Logique binaire et calcul booléen

Comme vous l'avez vu au chapitre précédent, vous pouvez écrire les nombres en base 2, encore appelé binaire, en utilisant la syntaxe `0b`. Les nombres binaires ont un intérêt particulier dans le monde de l'informatique. Les ordinateurs fonctionnent à la base avec des signaux électriques, il est très facile de représenter une valeur pouvant prendre deux états en électronique. Par exemple, on peut dire qu'un tension basse (autour de 0 volts) représente un état et une tension haute (autour de 5 volts) à un autre état.

Cette représentation électronique des valeurs binaires en +0V et +5V était vrai dans les premiers ordinateurs, mais avec l'évolution des besoins et technologies, la situation actuelle est un peu plus compliquée. Le plus important à retire dans ce cours C++ est qu'une valeur binaire peut prendre deux états.
Il est possible de nommer ces deux états de différentes manières, selon le contexte ou les habitudes. On peut par exemple parler d'états “haut” et “bas”, “oui” et “non”, “vrai” ou “faux”, “0” ou “1”, etc. En C++, vous avez deux manière de représenter une valeur binaire :

- pour écrire un nombre entier en utilisant le mode binaire avec `0b` et des 0 et 1 ;
- pour écrire une valeur binaire, en utilisant les mots-clés `true` (vrai) et `false` (faux).

Les booléens

Voyons dans un premier temps les valeurs binaires, encore appelées booléens (nom donné en l'honneur du mathématicien George Boole, qui a créé cette branche des mathématiques).

Le type bool

Les opérateurs de comparaison

Les opérateurs logiques

L'algèbre de bool

http://fr.wikipedia.org/wiki/Alg%C3%A8bre_de_Boole_(logique)

La représentation binaire

Voyons maintenant la représentation binaire des nombres entiers. Vous avez vu dans le chapitre précédent comment écrire un nombre selon cette représentation. Dans la représentation binaire, chaque chiffre 0 ou 1 est appelé un bit. Lorsque vous affichez ce nombre, la valeur est affichée (par défaut) selon la base 10 (décimale).

main.cpp

```
#include <iostream>

int main() {
    std::cout << 0b101010 << std::endl;
}
```

affiche :

```
42
```

Il n'existe pas de directive pour afficher les nombres directement en

binaire, on utilise souvent à la place le représentation hexadécimale. La raison est qu'il est relativement facile de faire la conversion entre hexadécimale et le binaire. En effet, un chiffre hexadécimal correspond exactement à quatre chiffres binaires, il faut donc utiliser la conversion suivante :

hexadécimal	binaire	hexadécimal	binaire
0	0000	8	1000
1	0001	9	1001
2	0010	a	1010
3	0011	b	1011
4	0100	c	1100
5	0101	d	1101
6	0110	e	1110
7	0111	f	1111

Ainsi, pour convertir la valeur `0x42` en binaire, vous devez prendre le premier chiffre (`4`), le convertir en binaire (`0100`), puis faire la même chose avec le second chiffre (`2`, ce qui donne `0010`). La représentation binaire finale est donc `0b01000010`.

La conversion entre nombres entier décimaux et leur représentation binaire est relativement simple, mais retenez que toutes les informations contenu dans un ordinateur sont enregistrée au format binaire, que ça soit du texte ou des nombres réels. La conversion n'est alors pas aussi directe qu'avec les nombres entiers et il existe des normes pour faire ces conversions.
Une séquence de 8 bits (ou de 2 chiffres hexadécimaux, cela revient au même) est appelée un octet (1 o), 1024 octets donnent 1 kilo octet (1 ko), 1024 ko donnent 1 méga octets (1 Mo), 1024 méga octets donnent 1 giga octets (1 Go), 1024 giga octets donnent 1 tera octets (1 To).

Les opérateurs arithmétiques

Maintenant que vous savez écrire et lire les nombres binaires, vous allez pouvoir les manipuler. Comme ce sont des nombres entiers, les opérateurs arithmétiques présentés dans le chapitre précédent peuvent être utilisé :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 0b1010 + 0b1011 << std::endl; // addition
    std::cout << 0b0011 - 0b1101 << std::endl; //
soustraction
    std::cout << 0b1000 * 0b1011 << std::endl; //
multiplication
    std::cout << 0b1001 / 0b0010 << std::endl; // division
entière
}
```

affiche :

```
21
-10
88
4
```

en exo : comment faire les opérateurs arithmétique à la main ? exo sur les grand nombres

L'opérateur négation

En complément de ces opérateurs arithmétiques, il existe des opérateurs travaillant sur la représentation binaire, que l'on appelle opérateurs logiques bit à bit. Le premier opérateur est la négation `~`, qui permet d'inverser tous les bits d'un nombre (les 0 deviennent de 1 et les 1 deviennent des 0). Par exemple :

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << std::hex << std::showbase;
    std::cout << ~0b1 << std::endl;
    std::cout << ~0b110011 << std::endl;
}
```

affiche :

```
0xfffffffe
0xffffffcc
```

Le résultat peut paraître surprenant. Si on réécrit ces deux nombres en binaire et qu'on les aligne avec les valeurs binaires d'origine, on obtient :

```
0b1 =                                                 1
0xfffffffe = 1111 1111 1111 1111 1111 1111 1111 1110

0b110011 =                                      11 0011
0xffffffcc = 1111 1111 1111 1111 1111 1111 1100 1100
```

Si on se rappelle que les 0 devant un nombre peuvent être ignorés (1 = 01 = 001 = 0001, etc.), on comprend que l'opération est réalisée sur des nombres entiers de 32 bits (ou 4 octets), quelque soit le nombre de bits que l'on utilise pour écrire le nombre. Les 0 manquant devant le nombre sont ajoutés avant l'opération.

On pourrait se dire que c'est un gâchis au niveau de la mémoire que l'ordinateur utilise systématiquement 32 bits, alors que l'on écrit des nombres sur 1 ou 6 bits. La raison est qu'un ordinateur est conçu pour optimiser le travail sur des nombres d'une certaine taille (32 ou 64 bits par exemple pour les ordinateurs de bureau), ce qui explique cette conversion. La taille optimale pour représenter un nombre dépend du type de processeur et du système d'exploitation, mais il est possible en C++ d'utiliser des

nombres de taille différentes. Vous verrez cela dans un prochain chapitre.
Un autre type d'opérateur logique sont les opérations de décalage à droite `»` et à gauche `«`. Ces opération permettent de décaler les bits à droite ou à gauche d'un certain nombre de bits. Par exemple :

main.cpp

```
#include <iostream>

int main() {
    std::cout << std::hex << std::showbase;
    std::cout << (0b11011000 << 1) << std::endl; // décalage de 1 bit à gauche
    std::cout << (0b11011000 << 2) << std::endl; // décalage de 2 bit à gauche
    std::cout << (0b11011000 >> 1) << std::endl; // décalage de 1 bit à droite
    std::cout << (0b11011000 >> 2) << std::endl; // décalage de 2 bit à droite
}
```

Remarquez bien les parenthèses. Sans celle-ci, le compilateur ne pourra pas faire la différence entre les opérateur de décalage de bits `«` et `»` et les opérateurs de flux `«` et `»`, ce qui ne produira pas le comportement attendu.
Plus généralement, il faudra faire attention en C++ à la syntaxe, un même opérateur pouvant signifiant des choses différentes selon le contexte.

affiche :

```
0x1b0
0x360
0x6c
0x36
```

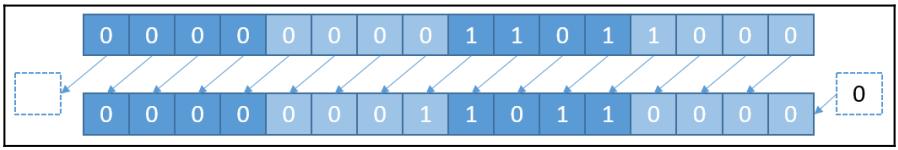
Affichons les valeurs en binaire et alignons les pour mieux comprendre :

```
0b11011000 = 0000 0000 0000 0000 0000 0000 1101 1000

0x1b0 =      0000 0000 0000 0000 0000 0001 1011 0000 //
décalage de 1 bit à gauche
0x360 =      0000 0000 0000 0000 0000 0011 0110 0000 //
décalage de 2 bit à gauche

0x6c =       0000 0000 0000 0000 0000 0000 0110 1100 //
décalage de 1 bit à droite
0x36 =       0000 0000 0000 0000 0000 0000 0011 0110 //
décalage de 2 bit à droite
```

Prenons par exemple le premier décalage (décalage de 1 bit à gauche). Avec un schéma, cela devrait être encore plus clair :

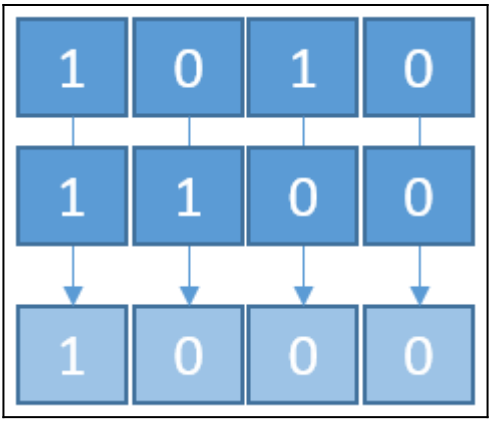


On voit :

- que la séquence de 0 et de 1 est identique ;
- qu'elle est décalée de 1 bit vers la gauche ;
- qu'un 0 est inséré à droite ;
- que le 0 à gauche est perdu.

Les opérateurs logiques bit à bit

Pour terminer, il existe les opérateurs logique AND (ET) `&`, OR (OR) `|` et XOR (OU Exclusif) `^` pour les nombres. Ils sont similaire aux opérateurs de même nom que vous avez vu précédemment pour les booléens, sauf qu'ils s'appliquent sur chaque bit d'un nombre. Ainsi, le premier bit du résultat est calculé à partir du premier bit de chaque nombre, le deuxième bit du résultat à partir du deuxième bit de chaque nombre, et ainsi de suite.



Le code suivant permet de vérifier les différents opérateurs logique :

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << std::hex << std::showbase;
    std::cout << (0b1010 & 0b1100) << std::endl; // AND
    std::cout << (0b1010 | 0b1100) << std::endl; // OR
    std::cout << (0b1010 ^ 0b1100) << std::endl; // XOR
}
```

affiche :

```
0x8
0xe
0x6
```

Si on convertie en binaire, on obtient :

```
0b1010 = 1 0 1 0
0b1100 = 1 1 0 0
0x8 =    1 0 0 0 // AND
0xe =    1 1 1 0 // OR
0x6 =    0 1 1 0 // XOR
```

On retrouve les tables logiques données pour les booléens.