Chapitre 3

Représentation des données

Représentation des entiers

« Pour tout comprendre, lire ce chapitre en entier. »

Nous avons vu au chapitre précédent comment écrire les entiers naturels en binaire ou en hexadécimal. Maintenant nous allons étudier comment les entiers *relatifs*, c'est-à-dire positifs ou négatifs (on dit aussi *signés* en Informatique) sont représentés en machine.
On va d'abord se limiter aux entiers naturels et on va voir qu'il n'y a pas de difficulté majeure à comprendre leur représentation en machine.

Remarque importante

Il existe des *centaines* de langages de programmation. Les principaux sont : C, C++, C#, JAVA, PYTHON, PHP et JAVASCRIPT (cette liste est non exhaustive). Chaque langage utilise ses propres *types de variable* mais en général il y a beaucoup de ressemblances. On travaillera donc sur des exemples de types de variables utilisés dans tel ou tel langage...sachant que ce type n'existe pas nécessairement en PYTHON.

I Représentation des entiers naturels : l'exemple du unsigned char

En Informatique on dit souvent qu'un entier naturel est *non signé* (ou *unsigned* en Anglais). Le type *unsigned char* se rencontre en C et en C++ (entre autres).

Un *unsigned char* est stocké sur un octet, c'est à dire 8 bits :

- l'octet 0000 0000 représente l'entier 0 ;
- 0000 0001 représente 1 ;
- et ainsi de suite jusqu'au plus grand entier représentable sur un octet : $(1111\ 1111)_2 = 255$.

On peut donc représenter les 256 premiers entiers avec un *unsigned char* et c'est logique : un octet, c'est 8 bits, chaque bit peut prendre 2 valeurs et $2^8 = 256$.
Si on a besoin de représenter des entiers plus grands, on pourra utiliser l'*unsigned short* : c'est la même chose mais ce type est représenté sur 2 octets. Donc on peut représenter les 2^{16} premiers entiers naturels, c'est à dire les nombres compris entre 0 et 65 535 inclus.

Cela continue avec l'*unsigned int* (sur 4 octets) et l'*unsigned long* (8 octets).

Remarque

En PYTHON c'est différent : le type `int` (abréviation de *integer*, qui veut dire « entier » en Anglais) permet de représenter des entiers arbitrairement grands, les seules limitations étant la mémoire de la machine. Il n'y a qu'à évaluer 2^{100000} dans un *shell* PYTHON pour s'en convaincre.

II Représentation des entiers relatifs : l'exemple du type char

Le type *char* (qui n'existe pas en PYTHON) utilise un octet et l'on veut représenter des entiers relatifs (donc plus seulement positifs).

1 Une première idée... qui n'est pas si bonne

On pourrait décider que le bit de poids fort est un bit de signe : 0 pour les positifs et 1 pour les négatifs, par exemple. Les 7 autres bits serviraient à représenter la valeur absolue du nombre. Puisqu'avec 7 bits on peut aller jusqu'à $(111\ 1111)_2 = 127$, ce format permettrait de représenter tous les nombres entiers de -127 à 127.

Par exemple 1000 0011 représenterait -3 et 0001 1011 représenterait 27.

Il est un peu dommage que zéro ait 2 représentations : 0000 0000 et 1000 0000, mais ce qui est encore plus dommage c'est que lorsqu'on ajoute les représentations de -3 et de 27 voici ce qui se passe :

	1	0	0	0	0	0	1	1
+	0	0	0	1	1	0	1	1
=	1	0	0	1	1	1	1	0

On obtient 10001 1110, qui représente -30... On aurait bien sûr préféré que cela nous donne 24.

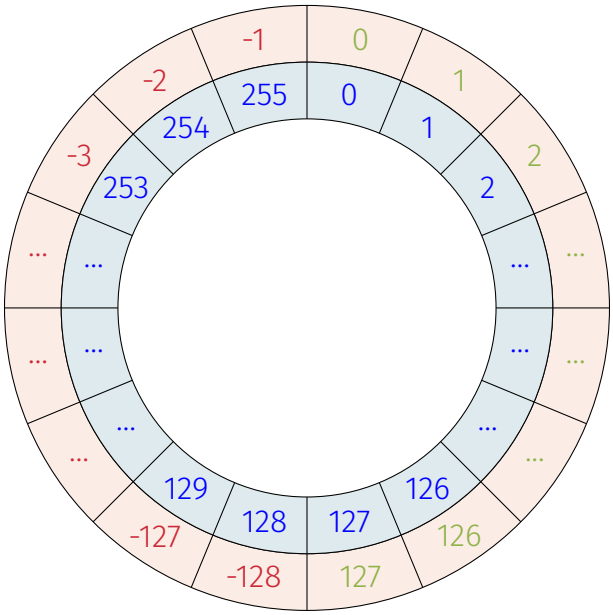
2 La bonne idée : le complément à deux

Ce format, qui est utilisé avec le type *char*, permet de représenter les entiers de -128 à 127 :

Propriété

Représentation en complément à 2 sur un octet :

- Soit x un entier positif plus petit ou égal à 127, alors on représente x par son écriture binaire (qui comprend 7 bits) et donc le bit de poids fort de l'octet (le 8^e) est égal à zéro.
- Sinon si x est un entier strictement négatif plus grand que -128, on le représente par l'écriture binaire de $256 + x$, qui est toujours représenté par un octet avec un bit de poids fort égal à 1.



Exemples

Comment représenter 97 ? Ce nombre est positif, on le représente par son écriture binaire sur 8 bits : 0110 0001

Comment représenter -100 ? Ce nombre est négatif, il est donc représenté en machine par 256-100=156, c'est-à-dire 1001 1100

Que représente 0000 1101 ? Le bit de poids fort est nul donc cela représente $(0000\ 1101)_2$, c'est à dire 13.

Que représente 1000 1110 ? Le bit de poids fort est non nul. $(1000\ 1110)_2 = 142$ représente x avec donc $256 + x = 142$, c'est-à-dire $x = -114$.

Méthode

Pour passer d'un nombre à son opposé en complément à 2, en binaire on procède de *la droite vers la gauche*

- On garde tous les zéros et le premier 1.
- On « inverse » tous les autres bits.

Exemple

Si on veut l'écriture en complément à 2 de -44 on commence par écrire 44 en base 2 :

$$44 = (0010\ 1100)_2$$

Puis on applique la méthode précédente :

$$\underbrace{00101}_{\text{on change}}\ \underbrace{100}_{\text{on garde}}$$

Ce qui nous donne

$$\underbrace{11010}_{\text{on a changé}}\ \underbrace{100}_{\text{on a gardé}}$$

Ainsi la représentation de -44 en complément à 2 sur 8 bits est 1101 0100.

Ce qui est agréable, c'est que **l'addition naturelle est compatible avec cette représentation** dans la mesure où :

- On ne dépasse pas la capacité : on n'ajoutera pas 100 et 120 car cela dépasse 127.
- Si on ajoute deux octets et que l'on a une retenue à la fin de l'addition (ce serait un 9^{e} bit), alors celle-ci n'est pas prise en compte.

Exemple

Ajoutons les représentations de 97 et -100 :

	0	1	1	0	0	0	0	1
+	1	0	0	1	1	1	0	0
=	1	1	1	1	1	1	0	1

On a $(1111\ 1101)_2 = 253$, il représente donc x sachant que $256 + x = 253$, c'est à dire x=-3.

On retrouve bien 97+(-100)=-3.

Attention à ne pas dépasser la capacité du format : en ajoutant 120 et 20 on obtient 140 qui, étant plus grand que 128, représente 140-156 = -116. C'est ce qu'affiche le programme suivant.

```cpp
#include <iostream> // nécessaire pour utiliser cout

int main() // début de la fonction main
{
    char c1 = 120; // on définit une première variable
    char c2 = 20; // puis une deuxième
    char c3 = c1+c2; // on les ajoute
    std::cout << (int) c3; // on affiche le résultat en base 10
    return 0; // la fonction main renvoie traditionnellement zéro
}
```

III Les principaux formats

En général, dans la majorité des langages (C, C++, C#, Java par exemple) les types suivants sont utilisés pour représenter les entiers relatifs (les noms peuvent varier d'un langage à l'autre) :

- `char` : Pour représenter les entiers compris entre -128 et +127. Nécessite un octet.
- `short` : Pour des entiers compris entre −32 768 et 37 267 (-2^{15} et $2^{15} - 1$). Codage sur 2 octets.
- `int` : (4 octets) entiers compris entre −2 147 483 648 et +2 147 483 647 (-2^{31} et $2^{31} - 1$).
- `long` : (8 octets) entiers compris entre −9 223 372 036 854 775 808 et +9 223 372 036 854 775 807 (-2^{63} et $2^{63} - 1$).

IV Quelques ordres de grandeur

- 1 kilooctet = 1 ko = 1 000 octets (fichiers textes)
- 1 mégaoctet = 1 Mo = 1 000 (fichiers .mp3)
- 1 gigaoctet = 1 Go = 1 000 (RAM des PC actuels, jeux)
- 1 téraoctet = 1 To = 1 000 (Disques durs actuels)
- 1 pétaoctet = 1 Po = 1 000 To
- 1 exaoctet = 1 Eo = 1 000 Po = 10^{18} octets (trafic internet mondial mensuel prévu en 2022 : 376Eo)

Exercices

Exercice 1

On considère le code C++ suivant :

```
#include <iostream> // bibliothèque d'affichage
int main() // début de la fonction principale
{
    unsigned char c = 0; // on définit la variable c
    for (int i = 0; i < 300; i++) // on fait une boucle pour
    {
        std::cout << "valeur de i : " << i;
        // on affiche la valeur de i
        std::cout << "  et valeur de c : " << (int)c;
         // on affiche la valeur de c en base 10
        std::cout << endl; // on revient à la ligne (END Line)
        c++; // on augmente c
    }
    return 0; // la fonction principale renvoie zéro
}
```

Voilà ce que la console affiche :

```
valeur de i : 0 et valeur de c : 0
valeur de i : 1 et valeur de c : 1
et cætera
valeur de i : 254 et valeur de c : 254
valeur de i : 255 et valeur de c : 255
valeur de i : 256 et valeur de c : 0
valeur de i : 257 et valeur de c : 1
et cætera
valeur de i : 298 et valeur de c : 42
valeur de i : 299 et valeur de c : 43
```

Comment expliquer ceci ?

Exercice 2

On considère le code C++ suivant :

```
#include <iostream> // bibliothèque d'affichage
```

```cpp
int main() // début de la fonction principale
{
    char c = 0; // on définit la variable c
    for (int i = 0; i < 257; i++) // on fait une boucle pour
    {
        std::cout << "valeur de i : " << i;
        // on affiche la valeur de i
        std::cout << "  et valeur de c : " << (int)c;
        // on affiche la valeur de c en base 10
        std::cout << endl; // on revient à la ligne (END Line)
        c++; // on augmente c
    }
    return 0; // la fonction principale renvoie zéro
}
```

Voilà ce que la console affiche :

```
valeur de i : 0 et valeur de c : 0
valeur de i : 1 et valeur de c : 1
et cætera
valeur de i : 126 et valeur de c : 126
valeur de i : 127 et valeur de c : 127
valeur de i : 128 et valeur de c : -128
valeur de i : 129 et valeur de c : -127
et cætera
valeur de i : 254 et valeur de c : -2
valeur de i : 255 et valeur de c : -1
valeur de i : 256 et valeur de c : 0
```

Comment expliquer ceci ?

Exercice 3

Donner les représentations en complément à deux sur un octet de :
88, 89, 90, -125, -2 et -3.

Exercice 4

Donner les représentations en complément à 2 (sur un octet) de -1 et de 92.
Ajouter ces deux représentations (en ignorant la dernière retenue). Quel nombre repré-

sente cette somme ?