Chapitre 3
Les opérations

1. Introduction

Voici quelques questions pour vous aider à synthétiser et retenir des contenus qui nous paraissent essentiels dans cette section :

- Quels sont les opérateurs arithmétiques ?
- Qu'est-ce qu'une affectation combinée ?
- Qu'est-ce qu'une post ou une préincrémentation ?
- Qu'est-ce qu'une post ou une prédécrémentation ?
- Qu'est-ce qu'un cast ?
- Quand utiliser un cast ?
- Y a-t-il des priorités à l'exécution entre les opérateurs ?

2. La notion d'expression

Dans le code informatique, tout élément ou ensemble d'éléments qui fait l'objet d'une évaluation numérique est appelé une expression :

```
a + b,  a / b,  a = b,  &b, !b sont des expressions.
```

Dans ces expressions, +, /, =, & et ! sont des opérateurs et les variables a et b sont les opérandes.

Ces combinaisons de variables avec des opérateurs sont appelées des expressions plutôt que des opérations car il n'y a pas que des opérateurs arithmétiques. De plus, une valeur constante, une variable ou un appel de fonction seuls sont également considérés comme des expressions. Ce sont des expressions élémentaires. Les expressions qui font appel à des opérateurs et plusieurs arguments sont dites expressions composées.

La valeur numérique d'une expression, composée ou élémentaire est le résultat de l'expression elle-même. S'il s'agit d'une opération arithmétique ou d'une affectation, la valeur numérique de l'expression est le résultat de cette opération. S'il s'agit d'un appel de fonction, c'est la valeur de retour de la fonction. L'expression vaut cette valeur, elle est cette valeur et, à ce titre, une expression peut être intégrée dans une expression plus large, dans une affectation ou encore comme opérande dans une opération. Par exemple, le programme :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int a, b=20;
    printf("%d--",a=10);     // affectation
    printf("%d--",a=a*b);    // multiplication
    printf("%d\n",a%b);      // modulo (reste de la division)
    return 0;
}
```

imprime :

```
10--200--0
```

3. Opérations arithmétiques

3.1 Les opérateurs +, -, *, /, %

Les opérateurs arithmétiques sont les suivants :

```
+     plus     :
le résultat de l'expression   a+b   est la somme

-     moins     :
le résultat de l'expression   a-b   est la soustraction

*     multiplier     :
le résultat de l'expression   a*b   est la multiplication
```

```
/     diviser :
le résultat de l'expression   a/b   est la division

% modulo :
le résultat de l'expression   a%b  est le reste de la division entière
de a par b
```

Que donne le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int a=10,b=20,c=0;                          // 1
    c=a+b;                                  // 2
    a=b/c;                                  // 3
    b=a*c;                                  // 4
    printf("a=%d, b=%d, c=%d\n",a,b,c);     // 5
    printf("res=%d\n",c%4);                 // 6
    printf("c=%d\n",c);                     // 7
    return 0;
}
```

```
ligne 1 : a vaut 10, b vaut 20, c vaut 0
ligne 2 : a vaut 10, b vaut 20, c vaut 30
ligne 3 : a vaut  0, b vaut 20, c vaut 30
ligne 4 : a vaut  0, b vaut  0, c vaut 30
ligne 5 : affichage :  a=0, b=0, c=30
ligne 6 : affichage : res=2
ligne 7 : affichage : c=30
```

Remarque

Attention ! À la ligne 6, le résultat de l'opération est affiché mais il n'y a pas d'affectation qui modifie la valeur de c et c conserve sa valeur qui est affichée à la ligne 7.

3.2 Les affectations combinées

Les opérateurs suivants permettent d'associer une opération avec une affectation. L'opération est faite en premier et ensuite l'affectation :

```
a += b          est une contraction de     a = a+b
a -= b          est une contraction de     a = a-b
a *= b          est une contraction de     a = a*b
a /= b          est une contraction de     a = a/b
a %= b          est une contraction de     a = a%b
```

Qu'imprime le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int a=7,b=232,c=4;                              // 1

        c+=a;                                   // 2
        b-=c;                                   // 3
        a%=10;                                  // 4
        printf("a=%d, b=%d, c=%d\n",a,b,c);     // 5
        c+=(b-a);                               // 6
        printf("c=%d\n",c);                     // 7
        return 0;
}
```

```
ligne 1 : a vaut 7, b vaut 232, c vaut 4
ligne 2 : c vaut 11
ligne 3 : b vaut  221
ligne 4 : a vaut  7
ligne 5 : affichage :  a=7 b=221 c=11
ligne 6 : c vaut 225
ligne 7 : affichage : c=225
```

3.3 Opérateurs d'incrémentation et de décrémentation

À maîtriser également, les opérateurs ++ et --. Ils sont très souvent utilisés. Ils s'emploient à la gauche ou à la droite de leur opérande.

Soit par exemple un entier i :

```
i++ et ++i     sont équivalents à i = i+1, de même
i-- et --i     sont équivalents à i = i-1
```

Mais attention ! Pour ces deux opérateurs, il y a toutefois une différence entre l'opérateur placé avant la variable et l'opérateur placé après la variable. Cette différence est sensible lorsque i++ ou ++i sont utilisés dans des expressions, par exemple dans un appel de fonction ou une opération :

```
printf("%d %d",++i, i++);
```

Si l'opérateur ++ est placé avant l'opérande : ++i, alors, dans une expression, la valeur de i est la valeur de i+1, c'est-à-dire que i est incrémenté de 1 avant d'être utilisé dans l'expression.

En revanche si ++ est placé après l'opérande : i++ alors, dans une expression, la valeur de i reste i et i est incrémenté de 1 après que l'expression a été évaluée.

Par exemple, qu'affiche la séquence suivante ?

```
int i=0;
    printf("%d",i++);   // résultat donne  0
    printf("%d",i);     // mais i est incrémenté après avoir été
                        // utilisé et vaut 1 après
    printf("%d",++i);   // i est incrémenté avant son utilisation,
                        // le résultat donne  2
```

Le principe est le même avec l'opérateur --. S'il est placé avant la variable, une soustraction de 1 est opérée avant que la variable soit utilisée dans l'expression. S'il est placé après la variable, la variable est utilisée sans modification dans l'expression puis est décrémentée de 1 après.

3.4 Opérations entre types différents, opérateur de conversion (cast)

Que se passe-t-il si des opérations avec des variables de types différents sont effectuées ? Par exemple :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int entier = 10;
float flottant = 34.9;
int eRes;
float fRes;
    eRes=entier+flottant;
    fRes=entier+flottant;
    printf("eRes : %d, fRes : %f\n", eRes, fRes);      // 1

    eRes=entier / (entier+5);
    fRes=entier / (entier+5);
    printf("eRes : %d, fRes : %f\n", eRes, fRes);      // 2

    eRes=flottant / entier;
    fRes=flottant / entier;
    printf("eRes : %d, fRes : %f\n", eRes, fRes);      // 3
    return 0;
}
```

Voici la règle à connaître pour le savoir :

L'opération arithmétique s'effectue dans le type de l'opérande le plus fort. L'affectation s'effectue dans le type du membre de gauche, celui de la variable qui reçoit la valeur.

1) Le premier affichage donne eRes : 44, fRes : 44.900000.

L'opération est faite en float, le type le plus fort. Avec l'affectation, le résultat est ramené dans le type int à 44 pour eRes et reste en float avec fRes.

2) Le second affichage donne eRes : 0, fRes : 0.000000.

L'opération est faite en int, le type le plus fort. Il n'y a donc pas de chiffre après la virgule et le résultat est 0 dans les deux cas.

3) Le troisième affichage donne eRes : 3, fRes : 3.490000.

L'opération est faite en float : le résultat est converti en int dans le premier cas et reste en float dans le second.

Opérateur de conversion

Parfois, il est nécessaire de pouvoir forcer une opération à s'effectuer dans un type particulier. Pour ce faire, il existe un opérateur qui permet de convertir le type d'une expression dans un autre type. C'est l'opérateur de conversion dit "cast" en anglais. Par exemple, dans le cas //2 ci-dessus :

```
fRes = entier / (entier+5);
```

si l'on a besoin du résultat en float de l'opération, il faut forcer cette opération à s'effectuer en float.

Pour cela, il suffit de spécifier entre parenthèses le type souhaité à gauche d'une des deux variables :

```
fRes = (float)entier / (entier+5);
ou
fRes = entier / (float)(entier+5);
```

Attention toutefois, ce n'est pas le résultat de l'opération qui est converti mais une des deux variables afin que l'opération s'effectue dans un type plus fort. Cela marche aussi dans l'autre sens, si l'on souhaite par exemple forcer une opération faite en float à se faire dans le type int :

```
fRes= (int)flottant / entier;
```

3.5 Priorités entre opérateurs

La machine exécute toujours les instructions les unes à la suite des autres et elle n'exécute jamais deux opérations en même temps. Pour pouvoir effectuer une opération composée de plusieurs calculs, il y a un ordre de priorité entre les opérateurs. Par exemple, l'opérateur * est prioritaire par rapport à l'opérateur + et une expression comme :

```
a + b * c
```

est interprétée comme :

```
a + (b * c)
```

Voici un tableau des priorités pour les opérateurs que nous avons abordés jusqu'ici (le tableau complet comprenant tous les opérateurs du C est donné en annexe et nous nous y référerons dans les chapitres suivants). Chaque ligne correspond à un niveau. Nous avons cinq niveaux et ils sont classés en ordre décroissant de la priorité la plus forte à la priorité la plus faible.

Opérateurs	Fonction de l'opérateur	Associativité
++ -- + - (type) sizeof	pré et post incrémentation pré et post décrémentation signe plus et signe moins conversion de type taille	droite
* / %	multiplication, division, modulo	gauche
+ -	addition et soustraction	gauche
= += - = *= /= %=	affectation et affectations combinées	droite
,	évaluation séquentielle (virgule)	gauche

Mais que se passe-t-il si les opérateurs ont le même niveau de priorité ? Il y a une règle d'associativité qui détermine l'ordre d'évaluation de l'expression et des sous-expressions s'il y en a. Par exemple, * et / sont associatifs à gauche, c'est-à-dire que le calcul est décomposé en partant de la gauche :

```
a * b / c
```

est interprété comme :

```
(a * b) / c
```

Il n'est pas obligatoire de s'appuyer exclusivement sur les priorités définies avec les opérateurs et il est possible de forcer une priorité en ajoutant des parenthèses, par exemple avec :

```
( a + b ) * c
```

c'est a + b qui est effectué en premier et ensuite la multiplication par c.

Dans le doute, il est bon de clarifier une expression avec les parenthèses appropriées.

Qu'imprime le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
```

```
int x,y,z;

        x= -3+4*5-6;
        printf("%d\n",x);

        x= 3+4%5-6;
        printf("%d\n",x);

        x=-3*4%-6/5;
        printf("%d\n",x);

        x= (7+6)%5/2;
        printf("%d\n",x);

        x=2;
        x*=3+2;
        printf("%d\n",x);

        x*=y=z=4;
        printf("%d\n",x);
        return 0;
}
```

Il affiche :

```
11
1
0
1
10
40
```

Pour trouver, le plus simple est d'ajouter les parenthèses correspondant aux priorités, et de décomposer progressivement l'expression dans l'ordre exécuté par la machine.

```
x = -3+4*5-6
x = (-3)+4*5-6              - comme signe
x = (-3)+(4*5)-6            * multiplier
x = ((-3)+(4*5))-6          + addition cause associativité gauche
x = (((-3)+(4*5))-6)        - moins avant affectation
(x = (((-3)+(4*5))-6))      = affectation

ce qui donne :
(x= ((-3+20)-6))
(x= (17-6))
(x= 11)
11
```

Décomposition premier exemple :

```
x = -3+4*5-6
x = (-3)+4*5-6              - comme signe
x = (-3)+(4*5)-6            * multiplier
x = ((-3)+(4*5))-6          + addition cause associativité gauche
x = (((-3)+(4*5))-6)        - moins avant affectation
(x = (((-3)+(4*5))-6))      = affectation

ce qui donne :
(x= ((-3+20)-6))
(x= (17-6))
(x= 11)
11
```

Décomposition deuxième exemple :

```
x = 3+4%5-6
x = 3+(4%5)-6
x = (3+(4%5))-6
x = ((3+(4%5))-6)
(x = ((3+(4%5))-6))
```

Décomposition troisième exemple :

```
x = -3*4%-6/5
x = (-3)*4%(-6)/5
x = ((-3)*4)%(-6)/5
x = (((-3)*4)%(-6))/5
x = ((((-3)*4)%(-6))/5)
(x = ((((-3)*4)%(-6))/5))
```

Décomposition quatrième exemple :

```
x = (7+6)%5/2           // le calcul entre parenthèses est effectué
                        // en premier
x = ((7+6)%5)/2
x = (((7+6)%5)/2)
(x = (((7+6)%5)/2))
```

Décomposition cinquième exemple :

```
x vaut 2
x *= 3+2
x *= (3+2)
(x *= (3+2))
```

Décomposition sixième exemple :

```
x vaut 10
x *= y = z = 4
x *= y = (z = 4)        associativité à partir de la droite
x *= (y = (z = 4))
(x *= (y = (z = 4)))
```