Chapitre précédent Chapitre suivant

Représentation des nombres en machine

La représentation des nombres en machine repose bien évidemment sur la représentation en base deux que nous venons d'étudier mais doit prendre en compte un certain nombre d'autres éléments :

- la finitude : la représentation d'un nombre utilise un nombre fini donné d'octets et donc de bits, que ce soit pour les nombres entiers ou les nombres non entiers
- la nécessité de définir une représentation des nombres négatifs qui permettent leur manipulation et en particulier la réalisation des opérations arithmétiques usuelles

Les types de base d'un langage

Nous prendrons ici l'exemple du langage Java (d'autres langages comme le langage C proposent des types de base supplémentaires mais les idées sont les mêmes).
Il propose un certain nombre de types de base :

- :
  `byte` `short` `int` `long`
- :
  `float` `double`

De manière générale, définir le type d'une variable permet :

- d'en fixer la taille, c'est-à-dire le nombre d'octets (et donc de bits) qui lui sont alloués
- d'en fixer l'interprétation, c'est-à-dire la manière dont la suite des bits doit être interprétée
- de définir les opérations possibles et la sémantique de ces opérations

Les nombres entiers

→ *Principes généraux*

La représentation des nombres entiers repose sur la représentation binaire des entiers.
Ainsi, si une variable destinée à recevoir des valeurs entières correspond en mémoire à une zone de octets (soit $8\times$ bits), la variable est a priori susceptible de prendre $2^{8\times}$ valeurs correspondant aux entiers représentables en base deux avec $8\times$ chiffres binaires.

Par exemple, si on considère un ordinateur utilisant 16 bits pour coder un nombre entier (typiquement le type short de Java), il existe 2^{16} configurations différentes pour ces 16 bits : vues comme les représentations binaires de nombres, elles correspondent à tous les nombres entiers compris entre `0` (suite `0000 0000 0000 0000` de seize bits tous égaux à `0`) et $2^{16}-1$, c'est-à-dire $(65\ 535)_{dix}$ (suite `1111 1111 1111 1111` de seize bits tous égaux à `1`)

→ Cependant, à moins de considérer que les variables ne prennent que des valeurs positives (ce que permet par exemple le langage C en qualifiant les variables de `unsigned`), les choses se compliquent un peu si on considère que les variables entières sont susceptibles de prendre également des valeurs négatives, et cela de manière a priori équiprobable.

L'espace de valeurs possibles doit ainsi être divisé en deux parties égales et donc, pour un codage des entiers

sur bits, il sera possible de coder :

- 2^{-1} valeurs positives
- 2^{-1} valeurs négatives

Entrons dans les détails :

- **Pour l'espace des valeurs positives** représentables, l'intervalle contient de manière naturelle les 2^{-1} premiers entiers,
  c'est-à-dire correspond à l'intervalle $[0 : 2^{n-1} - 1]$.
  L'ensemble de ces valeurs correspond à l'ensemble des suites de bits de longueur dont le premier bit est 0, d'où l'idée naturelle de représenter les valeurs positives par des suites de bits commençant par un bit de valeur 0.
- Il en découle que **l'espace des valeurs négatives** est associé à l'ensemble des séquences de bits commençant par un bit de valeur 1.

  Ce bit dont la valeur détermine si le nombre représenté est positif ou négatif est appelé **bit de signe**.

  Le problème se pose alors de choisir le type de codage des entiers négatifs. Il doit satisfaire un certain nombre de propriétés :

  - simplicité du calcul de l'opposé d'un nombre
  - réalisation simple des opérations arithmétiques, indépendamment du signe des opérandes

  On peut imaginer différentes représentations :

  - la première qui vient à l'esprit, la valeur absolue signée, consiste à utiliser la valeur 1 du premier bit pour coder le signe - et les autres bits pour coder la valeur absolue.
    Cela a le mérite de la simplicité mais présente deux inconvénients :
    - il y a deux représentations de zéro (un zéro positif et un zéro négatif)
    - elle ne permet pas l'addition simple de nombres de signes opposés (le vérifier par exemple avec l'addition de 13 et -6 représentés sur 8 bits)

  - la deuxième solution qui a été utilisée sur certaines machines est appelée **complément à 1**. Elle consiste, pour représenter un nombre négatif, à inverser les bits de la représentation de sa valeur absolue.

    Ses avantages et inconvénients sont :

    - le passage à l'opposé est simple
    - l'existence de deux codes différents pour le nombre zéro
    - une mise en œuvre difficile de l'addition
    - une gestion des dépassements de capacité assez compliquée

  - **la solution adoptée** consiste à représenter un entier relatif négatif $-$ sur bits en **complément à deux**, c'est-à-dire par 2^n

    Ainsi la représentation sur 8 bits de -75 sera ceci 

L'intervalle des valeurs négatives représentables sur bits est : $[-2^{n-1} : -1]$.

Question : pour une représentation sur 8 bits, quelles sont les valeurs décimales des entiers représentés respectivement par `11111111` et `10000000` ?

→ ***Comment calculer simplement le complément à deux***

Soit à représenter un nombre négatif.

- une première méthode consiste en l'application directe de la définition du complément à deux.
  Le calcul peut se faire, soit en passant par la base dix ou directement en base deux (ou dans l'une des bases huit ou seize) comme nous l'avons fait sur un exemple.

  Question : calculer la représentation de `-3118` pour une représentation sur 16 bits en utilisant la définition et des calculs en base dix, deux, huit et seize

- la deuxième méthode consiste à :
  - représenter son opposé (un nombre positif) en base deux sur `-1` bits
  - complémenter chaque bit (on inverse chaque bit, c'est-à-dire que l'on remplace les `0` par des `1` et vice-versa)
  - ajouter `1` à la valeur obtenue

  Ainsi, pour coder `-75` sur 8 bits
  - on écrit `75` sur 8 bits : `01001011`
  - on complémente chacun des 8 bits : `10110100`
  - on ajoute 1 : `10110101`

  La représentation sur 8 bits de `-75` est donc `10110101`.

  Question : déterminer par cette méthode la représentation sur 16 bits de `-3118`

- la troisième méthode consiste à
  - rechercher le bit `1` le plus à droite dans la représentation de la valeur absolue du nombre
  - inverser tous les bits à la gauche du précédent (tous les bits à sa droite et lui-même sont inchangés)

  Dans les exemples suivants, les bits de la représentation binaire de la valeur absolue à inverser sont en bleu et ceux qui sont inchangés dans le passage à l'opposé sont en rouge :

  - pour le nombre `-75` à coder sur 8 bits, `01001011` → `10110101`
  - pour le nombre `-103` à coder sur 8 bits, `01100111` → `10011001`
  - pour le nombre `-100` à coder sur 8 bits, `01100100` → `10011100`

  Question : déterminer par cette méthode la représentation sur 16 bits de `-3118`

  Question : quelle rel

opérations arithmétiques.
Ceci explique que tous les processeurs actuels utilisent ce code pour effectuer des calculs en arithmétique signée, et disposent de fonctions spéciales pour l'implémenter (cf. indicateur de dépassement).

→ Le tableau suivant donne la taille (nombres d'octets) de la zone allouée en mémoire à une variable d'un type de base du langage Java :

nom du type	byte	short	int	long
taille exprimée en nombres d'octets [de bits]	**1 [8]**	**2 [16]**	**4 [32]**	**8 [64]**
plus petite valeur	-2^7 -128	-2^{15} -32768	-2^{31} -2147483648	-9223372036854775808 -2^{63}
plus grande valeur	2^7-1 127	$2^{15}-1$ 32767	$2^{31}-1$ 2147483647	$2^{63}-1$ 9223372036854775807

Question : comment passe-t-on de la représentation sur 16 bits d'un entier à sa représentation sur 32 bits ?

Question : vous souhaitez affecter à une variable *n* de type `int` d'un programme JAVA la valeur de l'entier ayant comme représentation binaire sur 32 bits `01100001111111000010101001111101` sans en calculer la valeur décimale.
Comment allez-vous procéder ?

→ ***L'arithmétique***

Commençons par les additions :

- **deux nombres de même signe dont la somme est représentable avec le nombre de bits dont on dispose (ici 16)**
  - ***deux nombres positifs***

```
  --111111111----1
  0001101010111001      6841
+ 0000011101010001    + 1873
------------------    -------
= 0010001000001010    = 8714
```

    On a ainsi réalisé l'opération `6841 + 1873`

  - ***deux nombres négatifs***

```
  1110010101000111      -6841
+ 1111100010101111    + -1873
------------------    -------
=11101110111110110    = -8714
```

    Le bit de gauche est ignoré (il est néanmoins conservé sous forme d'un indicateur) et le complément à 2 du résultat est `0010001000001010` (représentation binaire de `8714`).
    On a donc réalisé effectivement l'opération `-6841 + -1873`

- **deux nombres de signe différent**
  Ici encore nous nous placerons dans l'hypothèse qu'on dispose de 16 bits pour représenter les entiers.

- ***la somme des représentations d'un nombre et de son opposé produit directement `0`***
- ***nombre positif supérieur à la valeur absolue du nombre négatif dont la somme est donc positive***

```
  11-11--11-------
  1110010101000111      -6841
+ 0110110011000000    + 27840
------------------    --------
=10101001000000111    = 20999
```

  Un 17-ème bit apparaît à gauche (comme retenue) qui est purement et simplement ignoré. Les 16 bits restant à droite, à savoir `0101001000000111` correspondent à la représentation binaire sur 16 bits du résultat (`20999`)de l'opération `-6841 + 27840`

- ***nombre positif inférieur à la valeur absolue du nombre négatif dont la somme est donc négative***

```
  --1--1----------
  0001101010111001       6841
+ 1001001101000000   + -27840
------------------   ---------
= 1010110111111001   = -20999
```

  Le résultat est le complément de `0101001000000111` c'est-à-dire `-20999`

- **deux nombres de même signe dont la somme n'est pas représentable avec le nombre de bits dont on dispose (ici 16)**
  Considérons les deux nombres `18931` et `23113` : chacun d'eux est inférieur à `32767` et est donc affectable à des variables de type `short`.
  Cependant leur somme (`42044`) est supérieur à `32767` et n'est donc pas affectable à une variable de type `short`.

  Examinons ce qui se passe à l'exécution de la séquence Java suivante et expliquons les résultats :

```
short m = 18931;     // 0100100111110011
short n = 23113;     // 0101101001001001
Deug.println(m + n);
short p = (short) (m + n);
Deug.println(p);
```

  qui produit les résultats suivants :

```
42044
-23492
```

  Lorsqu'on écrit `m + n`, l'opération est réalisée sur 32 bits (l'arithmétique est faite dans le type `int`). Chacun des deux opérandes est étendu sur 32 bits, par propagation du bit de signe (ici `0`). Puis l'addition est réalisée :

```
 00000000000000000100100111110011
+00000000000000000101101001001001
--------------------------------
=00000000000000001010010000111100
```

  Le résultat affiché par le premier `println` affiche la valeur correcte de la somme (entier naturel codable sur 31 bits).

La ligne suivante (`short p = (short) (m + n);`) est une demande d'affecter une grandeur codée sur 32 bits dans un espace de seulement 16 bits, ce que n'autorise pas le langage.
Si on avait écrit :

```
short p = m + n;
```

la compilation aurait produit une erreur

```
...... : possible loss of precision found : int required: short
```

D'où la nécessité d'utiliser l'opération de coercition ( ) qui force le changement de type.

Ce faisant on ne conserve que les seize bits de poids faible (de fait cela revient à réaliser une addition de nombres entiers sur 16 bits) :

```
 0100100111110011
+0101101001001001
-----------------
=1010010000111100
```

Le bit de signe obtenu pour la somme vaut `1` et la valeur de la somme est donc interprétée comme une valeur négative. Le nombre est l'opposé du nombre positif ayant comme représentation `0101101111000100`. Il s'agit de la représentation en machine de `-23492` sur 16 bits.

Remarque : Si dans un programme on écrit

```
short p = 17 + 38;
```

il n'y a pas d'erreur à la compilation : le compilateur sait que le résultat est `55` et que ce nombre est codable dans un `short`.

Par contre, si on écrit

```
short p = 12000 + 26000;
```

il y a une erreur à la compilation (de même nature que pour l'instruction `p = m + n;`) : le compilateur sait que le résultat est `38000` et que ce nombre n'est pas codable dans un `short`.

Question : que se passe-t-il pour la séquence Java utilisant des nombres négatifs :

```
short m = -18931;
short n = -23113;
Deug.println(m + n);
short p = (short) (m + n);
Deug.println(p);
```

Question : on suppose les déclarations suivantes de variables Java :

```
byte b1; // un octet
short s1; // deux octets
int i1; // 4 octets
long l1; // 8 octets
```

Vous voulez affecter

- **à b1 la plus grande valeur positive possible**

- **à s1 la plus grande valeur positive possible**
- **à i1 la plus grande valeur positive possible**
- **à l1 la plus grande valeur positive possible**

- Vous ne voulez pas calculer les représentations décimales de ces valeurs. Comment pouvez-vous procéder ?

- Vous incrémentez ensuite les valeurs de ces variables par les instructions :

```
b1++; s1++; i1++; l1++;
```

puis demandez l'impression de leur valeur. Qu'obtenez vous ?

→ ***Un autre problème : l'ordre de lecture des octets***

Dans tout ce que nous avons fait, nous avons supposé que l'octet de poids fort était à gauche, et donc que les nombres se lisaient en mémoire par adresse croissantes comme nous en avons l'habitude. Du point de vue des adresses des octets, nous avons donc fait l'hypothèse que les octets de poids fort étaient en mémoire avant les octets de poids faible.

Ce n'est malheureusement pas toujours le cas !

On distingue :

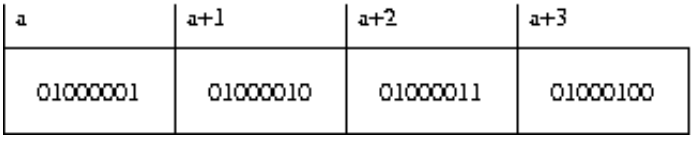
- la représentation «par le grand bout» (***big endian***) : les octets sont enregistrés en mémoire en commençant (c'est-à-dire à l'adresse la plus petite) par l'octet de plus fort poids (**M S B**). L'octet de poids fort est donc stocké à une adresse plus petite que l'octet de poids faible. C'est le cas pour les processeurs Motorola ou Sparc (de Sun) par exemple.
- la représentation «par le petit bout» (***little endian***) : les octets sont enregistrés en mémoire en commençant (c'est-à-dire à l'adresse la plus petite) par l'octet de plus faible poids (**L S B**). L'octet de poids fort est donc stocké à une adresse plus grande que l'octet de poids faible. C'est le cas pour les processeurs Intel.

  Ainsi, pour l'initialisation suivante de la variable :

  ```
  int n = 65*256*256*256 + 66*256*256 + 67*256 + 68;
  ```

  si les octets associés à la variable ont comme adresses , *+1*, *+2*, et *+3*, on aura

  - sur une machine

    on place d'abord en mémoire l'octet de poids fort (contenu de valeur 65 sous forme décimale) et on finit avec l'octet de poids faible (contenu de valeur 68 sous forme décimale)

  - sur une machine

    on place d'abord en mémoire l'octet de poids fable (contenu de

a	a+1	a+2	a+3
01000100	01000011	01000010	01000001

valeur 68 sous forme décimale) et on finit avec l'octet de poids fort (contenu de valeur 65 sous forme décimale)

→ ***Les indicateurs : retenue, dépassement de capacité***

Les nombres non entiers

→ Le tableau suivant donne la taille (nombres d'octets) de la zone allouée en mémoire à une variable d'un type de base du langage :

nom du type	`float`	`double`
taille exprimée en nombres d'octets [de bits]	**4 [32]**	**8 [64]**

→ Une représentation normalisée est proposée (IEEE-754), simple précision sur 32 bits (utilisée pour le type `float`) et double précision pour 64 bits (utilisée pour le type `double`) .

Pour chacune, la représentation comporte 3 champs :

- un bit de signe : sa valeur est `0` pour un nombre positif et `1` pour un nombre négatif
- un champ correspondant à un exposant
- un champ correspondant à une mantisse

Le tableau suivant donne pour les deux types, la taille de ces différents champs :

nom du type	exposant	excès pour l'exposant	mantisse
`float`	8 bits	`127`	23 bits
`double`	11 bits	`1023`	52 bits

A partir de la représentation en base deux du nombre, la mantisse est normalisée pour obtenir une valeur binaire dans l'intervalle ouvert `[1.0 : 10.0[`.

Cela revient à écrire le nombre sous une forme telle qu'il n'y ait qu'un seul chiffre dans la partie entière, un chiffre `1`.

→ Prenons comme exemple, le nombre `13.2890625`

Ce nombre est finiment représentable en base deux (en effet sa partie décimale `,2890625` est égale à `1/4+1/32+1/128`).
Il s'écrit sous forme binaire en «virgule fixe»:

```
1101.0100101
```

On place le point (la virgule) à la droite du `1` le plus à gauche. Cela donne une valeur qu'il faut corriger pour retrouver le nombre de départ en le mutipliant par deux à la puissance trois :

mantisse : `1.1010100101`

exposa[illegible]:11

Cela tradu[illegible]e fa[illegible]n bas[illegible]eux) :

1[illegible]1.va00101[illegible] 10^{11}

La rep[illegible]entation de c[illegible]

- la vam[illegible] .. [illegible]s -

Le 24-ème bit étant égal à `1`, il y a un arrondi.

Ce qui sera mémorisé dans la mantisse est alors : `00110011001100110011010`

La valeur mémorisée dans le champ exposant est : `01111111 + (-10) = 01111101`

La représentation sur 32 bits est donc : `00111111010011001100110011001101`0

- **en ce qui concerne** `0.2`, sans entrer dans les détails, sa représentation en tant que `float` est :

  `00111110010011001100110011001101`

- **enfin pour** `0.1` :

  `00111101110011001100110011001101`

  La représentation de 0.3f - 0.2f exprimée sous forme hexadécimale est **`3DCCCCCE`** alors que celle de **`0.2f - 0.1f`** est **`3DCCCCCD`**

  La valeur du test d'égalité des deux expressions est donc inévitablement `false`

- deuxième exemple : les entiers exprimés sous forme `float`

  Question : soit la séquence suivante :

```
int n = 0x3456789A; // valeur décimale : 878082202
float f = n;
int m = (int)f;
Deug.println(n);
Deug.println(f);
Deug.println(m);
```

  Justifier les résultats obtenus :

```
878082202
8.7808218E8
878082176
```

→ ***Les valeurs spéciales***

- le zéro : il est représenté par une valeur nulle des champs exposant et mantisse :

valeur	signe	exposant	mantisse
-0	1	00000000	00000000000000000000000
+0	0	00000000	00000000000000000000000

- Les infinis :

valeur	signe	exposant	mantisse
-∞	1	11111111	00000000000000000000000
+∞	0	11111111	00000000000000000000000

- `NaN` (- -) : un certain nombre de configurations ne correspondent pas à un nombre. Ainsi,

les valeurs `00000000` et `11111111` du champ exposant sont réservées respectivement au nombre `0` et aux infinis et supposent donc un champ mantisse nul. Par ailleurs, certaines opérations sont interdites (par exemple une division par zéro).

Un exemple de configuration interdite est `0 11111111 00001001001111000000100`

→ *Quelques précautions à prendre avec les types `float` et `double`*

Toutes ces précautions résultent du fait que la plupart des nombres ne sont pas représentables finiment en machine : nous l'avons vu avec par exemple `0,1`, c'est le cas de 1/3, ...

Donc, lorsqu'un nombre est stocké en mémoire, c'est une valeur approchée qui est utilisée. Cette mémorisation donne lieu à un arrondi. Ce mécanisme se solde par une perte de précision. Les opérations successives donnent elle-même lieu à de nouveaux arrondis et ainsi une suite de calculs est susceptible de conduire au bout du compte à des erreurs importantes.
Le contrôle de ces erreurs et leur évaluation est un domaine d'études en lui-même.

Citons des sources d'erreur importantes :

- lors de la soustraction de nombres très proches, des `0` peuvent apparaître à droite lors de la normlisation et le nombre de chiffres significatifs tendre vers zéro
- lors de l'addition de deux nombres ayant des ordres de grandeur très différents, on peut «perdre» toute l'information relative au plus petit des deux nombres

L ' : - ....
!

Faites par exemple le test en machine d'atteinte de la valeur entière `20` par incrémentations successives de `0.1` d'une variable initialisée à `0`.

→ *En guise de conclusion : pourquoi la fusée ARIANE 5 a explosé*

Le 4 juin 1996, lors de son premier vol, la fusée européenne Ariane 5 explose 30 secondes après son décollage causant la perte de la fusée et de son chargement estimé à 500 M$. Le système de guidage de la fusée s'est arrêté à la suite de l'arrêt des deux unités flottantes qui contrôlaient son programme.
Après deux semaines d'enquête, un problème a été isolé dans le système de référence inertiel : la vitesse horizontale de la fusée par rapport au sol était calculée avec des flottants sur 64 bits et dans le programme du calculateur de bord, il y avait une conversion de cette valeur flottante 64 bits vers un entier signé 16 bits.
Rien n'était fait pour tester que cette conversion était bien possible mathematiquement (sans dépassement de capacité).
Ces tests avaient été faits pour Ariane 4 qui était moins puissante qu'Ariane 5 et avait une vitesse horizontale suffisamment faible pour tenir sur un entier 16 bits, ce qui n'était malheureusement plus le cas pour Ariane 5.

Chapitre précédent Chapitre suivant