Opérateurs bitwise

Lycée Thiers

Crédits

- Une page de Dev4all

Crédits

- Une page de Dev4all
- Cette page de Wikipedia

Présentation

Dans un ordinateur, les données et les fichiers sont constitués de bits en nombre variable selon leurs types.
En C, il existe 6 opérateurs de gestion binaires (opérateurs *bitwise* ou *bit à bit*) :

- `&` : Opérateur bitwise AND

Présentation

Dans un ordinateur, les données et les fichiers sont constitués de bits en nombre variable selon leurs types.
En C, il existe 6 opérateurs de gestion binaires (opérateurs *bitwise* ou *bit à bit*) :

- `&` : Opérateur bitwise AND
- `|` : Opérateur bitwise OR

Présentation

Dans un ordinateur, les données et les fichiers sont constitués de bits en nombre variable selon leurs types.
En C, il existe 6 opérateurs de gestion binaires (opérateurs *bitwise* ou *bit à bit*) :

- `&` : Opérateur bitwise AND
- `|` : Opérateur bitwise OR
- $\wedge$: Opérateur bitwise XOR (OU exclusif). Noté aussi $\oplus$ en maths.

Présentation

Dans un ordinateur, les données et les fichiers sont constitués de bits en nombre variable selon leurs types.
En C, il existe 6 opérateurs de gestion binaires (opérateurs *bitwise* ou *bit à bit*) :

- `&` : Opérateur bitwise AND
- `|` : Opérateur bitwise OR
- $\wedge$: Opérateur bitwise XOR (OU exclusif). Noté aussi $\oplus$ en maths.
- `~` : Opérateur bitwise de complément

Présentation

Dans un ordinateur, les données et les fichiers sont constitués de bits en nombre variable selon leurs types.
En C, il existe 6 opérateurs de gestion binaires (opérateurs *bitwise* ou *bit à bit*) :

- `&` : Opérateur bitwise AND
- `|` : Opérateur bitwise OR
- `^` : Opérateur bitwise XOR (OU exclusif). Noté aussi $\oplus$ en maths.
- `~` : Opérateur bitwise de complément
- `>>` : Opérateur de redirection vers la droite

Présentation

Dans un ordinateur, les données et les fichiers sont constitués de bits en nombre variable selon leurs types.
En C, il existe 6 opérateurs de gestion binaires (opérateurs *bitwise* ou *bit à bit*) :

- `&` : Opérateur bitwise AND
- `|` : Opérateur bitwise OR
- `^` : Opérateur bitwise XOR (OU exclusif). Noté aussi $\oplus$ en maths.
- `~` : Opérateur bitwise de complément
- `>>` : Opérateur de redirection vers la droite
- `<<` : Opérateur de redirection vers la gauche

Principe

- Un opérateur bit à bit binaire comme `&` traite ses données d'entrée (deux objets de même type) selon leur représentation binaire (une séquence de 0 et de 1).

Principe

- Un opérateur bit à bit binaire comme `&` traite ses données d'entrée (deux objets de même type) selon leur représentation binaire (une séquence de 0 et de 1).
- Il calcule une nouvelle séquence binaire et traduit ce résultat dans le format d'origine des données.

L'opérateur &

- Cet opérateur binaire prend deux séquences de bits de même longueur et effectue un ET logique bit-à-bit.

L'opérateur &

- Cet opérateur binaire prend deux séquences de bits de même longueur et effectue un ET logique bit-à-bit.
- Il calcule le ET logique du premier bit des deux séquences, puis celui des seconds bits etc. Le résultat est une séquence de bits qui est interprétée selon le format d'origine des séquences d'entrée.

L'opérateur &

- Dans la fonction **main**, entrons ceci :

```
1  int x = 9, y = 12;
2  printf("%d&%d=%d\n",x,y,x&y);
```

Après compilation et exécution, on obtient l'affichage :

```
x&y=8
```

L'opérateur &

- Dans la fonction **main**, entrons ceci :

```
1  int x = 9, y = 12;
2  printf("%d&%d=%d\n",x,y,x&y);
```

Après compilation et exécution, on obtient l'affichage :

```
x&y=8
```

- L'opérateur **&** considère l'entier 9 comme **1001** et 12 comme **1100** :

$$\begin{array}{rr} & 1001 \\ \& & 1100 \\ \hline & 1000 \end{array}$$

Le résultat obtenu est **1000** qui est transformé en **int**. On obtient 8.

L'opérateur |

- Cet opérateur binaire fonctionne comme le précédent mais effectue un OU logique bit-à-bit et non un ET.
  Dans la fonction **main**, entrons ceci :

```
1  int x = 9, y = 12;
2  printf("%d | %d=%d\n",x,y,x | y);
```

Après compilation et exécution, on obtient l'affichage :

```
x|y=13
```

L'opérateur |

- Cet opérateur binaire fonctionne comme le précédent mais effectue un OU logique bit-à-bit et non un ET.
  Dans la fonction **main**, entrons ceci :

```
1  int x = 9, y = 12;
2  printf("%d | %d=%d\n",x,y,x | y);
```

Après compilation et exécution, on obtient l'affichage :

```
x|y=13
```

- On fait l'opération

```
   1001
|  1100
--------
   1101
```

Le résultat obtenu est **1101** qui est transformé en **int**. On obtient 13.

L'opérateur ^

- Cet opérateur binaire réalise le XOR bit-à-bit de ses deux opérandes.
- Dans la fonction **main**, entrons ceci :

```
1  int x = 9, y = 12;
2  printf("%d ^ %d=%d\n",x,y,x ^ y);
```

Après compilation et exécution, on obtient l'affichage :

```
x^y=5
```

- On fait l'opération

```
   1001
^  1100
---------
   0101
```

Le résultat obtenu est **0101** qui est transformé en **int**. On obtient 5.

L'opérateur ~

- Cet opérateur unaire inverse tous les bits de la séquence

L'opérateur ~

- Cet opérateur unaire inverse tous les bits de la séquence
- Dans la fonction **main**, entrons ceci :

```
1  uint8_t z = 5; uint8_t t = ~z;// entiers 8 bits
2  printf("~%u=%u\n",z,t); // %u pour unsigned
3
```

Après compilation et exécution, on obtient l'affichage :

```
~5=250
```

L'opérateur ~

- Cet opérateur unaire inverse tous les bits de la séquence
- Dans la fonction **main**, entrons ceci :

```
1  uint8_t z = 5; uint8_t t = ~z;// entiers 8 bits
2  printf("~%u=%u\n",z,t); // %u pour unsigned
3
```

Après compilation et exécution, on obtient l'affichage :

```
~5=250
```

- N'oublions pas qu'on travaille sur 8 bits

```
~  0000 0101
-------------
   1111 1010
```

Le résultat obtenu est **1111 1010** qui est transformé en **uint8_t** (entier non signé). On obtient 250 c.a.d 255-5.

L'opérateur ~

Dans la fonction **main**, entrons ceci :

```
1  int8_t u = 5; int8_t v = ~u;
2  printf("~%d=%d\n",u,v);
3
```

Après compilation et exécution, on obtient l'affichage :

```
~5=-6
```

- N'oublions pas qu'on travaille en complément à 2 sur 8 bits.

```
~  0000 0101
-------------
   1111 1010
```

Le résultat obtenu est **1111 1010** qui est transformé en **int8_t**. En base 10, ce nombre vaut 250. On lui retranche $2^8 = 256$ (entiers signés) et on obtient -6

L'opérateur >>

- Cet opérateur dit de *décalage à droite*, décale les bits vers la droite. Les bits qui sortent à droite sont perdus, tandis que le bit de poids fort est recopié à gauche.

L'opérateur >>

- Cet opérateur dit de *décalage à droite*, décale les bits vers la droite. Les bits qui sortent à droite sont perdus, tandis que le bit de poids fort est recopié à gauche.
- Ecrivons puis compilons et exécutons :

```
1  int32_t x = 13;
2  printf("%d", 13 >> 2);
```

On obtient l'affichage de 3.

L'opérateur >>

- Cet opérateur dit de *décalage à droite*, décale les bits vers la droite. Les bits qui sortent à droite sont perdus, tandis que le bit de poids fort est recopié à gauche.
- Ecrivons puis compilons et exécutons :

```
1  int32_t x = 13;
2  printf("%d", 13 >> 2);
```

On obtient l'affichage de 3.

- Expliquons la raison de ce résultat. L'écriture binaire de 13 sur 32 bits est :

$$\mathbf{0}00000000000000000000000000001101$$

L'opérateur >>

- Cet opérateur dit de *décalage à droite*, décale les bits vers la droite. Les bits qui sortent à droite sont perdus, tandis que le bit de poids fort est recopié à gauche.
- Ecrivons puis compilons et exécutons :

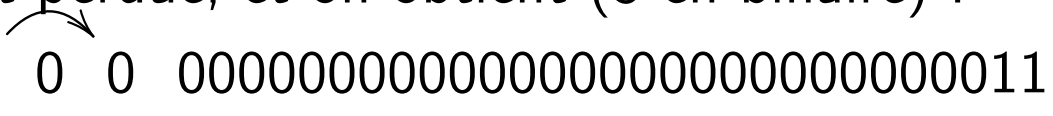
```
1  int32_t x = 13;
2  printf("%d", 13 >> 2);
```

On obtient l'affichage de 3.

- Expliquons la raison de ce résultat. L'écriture binaire de 13 sur 32 bits est :

$$\mathbf{0}00000000000000000000000000001101$$

- On décale les bits de deux rangs vers la droite. La séquence finale **01** est perdue, et on obtient (3 en binaire) :



L'opérateur >> (suite)

- Il a fallu combler les deux cases vides laissées par le départ de **01**. On a rajouté deux zéros à gauche parce que le bit de poids fort de l'expression initiale est à zéro. Ce zéro désigne en fait le signe du nombre 13 : ce dernier est positif comme $(-1)^0$.

L'opérateur >> (suite)

- Il a fallu combler les deux cases vides laissées par le départ de **01**. On a rajouté deux zéros à gauche parce que le bit de poids fort de l'expression initiale est à zéro. Ce zéro désigne en fait le signe du nombre 13 : ce dernier est positif comme $(-1)^0$.
- Le résultat de **13** >> **2** est donc **13/4** c'est à dire 3 (la division par 4 parce que $2^2 = 4$)

L'opérateur >> (suite)

- Effectuons maintenant ceci :

```
1  int32_t x = -13;
2  printf("%d\n", x >> 2);
3
```

Cette fois-ci le résultat est -4 alors qu'on se serait attendu à -3.

L'opérateur >> (suite)

- Effectuons maintenant ceci :

```
1  int32_t x = -13;
2  printf("%d\n", x >> 2);
3
```

Cette fois-ci le résultat est -4 alors qu'on se serait attendu à -3.

- L'écriture de -13 en complément à 2 sur 32 bits est en effet :

11111111111111111111111111110011

L'opérateur >> (suite)

- Effectuons maintenant ceci :

```
1  int32_t x = -13;
2  printf("%d\n", x >> 2);
3
```

Cette fois-ci le résultat est -4 alors qu'on se serait attendu à -3.

- L'écriture de -13 en complément à 2 sur 32 bits est en effet :

  11111111111111111111111111110011

- On décale de deux rangs vers la droite, et on ajoute la séquence **11** à gauche (on comble les vides par le bit de poids fort de la séquence initiale). On se retrouve donc avec :

  11111111111111111111111111111100

  C'est l'expression de -4 en complément à deux sur 32 bits.

L'opérateur <<

- Cet opérateur de décalage vers la gauche correspond à une multiplication par une puissance de deux.

L'opérateur <<

- Cet opérateur de décalage vers la gauche correspond à une multiplication par une puissance de deux.
- Considérons

```
1   printf("%d\n", 3 << 4);
```

Le codage de 3 en complément à 2 sur 8 bits est **0000011**.

L'opérateur <<

- Cet opérateur de décalage vers la gauche correspond à une multiplication par une puissance de deux.
- Considérons

```
1   printf("%d\n", 3 << 4);
```

Le codage de 3 en complément à 2 sur 8 bits est **0000011**.

- Le décalage de 4 rangs à gauche produit **0110000**, ce qui fait 48.

L'opérateur <<

- Cet opérateur de décalage vers la gauche correspond à une multiplication par une puissance de deux.
- Considérons

```
1  printf("%d\n", 3 << 4);
```

Le codage de 3 en complément à 2 sur 8 bits est **0000011**.

- Le décalage de 4 rangs à gauche produit **0110000**, ce qui fait 48.
- Il se trouve que $3 \times 2^4 = 48$. On a donc multiplié 3 par 2^4 en faisant notre décalage de 4 bits.