compilateur favori et/ou de votre plateforme cible.

Toutefois, si vous tenez vraiment à faire cette vérification avec du code, je peux vous offrir des pistes...

- dynamiquement (c'est-à-dire au moment de l'exécution de votre programme) : Vous pouvez par exemple déclarer un certain nombre de variables de type à virgule flottante, puis vérifier que leur représentation mémoire correspond à celle attendue en se basant sur IEEE 754. Il faudrait effectuer cette série de tests pour les différents types de nombres (zéros, dénormalisés, normalisés, infinis, NaNs).

Un tel code ne serait pas infaillible (on peut très bien imaginer des formats ressemblant à IEEE 754, qui passeraient avec succès tous les tests) mais il permet d'éliminer certains formats.

L'inconvénient est que du code inutile est intégré à l'exécutable, et que l'on perd du temps à chaque exécution du programme lorsqu'on effectue ces vérifications. À éviter en pratique, donc.

D'un point de vue technique, cela reste cependant un bon exercice. 🤖

- statiquement (c'est-à-dire lors de la compilation) : Pour cela, il faut vous appuyer sur votre ami le préprocesseur. Sans détailler, vous pouvez tester la valeur des macros définies dans le header `<float.h>` du C99 ; celles-ci caractérisent l'implémentation des nombres flottants : précision, exposants minimum et maximum, valeurs minimales et maximales... Tout cela en 3 variantes pour chacun des 3 types à virgule flottante du C.

Cette approche présente l'avantage de ne garder que le code nécessaire lors de la compilation et de ne pas faire cette vérification à l'exécution.

Mais avouez que c'est fichtrement moins rigolo.

L'avantage serait que le bon fonctionnement du code est indépendant du compilateur utilisé, facilitant ainsi les échanges de code.

Notez toutefois que les propositions ci-dessus vérifient la représentation en mémoire, mais pas les différentes opérations sur les flottants. C'est donc incomplet.

Q.C.M.

Quels sont les types à virgule flottante dont on dispose en C ?

- ☐ float et double ;
- ☐ float, double et long double ;
- ☐ float et long float ;
- ☐ float, long float et long long float ;
- ☐ float et double float ;
- ☐ float, double float et long double float.

Comment s'appelle la norme dont il est question dans ce tutoriel, qui définit des formats de nombre flottants ?

- ☐ ISO/IEC 8859-15:1999 ;
- ☐ ANSI/IEEE Std 754-1985 ;
- ☐ ISO/IEC 9899:1999 ;
- ☐ ISO 8899:2003.

Quelle est sa relation vis à vis de la norme du langage C ?

- ☐ Elle est pleinement supportée par la norme C.
- ☐ Le C90 l'imposait, mais plus le C99.
- ☐ Elle est imposée depuis le C99.
- ☐ Le C90 la supportait, mais plus le C99.
- ☐ Elle est supportée depuis le C99.
- ☐ Elle n'est pas supportée par la norme C, et le format des nombres flottants dépend entièrement de l'implémentation.

Selon elle, quels sont les deux formats principaux ?

- ☐ *pas précis* (8 bits) & *précision* (64 bits) ;
- ☐ *précision simple* (16 bits) & *précision double* (32 bits) ;
- ☐ *précision simple* (32 bits) & *précision double* (64 bits) ;
- ☐ *précision simple* (32 bits) & *précision complexe* (128 bits) ;
- ☐ *précision quarte* (32 bits) & *précision double* (64 bits) ;
- ☐ Cela dépend de l'implémentation (machine, système d'exploitation et compilateur).

Qu'est-ce que la mantisse d'un nombre flottant ?

- ☐ Son bit de poids fort, qui indique le signe du nombre flottant ;
- ☐ le champ contenant l'exposant du nombre flottant (en termes de puissance de 2) ;
- ☐ le champ contenant la partie fractionnaire du nombre flottant ;
- ☐ une créature mythologique monstrueuse pourvue d'une tête humaine, d'un corps de lion et d'une queue de serpent.

Quels sont les exposants (en termes de puissances de 2) permis par un `float` pour un nombre fini non nul (c'est-à-dire dénormalisé ou normalisé) écrit sous sa forme scientifique ? Réfléchissez bien.

- ☐ De 0 à 255 ;
- ☐ de -128 à 128 ;
- ☐ de -127 à 128 ;
- ☐ de -127 à 127 ;
- ☐ de -126 à 127 ;
- ☐ de -149 à 127.

Quelle est l'inconvénient de l'opérateur de comparaison `==` ?

- ☐ Il ne tient pas compte des arrondis.
- ☐ Il considère `+0.0` et `-0.0` comme différents.
- ☐ Il n'est pas ~~potable~~ portable.
- ☐ Il ne fonctionne correctement que si le programme est compilé du haut d'une colline par une nuit de pleine Lune.

Correction !

Statistiques de réponses au QCM

Voilà, ce cours touche à sa fin ! Il a été très théorique, j'espère que vous avez digéré.

Vous savez maintenant comment vous servir des nombres à virgule en C, et comment ils fonctionnent *under the hood* (sous le capot). On a aussi vu les difficultés de leur utilisation, et comment les contourner.

J'espère que vous avez apprécié le voyage, et bon code !

Faites-nous de beaux programmes mathématiques ! (*pas des antisèches, hein ?* 😊)

Si vous voulez aller encore plus loin, je ne peux que vous conseiller de lire [ce tutoriel](#) qui décrit comment sont gérés les

nombres flottants au niveau matériel (le processeur).

Sources :

- articles de Wikipédia : [virgule flottante](#), [IEEE 754](#) (n'hésitez pas à aller sur les articles anglais qui sont bien plus complets) ;
- wikilivre (actuellement en rédaction) : [Arithmétique flottante](#) ;
- page présentant l'astuce pour comparer des flottants : [http://www.cygnum-software.com/papers/ \[...\] ingfloats.htm](http://www.cygnum-software.com/papers/ingfloats.htm) (en) ;
- la norme C99 ! ou plutôt son draft (n1256), disponible [ici](#) en PDF (en).

Liens additionnels :

- page de manuel de `isinf`, `isnan`, etc. : <http://www.man-linux-magique.net/man3/isinf.html> ;
- pages du manuel de la lib GNU C concernant les flottants (concepts, constantes fournies par `<float.h>`, exemple pour IEEE 754) : [http://www.gnu.org/s/hello/manual/libc \[...\] e-Macros.html](http://www.gnu.org/s/hello/manual/libc [...] e-Macros.html) (en) ;
- **site très pratique permettant de calculer la représentation en mémoire d'un nombre à virgule et l'inverse, pour les deux formats principaux de IEEE 754 (32 et 64 bits) : <http://babbage.cs.qc.edu/IEEE-754/> (en).**



Je tiens à remercier tous ceux qui ont participé à [cette discussion](#) sur le forum, et en particulier **Taurre** (ainsi que **yoch**) ; ils m'ont beaucoup aidé pour la dernière partie de ce cours, consacrée à la norme C.

Un grand merci également au bêta-testeurs pour leur précieuse assistance et le temps qu'ils ont consacré à ce tutoriel : je citerai surtout **mewtow**, **Adroneus** et **yoch** (sans oublier **programLyrique** et **Duarna**).

Lire aussi

- [Forum](#)
- [Tutoriels](#)
- [News](#)
- [Je peux pas créer ma variable!](#)
- [Crash exo les types de variables](#)
- [Insérer variable dans texte d'une variable](#)
- [Compte le nombre d'occurrences d'une chaîne dans un texte](#)
- [\(gros\) problème avec les nombres décimaux en C++](#)
- [Chaînes de caractères à taille variable en C](#)
- [time.h et ses fonctions](#)
- [Mettez des accents dans vos programmes avec le type `wchar_t`](#)
- [Utiliser la mémoire](#)
- [Installation FMOD Ex avec Xcode](#)
- [Deux nouveaux livres complètent la collection d'ouvrages libres Framabook](#)