Complément C++ pour MT et EL

Tutoriel sur les opérateurs bit à bit (*bitwise operators*)

Les opérateurs bit à bit facilitent la manipulation d'un motif binaire à l'échelle du bit individuel ou d'un groupe de bits (dans une variable de type entier).

Pourquoi aurait-on besoin de ces opérateurs ?

De nos jours, la mémoire n'est plus une ressource coûteuse sur la plupart de plateformes (ordinateur individuel, laptop, smartphone, serveurs, etc…) sauf sur certaines classes de ***systèmes embarqués*** avec des processeurs plus simples qui souvent ne travaillent qu'avec des entiers. Dans ce contexte il devient pertinent d'un point de vue économique (et énergétique) de travailler à une échelle plus fine que l'octet.

Un autre cadre d'application est le ***contrôle temps-réel*** dans lequel un processeur veut *minimiser le temps de réponse* à un événement mesuré par des capteurs. Il faut donc *minimiser le temps de transfert* d'informations avec des périphériques (capteur, actionneur…).

La partie théorique du cours ICC (module 3) montrera que les temps de transferts augmentent « dramatiquement » avec la distance entre le périphérique et le processeur (on-chip, off-chip, flash, disque…). De ce fait, on préfère regrouper le plus de données possible dans la charge utile transférée entre le périphérique et le processeur (qui peut être réduite à un mot mémoire de 16, 32 ou 64 bits) de façon à ne *faire qu'un seul transfert plutôt que plusieurs*. Le coût calcul d'utilisation des opérateurs bit à bit sur le motif binaire transféré est largement compensé par l'économie en nombre de transferts.

A titre plus prospectif, l'avènement du big data peut réduire les performances du processeur simplement à cause du transfert des données vers la mémoire (ICC module3). On peut se demander si un compactage et une manipulation des données avec cette famille d'opérateur bit à bit pourrait apporter un avantage compétitif.

<u>Mise en œuvre :</u>

Chaque fois qu'on veut travailler sur un groupe de bits on doit définir deux informations :

- **Masque** : sa taille **N** en nombre de bits se traduit par une constante appelée un *masque* pour lequel on a des 1 pour les **N** bits de poids faibles.

  Ex : un groupe de **3** bits est associé à un masque dont la valeur entière est 7 car sa valeur binaire est $\mathbf{111}_2$.

- **Décalage** : traduit le nombre de bits **D** dont on doit décaler le masque pour le présenter vis à vis de la région du motif binaire où se trouve l'information à extraire ou modifier. Ce nombre est toujours positif ; on l'utilise avec << pour un décalage à gauche et avec >> pour un décalage à droite.

Lorsqu'on travaille avec une donnée représentée avec un groupe de N bits, on part du principe qu'il s'agit d'entiers positifs (non-signés) compris entre **0 et 2^N-1.**

Supposons une donnée G représentée sur un groupe de 5 bits et stockée dans un entier non-signé **n** avec un décalage de **9** bits. Donc, lorsque ce groupe de bits est inséré dans le motif binaire de **n,** la plage qui le concerne se situe entre les puissances **9** et **14** de l'entier **n**.

On aurait du mal à estimer la valeur de l'information de notre groupe de 5 bits en faisant un affichage tel que : **`cout << n ;`**

En effet, par défaut on obtient la valeur décimale de la variable **n** ; il est difficile d'en déduire facilement la valeur du groupe de 5 bits compris entre les puissances 9 et 14 de deux.

Il est certes possible de faire afficher la valeur de **n** en hexadécimal (cours ultérieur) ; ce qui permettrait à une personne familière avec cette base de savoir ce qui se trouve entre les puissances 9 et 14 du motif binaire de la variable n en mémoire.

Indépendamment de cette capacité à lire l'hexadécimal, il peut être nécessaire de récupérer la *valeur de la donnée G dans une variable pour la poursuite du programme*. Cela est détaillé ci-dessous avec la variable **my_data** qui récupère la valeur des 5 bits de G :

```
#include <iostream>
using namespace std;

constexpr unsigned mask5(31);
constexpr unsigned shift(9);

int main()
{
      unsigned int n(1234567);
      unsigned int my_data(0);
```

On utilise les opérateurs combinés avec l'affectation pour suivre les transformations effectuées sur n. Tout d'abord le décalage qui ramène G sur les poids faibles :

```
      n = n >> shift ;
```

Puis le masquage qui supprime tout ce qui se trouve dans les puissances supérieures au 5 bits du groupe de G :

```
      my_data = n && mask5 ;
```

On dispose dans **my_data** de la valeur des 5 bits du groupe G ; cette valeur est comprise entre 0 et 31 puisqu'elle est limitée à 5 bits.

Le code est disponible dans le fichier archive tutoriel_operateurs_bit_a_bit