

Architecture et Système

Stefan Schwoon

Cours L3, 2023/2024, ENS Cachan

Entiers

Les entiers (tels qu'on les utilise dans les langages de programmation habituels) sont stockés dans un mot de taille fixe (typiquement 8, 16, 32 bits).

Types en C: `char`, `short int`, `int`, `long int`, `long long int`

En C, les tailles de ces types ne sont pas précisément définies par le langage de C, seulement leurs tailles minimales : 8, 16, 16, 32, 32, où `int` est un mot de registre.

On peut obtenir les valeurs concrètes avec `sizeof(char)` etc.

Big vs little endian

Un mot de taille > 8 s'étend sur plusieurs octets, p.ex. avec 32 bits:

12	34	56	78
----	----	----	----

Sous l'interprétation **big-endian**, ceci représente $(12345678)_{16}$ (les bits les plus significatifs sont dans le premier octet).

Little-endian: c'est l'inverse, on interprète cela comme $(78563412)_{16}$.

Le mode d'interprétation devient important lors des **échanges des données binaires** (fichiers, réseau). P.ex., l'Internet protocol (IP) définit cet ordre comme big-endian.

Fonctions en C : `ntohl`, `ntohs`, `htonl`, `htons`

Entiers avec/sans signe

Un mot de n bits peut représenter 2^n valeurs différentes. Les opérations arithmétiques travaillent implicitement modulo 2^n .

Deux interprétations :

`unsigned` (sans signe), le domaine est de 0 à $2^n - 1$.

`signed` (avec signe) / complément à deux : le domaine est de -2^{n-1} à $2^{n-1} - 1$

Pour les valeurs non-négatives, le bit le plus significatif (MSB) est de 0.

Pour les valeurs négatives, le MSB est de 1.

On obtient la représentation de $-i$ en prenant la négation (bit par bit) de i , puis en rajoutant 1. P.ex., $-1 \cong 11 \dots 1$ et $-2^{n-1} \cong 10 \dots 0$.

En C

Les types entiers peuvent être déclaré comme `signed` ou `unsigned` ;
par défaut ils sont `signed`.

Pour des `char` cette distinction n'est pas important lorsqu'on les interprète
comme des caractères.

Attention aux opérateurs de décalage (`<<` et `>>`) :

pour les `unsigned`, l'opération est dite *logique*, le décalage se fait sur
l'intégralité du mot, en inclus le bit le plus significatif;

pour les `signed`, l'opération est dite *arithmétique*, elle conserve le signe.

Valeurs réelles

Les valeurs réelles sont typiquement représentées dans un format **virgule flottante**, dans un mot de taille fixe, avec une précision limitée.

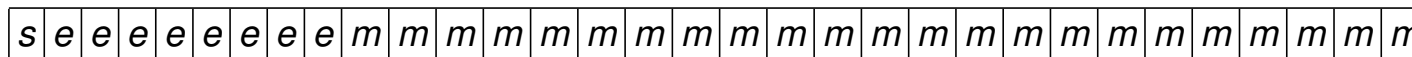
Idée en général : tuple $\langle s, m, e \rangle$ avec l'interprétation $\pm 2^e \cdot m$.

s est le **signe** (un seul bit, 0 non-négatif, 1 négatif);

m est la **mantisse**;

e est l'**exposant**.

→ compromis entre taille et précision des valeurs représentables.



Besoin des standards

Problèmes:

Comment répartir les bits entre taille et mantisse ?

Représentations non uniques : $\langle s, m, e \rangle \equiv \langle s, 2m, e - 1 \rangle$

Comment traiter des cas spéciaux (division par zéro), comment traiter les arrondis ?

Le standard le plus important pour régler ces questions s'appelle **IEEE 754**.

En C : `float` en C = IEEE 754, 32 bit; `double` = IEEE 754 (64-bit).

Dans le suivant, on discutera la partie 32 bit, les autres parties étant similaires.

IEEE 754 (variante 32-bit)

IEEE 754 spécifie les conventions suivantes :

Signe : 1 bit; 0 pour positif, 1 pour négatif

Exposant : 8 bits, interprété sans signe, mais décalé de 127. P.ex., $(10000001)_2 = 129$, mais l'exposant effectif est de 2. Si tous les bits de l'exposant sont 1, voir ci-dessous.

Mantisse : 23 bits, interprété comme $1 + (m/2^{23})$, ce qui donne un résultat dans $[1, 2)$.

Remarques :

Cette interprétation de la mantisse garantit une représentation unique.

Cas spéciaux, si exposant est 1111 1111 : $\pm\infty$ (avec $m = 0$)
ou NaN (not a number, avec $m \neq 0$)

Addition dans les flottantes

Le standard IEEE définit la procédure à suivre pour effectuer des opérations arithmétiques (comment arrondir, comment traiter des cas spéciaux, ...).

Discutons le cas d'une addition:

Soient $x = 2^{e_x} \cdot m_x$ et $y = 2^{e_y} \cdot m_y$ et $x > y$

(par simplicité on suppose qu'ils sont tous les deux positifs).

P.ex. $x = 2.5$ et $y = 0.75$, du coup,

$e_x = 1, m_x = 1.25, e_y = -1, m_y = 1.5$.

Représentation binaire :

0	1000 0000	0100 0000 0...0
0	0111 1110	1000 0000 0...0

On isole désormais la mantisse, mais en prenant compte de la “une cachée” :

1	0100	0000	0...	0
1	1000	0000	0...	0

L'exposant d'y est moins grand que celui de x de 2.
Du coup, on décale la mantisse d'y de 2 positions.

1	0100	0000	0...	0	
0	0	110	0000	0...	0

L'addition des deux mantisses donne désormais :

1	1010	0000	0...	0
---	------	------	------	---

Comme le résultat n'excède pas les 24 bits (une cachée plus 23 bits), on garde l'exposant de x et on enlève simplement l'une cachée de i_z pour obtenir la mantisse du résultat.

0	1000 0000	1010 0000 0...0
---	-----------	-----------------

Si l'addition des mantisses avait débordé les 24 bits, il aurait d'abord fallu décaler i_z à droit par une position (perdant un bit de précision).

Problèmes de virgule flottante

Précision limitée : certaines valeurs “rondes” comme 0.1 ou 2.3 ne sont pas représentables.

Erreurs d'arrondi; p.ex. $x + y$ donne x si x beaucoup plus grand qu' y .

Du coup, certaines lois comme distributivité ne sont plus valables.