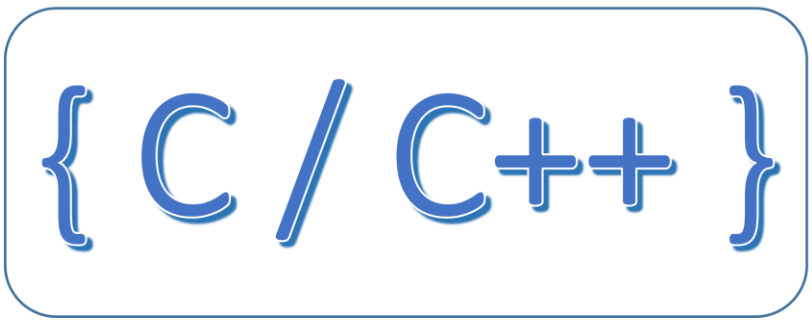
LYCÉE ST-JOSEPH HASPARREN BTS SN-IR	C++ OPERATEURS	COURS

Tâches professionnelles :	T7.2	Réaliser une maquette, un prototype logiciel/matériel
Compétences du référentiel :	C4.4	Développer un module logiciel.
Savoirs - Savoir faires :	S4.7	Langage de programmation C++

Table des matières



Nous avons découvert les variables et les constantes, nous pouvons donc commencer à travailler avec en utilisant des ***opérateurs***. Ce qui suit est une liste complète des opérateurs. À ce stade, il n'est probablement pas nécessaire de les connaître tous, mais ils sont tous énumérés ici pour servir également de référence.

1 OPERATEUR D'AFFECTATION (=)

L'opérateur d'affectation affecte une valeur à une variable : `x = 5;`

Cette instruction affecte la valeur entière ***5*** à la variable ***x***. L'opération d'affectation a toujours lieu de droite à gauche, et jamais l'inverse : `x = y;`

Cette instruction assigne à la variable ***x*** la valeur contenue dans la variable ***y***. A partir de l'instant où cette instruction est exécutée la valeur précédente de ***x*** est perdue et remplacée par la valeur de ***y***.

Notez aussi que nous ne faisons l'attribution de la valeur de ***y*** à ***x*** que au moment de l'opération d'affectation. Par conséquent, si ***y*** change un moment plus tard, cela n'affectera pas la nouvelle valeur prise par ***x***.

Par exemple, sur le code suivant, on voit sous forme de commentaires l'évolution du contenu stocké dans les variables:

```
1  // assignment operator
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int a, b;         // a:?,  b:?
8    a = 10;           // a:10, b:?
9    b = 4;            // a:10, b:4
10   a = b;            // a:4,  b:4
11   b = 7;            // a:4,  b:7
12
13   cout << "a:";
14   cout << a;
15   cout << " b:";
16   cout << b;
17 }
```

```
a:4 b:7
```

Les opérations d'affectation sont des expressions qui peuvent être évaluées. Cela signifie que l'affectation elle-même a une valeur, et cette valeur est celle affectée à l'opération.

Par exemple : `y = 2 + (x = 5);` équivaut à

```
1 x = 5;
2 y = 2 + x;
```

et ***y*** vaut donc ***7***.

L'expression suivante est également valable en C ++: `x = y = z = 5;`

Elle attribue ***5*** aux trois variables ***x***, ***y*** et ***z*** et toujours de droite à gauche.

2 LES OPERATEURS ARITHMETIQUES (+, -, *, /,%)

Les cinq opérations arithmétiques supportées par C ++ sont les suivantes :

opérateur	la description
+	une addition
-	soustraction
*	multiplication
/	division
%	modulo

Les opérations **d'addition**, **soustraction**, **multiplication** et **division** correspondent littéralement aux opérateurs mathématiques classiques.

Le dernier, ***opérateur modulo,*** représenté par un signe de pourcentage (**%**), donne le reste d'une division de deux valeurs.

Par exemple : `x = 11 % 3;` affecte ***2*** dans la variable ***x***, puisque la division de ***11*** par ***3*** donne ***3***, avec un reste de ***2***.

Modulo permet de :
- **dire si un nombre x est divisible par un autre y (le reste vaut 0),**
- **trouver la partie entière de la division ((x-reste)/y),**
- **trouver la partie décimale de la division (reste/y),**
- **ramener un nombre aléatoire dans une fourchette de valeurs donnée.**

EXERCICE :

Vous devez écrire et tester un programme qui aide un utilisateur à savoir combien d'emballages il doit prévoir pour expédier un certains nombres d'articles.

Le programme doit demander à l'utilisateur de saisir : le nombre d'articles à emballer, la quantité d'articles par emballage.

Le programme doit afficher pour l'utilisateur : le nombre d'emballages à prévoir et le nombre d'articles qui vont rester.

3 AFFECTATION COMPOSEES (+ =, - =, * =, / =, % =, >> =, << =, & =, ^ =, | =)

expression	équivalente à...
y += x;	y = y + x;
x -= 5;	x = x - 5;
x /= y;	x = x / y;
price *= units + 1;	price = price * (units+1);

Les exemples ci-contre montrent la logique de cette instruction qui conjugue opération et affectation :

Et ainsi de suite pour tous les autres opérateurs d'affectation composé.

Par exemple :

```
1  // compound assignment operators
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int a, b=3;
8    a = b;
9    a+=2;              // equivalent to a=a+2
10   cout << a;
11 }
```

5

Cette écriture est plus condensée que l'instruction classique, mais elle est aussi ***moins lisible, donc non recommandée***.

4 INCREMENTATION ET DECREMENTATION (++, --)

Certaines expressions peuvent être encore plus raccourcies : l'opérateur d'incrémentation (++) et l'opérateur de décrémentation (--) permettent d'augmenter de 1, ou de diminuer de 1, la valeur stockée dans une variable. Ils sont équivalents, respectivement à +=1 et -= 1.

Ainsi :

```
1 ++x;
2 x+=1;
3 x=x+1;
```

sont équivalentes, elles augmentent ***x*** de 1.

Une particularité de cet opérateur est qu'il peut être utilisé en préfixe ou en suffixe. Cela signifie qu'il peut être écrit avant le nom de la variable (++x) ou après (x++). Dans des expressions simples comme x++ ou ++x, les deux ont exactement le même sens.

Dans d'autres expressions pour lesquelles on évalue le résultat de l'opération d'incrémentation ou de décrémentation, il peut y avoir une différence importante dans le résultat de l'évaluation:

- Dans le cas de la **pré-incrémentation** (++x), le test évalue la valeur finale de x , une fois qu'elle est déjà augmentée.
- Dans le cas de la **post-incrémentation** (x++), la valeur de x est également augmenté, mais le test se fait sur la valeur avant l'incrémentation.

Dans l'*exemple 1,* la valeur attribuée à y est la valeur de x **après** avoir augmenté.

Dans l'*exemple 2,* la valeur attribuée à y est la valeur de x **avant** d'être augmenté.

Exemple 1	**Exemple 2**
x = 3; ***y = ++x;*** ***// x contient 4, y contient 4***	***x = 3;*** ***y = x++;*** ***// x contient 4, y contient 3***

5 OPERATEURS RELATIONNELS ET DE COMPARAISON (==, ! =,>, <,> =, <=)

Deux expressions peuvent être comparées à l'aide des opérateurs relationnels et d'égalité. Par exemple, pour savoir si les deux valeurs sont égales ou si l'une est supérieure à l'autre.

Le résultat d'une telle opération est vrai (**true**) ou faux (**false**) (une valeur booléenne).

Les opérateurs relationnels en C ++ sont :

opérateur	**description**
==	**Égal à**
!=	**Pas égal à**
<	**Inférieur**
>	**Supérieur**
<=	**Inférieur ou égal à**
>=	**Supérieur ou égal à**

Quelques exemples :

```
1 (7 == 5)     // evaluates to false
2 (5 > 4)      // evaluates to true
3 (3 != 2)     // evaluates to true
4 (6 >= 6)     // evaluates to true
5 (5 < 5)      // evaluates to false
```

Bien sûr, ce ne sont pas juste des constantes numériques qui peuvent être comparées, mais toute valeur, y compris, bien sûr, les variables.

Supposons que **a=2**, **b=3** et **c=6**, alors :

```
(a == 5)      // evaluates to false, since a is not equal to 5
(a*b >= c)    // evaluates to true, since (2*3 >= 6) is true
(b+4 > a*c)   // evaluates to false, since (3+4 > 2*6) is false
((b=2) == a) // evaluates to true
```

Dans la dernière expression **((b=2) == a)**, nous avons d' abord attribué la valeur **2** à **b**, puis nous l' avons comparé à **a** (qui contient également la valeur **2**), ce qui donne **true**.

Attention !

L'opérateur d'affectation (opérateur =, avec un signe égal) ne doit pas être confondu avec l'opérateur de comparaison d'égalité (opérateur ==, avec deux signes égal) !

Le premier (=) affecte à la variable sur sa gauche la valeur qui est dans la variable à sa droite , tandis que l'autre (==) compare si les valeurs des deux côtés de l'opérateur sont égales.

Une erreur classique, dans un test ***if*** (à voir au chapitre suivant) :

- Saisir : `if (x = y)` *va affecter y dans x*, et le *test* ***if*** *sera toujours* ***true*** → bug
- Au lieu de : `if (x == y)` qui *compare x et y*, et sera ***true*** *si x est égal à y* → c'était le but

EXERCICE :

Le programme demande à l'utilisateur de saisir tour à tour 2 nombres var1 et var2, puis affiche true (1) si var1==var2, ou affiche false (0) sinon.

6 LES OPERATEURS LOGIQUES (!, &&, ||)

- **L'opérateur ! est l'opération booléenne NOT** : il n'a qu'un seul opérande, à sa droite, et inverse l'état de celle-ci (il renvoie la valeur booléenne opposée de l'évaluation de son opérande).

Par exemple :

```
!(5 == 5)   // evaluates to false because the expression at its right (5 == 5) is true
!(6 <= 4)   // evaluates to true because (6 <= 4) would be false
!true       // evaluates to false
!false      // evaluates to true
```

- **L'opérateur && correspond à l'opération logique booléenne ET** : ce qui donne true si ses deux opérandes sont true, et false autrement. Le panneau suivant montre le résultat pour a&&b :

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

- **L'opérateur || correspond à l'opération logique booléenne OU :** ce qui donne true si au moins l'un de ses opérandes est true, et est ainsi false seulement lorsque les deux opérandes sont false. Voici les résultats possibles d'a||b :

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Par exemple :

```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false )
( (5 == 5) || (3 > 6) )  // evaluates to true ( true || false )
```

Attention !

Lorsque vous utilisez les opérateurs logiques, C ++ évalue seulement ce qui est nécessaire de gauche à droite pour arriver au bon résultat, en ignorant le reste. Par exemple, pour **((5==5)||(3>6))**, C ++ évalue d' abord si 5==5 est true, et comme c'est oui c'est suffisant pour que toute l'expression soit true, donc il ne vérifie jamais si 3>6 est true ou non. Ceci est connu comme le *court-circuit* de l'*évaluation,* et fonctionne comme ceci pour ces opérateurs :

opérateur	**court-circuit**
&&	si l'expression de gauche est false , le résultat combiné est false (l'expression de droite n'est jamais évaluée).
||	si l'expression de gauche est true , le résultat combiné est true (l'expression de droite n'est jamais évaluée).

Cela est surtout important lorsque l'expression de droite a des actions secondaires, tels que la modification des valeurs :

```
if ( (i<10) && (++i<n) ) { /*...*/ }   // note that the condition increments i
```

Ici, l'expression conditionnelle combinée augmenterait i de un, mais seulement si la condition sur la gauche de && est true, parce que sinon, la condition sur le côté droit (++i<n) n'est jamais évaluée.

➔ Eviter de faire des instructions qui font plusieurs choses en même temps

7 L'OPERATEUR CONDITIONNEL TERNAIRE (?)

L'opérateur conditionnel évalue une expression, retournant une valeur si cette expression est évaluée à true, et une autre si l'expression est évaluée comme false .

Sa syntaxe est: **condition ? result1 : result2**
Si la condition est true , l'expression entière est évaluée à result1 , et autrement result2 .

```
7==5 ? 4 : 3      // evaluates to 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3    // evaluates to 4, since 7 is equal to 5+2.
5>3 ? a : b       // evaluates to the value of a, since 5 is greater than 3.
a>b ? a : b       // evaluates to whichever is greater, a or b.
```

EXERCICE :

Vous devez écrire un petit programme qui détermine si une année sera bissextile (elle aura 366 jours) ou pas (365jours) :

Une année est bissextile si :

- L'année est divisible par 4 et non divisible par 100,
- Ou si l'année est divisible par 400.

L'utilisateur saisie l'année (ex : 2015) que l'on récupère dans une variable de type **int** appelée **annee**.

On teste que la saisie soit correcte : un int, et pas négatif (voir cours C++ - Entrées Sorties)

Pour savoir si un nombre est divisible par un autre on utilise **l'opérateur modulo**.

On utilise **les opérateurs logiques** pour tester les divisibilités.

Suivant le résultat des tests on affecte (**opérateur ternaire**) une variable **string** appelée **message** à « Bissextile » ou « Pas bissextile ». On affiche **message**.

8 OPERATEUR VIRGULE (,)

L'opérateur virgule (,) est utilisée pour enchainer deux ou plusieurs expressions entre parenthèses, là où normalement il est prévu une seule expression. Lorsque l'évaluation de l'ensemble des expressions donne une valeur, c'est l'expression la plus à droite qui donne cette valeur.

Exemple : `a = (b=3, b+2);` a pour effet, d'abord affecter la valeur ***3*** à ***b***, puis attribuer ***b+2*** à la variable ***a***. Ainsi, à la fin, la variable ***a*** contient la valeur ***5***, tandis que la variable ***b*** contient la valeur ***3***.

9 OPERATEURS BINAIRES (&, |, ^, ~, <<, >>)

Les opérateurs binaires permettent de modifier les variables en travaillant directement au niveau des bits.

Cela permet soit des gains de rapidité dans certains cas, soit de travailler directement au niveau du hard (registres des microcontrôleurs).

Opérateur	**Equivalent assembleur**	**Description**
&	AND	ET
|	OR	OU
^	XOR	OU exclusif
~	NOT	complément unaire (inversion de bits)
<<	SHL	décalage de bits à gauche
>>	SHR	décalage de bits à droite

Nous approfondirons cela lors d'applications sur cartes électroniques à microcontrôleur, en particulier pour accéder aux Entrées/Sorties sur lesquelles on raccorde des capteurs, relais, etc. ou pour la communication.

10 OPERATEUR DE CASTING DE TYPE EXPLICITE

Les opérateurs de ***casting*** ou ***transtypage*** de type permettent de convertir explicitement une valeur d'un type donné vers un autre type. Il y a plusieurs façons de le faire en C ++. La plus simple, qui a été hérité du langage C, fait précéder l'expression à convertir par le nouveau type enfermé entre parenthèses **()** :

```
1 int i;
2 float f = 3.14;
3 i = (int) f;
```

Le code précédent convertit le nombre à virgule flottante **3.14** en une valeur entière **3** : le reste est perdu.

Ici, l'opérateur de transtypage était **(int)**.

Une autre façon de faire la même chose en C ++ est d'utiliser la notation fonctionnelle, qui fait précéder l'expression à convertir par le type et renfermant l'expression entre parenthèses :

```
i = int (f);
```

Les deux manières de casting sont valides en C ++.

11 SIZEOF : TAILLE DE

Cet opérateur accepte un paramètre, qui peut être soit un type ou d'une variable, et renvoie la taille en octets de ce type ou de l'objet :

```
x = sizeof (char);
```

Ici, x prend la valeur 1, puisque char est un type qui a une taille d'un octet.

12 AUTRES OPERATEURS

Plus tard nous verrons quelques autres opérateurs, comme ceux faisant référence à des pointeurs ou ceux spécifiques à la programmation orientée objet.

13 PRIORITE DES OPERATEURS

Comme pour un calcul à la calculette, une seule expression peut avoir plusieurs opérateurs.

Par exemple : `x = 5 + 7 % 2;`

En C ++, l'expression ci-dessus assigne toujours 6 à la variable x, parce que l'opérateur % a une priorité plus élevée que l'opérateur +, et est donc toujours évaluée avant.

Certaines parties des expressions peuvent être mis entre parenthèses pour remplacer cet ordre de priorité, et faire en sorte que la priorité soit claire. Notez la différence :

```
1 x = 5 + (7 % 2);     // x = 6 (same as without parenthesis)
2 x = (5 + 7) % 2;     // x = 0
```

→ Dans le doute mettez ***toujours*** des parenthèses, même si c'est inutile, le code sera plus lisible.

De la plus grande à la plus petite priorité, les opérateurs du C++ sont évalués dans l'ordre suivant :

Level	Precedence group	Operator	Description	Grouping
1	Scope	`::`	scope qualifier	Left-to-right
2	Postfix (unary)	`++ --`	postfix increment / decrement	Left-to-right
		`()`	functional forms	
		`[]`	subscript	
		`. ->`	member access	
3	Prefix (unary)	`++ --`	prefix increment / decrement	Right-to-left
		`~ !`	bitwise NOT / logical NOT	
		`+ -`	unary prefix	
		`& *`	reference / dereference	
		`new delete`	allocation / deallocation	
		`sizeof`	parameter pack	
		`(type)`	C-style type-casting	
4	Pointer-to-member	`.* ->*`	access pointer	Left-to-right
5	Arithmetic: scaling	`* / %`	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	`+ -`	addition, subtraction	Left-to-right
7	Bitwise shift	`<< >>`	shift left, shift right	Left-to-right
8	Relational	`< > <= >=`	comparison operators	Left-to-right
9	Equality	`== !=`	equality / inequality	Left-to-right
10	And	`&`	bitwise AND	Left-to-right
11	Exclusive or	`^`	bitwise XOR	Left-to-right
12	Inclusive or	`|`	bitwise OR	Left-to-right
13	Conjunction	`&&`	logical AND	Left-to-right
14	Disjunction	`||`	logical OR	Left-to-right
15	Assignment-level expressions	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment / compound assignment	Right-to-left
		`?:`	conditional operator	
16	Sequencing	`,`	comma separator	Left-to-right

Lorsqu'une expression a deux opérateurs avec le même niveau de priorité, le *regroupement* détermine lequel est évalué en premier : soit de gauche à droite ou à droite-gauche.

EXERCICE :

Vous devez écrire un petit programme qui détermine si un nombre est pair, positif et différent de 0.

L'utilisateur saisie le nombre (entier positif ou négatif) que l'on récupère dans une variable de type **int** appelée **nombre**.

On teste que la saisie soit correcte : un int.

Pour savoir si un nombre est divisible par un autre on utilise **l'opérateur modulo**.

On utilise **l'opérateur ternaire** pour tester les conditions (**opérateurs logiques**)

Suivant le résultat du test on affecte une variable **string** appelée **message** que l'on affiche.