

Chapitre 3

Les opérateurs

Plan

- 1. Les opérateurs arithmétiques**
2. Les opérateurs d'affectation / assignation
3. Les opérateurs d'incrémentation
4. L'opérateur virgule
5. Les opérateurs de comparaison
6. Les opérateurs logiques
7. Les opérateurs bit-à-bit
8. Les opérateurs de décalage de bit

3.1 Les opérateurs arithmétiques

| Opérateur | Signification | Exemple |
|-----------|---|----------|
| + | Addition | $a + b$ |
| - | Soustraction | $a - b$ |
| * | Multiplication | $a * b$ |
| / | Division | a / b |
| % | Modulo : reste de la division d'entiers | $a \% b$ |

3.1 Les opérateurs arithmétiques

Quelle valeur prend **a** dans ces expressions ?

```
int a;  
a = 10 % 2;  
a = 10 % 3;  
a = 10 % 10;  
a = 5 % 10;  
a = 10 % 0;  
a = 0 % 10;
```



Remarques

L'opérateur `%` ne s'applique qu'à des opérandes de type entier et retourne un entier.

La fonction `fmod()` dans la librairie `<math.h>` travaille avec des valeurs de type double.

3.1 Priorité des opérateurs

| Opérateur | Sens d'évaluation | Priorité |
|--------------|-------------------|----------|
| () | → | haute |
| | | |
| * / % | → | |
| + - | → | |
| | | |
| | | |
| | | |
| | | |
| = | ← | basse |

3.1 Priorité des opérateurs

Dans chaque classe de priorité, les opérateurs ont la même priorité. S'il y a une suite d'opérateurs binaires de même classe, l'évaluation se fait de la **gauche vers la droite** dans l'expression.

Exemple

$$6 / 3 * 2 \rightarrow 4$$

Pour les opérateurs unaires (!, ++, --) et pour les opérateurs d'affectation (=, +=, -=, *=, /=, %=), l'évaluation se fait de **droite à gauche**.

3.1 Les opérateurs arithmétiques

Quelle valeur prend a dans ces expressions ?

```
int a = 0;
```

```
a = (10 + 6) / (3 - 1);
```

```
a = (10 + 6) / 3 - 1;
```

```
a = 10 + 6 / (3 - 1);
```

```
a = 10 + 6 / 3 - 1;
```



3.1 Types et domaines de définition

Le **type** d'une variable définit le **domaine des valeurs possibles**

Il faut être attentif à **ne pas dépasser les limites de validité** des variables.

⚠ lorsqu'on combine des variables de types différents à l'aide des opérateurs mathématiques.

Le tableau suivant rappelle les **domaines de validité** de quelques types du langage C.

3.1 Domaines de validité

| Type | Min | Max |
|-----------------------------|-------------|-------------|
| <code>char</code> | -128 | +127 |
| <code>unsigned char</code> | 0 | +255 |
| <code>short</code> | -32768 | +32767 |
| <code>unsigned short</code> | 0 | +65535 |
| <code>int</code> | -2147483648 | +2147483647 |
| <code>unsigned int</code> | 0 | +4294967295 |
| <code>long</code> | -2147483648 | +2147483647 |
| <code>unsigned long</code> | 0 | +4294967295 |
| <code>float</code> | 1.1E-38 | 3.4E+38 |
| <code>double</code> | 2.2E-308 | 1.7E+308 |

3.1 Domaines de validité

Le fichier `<limits.h>` définit des macros pour les nombres entiers. Les valeurs dépendent de l'implémentation et de la machine.

| Valeur | Remarques |
|---|---|
| CHAR_BIT | Taille d'un <code>char</code> en bits (minimum 8 bits) |
| SCHAR_MIN, SHRT_MIN, INT_MIN, LONG_MIN, LLONG_MIN (C99) | Valeur minimum possible des types entiers signés : <code>char, short, int, long, long long</code> |
| SCHAR_MAX, SHRT_MAX, INT_MAX, LONG_MAX, LLONG_MAX (C99) | Valeur maximum possible des types entiers signés : <code>signed char, signed short, signed int, ...</code> |
| UCHAR_MAX, USHRT_MAX, UINT_MAX, ULONG_MAX, ULLONG_MAX (C99) | Valeur maximum possible des types entiers non-signés : <code>unsigned char, unsigned short, ...</code> |
| CHAR_MIN | Valeur minimum possible pour un <code>char</code> |
| CHAR_MAX | Valeur maximum possible pour un <code>char</code> |
| ... | ... |

3.1 Domaines de validité

Le fichier `<float.h>` définit des macros pour les nombres en virgule flottante. Les valeurs dépendent de l'implémentation et de la machine.

| Valeur | Remarques |
|--------------------------|--|
| FLT_MIN, DBL_MIN | Valeur minimum possible d'un float , double |
| FLT_MAX, DBL_MAX | Valeur maximum possible d'un float ou double |
| FLT_EPSILON, DBL_EPSILON | Le plus petit nombre tel que $1.0 + x \neq 1.0$ pour float ou double |
| FLT_DIG, DBL_DIG | Nombre de décimales qui peuvent être représentées sans perte de précision pour float ou double |
| | |

3.1 «Overflow» – dépassement de capacité

Lorsqu'on sort des domaines de définition

a) Avec les types **réels** tels que `float` ou `double`, la variable prend une valeur spéciale

| | Valeurs | math.h |
|---------------------|---------------------|--|
| Valeur infinie | <code>1.#INF</code> | <code>HUGE_VAL</code> <code>INFINITY</code> |
| Valeur indéterminée | <code>1.#IND</code> | <code>NAN</code> |

b) Avec les types **entiers**, signés ou non, **aucun message d'erreur**, mais un certain nombre de bits sont pris en compte, les autres sont ignorés. Généralement, la valeur "boucle sur le codage", sans garanti à 100% hélas.

3.1 Les opérateurs arithmétiques

Que valent i3 et f3?

```
int    i1 = 5,  i2 = 2,  i3;  
float  f1 = 5., f2 = 2., f3;
```

```
f3 = f1 / f2;
```

```
i3 = i1 / i2;
```

```
f3 = i1 / i2;
```

```
i3 = f1 / f2;
```



3.1 Règles de conversions implicites

Une **conversion implicite** est une modification du type prise en charge par le compilateur.

- 1. Dans une affectation** : le résultat de l'expression à droite du signe = est converti de façon à **réduire au minimum les pertes de données**.
- 2. Dans une opération** : si les opérandes sont de types différents, il y a conversion **du type le plus FAIBLE vers le type le plus FORT**.

| FAIBLE |
|---------------|
| char |
| short |
| int |
| unsigned int |
| long |
| unsigned long |
| float |
| double |
| long double |
| FORT |

Conversion explicite (transtypage ou *casting*)

Une **conversion explicite** est **une modification du type**, réalisée par le programmeur grâce à la syntaxe suivante :

```
(<Type>) <valeur>
```

Exemples

```
(int) 132.45  
(double) 2/3 // 2.0/3
```

Remarque

On peut aussi forcer un transtypage de valeurs littérales avec des suffixes :
pour **double**, **f** pour **float**, **u** pour **unsigned**, **l** pour **long**.

Par exemple `x = 17f`

Exercice – Conversions

```
int x = 5;  
int y = 2;  
int r;  
r = x / y;
```

```
int x = 5;  
int y = 2;  
float r;  
r = x / y;
```

```
int x = 5;  
int y = 2;  
float r;  
r = (float)x / y;
```

```
float x = 5;  
int y = 2;  
int r;  
r = x / y;
```

```
float x = 5;  
int y = 2;  
float r;  
r = x / y;
```

```
int x = 5;  
int y = 2;  
float r;  
r = (float)(x / y);
```



Exemple

```
// This programs uses various operations using variables and mixing types
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    double result1, result2;
    long result3;
    unsigned int result4;
    result1 = 4. / 3;    // Division: double -> double
    result2 = 4 / 3;     // Division: int -> double
    result3 = result1;   // double -> long
    result4 = -result1;  // double -> unsigned int
    printf("double:      4. / 3  = %lg\n", result1);
    printf("double:      4  / 3  = %lg\n", result2);
    printf("long:        4. / 3  = %ld\n", result3);
    printf("unsigned int: -(4. / 3) = %u\n", result4);    // (2^32) - 1

    return 0;
}
```

Affichage des résultats

```
double:          4. / 3  = 1.33333
double:          4   / 3  = 1
long:            4. / 3  = 1
unsigned int: - (4./ 3) = 4294967295
```

Note : de nouvelles spécifications de format sont employées ici.
Voir chapitre sur les entrées-sorties.

| | |
|----------------------|-----------------|
| <i>long</i> | <i>%ld</i> |
| <i>unsigned long</i> | <i>%lu</i> |
| <i>double</i> | <i>%lg, %lf</i> |

Plan

1. Les opérateurs arithmétiques
- 2. Les opérateurs d'affectation / assignation**
3. Les opérateurs d'incrémentation
4. L'opérateur virgule
5. Les opérateurs de comparaison
6. Les opérateurs logiques
7. Les opérateurs bit-à-bit
8. Les opérateurs de décalage de bit

3.2 L'opérateur d'affectation

En C, **l'affectation peut jouer le rôle d'un opérateur en fournissant un résultat** pouvant être réutilisé dans le programme.

La partie à gauche de l'opérateur **=** s'appelle une ***lvalue***.

L'affectation retourne un résultat qui peut aussi être affecté à une variable :

b = (a = 4) ; ou **b = a = 4 ;**

est équivalent à:

a = 4 ;

b = a ;

3.2 L'opérateur d'affectation

<Var> = <Var> <Opérateur> <Expression>;

devient

<Var> <Opérateur composé> <Expression>;

Exemple

| | |
|---------------------------------------|---------------|
| <code>compteur = compteur + 2;</code> | } équivalents |
| <code>compteur += 2;</code> | |

3.2 L'opérateur d'affectation

| Opérateur | Signification | Exemple |
|-----------|-------------------------------|---------------------|
| = | Affectation | <code>a = b</code> |
| += | addition et assignation | <code>a += b</code> |
| -= | soustraction et assignation | <code>a -= b</code> |
| *= | multiplication et assignation | <code>a *= b</code> |
| /= | division et assignation | <code>a /= b</code> |
| %= | modulo et assignation | <code>a %= b</code> |

3.2 Priorité des opérateurs

| Opérateur | Sens d'évaluation | Priorité |
|------------------|-------------------|----------|
| () | → | haute |
| | | |
| * / % | → | |
| + - | → | |
| | | |
| | | |
| | | |
| | | |
| = += -= *= /= %= | ← | basse |

Plan

1. Les opérateurs arithmétiques
2. Les opérateurs d'affectation / assignation
- 3. Les opérateurs d'incrémentation**
4. L'opérateur virgule
5. Les opérateurs de comparaison
6. Les opérateurs logiques
7. Les opérateurs bit-à-bit
8. Les opérateurs de décalage de bit

3.3 Les opérateurs d'incrémentation

Opérateur qui **augmente**, ou **diminue, de 1** la valeur d'une variable entière.

| Opérateur | Signification | Exemple |
|-----------|----------------|--------------------------|
| ++ | incrémentation | a++ ++a |
| -- | décrémentation | a-- --a |

3.3 Les opérateurs d'incrémentation

Les opérateurs **++** et **--** peuvent être

(a) **préfixés**, à gauche de l'opérande
On **pré-incrémente**, puis on affecte

(b) **postfixés**, à droite de l'opérande
On affecte, puis on **post-incrémente**

3.3 Les opérateurs d'incrémentation

`y = x++;` est équivalent à

```
y = x;  
x = x + 1;
```

`y = ++x;` est équivalent à

```
x = x + 1;  
y = x;
```

`y = x--;` est équivalent à

```
y = x;  
x = x - 1 ;
```

`y = --x;` est équivalent à

```
x = x - 1 ;  
y = x ;
```

3.3 Opérateurs d'incrémentation

Que valent x et y ?

```
int x = 6, y;  
y = --x;
```

```
int x = 6, y;  
y = x--;
```



3.3 Opérateurs d'incrémentation

Que valent x et y ?

```
int x = 6, y;  
y = --x;
```

```
int x = 6, y;  
y = x--;
```



3.3 Priorité des opérateurs

| Opérateur | Sens d'évaluation | Priorité |
|------------------|-------------------|----------|
| () | → | haute |
| ++ -- | ← | |
| * / % | → | |
| + - | → | |
| | | |
| | | |
| | | |
| | | |
| = += -= *= /= %= | ← | basse |

Plan

1. Les opérateurs arithmétiques
2. Les opérateurs d'affectation / assignation
3. Les opérateurs d'incrémentation
- 4. L'opérateur virgule**
5. Les opérateurs de comparaison
6. Les opérateurs logiques
7. Les opérateurs bit-à-bit
8. Les opérateurs de décalage de bit

3.4 L'opérateur virgule : « , »

`<expression1> , ... , <expressionN>`

`expression1` est évaluée, puis `...`, puis finalement `expressionN`. La valeur de `expressionN`, la plus à droite, est conservée, c'est la valeur de l'opération virgule.

Utilisation courante dans les boucles :

```
for (i=1 , j=0 ; j<15 ; i++ , j=j+2) { ... }
```


Plan

1. Les opérateurs arithmétiques
2. Les opérateurs d'affectation / assignation
3. Les opérateurs d'incrémentation
4. L'opérateur virgule
- 5. Les opérateurs de comparaison**
6. Les opérateurs logiques
7. Les opérateurs bit-à-bit
8. Les opérateurs de décalage de bit

3.5 Opérateurs de comparaison

Comparaison de variables ou expressions dont le résultat est **0** ou **1** :

```
{type simple} x {type simple} -> {0, 1}
```

Exemples

```
_Bool b = 5 < (3 + 4) ;
```

```
b = 'A' > 'Z' ;
```

3.5 Exemple de valeurs booléennes

Les valeurs booléennes peuvent avoir diverses représentations

| Version | Type | vrai | faux | #include |
|------------|--------------------|----------------------------|--------------|--------------------------|
| K&R | <code>int</code> | 1 ($\neq 0$) | 0 | |
| <u>C99</u> | <code>bool</code> | true | false | <stdbool.h> |
| <u>C99</u> | <code>_Bool</code> | $\neq 0$ | 0 | |

Have g++ follow the coming C++0x (aka c++11) ISO C++ language standard [-std=c++0x]



```
#include <stdbool.h>
bool test = true;
test = (2*3) < 7;
printf("%d", test);
printf("%s", test ? "true" : "false");
```

```
_Bool test = 3; //!=0 => true
test = (2*3) < 7;
printf("%d", test);
printf("%s", test ? "true" : "false");
```

3.5 Opérateurs de comparaison

| Opérateur | Signification | Exemple |
|--------------|---------------------|------------------|
| == | égal à | a == b |
| != | différent de | a != b |
| > | supérieur à | a > b |
| < | inférieur à | a < b |
| >= | supérieur ou égal à | a >= b |
| <= | inférieur ou égal à | a <= b |

3.3 Priorité des opérateurs

| Opérateur | Sens d'évaluation | Priorité |
|------------------|-------------------|----------|
| () | → | haute |
| ++ -- | ← | |
| * / % | → | |
| + - | → | |
| < > <= >= | → | |
| == != | → | |
| | | |
| | | |
| = += -= *= /= %= | ← | basse |

Plan

1. Les opérateurs arithmétiques
2. Les opérateurs d'affectation / assignation
3. Les opérateurs d'incrémentation
4. L'opérateur virgule
5. Les opérateurs de comparaison
- 6. Les opérateurs logiques**
7. Les opérateurs bit-à-bit
8. Les opérateurs de décalage de bit

3.6 Opérateurs logiques

Opérateur qui agit sur des booléens et qui donne un booléen :

`{ _Bool } x { _Bool } -> { _Bool }`

Exemple :

```
bool a = true, b = false, c;  
c = a || b;
```

| Opérateur | Signification | Exemple |
|-----------|----------------|-----------------------------|
| && | et (AND) | <code>a && b</code> |
| | ou (OR) | <code>a b</code> |
| ! | négation (NOT) | <code>!a</code> |

3.6 Opérateurs logiques

| | | NOT | AND | OR | XOR |
|-------|-------|-----------------|-----------------------------|---------------------|---|
| p | q | <code>!q</code> | <code>p && q</code> | <code>p q</code> | <code>(!p && q)</code> <code> </code> <code>(p && !q)</code> |
| false | false | true | false | false | false |
| false | true | false | false | true | true |
| true | false | true | false | true | true |
| true | true | false | true | true | false |

3.6 Priorité des opérateurs

| Opérateur | Sens d'évaluation | Priorité |
|------------------|-------------------|----------|
| () | → | haute |
| ++ -- ! | ← | |
| * / % | → | |
| + - | → | |
| < > <= >= | → | |
| == != | → | |
| && | → | |
| | → | |
| = += -= *= /= %= | ← | basse |

3.6 Exemples

```
(true && false)
```

```
!true
```

```
false || !(32 > 12)
```

Remarque : les opérateurs **&&** et **||** ont un point d'arrêt entre l'évaluation du membre de gauche et du membre de droite : « *short-circuit evaluation* ». **Le membre de droite n'est évalué que si c'est nécessaire.**

Plan

1. Les opérateurs arithmétiques
2. Les opérateurs d'affectation / assignation
3. Les opérateurs d'incrémentation
4. L'opérateur virgule
5. Les opérateurs de comparaison
6. Les opérateurs logiques
- 7. Les opérateurs bit-à-bit**
8. Les opérateurs de décalage de bit

3.7 Opérateurs bit-à-bit

Opérateurs qui agissent directement au niveau des bits des mots. La valeur de **chaque bit** dépend **seulement** des valeurs des bit(s) à cette **position**

| Opérateur | Signification | Exemple |
|--------------|-----------------------|------------------|
| & | et : « AND » | a & b |
| | ou : « OR » | a b |
| ^ | ou exclusif : « XOR » | a ^ b |
| ~ | complément à 1 | ~a |

Les opérateurs composés sont également autorisés : **&=**, **|=** et **^=**

3.7 Opérateurs bit à bit – entiers

| | a | 0 | 0 | 1 | 1 |
|----------------|--------------|---|---|---|---|
| | b | 0 | 1 | 0 | 1 |
| Et | $a \ \& \ b$ | 0 | 0 | 0 | 1 |
| Ou | $a \ \ b$ | 0 | 1 | 1 | 1 |
| Ou exclusif | $a \ ^ \ b$ | 0 | 1 | 1 | 0 |
| Complément à 1 | $\sim a$ | 1 | 1 | 0 | 0 |

3.7 Opérateurs bit à bit – entiers

Exemples : `int a = 33333, b = -77777;`

| Expression | Représentation |
|----------------------------|-------------------------------------|
| <code>a</code> | 00000000 00000000 10000010 00110101 |
| <code>b</code> | 11111111 11111110 11010000 00101111 |
| <code>a & b</code> | 00000000 00000000 10000000 00100101 |
| <code>a ^ b</code> | 11111111 11111110 01010010 00011010 |
| <code>a b</code> | 11111111 11111110 11010010 00111111 |
| <code>~ (a b)</code> | 00000000 00000001 00101101 11000000 |
| <code>~ a & ~ b</code> | 00000000 00000001 00101101 11000000 |

3.7 Opérateurs bit à bit – entiers

Qu'affiche ce programme

```
unsigned char x=0xAA; // 1010 1010
printf("%X\n", x);
printf("%X\n", x & 0xFF);
printf("%X\n", x & 0x01);
printf("%X\n", x & 0x03);
```



Plan

1. Les opérateurs arithmétiques
2. Les opérateurs d'affectation / assignation
3. Les opérateurs d'incrémentation
4. L'opérateur virgule
5. Les opérateurs de comparaison
6. Les opérateurs logiques
7. Les opérateurs bit-à-bit
- 8. Les opérateurs de décalage de bit**

3.8 Opérateurs de décalage

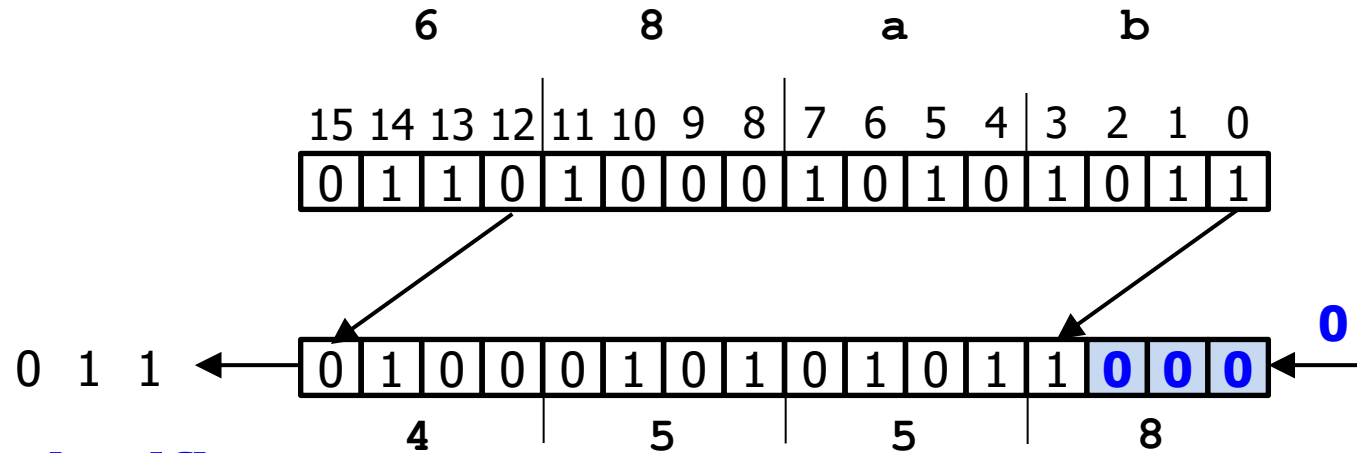
Les opérateurs de décalage vont décaler les bits soit vers la droite, soit vers la gauche.

| Opérateur | Signification | Exemple |
|-----------|--|----------------------|
| << | Décalage des bits de a vers la <u>gauche</u> de b bits | a << b |
| >> | Décalage des bits de a vers la <u>droite</u> de b bits | a >> b |
| <<= | Décalage à gauche et affectation | a <<= b |
| >>= | Décalage à droite et affectation | a >>= b |

3.8 Opérateur de décalage à gauche

```
short a = 0x68ab;
```

```
a = a << 3; /* shift left 3 bits */
```



Les **"Most Significant Bits"** sont perdus

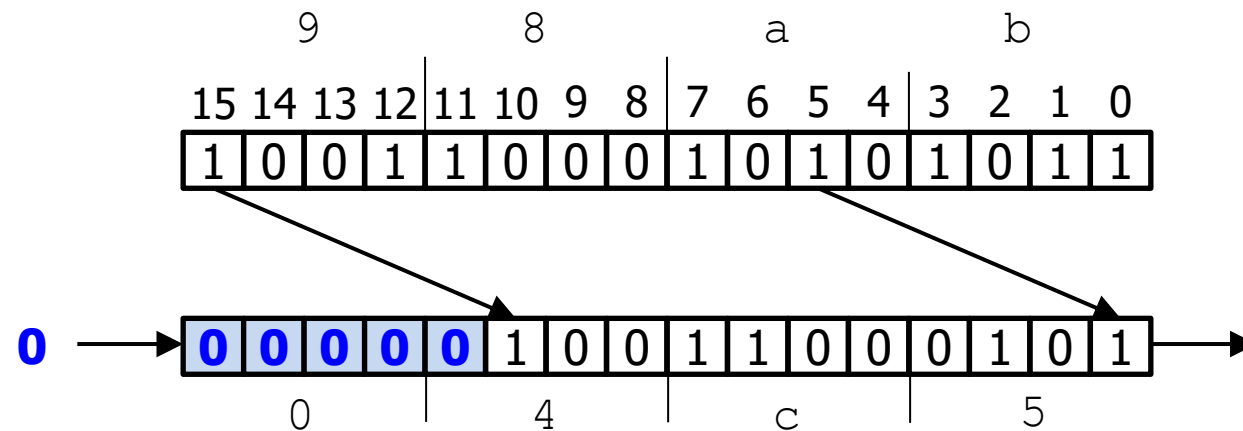
Les positions vides en entrée sont **remplies de zéros**

3.8 Opérateur de décalage à droite – non signé

```
unsigned short a = 0x98ab;
```

```
...
```

```
a >>= 5; /* shift right 5 bits */
```



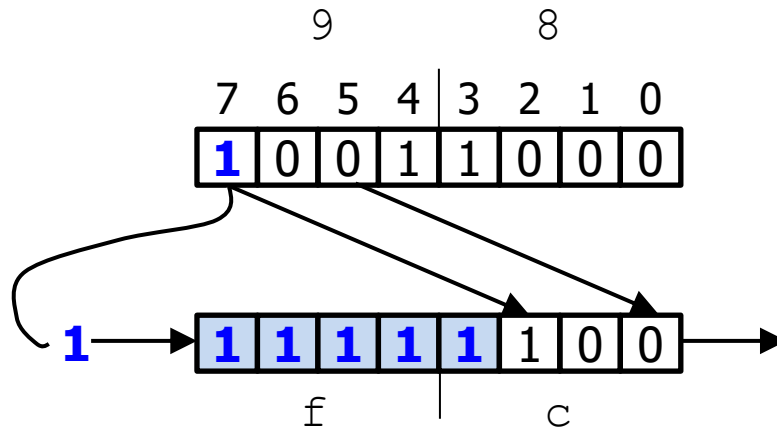
Pour les types **non signés**, les positions libérées sont **remplies avec des zéros**

3.8 Opérateur de décalage à droite – signé

```
char a = 0x98; // -104
```

```
. . .
```

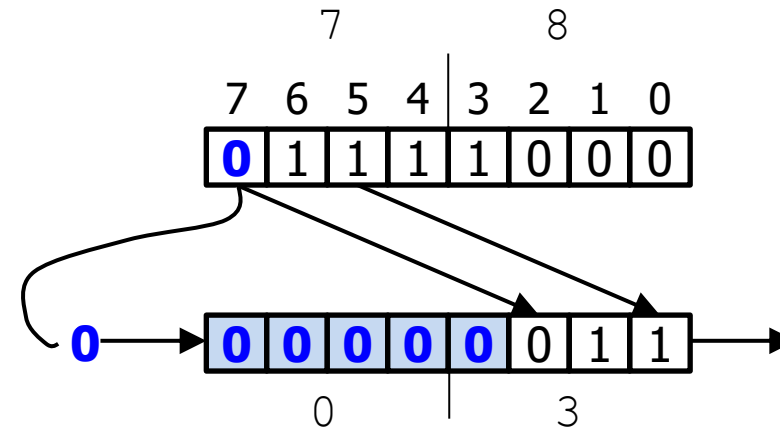
```
a >>= 5;
```



```
char a = 0x78; // +120
```

```
. . .
```

```
a >>= 5;
```



Les positions libérées sont remplies avec **une copie du bit de poids fort : bit de signe**

3.8 Remarques

$x \ll n$ est
équivalent à
multiplier par 2^n

$x \gg n$ est
équivalent à
diviser par 2^n

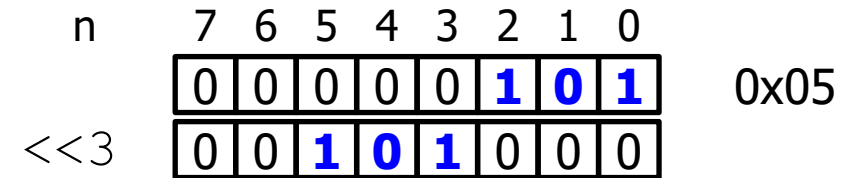
| n | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
|-----|---|---|---|---|---|---|---|---|------|-----|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0x01 | 1 |
| <<1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0x02 | 2 |
| <<2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0x04 | 4 |
| <<3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0x08 | 8 |
| <<4 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0x10 | 16 |
| <<5 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0x20 | 32 |
| <<6 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40 | 64 |
| <<7 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x80 | 128 |

Les multiplications et divisions de puissance de 2 peuvent être implémentées avec des **décalages** (et être plus rapides)

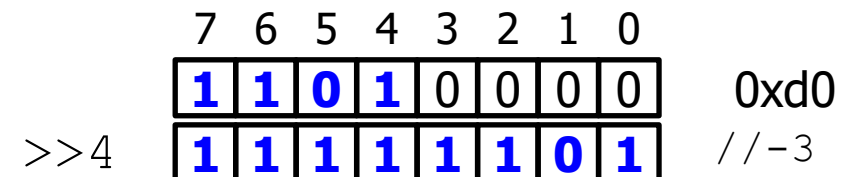
→ Inutile de le faire en pratique car les compilateurs récents le feront pour nous si possible.

3.8 Examples

```
char a = 0x05; // +5  
a <<= 3;
```



```
char a = 0xd0; // -48  
a >>= 4;
```



3.8 Extraction d'un byte d'un `int`

```
int i; char b;  
int noByte; // de 0 à 3  
  
unsigned int mask = 0xFF;  
int n = noByte * 8;  
  
mask <=< n;  
b = ( ( i & mask ) >> n );
```

| Opérateur | Sens d'évaluation | Priorité |
|--|-------------------|----------|
| <code>++ -- (postfixé) ()</code> | \rightarrow | haute |
| <code>+ - (unaire) ++ -- (préfixé) (<Type>) ! ~</code> | \leftarrow | |
| <code>* / %</code> | \rightarrow | |
| <code>+ -</code> | \rightarrow | |
| <code><< >></code> | \rightarrow | |
| <code>< > <= >=</code> | \rightarrow | |
| <code>== !=</code> | \rightarrow | |
| <code>&</code> | \rightarrow | |
| <code>^</code> | \rightarrow | |
| <code> </code> | \rightarrow | |
| <code>&&</code> | \rightarrow | |
| <code> </code> | \rightarrow | |
| <code>?:</code> | \leftarrow | |
| <code>= += -= *= /= %= >>= etc</code> | \leftarrow | basse |

Exercices



Exercices du chapitre 03