



LIF5 - Algorithmique et programmation procédurale

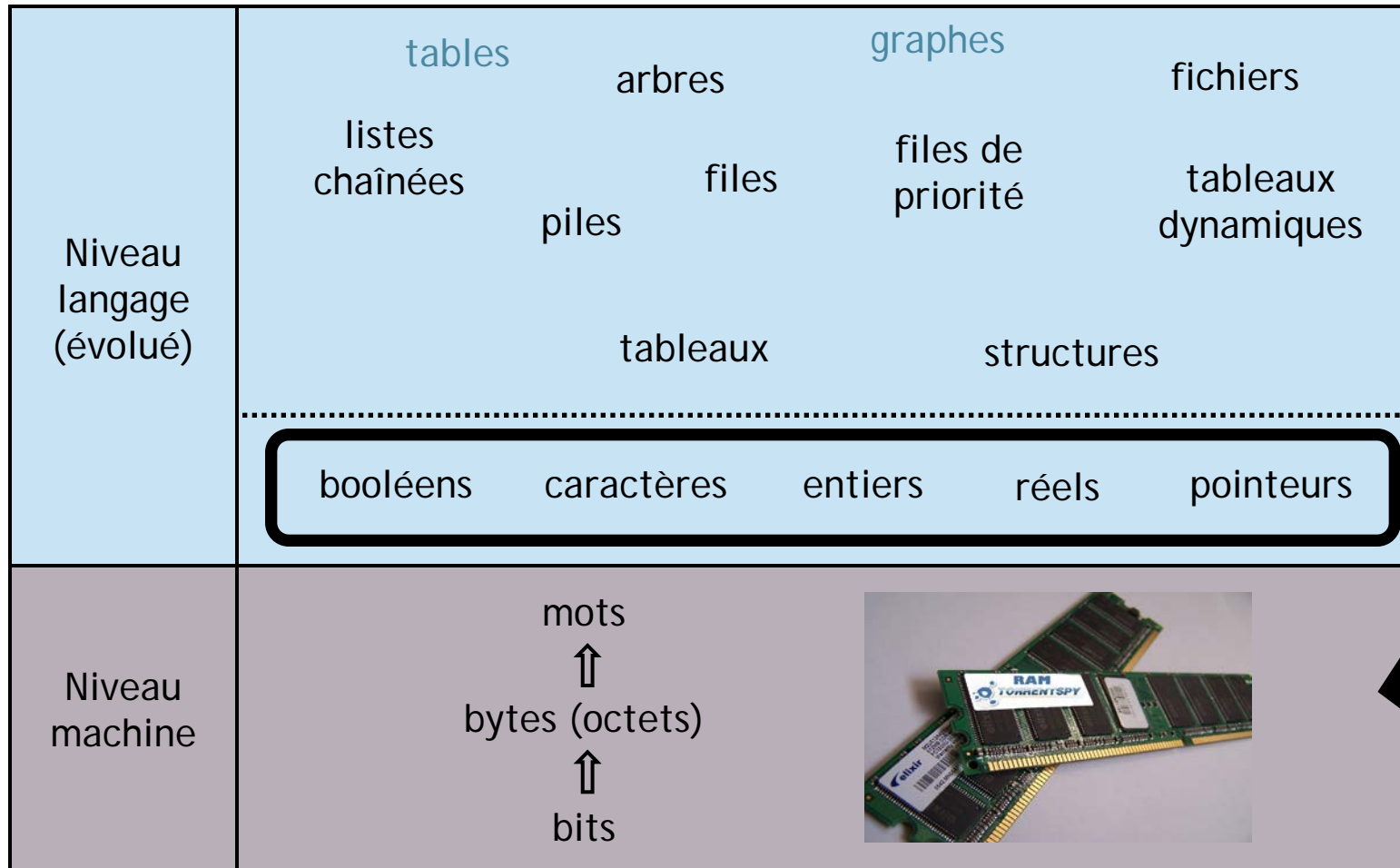
Carole Knibbe

Samir Akkouché

Chapitre 2

Codage des types de base

Comment les structures élémentaires sont-elles représentées en mémoire ?



Le monde merveilleux de l'algorithmique

- i : entier
- a : réel
- c : caractère
- b : booléen

... et on ne se soucie pas de la façon dont l'ordinateur code et manipule ces données

Pourtant...

Pourquoi se soucier du codage des nombres ?

- 23 août 1991, au large de la Norvège : premier tremblement de terre causé par un bug informatique



Coût estimé du crash :
700 millions de dollars



approximation de calcul dans le
logiciel de dessin des ballasts



sous-estimation de 47% de la taille
des parois des ballasts



à 65m de profondeur, rupture d'un
des ballasts



la plate-forme (90 000 tonnes)
coule et se brise à 220m de
profondeur



séisme de magnitude 3 sur
l'échelle de Richter

Pourquoi se soucier du codage des nombres ?

- 4 juin 1996, Kourou, Guyane française : la fusée Ariane 5 explose en vol à cause d'une conversion erronée de réel en entier



Coût estimé : 500 millions de dollars

le système de référence inertielle (SRI) principal et celui de secours exécutent une conversion réel - entier avec un réel trop grand



erreur d'overflow, les deux SRI « plantent » simultanément



utilisant des données incorrectes, le calculateur de bord provoque un braquage intempestif de la fusée



la fusée bascule et se brise sous les forces aérodynamiques



la fusée s'autodétruit

Pourquoi se soucier du codage des nombres ?

- 25 février 1991, Dharan, Arabie Saoudite : 28 morts et 100 blessés à cause d'une erreur d'arrondi

Les batteries de missile Patriot sont censées protéger la base américaine en détectant et détruisant les missiles ennemis



accumulation d'erreurs d'arrondi
(au 24^è bit) sur le calcul du temps



retard de 0,3 secondes sur le
temps réellement écoulé



erreur de 500m sur la prédiction
de la position d'un missile Scud
irakien (vitesse = 1,6 km/s)



pas de lancement de missile anti-
missile

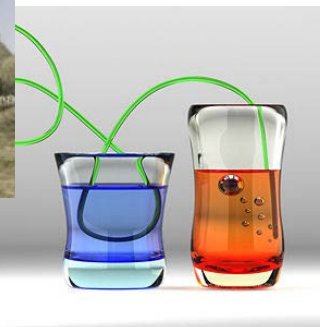
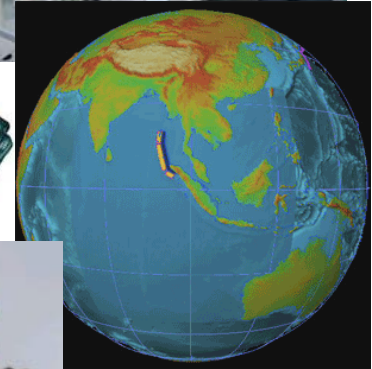
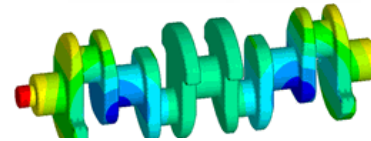


le missile irakien détruit le
baraquement américain

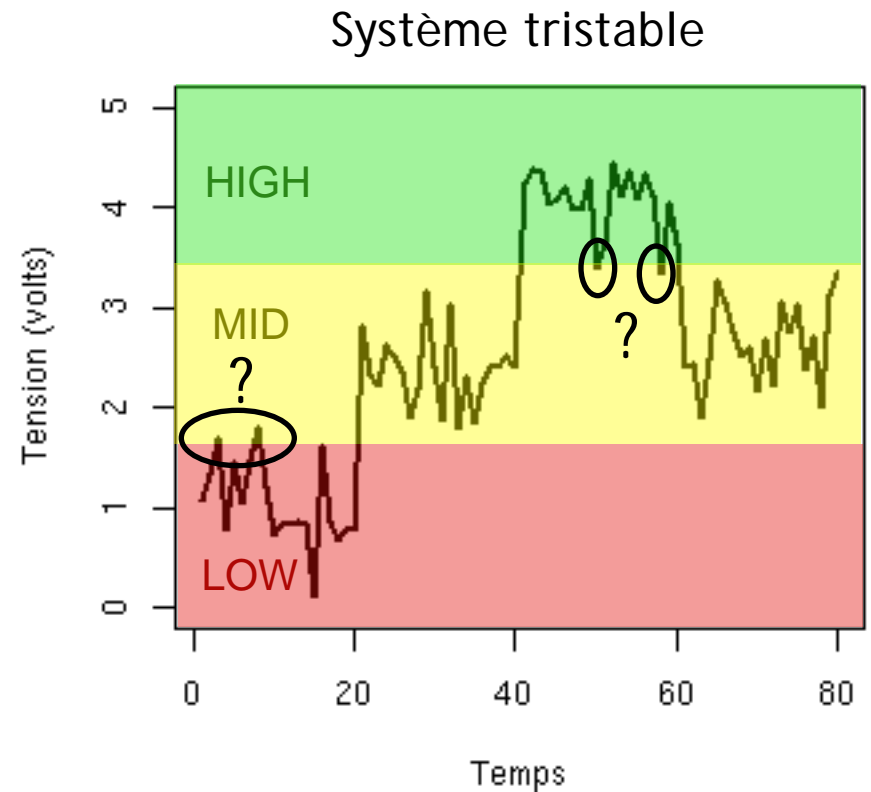
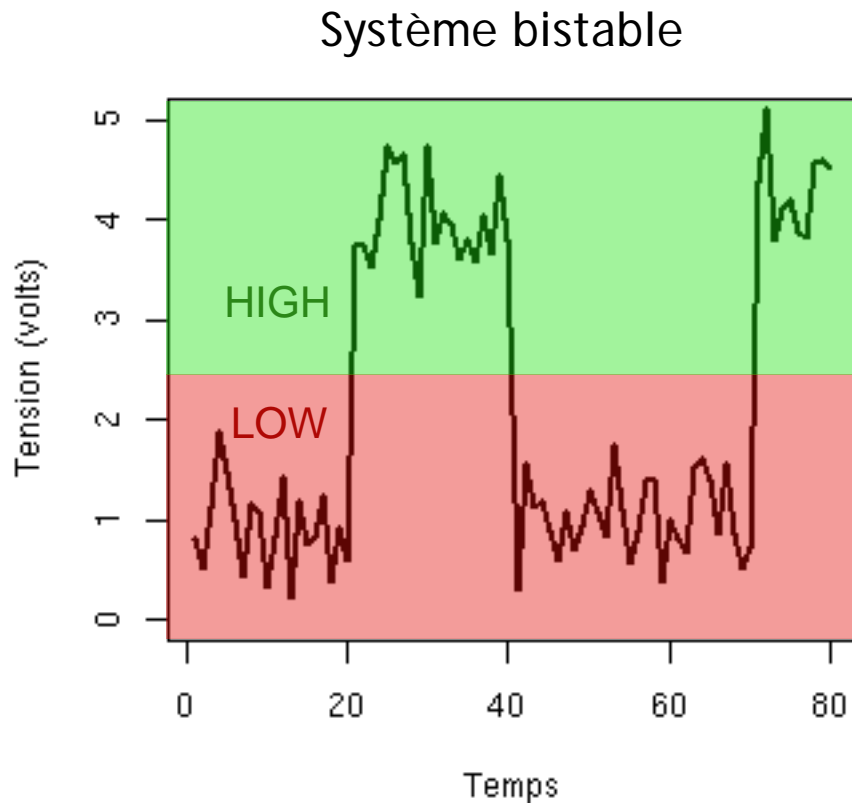
Pourquoi se soucier du codage des nombres ?

De nombreux domaines sont concernés :

- Contrôle de systèmes automatiques
 - Informatique industrielle
 - Robotique
- CAO
- Simulation numérique
 - événements climatiques, épidémies, ...
 - crash de voitures, simulateurs de vol, ...
- Jeux vidéo
- Images de synthèse
 - rendu de la lumière par l'algorithme de lancer de rayon



Représentation binaire : pourquoi ?



Pour un même niveau de bruit, se limiter à deux états permet de coder l'information de façon plus fiable

Codage des entiers positifs : le principe

- Décomposition du nombre en base 2
- Exemple : $45 = 32 + 8 + 4 + 1$
 $= 2^5 + 2^3 + 2^2 + 2^0$
 $= 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
 $2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0$

Donc $45_{10} = 1 \ 0 \ 1 \ 1 \ 0 \ 1_2$
 $5 \text{ volts} \ 0 \text{ volts} \ 5 \text{ volts} \ 5 \text{ volts} \ 0 \text{ volts} \ 5 \text{ volts}$

- Conversion décimal \rightarrow binaire par divisions par 2 successives (cf TD)

Question :

Quelle opération arithmétique est bien plus rapide en représentation binaire ?

Codage des entiers positifs : le principe

- **Notation plus compacte = notation hexadécimale (base 16)**
 - regroupement des bits par paquets de 4
 - 16 valeurs possibles pour chaque paquet : 0, 1, ..., 9, A, B, C, D, E, F
 - préfixe '0x' pour indiquer que le nombre est exprimé en base 16
- Exemple :

$$45_{10} = \overset{2}{1}\overset{2}{0}\overset{2}{1}\overset{2}{1}\overset{2}{0}\overset{2}{1}_2 = \underbrace{\overset{2}{0}\overset{2}{0}\overset{2}{1}\overset{2}{0}}_{\substack{2_{10} \\ 2_{16}}}\overset{2}{1}\overset{2}{1}\overset{2}{0}\overset{2}{1}_2 = 2D_{16} = 0x2D$$

Et on a bien $2 \cdot 16^1 + 13 \cdot 16^0 = 45_{10}$

Codage des entiers positifs : la pratique

- Codage sur un nombre prédéterminé de bits, multiple de 8
- En C :

Type	Nombre de bits ¹	Nombre d'octets ¹	Valeur min	Valeur max
unsigned char	8	1	0	$2^8 - 1 = 255$
unsigned short	16	2	0	$2^{16} - 1 = 65535$
unsigned int	32	4	0	$2^{32} - 1 \cong 4$ milliards
unsigned long	idem unsigned int ¹			

¹ Sur la plupart des ordinateurs actuels. Voir le fichier limits.h de votre machine.

- Ajout de 0 à gauche si nécessaire
 - 45 en unsigned char = 00101101 en hexa : (0x)2D
 - 45 en unsigned short = 0000000000101101 en hexa : (0x)002D
- Attention au risque de dépassement de capacité (overflow)



Dépassement de capacité (overflow)

Exemple : a, b et c sont des « unsigned char » (1 octet)

a = 200

b = 100

c = a + b = ... 44 !

$$\begin{array}{r} \text{x} \quad 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \\ + \quad 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \\ \hline \text{x} \quad 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \end{array}$$

La dernière retenue
et le dernier 1 sont
perdus !

A retenir :

L'arithmétique des ordinateurs n'est PAS celle (idéale) des mathématiques, car les ordinateurs travaillent avec un nombre fini de valeurs

Codage des entiers relatifs : le principe

- 1 bit de signe, et les autres bits pour la valeur absolue ? **NON !**
 - $x + (-x)$ ne donnerait pas 0 !
 - double codage du zéro
- Pour les entiers positifs, codage comme précédemment
- Pour les entiers négatifs, codage en « complément à 2 » :
 - codage de la valeur absolue en binaire
 - calcul du complément à 1 par inversion de tous les bits
 - calcul du complément à 2 par ajout de 1_2
- Exemple : codage de -45 (sur 8 bits)
 - valeur absolue = $45_{10} = 00101101_2$
 - complément à 1 = 11010010_2
 - complément à 2 = $11010010_2 + 1_2 = 11010011_2$

Codage des entiers relatifs : le principe

- On a bien une seule représentation pour 0
... parce que la dernière retenue est ignorée

$$\begin{array}{rcl}
 \text{✕ } 11111111 & \leftarrow & \text{« complément à 1 » de 0} \\
 + \quad 00000001 & \leftarrow & \text{incréméntation} \\
 \hline
 \text{✕ } 00000000 & \leftarrow & \text{« complément à 2 » de 0}
 \end{array}$$

- On a bien $x + (-x) = 0$... parce que la dernière retenue est ignorée

$$\begin{array}{rcl}
 \text{✕ } 00101101 & \leftarrow & 45 \\
 + \quad 11010011 & \leftarrow & -45 \text{ codé en « complément à 2 »} \\
 \hline
 \text{✕ } 00000000
 \end{array}$$

Codage des entiers relatifs : la pratique

En C :

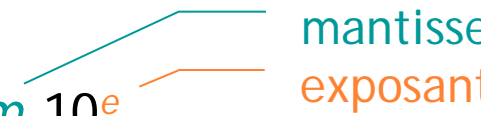
Type	Nombre de bits ¹	Nombre d'octets ¹	Valeur min	Valeur max
signed char	8	1	-128	127
signed short	16	2	-32768	32767
signed int	32	4	$\cong -2$ milliards	$\cong 2$ milliards
signed long	idem signed int ¹			

¹ Sur la plupart des ordinateurs actuels. Voir le fichier limits.h de votre machine.

Sur la plupart des machines, un « int » est par défaut un « signed int » (idem short, long)

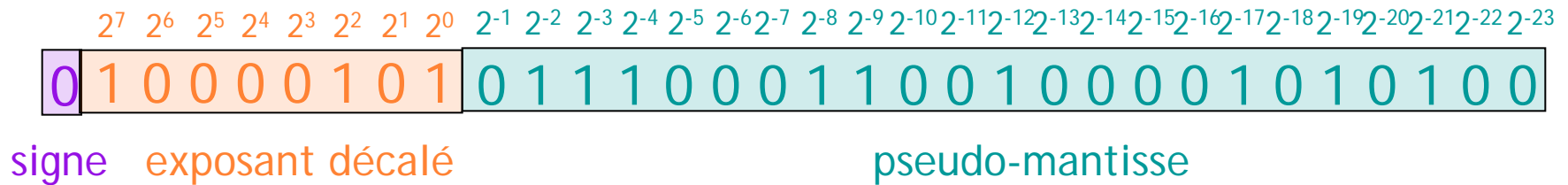


Codage des réels : la représentation en virgule flottante

- Notation scientifique usuelle : $a = m \cdot 10^e$

mantisse
exposant
- Un nombre donné peut être représenté de plusieurs façons
 - exemple : $3,14 = 0,314 \cdot 10^1 = 3,14 \cdot 10^0 = 314,0 \cdot 10^{-2} = \dots$
- L'une est choisie comme représentation standard (normalisation)
 - exemple de normalisation : $0, \text{---} \cdot 10^{\text{--}}$
- On peut aussi utiliser des puissances de 2 : $a = m' \cdot 2^{e'}$
 - exemple de normalisation (les « float ») : $\pm 1, \text{-----} \cdot 2^{\text{-----}}$
 - 1 bit de signe
 - 23 bits pour coder la valeur de la mantisse (puissances négatives de 2)
 - 8 bits pour coder la valeur de l'exposant (puissances positives de 2)

Codage des réels : le standard utilisé par les ordinateurs

Exemple :



$$s = 0$$

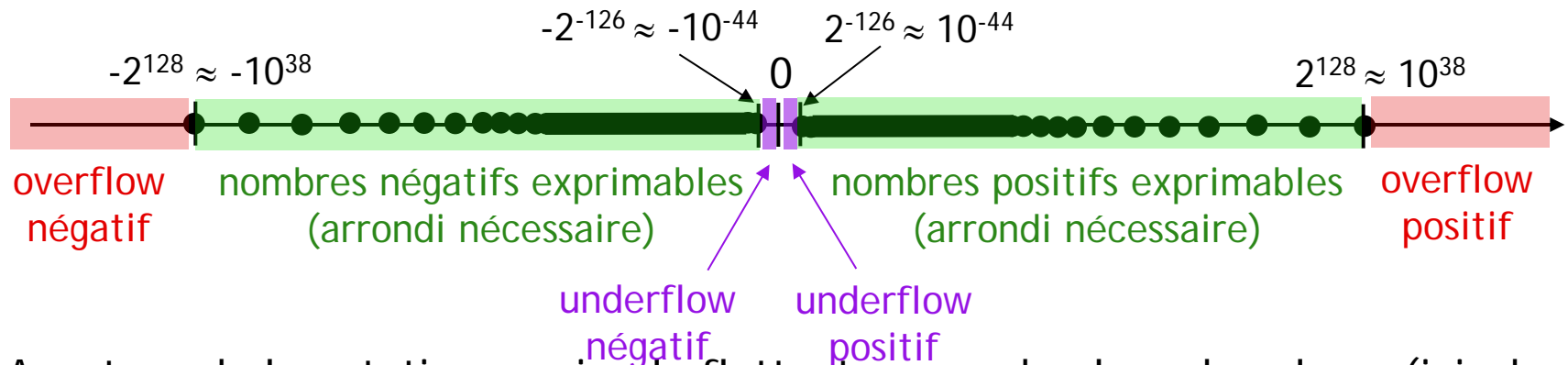
$$e = 2^0 + 2^2 + 2^7 = 1 + 4 + 128 = 133$$

$$\begin{aligned}
 f &= 2^{-2} + 2^{-3} + 2^{-4} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-17} + 2^{-19} + 2^{-21} \\
 &= 0,25 + 0,125 + 0,0625 + 0,00390625 + \dots + 0,000000476837158203125 \\
 &= 0,443613529205322
 \end{aligned}$$

$$\begin{aligned}
 \text{Nombre} &= (-1)^0 \cdot (1 + 0,443613529205322) \cdot 2^{(133-127)} \\
 &= 1 \cdot 1,443613529205322 \cdot 2^6 \\
 &= 92,3912658691405
 \end{aligned}$$

Codage des réels : la représentation en virgule flottante

Example : $\pm 1, \dots, .2$

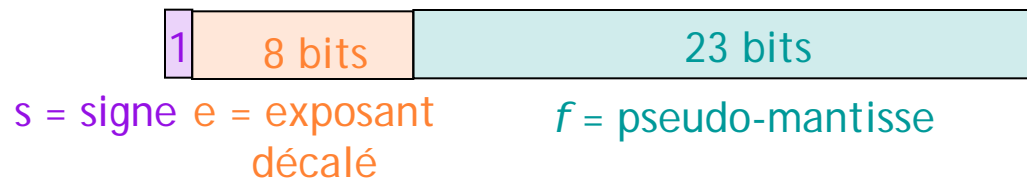


- Avantage de la notation en virgule flottante : grande plage de valeurs (ici, de -10^{38} à $+10^{38}$ avec seulement 32 bits)
- Inconvénients :
 - tous les nombres de la plage ne sont pas représentables exactement
 - les valeurs codables exactement ne sont pas régulièrement espacées
- Nombre de bits alloués à l'exposant → amplitude de la plage
- Nombre de bits alloués à la mantisse → précision du codage

Codage des réels : le standard utilisé par les ordinateurs

- Principe mantisse-exposant avec des puissances de 2
- Standard IEEE 754 simple précision (4 octets, type « float » en C) :

$(-1)^s \cdot (1, f) \cdot 2^{(e-127)}$ avec f = partie fractionnaire de la mantisse



- Standard IEEE 754 double précision (8 octets, type « double » en C) :

$(-1)^s \cdot (1, f) \cdot 2^{(e-1023)}$ avec f = partie fractionnaire de la mantisse



- e est codé sous forme de puissances positives de 2
- f est codé sous forme de puissances négatives de 2

Codage des réels : le standard utilisé par les ordinateurs

Norme IEEE 754	Simple précision	Double précision
Nombre de bits pour le signe	1	1
Nombre de bits pour l'exposant	8	11
Nombre de bits pour la pseudo-mantisse	23	52
Décalage exposant	127	1023
Plus petit nombre normalisé (valeur abs.)	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$
Plus grand nombre normalisé (valeur abs.)	$2^{128} \approx 10^{38}$	$2^{1024} \approx 10^{308}$

Codage des réels : le standard utilisé par les ordinateurs

	Signe	Exposant décalé	Pseudo-mantisse
Nombre normalisé	0 ou 1	toute configuration de bits, sauf « tous les bits à 0 » ou « tous les bits à 1 »	toute configuration de bits
Zéro	0 ou 1	tous les bits à 0	tous les bits à 0
Infini	0 ou 1	tous les bits à 1	tous les bits à 0
Not a Number (NaN)	0 ou 1	tous les bits à 1	toute configuration de bits non nulle

- Format spécial « Infini » : division par 0, $\infty \times \infty$, $\infty + \infty$
- Format spécial « Not a Number » :
 - $0/0$, ∞/∞ , $0 \times \infty$, $\infty - \infty$
 - racine carrée d'un nombre négatif
 - logarithme de 0 ou d'un nombre négatif
 - $\arcsin(x)$ ou $\arccos(x)$ d'un nombre non compris entre -1 et 1
 - etc.

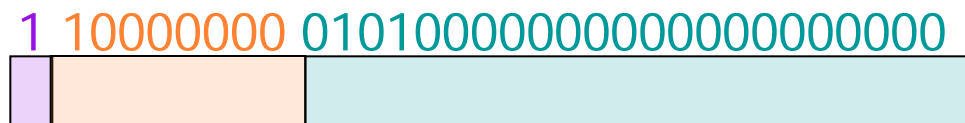
Codage des réels : exemple (sympathique) d'encodage

Quelle est la représentation IEEE 754 simple précision de -2,625 ?

- Nombre négatif, donc $s = 1$
- La valeur absolue peut s'exprimer comme somme de puissances positives et négatives de 2 :
 $2,625 = 2 + 0,5 + 0,125 = 1.2^1 + 0.2^0 + 1.2^{-1} + 0.2^{-2} + 1.2^{-3} = 10,101_2$
- Il faut normaliser pour que la mantisse s'écrive « 1, quelque chose »
Ici on décale la virgule d'un cran à gauche : $1,0101.2^1$
- Identification : $(1, f). 2^{(e-127)} = 1,0101.2^1$

=> mantisse = $1,010100000000000000000000_2$

=> $e - 127 = 1$ donc $e = 128_{10} = 10000000_2$





Codage des réels : exemple (moins sympathique) d'encodage

Quelle est la représentation IEEE 754 simple précision de $0,1_{10}$?

- Problème : $0,1$ s'exprime avec un nombre fini de décimales en base 10, mais pas en base 2 !

$$0,1_{10} = 0,0001100110011001100110011001100110011001100110011..._2$$

=> Avec un nombre fini de bits, l'ordinateur fait forcément une approximation !

- Normalisation : $0,00011001100110011..._2 = 1,1001100110011..._2 \cdot 2^{-4}$
- Identification : $(1 + f) \cdot 2^{(e-127)} = 1,1001100110011..._2 \cdot 2^4$

$$\Rightarrow f = 0,10011001100110011001101_2 \text{ (on arrondit au 23ème bit)}$$

$$\Rightarrow e - 127 = -4 \text{ donc } e = 123_{10} = 01111011_2$$



Codage des réels : exemple (moins sympathique) d'encodage

Quelle est la représentation IEEE 754 simple précision de $0,1_{10}$?

0	01111011	10011001100110011001101
---	----------	-------------------------

Décodage :

$$s = 0$$

$$e = 2^0 + 2^1 + 2^3 + 2^4 + 2^5 + 2^6 = 1 + 2 + 8 + 16 + 32 + 64 = 123$$

$$\begin{aligned} f &= 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} + 2^{-23} \\ &= 0,5 + 0,0625 + \dots + 0,000000119209289550781 \\ &= 0,599999904632568 \end{aligned}$$

$$\begin{aligned} \text{Nombre} &= (-1)^0 \cdot (1 + 0,599999904632568) \cdot 2^{(123-127)} \\ &= 1 \cdot 1,599999904632568 \cdot 2^{-4} \\ &= 0,100000001490116 \quad !!! \end{aligned}$$

L'ordinateur NE PEUT PAS manipuler exactement $0,1$!



Codage des réels : exemple (moins sympathique) d'encodage

L'ordinateur NE PEUT PAS manipuler exactement 0,1 !
(ni 0,2 ni 0,3 ...)

Conséquences :

- des comportements qui peuvent sembler étranges...
 - $0,1 + 0,2 == 0,3 \Rightarrow \text{false}$
 - $0,1 + 0,3 == 0,4 \Rightarrow \text{true}$
- des erreurs d'approximation qui peuvent s'accumuler
 - bug du système anti-missile Patriot



Codage des réels : opérations arithmétiques en virgule flottante

- Multiplication :
 - addition des exposants
 - multiplication des mantisses
 - renormalisation (si nécessaire) :
décaler la virgule pour que m soit
de la forme $1, \dots$ en ajustant
l'exposant
- Division :
 - soustraction des exposants
 - division des mantisses
 - renormalisation (si
nécessaire)

Attention

Risque d'overflow ou d'underflow si le résultat est en dehors de la plage de valeurs exprimables

Codage des réels : opérations arithmétiques en virgule flottante

- Addition (ou soustraction) :
 - dénormalisation de la plus petite valeur pour l'exprimer dans l'exposant de la plus grande
 - addition (ou soustraction) des mantisses
 - renormalisation si nécessaire

Attention

Risque d'overflow ou d'underflow

+ Risque d' « absorption » lors de l'addition

+ Risque de « cancellation catastrophique » lors de la soustraction

Codage des réels : phénomène d'absorption lors d'une addition

$$\begin{aligned} & (1,1000000000000000100010000.2^{-10}) \\ + & (1,0010000000000011000000000.2^{14}) \\ = & (0,00000000000000000000000000\textcolor{red}{11000000000000000000100010000}).2^{14} \\ + & (1,0010000000000011000000000.2^{14}) \\ = & 1,0010000000000011000000000.2^{14} \quad \text{pour l'ordinateur} \end{aligned}$$


Attention

Si les deux opérandes sont trop différents (en exposant), alors la plus petite valeur sera, au cours de la dénormalisation, confondue avec 0 et donc négligée dans le calcul...

=> si $a \gg b$, alors $a + b = a$ (phénomène d'absorption)

Codage des réels : phénomène de cancellation lors d'une soustraction

- Quand on soustrait deux nombres quasiment égaux :
 - annulation des digits de poids fort, qui sont identiques
 - seuls les digits de poids faible restent non nuls
- Opérandes exacts : cancellation bénigne
- Opérandes arrondis : cancellation catastrophique
 - exemple: calcul de $\sqrt{n+1} - \sqrt{n}$ avec n grand

$$\begin{array}{r} 1, \text{xxxxxxxxxxxxxxxxxxxxxxxxxx}111???? \cdot 2^y \\ - 1, \text{xxxxxxxxxxxxxxxxxxxxxxxxxx}001???? \cdot 2^y \\ \hline 0, \text{000000000000000000000000}110???? \cdot 2^y \\ 1, 10???????????????????????????? \cdot 2^{y-2} \end{array}$$


Lors de la renormalisation, l'incertitude « conquiert » les deux derniers digits stockés : la soustraction révèle les approximations précédentes

Codage des réels : bilan

- La notation en virgule flottante permet de représenter un intervalle très grand sur un nombre limité de bits

MAIS...

- Seuls les rationnels qui, sous forme réduite, ont une puissance de 2 pour dénominateur peuvent s'exprimer de façon exacte en binaire
- Les autres rationnels (0,1 par ex.) et les irrationnels (Π , $\sqrt{2}$, ...) sont manipulés sous forme approchée
- Les valeurs codables exactement ne sont pas régulièrement espacées entre elles
- Risques d'overflow et d'underflow
- Risque d'absorption lors de $a+b$, si $a \gg b$
- Risque de cancellation catastrophique lors de $a - b$, si $a \approx b$

Codage des réels : bilan

A retenir (bis !)

L'arithmétique des ordinateurs n'est PAS celle (idéale) des mathématiques, car les ordinateurs travaillent avec un nombre fini de valeurs

Codage des réels : précautions

- Utiliser les « double » plutôt que les « float »
- Méfiance envers la soustraction, surtout si les deux opérandes peuvent être proches
 - Si possible, réécrire l'expression sous une autre forme pour éviter cette situation
- Attention aux tests d'égalité entre flottants
 - Remplacer `if (x == y)` par `if(abs((x - y)/x) < epsilon)`
- Ne pas faire une confiance aveugle aux codes récupérés

Codage des booléens

- En principe, un seul bit suffirait : 0 = false, 1 = true
- En pratique, les booléens sont codés comme des entiers (donc sur au moins 8 bits)
 - 0 = false
 - toute valeur non nulle = true
- En C :
 - avant la norme C99, pas de type « bool », mais codage possible avec un entier, de préférence petit en mémoire (donc char : 1 octet)
 - depuis la norme C99, on peut utiliser un type bool (1 octet) à condition d'inclure le fichier stdbool.h

Codage des caractères

- A chaque caractère est associé un entier : son « code ASCII »

000 (nul)	016 (dle)	032 spc	048 0	064 @	080 P	096 `	112 p
001 (soh)	017 (dc1)	033 !	049 1	065 A	081 Q	097 a	113 q
002 (stx)	018 (dc2)	034 "	050 2	066 B	082 R	098 b	114 r
003 (etx)	019 (dc3)	035 #	051 3	067 C	083 S	099 c	115 s
004 (eot)	020 ¶(dc4)	036 \$	052 4	068 D	084 T	100 d	116 t
005 (enq)	021 §(nak)	037 %	053 5	069 E	085 U	101 e	117 u
006 (ack)	022 (syn)	038 &	054 6	070 F	086 V	102 f	118 v
007 (bel)	023 (etb)	039 '	055 7	071 G	087 W	103 g	119 w
008 (bs)	024 (can)	040 (056 8	072 H	088 X	104 h	120 x
009 (tab)	025 (em)	041)	057 9	073 I	089 Y	105 i	121 y
010 (lf)	026 (eof)	042 *	058 :	074 J	090 Z	106 j	122 z
011 (vt)	027 (esc)	043 +	059 ;	075 K	091 [107 k	123 {
012 (np)	028 (fs)	044 ,	060 <	076 L	092 \	108 l	124
013 (cr)	029 (gs)	045 -	061 =	077 M	093]	109 m	125 }
014 (so)	030 (rs)	046 .	062 >	078 N	094 ^	110 n	126 ~
015 ¢(si)	031 (us)	047 /	063 ?	079 O	095 _	111 o	127 DEL

Codage des caractères

- Un caractère est donc représenté en mémoire comme un petit entier (unsigned char en C)

- Exemple :

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	0	1	0	0	1	0	1	0

$$\begin{aligned}\text{Code} &= 2^1 + 2^3 + 2^6 \\ &= 2 + 8 + 64 \\ &= 74\end{aligned}$$

D'après la table ASCII, il s'agit du caractère 'J'

Codage des caractères

- Cette représentation permet de faire des opérations arithmétiques et des comparaisons sur des caractères (cf LIF1)

```
monChar : caractère  
monChar ← 't'  
monChar ← monChar + 1
```

```
afficher(« Le caractère est maintenant », monChar)
```

```
Si (monChar >= 'A' et monChar <= 'Z') Alors  
    afficher(« C'est une majuscule »)
```

```
Sinon  
    afficher(« Ce n'est pas une majuscule »)
```

```
Fin Si
```

Codage des caractères

- Cette représentation permet de faire des opérations arithmétiques et des comparaisons sur des caractères (cf LIF1)

```
unsigned char monChar;  
monChar = 't';  
monChar = monChar + 1;  
  
printf("Le caractère est maintenant %c \n", monChar);  
printf("Son code ASCII est %d \n", monChar);  
  
if ((monChar >= 'A') && (monChar <= 'Z'))  
{  
    printf("C'est une majuscule \n");  
}  
else  
{  
    printf("Ce n'est pas une majuscule \n");  
}
```



Les opérateurs en C

- 1, 2 ou 3 opérandes
- Considérés comme des expressions : renvoient une valeur
- Mais peuvent avoir des effets de bord (ex. affectation)



Les opérateurs en C

Affectation (élargie)	= ++ -- += -= *= /=	b=-3; b++ met -2 dans b et renvoie -2 a=7; a/=3 équivaut à a=a/3 : on met 2 dans a et on renvoie 2
Arithmétiques	+ - * / % (modulo=reste de la div. entière)	7/3 vaut 2; 7.0/3 vaut ~2.3333 7%3 vaut 1
Comparaison	< <= > >= == !=	6.5 > -2.2 vaut 1 0.33333 == 0.33334 vaut 0
Logiques	! NON logique && ET logique OU logique	!1 vaut 0, !0 vaut 1 1 && 1 vaut 1, 0 && 1 vaut 0 1 1 vaut 1, 0 1 vaut 1

Les opérateurs en C

Manipulation bit à bit (sur des entiers non signés uniquement)	~	NON bit à bit (complément à 1)	~295 vaut 4 294 967 000
	&	ET bit à bit	6 & 3 vaut 2
		OU bit à bit	6 3 vaut 7
	^	OU EXCLUSIF bit à bit	6 ^ 3 vaut 5
	<<	décalage de n crans vers la gauche, perte des bits de poids fort, bits entrés à droite sont à 0	6 << 3 vaut 48
	>>	décalage de n crans vers la droite, perte des bits de poids faible, les bits entrés à gauche sont à 0	6 >> 3 vaut 0

Usages : compression de données, commande de registres matériels, générateurs de nombres aléatoires...

Exemple : pour positionner un bit à 1, on combine la valeur avec un masque grâce à l'opérateur OU.

```
/* mettre à 1 le bit 4 de a : */  
unsigned int a = 0x000F; /* 0000 0000 0000 1111 */  
unsigned int b = 1 << 4; /* 0000 0000 0001 0000 */  
unsigned int c = a | b; /* 0000 0000 0001 1111 */
```

Les opérateurs en C

Référence	<p>[] accès à un élément d'un tableau</p> <p>* accès à la valeur se trouvant à une adresse mémoire donnée (déréférencement)</p> <p>& adresse mémoire d'une variable</p> <p>. accès à un champ d'une struct</p> <p>-> accès à un champ de la struct se trouvant à une adresse donnée</p>	<p>tab[5] vaut 8.56</p> <p>*(0xbfff1a65) vaut -13.9</p> <p>&mavar vaut 0xbfff1a65</p> <p>etu.nom vaut "Dupont"</p> <p>p = &etu; p->nom vaut "Dupont"</p>
Divers	<p>?: opérateur conditionnel (si alors sinon compact, à éviter car peu lisible)</p> <p>, calculs successifs ds une expression, prend comme valeur la dernière calculée</p> <p>sizeof taille en octets d'un type ou d'une variable</p>	<p>max = (a>b) ? a : b</p> <p>(b=5, i++, 3*b) vaut 15</p> <p>sizeof(float) vaut 4</p>



Conversions implicites mises en place par le compilateur

- Opérateurs arithmétiques :
 - opérande de type char ou short → promotion numérique en int
 - opérandes de types différents → conversion vers le type commun le plus large : $\text{int} < \text{long} < \text{float} < \text{double} < \text{long double}$
 - attention : $\text{int} + \text{unsigned int}$ → conversion vers unsigned int !!
- Opérateur d'affectation :
 - conversion dans le type de l'opérande de gauche
 - possible perte de précision

Conversions implicites mises en place par le compilateur

- Exemple :

```
char monChar = 24;  
double monDouble = 2.25;  
float monFloat;  
  
monFloat = monDouble + monChar;
```

double + char

↓ promotion de monChar en int

double + int

↓ ajustement vers le type le plus large

double + double

↓ calcul de la somme

double

↓ conversion du double en float pour l'affectation

float

Conversions explicites (cast)

- Utile notamment pour réaliser la division non entière de deux entiers
- Exemple en C :

```
int i = 5;  
int j = 3;  
double monDouble;  
  
monDouble = i / (double) j;
```