



Un programme effectue des opérations sur des nombres afin de générer un résultat. Ces nombres sont stockés dans des variables. Les opérations sur ces nombres sont effectuées par l'intermédiaire d'opérateurs. Ils caractérisent le type d'opération que l'on désire exécuter sur ces données.

L'opérateur le plus connu est l'opérateur d'assignation : le '='. Il permet d'affecter le contenu d'une variable. `A = 12 ;` // On assigne à la variable **A**, la valeur **12**. L'ancienne valeur contenue dans **A** est perdue.

## Assignation

Cet opérateur est l'opérateur fondamental de tout langage de programmation, en effet il permet d'affecter le contenu d'une variable. En C++, on utilise le symbole '='.

**Attention** : '=' ne correspond pas à une égalité, mais bien à une affectation.

### Syntaxe :

Algorithme	Représentation C++
variable ← expression variable1 ← variable2	variable = expression ; variable1 = variable2 ;

Comme toute ligne d'instruction en C++, il faut lire la ligne de la droite vers la gauche. C'est **expression** (constante littérale) qui est stocké à l'intérieur de la variable : **variable**.

### Exemples :

`A = 1854.7 ;` // On stocke la valeur **1854.7** (constante littérale double) dans la variable **A**.  
`B = A ;` // On stocke le contenu de **A** dans la variable **B**, ce qui fait que `B <= 1854.7`  
`D = C = 43 ;` // Il est possible d'avoir des affectations successives : **43** est stockée dans **C**, et ensuite, le contenu de **C** est stocké dans **D**.

## Opérateurs arithmétiques

Ils permettent d'effectuer les opérations dites mathématiques. Ces opérations ne s'appliquent qu'à des valeurs ou des variables de type numériques (entières ou réelles).

Opérateurs arithmétiques	Opération effectuée
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo (reste d'une division entière)

### Syntaxe :

Algorithme	Représentation C++
variable ← expression + 24 variable ← variable1 / variable2 variable ← variable1 modulo 2	variable = expression + 24 ; variable = variable1 / variable2 ; variable = variable1 % 2 ;

### Exercices :

Soit **A** ayant pour contenu la valeur numérique **65**, **B** la valeur **200**, et **C** la valeur **-98**.

Opérateurs	Résultat
A + B	
B / C	
B % C	
A = B * C	
B + C	
A - C	
B = A / B	

**Corrections** : [Visualisation du résultat](#)

## Opérateurs d'égalité, relationnels et logiques

Les opérateurs relationnels permettent de comparer des valeurs relatives. Ceux sont des opérateurs dits de test, ils sont donc généralement utilisés pour les structures de tests, ou les structures de boucle.

Le résultat de ces opérateurs est de type **booléen**, donc : soit **Vrai**, soit **Faux**.

- Faux** : représentée par la constante littérale : `false`.
- Vrai** : représentée par la constante littérale : `true`.

Opérateurs relationnels	Opération effectuée
>	Supérieur à
>=	Supérieur ou égal à
<	Inférieur à

<=	Inférieur ou égal à
==	Egal à
!=	Différent de, ou non égal à
&&	ET logique
	OU logique
!	NON logique

**Syntaxe :**

Algorithme	Représentation C++
<i>variable = expression</i> <i>expression1 &gt; expression2</i> ...	<i>variable == expression ;</i> <i>expression1 &gt; expression2 ;</i> ...

Où **variable** est le nom d'une variable de type numérique (char, entier ou réel). Et **expression** est une constante littérale d'une valeur numérique (Ex. 12, 98.566, ...).

**Exercices :**

Soit la déclaration suivante :

```
short A=65, B=200, C= -98 ;
```

Opérateurs	Résultat
A > B	
C >= B	
C < A	
C <= -98	
A == 0	
A == 65	
B != C	
C > -98	
A != B == C	
(A==65) && (B==200)	
(A==65) && (B==2000)	
(A==65)    (B==200)	
(A==7)    (B==200)	
(A==7)    (B==0)	
(A==65)    !(B !=200)	

 Ces opérateurs peuvent être utilisés en dehors des structures de test et de boucles.

 **Corrections :** [Visualisation du résultat](#)

**Opérateurs de bit (spécialisés pour le traitement binaire)**

Le langage C++ fourni cinq opérateurs qui permettent de manipuler les opérations au niveau du bit

Opérateurs de bit	Opération effectuée
&	ET bit à bit
^	OU exclusif bit à bit
	OU bit à bit
variable << décalage	Décalage à gauche
variable >> décalage	Décalage à droite
~	Complément (NON) bit à bit

**Exercices :**

Soit la déclaration suivante :

```
short A=65, B=200, C= -98 ;
```

Opérateurs	Résultat en binaire	Résultat en décimal
A		
B		
C		
A & B		
A   B		
A ^ B		
~A		
~C		
B << 1		
A >> 2		

 **Corrections :** [Visualisation du résultat](#)

**Opérateurs d'affectations composés**

Il existe 11 opérateurs d'affectation. L'opérateur "=" est le plus simple d'entre eux; les autres sont les opérateurs d'affectation composés. C'est une combinaison entre l'opérateur d'assignation et un autre des opérateurs déjà vus.

Opérateurs d'affectation composés	Opération effectuée
-----------------------------------	---------------------

Expression1 = Expression2	Expression1 = Expression 2
Expression1 *= Expression2	Expression1 = Expression1 * Expression2
Expression1 /= Expression2	Expression1 = Expression1 / Expression2
Expression1 %= Expression2	Expression1 = Expression1 % Expression2
Expression1 += Expression2	Expression1 = Expression1 + Expression2
Expression1 -= Expression2	Expression1 = Expression1 - Expression2
Expression1 <= Expression2	Expression1 = Expression1 << Expression2
Expression1 >= Expression2	Expression1 = Expression1 >> Expression2
Expression1 &= Expression2	Expression1 = Expression1 & Expression2
Expression1 ^= Expression2	Expression1 = Expression1 ^ Expression2
Expression1  = Expression2	Expression1 = Expression1   Expression2

### Syntaxe :

Algorithme	Représentation C++
<code>variable &lt;- variable opération expression</code>	<code>variable opération= expression ;</code>

### Exercices :

Donnez l'affectation composée relative à l'expression de gauche

Syntaxe classique	Avec l'affectation composée
<code>i = i + 5 ;</code>	
<code>j = j - 2 ;</code>	
<code>k = k * 3 ;</code>	
<code>x = y / 5 ;</code>	


 **Corrections :** [Visualisation du résultat](#)

### Opérateurs d'incrément et de décrémentation

Les affectations les plus fréquentes sont du type `i = i + 1 ;` ou `i = i - 1 ;`. En C++, nous disposons de deux opérateurs supplémentaires pour ces affectations :

<code>i++</code>	<code>++i</code>	Pour l'incrément (augmentation de une unité)
<code>i--</code>	<code>--i</code>	Pour la décrémentation (diminution de une unité)

Les opérateurs `++` et `--` sont employés dans les cas suivants :

-  incrémenter/décrémenter une variable (par exemple : dans une boucle). Dans ce cas il n'y a pas de différence entre la notation préfixe (`++i`, `--i`) et la notation postfixe (`i++`, `i--`).
- incrémenter/décrémenter une variable et dans le même temps affecter sa valeur à une autre variable. Dans ce cas, nous devons choisir entre la notation préfixe et postfixe.

### Exemples :

```
x = i++; <=> x=i; suivi de : i=i+1 ; // l'incrément se fait après l'affectation
x = i--; <=> x=i; suivi de : i=i-1 ; // la décrémentation se fait après l'affectation
x = ++i; <=> i=i+1; suivi de : x=i; // l'incrément se fait avant l'affectation
x = --i; <=> i=i-1; suivi de : x=i; // la décrémentation se fait avant l'affectation
```

### Exercices :


Avant chacune des instructions suivantes, `i = 3` et `j = 15` :

Instructions	Ecriture équivalente	Résultats
<code>i = ++j ;</code>		<code>i = ..... j = .....</code>
<code>i = j++ ;</code>		<code>i = ..... j = .....</code>
<code>i++ ;</code>		<code>i = ..... j = .....</code>
<code>j = ++i + 5 ;</code>		<code>i = ..... j = .....</code>
<code>j = i++ + 5 ;</code>		<code>i = ..... j = .....</code>

 **Corrections :** [Visualisation du résultat](#)

 Ces opérateurs sont généralement utilisés avec des variables de type entier et plus précisément des variables de type scalaire.

### Type scalaire

 Les variables ou les entités de type scalaire, sont des variables dont on peut déterminer le nombre exact de valeurs possibles. Ce nombre est limité et connu. Par exemple, le type `char` procure uniquement 256 valeurs possibles. D'une façon générale, tous les types entiers correspondent à ce genre de critère (`bool`, `char`, `short`, `int`). Par contre, les types réels, ne sont pas des types scalaires. En effet, entre la valeur 0.1 et 0.2, il existe normalement une infinité de valeur.

### Priorité des opérateurs

L'ordre de l'évaluation des différentes parties d'une expression correspond en principe à celle que nous connaissons en mathématiques. Par exemple, la multiplication est prioritaire sur l'addition et que la multiplication et l'addition sont prioritaires sur l'affectation.

Priorité	Opérateurs
1 (la plus forte)	<code>() [] -&gt; . ::</code>
2	<code>! ~ ++ -- (casting) * &amp; new delete</code>
3	<code>* / %</code>
4	<code>+ -</code>
5	<code>&lt;&lt; &gt;&gt;</code>
6	<code>&lt; &lt;= &gt; &gt;=</code>
7	<code>== !=</code>

8	&
9	^
10	
11	&&
12	
13	? :
14 (la plus faible)	= += -= *= /= %= ....

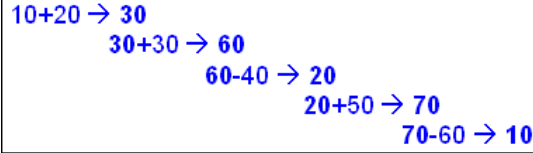


**Attention :**

Dans chaque classe de priorité, les opérateurs ont la même priorité. Si nous avons une suite d'opérateurs binaires de la même classe, l'évaluation se fait en passant de la gauche vers la droite dans l'expression. Pour les opérateurs unaires (!, ++, --) et pour les opérateurs d'affectation (=, +=, -=, \*=, /=, %=), l'évaluation se fait de droite à gauche dans l'expression.

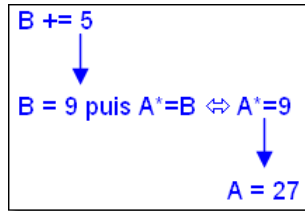
**Exemple :**

L'expression  $10+20+30-40+50-60$  donnera le résultat qui suit.



**Exemple :**

Pour  $A=3$  et  $B=4$ , l'expression `A *= B += 5;` donnera le résultat qui suit.



**Parenthèse :**

Les parenthèses sont seulement nécessaires si nous devons forcer la priorité, mais elles sont aussi permises si elles ne changent rien à la priorité. En cas de parenthèses imbriquées, l'évaluation se fait de l'intérieur vers l'extérieur.

**Exemple :**

En supposant à nouveau que  $A=5$ ,  $B=10$ ,  $C=1$ , l'expression suivante s'évaluera à 134 :

$$X = ((2*A+3)*B+4)*C ;$$

**Exemple :**

Observez la priorité des opérateurs d'affectation :

$X *= Y + 1 <==> X = X * (Y + 1)$  // équivalent  
 $X *= Y + 1 <==> X = X * Y + 1$  // non équivalent