

Expressions et opérateurs

1. Expressions

On peut trouver les expressions à plusieurs endroits dans un programme C/C++. Par exemple, on peut trouver une expression à droite du signe égal (=) d'une affectation.

```
nom_variable = expression;
```

Une expression peut être :

- tout simplement une valeur littérale, comme :
7 -12 5.94 -31.89
- ou une formule qui met en œuvre des variables et des opérateurs, comme :
a * 3
a + (b - 1) + 2 * a * b - 1

Exemple

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

int main(){
    int n;
    double x;
    int a, b;

    n = 7;
    x = 5.94;
    a = 3;
    b = 5;
    b = a * 3;
    a = a + (b - 1) + 2 * a * b - 1;

    return 0;
}
```

Remarquez bien qu'une expression est quelque chose d'assez général. On peut utiliser :

+ - * () ...

Types des valeurs littérales

On a déjà vu que toute variable doit avoir un type. Par exemple :

```
int a;
double x;
```

`a` est une variable de type `int` (stocke des nombres entiers) alors que `x` est une variable de type `double` (stocke des nombres réels).

De même, les valeurs littérales ont leurs propres types, juste comme les variables. Par exemple :

La valeur littérale `5` est de type `int`.

La valeur littérale `17.52` est de type `double`.

Remarque

`7.` est équivalent à `7.0`, donc de type `double`.

On peut écrire : `double y = 7.;`

au lieu de : `double y = 7.0;`

Mais il vaut mieux écrire `7.0` au lieu de `7.` (plus lisible).

Affectation d'une valeur décimale à une variable entière

Dans une affectation,

```
variable = expression;
```

l'expression doit retourner (calculer) une valeur de même type que la variable située à gauche du signe d'affectation.

Exemple

```
int a = 12, b;  
b = a + 2;
```

Après évaluation de l'expression "`a + 2`", on obtient une valeur de type `int` (`12 + 2 = 14`) parce que `a` est une variable de type `int` et `2` est une valeur littérale de type `int`. Notez bien que la valeur retournée (`14`), après évaluation de l'expression, est de même type que la variable `b`.

Exemple

```
double x = 2.1;  
x = x * 3.2;
```

Après évaluation de l'expression "`x * 3`", on obtient une valeur de type `double` (`2.1 * 3.2 = 6.72`). Cette valeur (`6.72`) sera affectée à la variable `x` qui est de type `double`.

On a dit qu'une expression doit calculer une valeur de même type que la variable qui se trouve à gauche du signe de l'affectation. Maintenant, que se passe-t-il si on essaye d'affecter une valeur décimale (de type `double`) à une variable entière (de type `int`).

Exemple

```
double x = 2.5;  
int n;  
n = x * 3;
```

Que contient la variable `n` après exécution des lignes de code ci-dessus? Voici la réponse :

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

int main(){
    double x = 2.5;
    int n;
    n = x * 3;
    cout << "n = " << n << endl;

    return 0;
}
```

Mémoire	
	...
x	2.5
n	7
	...

On remarque bien que la variable `n` contient la valeur 7 et non pas 7.5. En fait, le compilateur a converti la valeur littérale 7.5 qui est de type `double` en une valeur de type `int`.

Pourquoi? Quand on affecte une valeur décimale (de type `double`) à une variable de type `int`, la partie fractionnaire est perdue.

Voici un autre exemple :

```
double y = 23.75;
int n = y;
```

Que contient la variable `n`?

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

int main(){
    double y = 23.75;
    int n = y;
    cout << "n = " << n << endl ;

    return 0;
}
```

Mémoire	
	...
y	23.75
n	23
	...

2. Opérateurs

2.1. Opérateurs arithmétiques

En C/C++, comme dans tout langage de programmation, on dispose des quatre opérateurs usuels:

- + Addition
- Soustraction
- * Multiplication
- / Division

Piège de la division entière

Si la division est faite entre des entiers (`int`), il s'agit alors de la division entière.

Exemple

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

int main(){
    double x;
    x = 1 / 2;
    double y = 5 / 2;
    double z = 1 / 2.0;

    cout << "x = " << x << ", y = " << y << ", z = " << z;
    return 0;
}
```

Que contiennent les variables x, y et z?

	Mémoire
	...
X	0
Y	2
z	0.5
	...

Pourquoi?

⇒ `int / int` alors il s'agit de la division entière

`x = 1 / 2;`
 ↑ ↑
 int int

$1 = 0 * 2 + 1$
 ↑
 le résultat de la
 division entière

1		2
1		0

Ce qui veut dire que x va contenir 0 au lieu de 0.5.

Le fait que x est de type `double` ne change rien. Dans une affectation, on évalue ce qui est droite du signe '=', quel que soit ce qui se trouve à gauche.

`double y = 5 / 2;`
 ↑ ↑
 int int

$5 = 2 * 2 + 1$
 ↑
 le résultat de la
 division entière

Passons à la dernière affectation:

`double z = 1 / 2.0;`
 ↑ ↑
 int double

Si l'une des deux valeurs qui apparaissent dans la division est de type `double` alors le compilateur convertit l'autre valeur vers le type `double`.

`z = 1.0 / 2.0`

z va contenir 0.5

Exercice: Ecrire un programme C++ qui calcule la moyenne de deux valeurs entières.

Solution

```
#include<iostream>
using namespace std;
int main(){
    int note1 = 12;
    int note2 = 17;

    double moyenne;
    moyenne = (note1 + note2) / 2;
    cout << "moyenne = " << moyenne;
}
```

Mémoire	
	...
note1	12
note2	17
moyenne	14
	...

On cherche à obtenir la moyenne des deux valeurs 12 et 17, mais la variable `moyenne` vaut 14 au lieu de 14.5. Pourquoi? Parce que `note1` et `note2` sont deux variables de type entier. Donc, il s'agit de la division entière.

Solutions possibles

1.

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

int main(){
    int note1 = 12;
    int note2 = 17;

    double moyenne = (note1 + note2) / 2.0;
    cout << "moyenne = " << moyenne;
}
```

2.

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

int main(){
    int note1 = 12;
    int note2 = 17;

    double moyenne = note1 + note2;
    moyenne = moyenne / 2;
    cout << "moyenne = " << moyenne;
}
```

2.2. Opérateurs d'affectation

En plus du signe d'affectation '=', en C/C++, on trouve d'autres opérateurs d'affectation

+=
-=
*=
/=

Exemple 1

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

int main(){
    int a = 3;
    a += 5;
    cout<< "a = " << a << endl;
}
```

Mémoire

	...
a	3 8
	...

`a += 5;` est équivalent à `a = a + 5;`

Exemple 2

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

int main(){
    int i = 2;
    int j = 3;
    i *= j + 2;
    cout << i << ", " << j; 0;
}
```

Mémoire

	...
i	2 10
j	3
	...

`i *= j + 2;` est équivalent à `i = i * (j + 2);`

2.3. Opérateur modulo %

L'opérateur modulo, noté %, renvoie le reste de la division entière.

Exemple

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

int main(){
    int a;
    a = 11 % 4;
    int b = 12;
    b = b % 4;
    cout << a << ", " << b;
}
```

Mémoire

	...
a	3
b	12 0
	...

`a = 11 % 4;`
 $11 = 2 * 4 + \underline{3}$
3 est le reste de la division de 11 sur 4, alors : $11 \% 4 = 3$

`b = b % 4;`
 $12 = 3 * 4 + \underline{0}$
0 est le reste de la division de 12 sur 4, alors : $12 \% 4 = 0$

11		4
3		2

Remarque : L'instruction suivante est erronée : `int b = 12.0 % 4;`

L'opérateur module (%) n'est disponible que pour le type `int`.

2.4. Opérateurs d'incrémentation (++) et de décrémentation (--)

L'opérateur ++ permet d'incrémenter, c.-à-d, ajouter 1 à une variable.

L'opérateur -- permet décrémentation, c.-à-d, soustraire 1 à une variable.

Exemple 1

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

int main(){
    int x = 5;
    cout << "x = " << x << endl;
    x++;
    cout << "x = " << x << endl;
}
```

	Mémoire
	...
x	5 6
	...

x++; est équivalent à x = x + 1; En d'autres termes, dans un programme, on peut écrire x++ ou x = x + 1.

Exemple 2

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

int main(){
    double y = 5.5;
    cout << "y = " << y << endl;
    y--;
    cout << "y = " << y << endl;
}
```

	Mémoire
	...
y	5.5 4.5
	...

y--; est équivalent à y = y - 1;

Écriture des valeurs littérales avec la notation scientifique

Les valeurs littérales peuvent être écrites en utilisant la notation scientifique. Voici un exemple :

```
double x = 2.5e4;
double y = 3.7e-2;
int n = 2e2;
```

	Mémoire
	...
x	25000
y	0.037
n	200
	...

De façon générale : $aeb = a * 10^b$

Alors :

```
x = 2.5e4 = 2.5 * 104 = 25000
y = 3.7e-2 = 3.7 * 10-2 = 0.037
n = 2e2 = 2 * 102 = 200
```

Fonctions mathématiques

Dans une expression, on peut utiliser les fonctions mathématiques usuelles. Comme par exemple, les fonctions trigonométriques : sin, cos, tan, asin, acos, atan, ...

Exemple 1

```
#include<iostream>
#include<cmath>

using namespace std;

int main(){
    double x = 1;

    double y = sin(x);
    cout << "y = " << y << endl;
}
```

En fait, les fonctions mathématiques, en C++, sont fournies par la bibliothèque `cmath`. En d'autres termes, pour pouvoir utiliser les différentes fonctions mathématiques fournies, il faut ajouter la ligne suivante au début du programme:

```
#include<cmath>
```

Quelques fonctions

<code>pow</code>	<code>pow(x, y)</code> : calcule x^y .
<code>sqrt</code>	<code>sqrt(x)</code> : racine carré (square root) de x .
<code>ceil</code>	<code>ceil(x)</code> : renvoie le plus petit entier qui ne soit pas inférieur à x . Exemple: <code>ceil(2.6) = 3</code>
<code>floor</code>	<code>floor(x)</code> : renvoie le plus grand entier qui ne soit pas supérieur à x . Exemple: <code>floor(2.6) = 2</code>
<code>abs</code>	<code>abs(x)</code> : renvoie la valeur absolue de x .
<code>log</code>	Logarithme népérien ou \ln
<code>log10</code>	Logarithme à base 10 ou \log
<code>exp</code>	Exponentiel
...	...

Exemple 2

```
#include<iostream>
#include<cmath>

using namespace std;

int main(){
    double a = 16;

    double b = 2 * sqrt(a) + 1;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
}
```

Mémoire	
	...
a	16
b	9
	...

Constantes mathématiques

Les constantes suivantes sont aussi définies dans la bibliothèque `cmath` :

`M_PI` : $\pi = 3.14159\dots$

`M_E` : $e = 2.71828\dots$

Exemple : Calcul de la circonférence d'un cercle.

```
#include<iostream>
#include<cmath>

using namespace std;

int main(){
    double rayon;
    double circonference;

    // Lecture du rayon
    cout << "Entrez le rayon : ";
    cin >> rayon;

    // Calculer la circonference
    circonference = 2 * M_PI * rayon;

    // Afficher le resultat
    cout << " La circonference du cercle est : " << circonference << endl;
}
```

Remarquez bien qu'on a utilisé la constante `M_PI` fournie par la bibliothèque `cmath` au lieu de déclarer notre propre constante comme suit :

```
const double PI = 3.14159;
```