Encodage et représentation des nombres

Binaire / Octet

Les nombres sont représentés en binaires:

ex. pour des entiers

2	10
6	110
156	10011100
86751	10101001011011111

un '1' ou un '0'
est un "bit"

Un **octet** est un groupe de 8 bits

10011100 00101101

1er octet 2eme octet

Binaire / Octet / Hexadécimale

Il est pratique de représenter les nombres binaires en hexadécimale

1 octet (8 bits) se représente par 2 caractères hexadécimaux

ex.

10011100

9 C => 9C en hexadécimale

On a bien: 156 en base 10, 10011100 en base 2, 9C en base 16

ex2.

Le nombre 8A4F en hexadécimale vaut:
100010100100111 en base 2
35407 en base 10

Ce nombre peut se représenter sur 2 octets

Utilisation mémoire

Si on considère un nombre stocké sur 4 octets, on peut compter de

00 00 00 00 à FF FF FF FF

0 à 4294967295 en base 10

Rem. 4294967295 octets ~ 4Go
= limite maximum de RAM sur les anciens PC
(car adresses stockés sur 4 octets)

Mais, uniquement nombre positifs !!

Comment représenter -5 ?

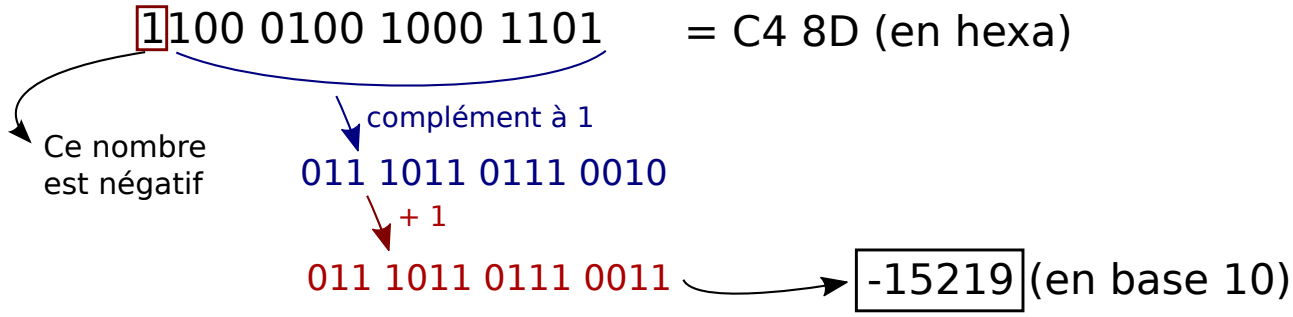
Nombres signés

On utilise un bit pour savoir si le nombre est positif ou négatif

ex.

Attention: les nombres négatifs sont évalués avec le complément à 1+1

ex. un nombre de 2 octets 1100 0100 1000 1101



Exemples nombres entiers

Ex. Pour des nombres entiers stockés sur 4 octets

Binaire	hexa	Valeur en base 10
0110 0010 0000 1111 0010 1000 1111 0011	62 0F 28 F3	1645160691
1110 0010 0000 1111 0010 1000 1111 0011	E2 0F 28 F3	-502322957
0000 0000 0000 0000 0000 0000 0000 0001	00 00 00 01	1
1000 0000 0000 0000 0000 0000 0000 0000	80 00 00 00	-2147483648
1111 1111 1111 1111 1111 1111 1111 1111	FF FF FF FF	-1
0111 1111 1111 1111 1111 1111 1111 1111	7F FF FF FF	2147483647

La taille des données

L'encodage dépend du nombre d'octet alloué pour chaque nombre

- **short**: nombre entier sur 2 octets
- **int**: nombre entier sur 4 octets
- **long**: nombre entier sur 8 octets
- **long long**: nombre entier sur 16 octets (parfois 8 octets)

Exemple de type d'entiers

ex. Le nombre 4876 en base 10

4876 base 10 => 13 0C en hexa

Soit 00010011 00001100 en binaire

Sur une architecture Big Endian:

Type	en binaire	Stockage en hexa
short	00010011 00001100	13 0C
int	00000000 00000000 00010011 00001100	00 00 13 0C
long	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00010011	00 00 00 00 00 00 13 0C
long long	00000000 00000000 . . . . . . 00000000 00000000 00001100 00010011	00 00 00 00 00 00 00 00 00 00 00 00 00 00 13 0C

Exemple de type d'entiers (2)

ex. Le nombre -4876 en base 10

Sur une architecture Big Endian:

Type	Stockage en binaire	Stockage en hexa
short	11101100 11110100	EC F4
int	11111111 11111111 11101100 11110100	FF FF EC F4
long	11111111 11111111 11111111 11111111 11111111 11111111 11101100 11110100	FF FF FF FF FF FF EC F4
long long	11111111 11111111 . . . 11101100 11110100	FF FF FF FF FF FF FF FF FF FF FF FF FF FF EC F4

Nombres flottants

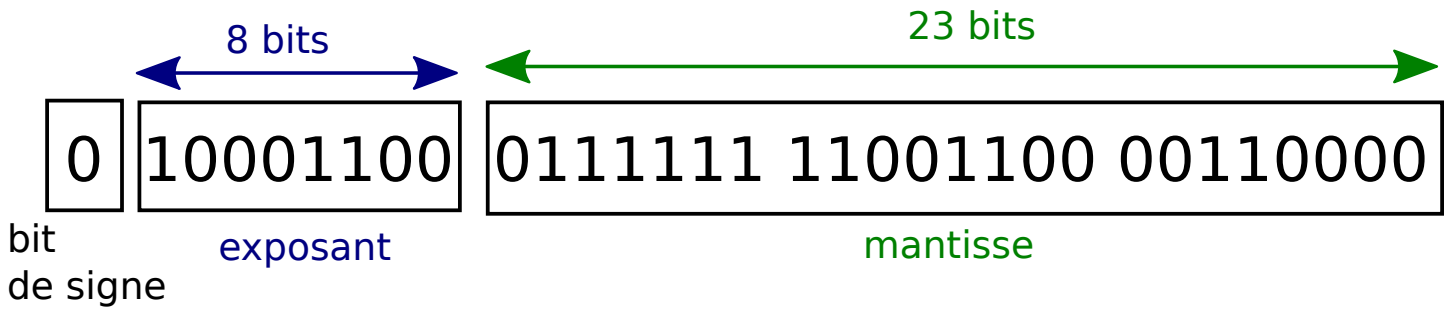
L'encodage des nombres flottants est codifié par la norme IEEE 754

Le type 'float' du C est un nombre flottant stocké sur 4 octets

Le type 'double' du C est un nombre flottant stocké sur 8 octets
(double précision)

Nombres flottants

Soit le nombre binaire: 01000110 00111111 11001100 00110000



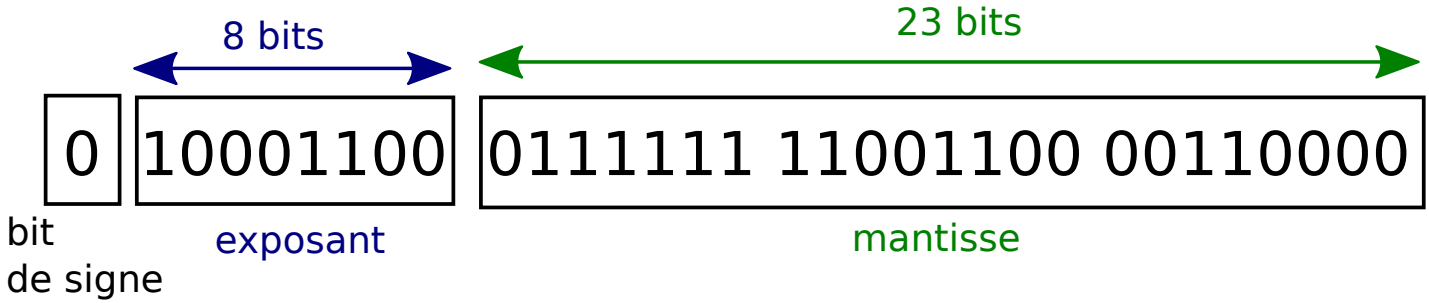
Soit
s: le bit de signe
e: le nombre décimale correspondant à l'exposant
b_i: les bits de la mantisse (i dans [1,23])

Le nombre flottant associé à ce binaire est donné par

$$(-1)^s \; 2^{e-127} \left(1 + \sum_{i=1}^{i=23} b_i \, 2^{-i} \right)$$

Nombres flottants

Soit le nombre binaire: 01000110 00111111 11001100 00110000



s=0, e=140

nombre= $(-1)^s \; 2^{e-127} \left(1 + \sum_{i=1}^{i=23} b_i \, 2^{-i}\right)$

= $2^{140-127}$ (1+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}+2^{-7}+2^{-8}+2^{-9}+2^{-12}+2^{-13}+2^{-18}+2^{-19})

= 12275.046875

Nombres flottants: stockage

Le nombre 12275.046875

Est représenté en binaire en 'float' simple précision par
01000110 00111111 11001100 00110000

Qui vaut en hexa: 46 3F CC 30

Nombres flottants

Remarques:

On ne peut pas représenter tous les nombres exactement

ex. 0.4 ne se représente pas de manière exacte en flottant

00111110 11001100 11001100 11001101

vaut: $2^{125-127}\ (1+2^{-1}+2^{-4}+2^{-5}+2^{-8}+2^{-9}+2^{-12}+2^{-13}+2^{-16}+2^{-17}+2^{-20}+2^{-21}+2^{-23})$

~ 0.400000005960465

00111110 11001100 11001100 11001100

vaut: $2^{125-127}\ (1+2^{-1}+2^{-4}+2^{-5}+2^{-8}+2^{-9}+2^{-12}+2^{-13}+2^{-16}+2^{-17}+2^{-20}+2^{-21}\ \ )$

~ 0.399999976158142

Nombres flottants

Remarques:

On ne peut pas représenter tous les nombres exactement

ex. 0.4 ne se représente pas de manière exacte en flottant

00111110 11001100 11001100 11001101

vaut: $2^{125-127}$ $(1+2^{-1}+2^{-4}+2^{-5}+2^{-8}+2^{-9}+2^{-12}+2^{-13}+2^{-16}+2^{-17}+2^{-20}+2^{-21}+2^{-23})$

~ 0.400000005960465

00111110 11001100 11001100 11001100

vaut: $2^{125-127}$ $(1+2^{-1}+2^{-4}+2^{-5}+2^{-8}+2^{-9}+2^{-12}+2^{-13}+2^{-16}+2^{-17}+2^{-20}+2^{-21})$

~ 0.399999976158142

En C:

```
int a=0.1;
int b=0.3;
int c=0.4;
if(a+b == c)
{...
```

peut être faux

Double précision

'Double' est similaire aux 'float' avec plus de précision

Un double est stocké sur 8 octets

1 bit de signe
11 bits pour l'exposant e
52 bits de mantisse

$$N = (-1)^s \; 2^{e-1023} \left(1 + \sum_{i=1}^{i=52} b_i \, 2^{-i} \right)$$

Attention:
Double n'est pas exacte !
Même problèmes que float.

Double précision: approximation

Rem. 0.4 n'est toujours pas représentable exactement en double précision.

0.4 est stocké comme: 9A 99 99 99 99 99 D9 3F

Soit, le nombre binaire:

00111111 11011001 10011001 10011001
10011001 10011001 10011001 10011010

Qui correspond au nombre décimale:
0.4000000000000002220446...

Nombres flottants

Pour aller plus loin:

- http://floating-point-gui.de/
- *[David Goldberg, What Every Programmer Should Know About Floating-Point Arithmetic]*