

Représentation des données en machine

2/2

Mardi 12 Décembre 2017

Michael FRANÇOIS

francois@esiea.fr

<https://francois.esiea.fr/>

Objectif de ce cours

- Comprendre comment sont représentés les entiers relatifs et les réels en machine.
- Comprendre les limites de ces représentations et les conséquences possibles.
- Introduction sur la norme IEEE-754.
- Introduction sur les tableaux multi-dimensionnels.

Entier naturel (Rappels)

Représenter un entier naturel

- Un entier naturel est un entier positif ou nul (*i.e.* appartenant à $\mathbb{N}=\{0, 1, 2, 3, 4, 5, \dots \infty\}$)
- En informatique, que vaut ∞ ?
 - Problème : avec un nombre fini de bits peut-on représenter un nombre aussi grand qu'on veut ?
 - Pragmatiquement, ∞ c'est le plus grand nombre représentable sur un nombre fini de bits, et dépend donc du type choisi pour représenter la variable
- Choisir le type adapté à la variable revient à chercher le meilleur compromis entre la plage des valeurs possibles et la taille mémoire à réserver.

ATTENTION : grande taille \Rightarrow coût en puissance et coût en mémoire !!!

Type	Nb octets	Descripteur
unsigned char	1	"%hhu"
unsigned short unsigned short int	2	"%hu"
unsigned int	4	"%u"
unsigned long unsigned long int	4	"%lu"
unsigned long long unsigned long long int	8	"%llu"

Remarque : rien n'interdit que deux types différents possèdent la même taille. Dans une implémentation donnée, la taille des entiers courts est inférieure ou égale à celle des entiers et que celle des entiers est inférieure ou égale à celle des entiers longs.

- Les algorithmes de conversion de la base 10 en base 2 s'appliquent (méthode de Hörner/méthode des puissances)
- Exemple d'une variable codée en `unsigned char`

Valeur bit à bit		N
0000	0000	0
0000	0001	1
...		...
0111	1111	127
...		...
1111	1111	255

Et que vaut $255 + 1$?

Entier relatif

Représenter un entier relatif

- Un entier relatif est un entier négatif, nul ou positif (i.e. appartenant à $\mathbb{Z} = \{-\infty, \dots, -2, -1, 0, 1, 2, \dots, \infty\}$)
- Il faut adapter le type de la variable en fonction des valeurs possibles comme pour le cas des entiers naturel.
- Les types utilisés pour représenter un entier relatif sont :
 - Les entiers très courts : 8 bits (1 octet), de $-128(-2^7)$ à $127(2^7 - 1)$
 \implies `char`
 - Les entiers courts : 16 bits (2 octets), de $-32768(-2^{15})$ à $32767(2^{15} - 1)$
 \implies `short int`
 - Les entiers (longs) : 32 bits (4 octets), de -2^{31} à $2^{31} - 1 \implies$ `int`
 - Les entiers très longs : 64 bits (8 octets), de -2^{63} à $2^{63} - 1$
 \implies `long int`

Représentation bit à bit : 1ère intuition

- Pour représenter les entiers positifs, on utilise tout simplement la représentation des entiers naturels vue précédemment. Par exemple, pour une variable de type `char` ayant comme valeur 8 on a :

Valeur bit à bit	\mathbb{Z}
0000 1000	8

- Pour représenter les entiers négatifs, on **peut utiliser** le premier bit pour jouer le rôle du signe : 0 pour + et 1 pour - et coder la valeur absolue comme un entier positif. Par exemple pour -8 on a :

Valeur bit à bit	\mathbb{Z}
1000 1000	-8

PROBLÈME : l'addition bit à bit n'est plus respectée !!!!!!!

- Exemple : $8 + (-8)$ doit donner 0 et pourtant on obtient :

Valeur bit à bit		\mathbb{Z}
	0000 1000	8
+	1000 1000	$+(-8)$
	1001 0000	-16

- Cas particulier de la double représentation du 0 :

Valeur bit à bit		\mathbb{Z}
0000	0000	0
1000	0000	-0

Représentation bit à bit : 2ème intuition

Le problème ne se pose que pour les entiers négatifs !!!

- Une autre façon de définir l'opposé d'un nombre est d'inverser tous les bits ($1 \Rightarrow 0$ et $0 \Rightarrow 1$) "méthode appelée complément à 1".

Par exemple, -8 devient alors :

Valeur bit à bit	\mathbb{Z}
1111 0111	-8

- Dans ce cas que devient $8 + (-8)$? :

	Valeur bit à bit	\mathbb{Z}
	0000 1000	8
+	1111 0111	$+(-8)$
	1111 1111	$8+(-8)$ en complément à 1

- Si on ajoute 1 au résultat précédent, on obtient alors :

Valeur bit à bit			\mathbb{Z}
	1111	1111	$8+(-8)$ en complément à 1
+	0000	0001	1
1	0000	0000	0 en complément à 2

- Finalement, si on oublie le bit de retenue qui vient d'apparaître (après tout, en **char**, on ne dispose que de 8 bits), on trouve enfin 0. Cet algorithme s'appelle le complément à 2.

Remarque : donc l'opposé d'un nombre correspond tout simplement au complément à 1 de ce nombre auquel on ajoute 1.

Comment trouver la représentation bit à bit d'un nombre ? (2ème intuition)

- Si le nombre à représenter est un entier positif ou nul, sa représentation binaire s'obtient par la méthode de Hörner ou celle des puissances.
- Si le nombre est un entier strictement négatif :
 - on code la représentation binaire de sa valeur absolue
 - on inverse bit à bit le résultat précédent
 - on ajoute 1 au résultat précédent
- Tous les nombres positifs ont une représentation binaire qui commence par 0.
- Tous les nombres négatifs ont une représentation binaire qui commence par 1.

Comment trouver la valeur d'un nombre à partir de sa représentation bit à bit ? (2ème intuition)

- Si la représentation binaire commence par 0, alors le nombre est **positif** et il suffit d'appliquer directement la forme polynomiale.
- Si la représentation binaire commence par 1, alors le nombre est **négatif**. Pour trouver sa valeur absolue, il suffit d'appliquer les opérations dans le sens inverse :
 - soustraire 1 au nombre
 - inverser bit à bit le résultat précédent
 - utiliser directement la forme polynomiale

Essayons de voir alors ce que devient la double représentation de 0 :

- $+0$ a pour représentation :

Valeur bit à bit	\mathbb{Z}
0000 0000	0

- -0 à pour représentation en complément à 2 :

Valeur bit à bit	\mathbb{Z}
0000 0000	$ 0 $
1111 1111	complément à 1
+ 0000 0001	+1
1 0000 0000	0 en complément à 2

- Cette fois-ci $+0$ et -0 sont représentés par une même valeur.

Exemple d'une variable codée en `char` :

Valeur bit à bit		\mathbb{Z}	Valeur bit à bit		\mathbb{Z}
1000	0000	-128	0000	0000	0
1000	0001	-127	0000	0001	1
1000	0010	-126	0000	0010	2
...
1111	1110	-2	0111	1110	126
1111	1111	-1	0111	1111	127

Attention : $127 + 1 = -128$! ! !

Type	Nb octets	Descripteur
char	1	"%hhhd"
short short int	2	"%hd"
int	4	"%d"
long long int	4	"%ld"
long long long long int	8	"%lld"

Réels

Représenter un nombre réel

- Un nombre réel est représenté en décimal sous la forme suivante :

$$d_m d_{m-1} \dots d_1 d_0 . d_{-1} d_{-2} \dots d_{-n}$$

La valeur du nombre est :

$$d = \sum_{i=-n}^m d_i \times 10^i$$

- Exemple : 12.94_{10} peut être détaillé comme suit

$$1 \times 10^1 + 2 \times 10^0 + 9 \times 10^{-1} + 4 \times 10^{-2} = 12 + 94/100$$

- En binaire, on a :

$$b_m b_{m-1} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-n}$$

La valeur du nombre est :

$$b = \sum_{i=-n}^m b_i \times 2^i$$

- Exemple : 101.11_2 peut être détaillé comme suit

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 5 + 3/4$$

Passage en binaire d'un nombre réel en base 10

- **Codage de la partie entière** : le codage de la partie entière ne pose pas de problèmes particuliers (conformément au précédent cours). Pour la partie décimale, on doit procéder autrement.
- **Codage de la partie décimale** : l'expression de la partie décimale dans une nouvelle base B est obtenue par multiplication successive par B de la partie décimale du résultat précédent. L'unité obtenue correspond à un élément de la décomposition. On procède ainsi de suite jusqu'à ce qu'il n'y ait plus de partie décimale ou que le nombre de bits obtenus corresponde à la taille du mot mémoire dans lequel on stocke cette partie.

Exemple 1 :

$$0.375 = ?$$

$$0.375 \times 2 = 0.75$$


$$0.75 \times 2 = 1.5$$

$$0.5 \times 2 = 1.0$$

$$0.375 = 0.011$$

Exemple 2 : la décomposition de la partie décimale peut conduire à une suite infinie de termes

$$0.3 = ?$$

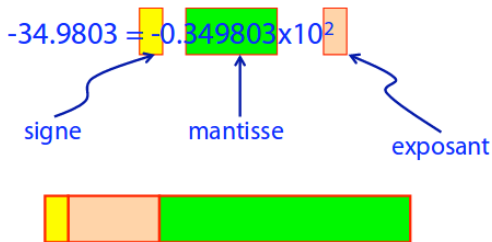


$0.3 \times 2 =$	$\boxed{0}.6$
$0.6 \times 2 =$	$\boxed{1}.2$
$0.2 \times 2 =$	$\boxed{0}.4$
$0.4 \times 2 =$	$\boxed{0}.8$
$0.8 \times 2 =$	$\boxed{1}.6$
$0.6 \times 2 =$	$\boxed{1}.2$
...	

$$0.3 = 0.01001[1001]$$

Codage binaire des nombres réels

- La représentation précédente ne peut pas être utilisée par un ordinateur en raison de la présence de la virgule. En plus, on ne pourra pas représenter des nombres très grands ou très petits (ex. constante de Planck $h = 6.62606957 \times 10^{-34}$ J.s)
- Format utilisé (le nombre n de bits est fixe) :



Remarque : le codage sur un nombre n de bits fixe, implique un nombre fini de valeurs au niveau du stockage :

- les calculs sont nécessairement arrondis ;
- il y aura forcément des erreurs d'arrondi et de précision.

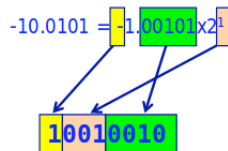
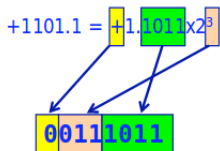
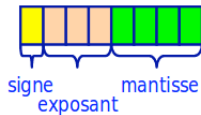
PROBLÈME : un nombre réel peut être écrit de différentes façons. Par exemple : $0.101 \times 2^5 = 101 \times 2^2 = 0.0101 \times 2^6$

- Afin d'éviter des représentations différentes du même nombre, la mantisse est normalisée. Dans la convention la plus courante, un nombre binaire normalisé différent de zéro possède la forme :

$$\pm 1 . bbbb \dots b \times 2^{\pm e}$$

- Sachant que sous cette forme, le bit le plus significatif est toujours égal à 1, il n'est pas nécessaire de le coder, car il est implicite.

Exemple : on considère la représentation suivante :



Est-ce que le nombre zéro peut être codé sous cette forme ?

- **PROBLÈME** : sous cette forme, il est impossible de coder le nombre zéro.
- **Solution** : le nombre zéro est représenté avec tous les bits à 0. Avec la représentation de l'exemple précédent, nous avons: $0.0 = 0\ 000\ 0000$
- Par extension, tous les nombres avec exposant égal à 0 sont dits **non normalisés** : le bit à gauche du point décimal est égal à 0 et non pas à 1, comme c'est le cas pour les autres nombres normalisés. Les nombres non normalisés différents de 0.0 sont utilisés pour représenter de très petites valeurs, proches de 0.0

- **PROBLÈMES** : comment représenter le nombre 1.0 ? Comment représenter les exposants négatifs ?
- Comme un exposant est un nombre entier signé, une solution serait de le représenter en complément à 2. Mais ce n'est pas la solution qui a été choisie !
- En général, l'exposant est représenté de façon biaisée : une constante, le biais, doit être soustrait de la valeur dans le champ pour obtenir la vraie valeur de l'exposant (conversion dite par excès). Donc :

$$\text{Champ exposant} = \text{Exposant} + \text{Biais}$$

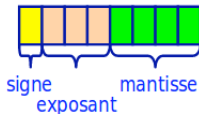
- Typiquement, la valeur du biais est $2^{k-1} - 1$, où k correspond au nombre de bits du champ de l'exposant.
- Cependant, les deux valeurs extrêmes du champ exposant sont réservées pour des cas particuliers :
 - 00...00 : pour les nombres non normalisés ($0 \leq x < 1$)
 - 11...11 : pour infini (positif et négatif) et NaN (Not a Number)

Exemple : pour $k = 4$, la valeur du biais sera $2^3 - 1 = 7$. L'exposant pourra avoir une valeur comprise strictement entre $-2^3 + 1$ et 2^3 , même si la valeur représentée dans le champ exposant sera **exposant+biais**.

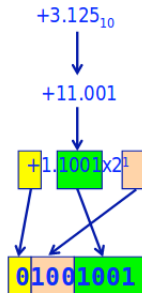
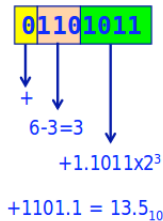
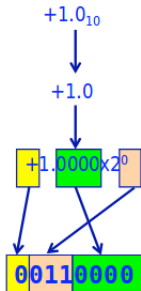
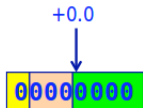
champ exposant	exposant	champ exposant	exposant
0000	non normalisé	1000	1
0001	-6	1001	2
0010	-5	1010	3
0011	-4	1011	4
0100	-3	1100	5
0101	-2	1101	6
0110	-1	1110	7
0111	0	1111	infini

Remarque : avec cette représentation, la comparaison entre deux nombres est facilitée ainsi que la représentation des nombres 0.0 et 1.0

Exemple : on considère la représentation suivante



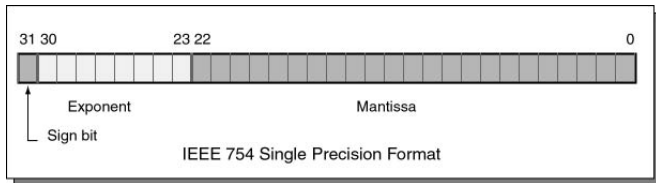
$$\text{biais} = 2^{3-1}-1 = 2^2-1 = 3$$



Représentation d'un nombre réel avec la NORME IEEE-754 (virgule flottante) - type float

IEEE - Institute of Electrical and Electronics Engineers. Association professionnelle constituée d'ingénieurs électriciens, d'informaticiens, de professionnels du domaine des télécommunications, etc. Elle a pour but de promouvoir la connaissance dans le domaine de l'ingénierie électrique (électricité et électronique). L'IEEE-754 est une norme pour la représentation des nombres à virgule flottante en binaire.

- En **simple precision**, la norme code un nombre réel sur 32 bits selon ces trois composantes :
 - le **signe** est représenté par un seul bit, le bit de poids fort (celui le plus à gauche), 0 si positif, 1 si négatif ;
 - l'**exposant** est codé sur les 8 bits consécutifs au signe ;
 - la **mantisse** (seuls les bits situés après la virgule sont retenus) sur les 23 bits restants.



- Récapitulatif : en simple précision on a 32 bits repartis comme suit
 - Signe \Rightarrow 1 bit
 - Exposant \Rightarrow 8 bits
 - Mantisse \Rightarrow 23 bits
- La valeur d'un nombre est donnée par l'expression :

$$(-1)^s \times \left(1 + \sum_{i=0}^{22} m_i 2^{i-23}\right) \times 2^{e-127}$$

- Exemple : $B = 1\ 10000010\ 001100000000000000000000$

$$B = (-1)^1 \times (1 + 1 \times 2^{-3} + 1 \times 2^{-4}) \times 2^{(2^7+2^1)-127}$$

$$B = -1 \times (1 + 0.125 + 0.0625) \times 2^{130-127}$$

$$B = -1 \times (1.1875) \times 2^3$$

$$B = -1.1875 \times 2^3$$

$$B = -9.5$$

Remarque : les réels double précision (type `double`) sont stockés sur 64 bits repartis comme suit

- Signe \Rightarrow 1 bit
- Exposant \Rightarrow 11 bits
- Mantisse \Rightarrow 52 bits

Tableaux multidimensionnels

Tableaux multidimensionnels

Intérêt : représenter des données qui sont naturellement multidimensionnelles comme :

- une image (2D) ;
- un volume (3D) ;
- une vidéo (3D) ;
- un volume dynamique (*i.e.* qui varie avec le temps) (4D) ;
- etc.

Déclaration de tableau

- En C, ce n'est que la généralisation de la déclaration des tableaux à 1 dimension. En effet :
 - il y a autant de [] que de dimensions ;
 - les nombres entre chaque [] indiquent le nombre de cases à réserver en mémoire.
- Exemple :

```
int matrix[2][3];
```

matrix représente un tableau d'entiers de 2 lignes, chacune composée de 3 colonnes.

Accès aux éléments en lecture / écriture

- Comme pour le cas à 1 dimension dans la partie action du code, il suffit de donner le nom du tableau suivi des indices de la case pour l'accès en lecture ou écriture.
- Rappel : les indices commencent à 0 !

```
/* Read access */  
printf("La case (1,2) contient %d\n", matrix[1][2]);  
/* Write access */  
matrix[0][0] = 1;
```

Comment parcourir un tableau multidimensionnel ?

- Il s'agit là encore d'une généralisation du cas à 1 dimension.
- Il suffit de faire une boucle itérative par dimension.

Exemple : initialisation à 0 du tableau `matrix` de dimensions `nbLines` lignes et `nbCols` colonnes

```
for(numLine=0; numLine < nbLines; numLine++)  
    for(numCol=0; numCol < nbCols; numCol++)  
        matrix[numLine][numCol] = 0;
```

Passage d'un tableau multidimensionnel en paramètre d'une fonction

- Lors du passage d'un tableau multidimensionnel comme paramètre d'une fonction, il suffit de préciser le type ,le nom, le nombre de cases à réserver par dimension entre [].
- **ATTENTION** : il n'est pas toléré de laisser des crochets vides contrairement au cas à 1 dimension ! ! !
- Il faut utiliser des variables précédemment déclarées.

Exemples :

```
void myFunction(int nbLin, int nbCol, int iArray[nbLin] [
    nbCol]);
void myFunction1(int nbDim1, int nbDim2, int nbDim3, int
    iArray[nbDim1] [nbDim2] [nbDim3]);
```


Bibliographie

- L. BEAUDOIN, Introduction à l'algorithmique et au langage C (Représentation des données en machine 1/2), cours 1A 2016-2017 ESIEA-Paris.
- Eduardo Sanchez, Représentation des nombres réels, Ecole Polytechnique Fédérale de Lausanne.
- C. DELANNOY, Langage C, éditions EYROLLES, 4ème tirage 2005.