Chapitre 8

Opérateurs arithmétiques et bit à bit

Maintenant que nous savons ce que sont les variables, quels sont les différents types arithmétiques, que nous avons appris comment en déclarer et y mémoriser des données, et que nous sommes en mesure d'afficher le contenu d'une de ces variables, nous pouvons à présent étudier les opérateurs arithmétiques, et bits à bits, permettant de travailler avec, et sur, ces variables.

Tout d'abord, nous étudierons les opérateurs arithmétiques les plus souvent utilisés, ceux permettant d'additionner, diviser, ... Puis, nous verrons ceux qu'il est moins fréquent de rencontrer, mais dont l'utilité ne peut être niée. Naturellement, nous n'oublierons pas de faire mention des formes concises de ces opérateurs.
Ensuite, après un bref rappel sur la logique booléenne, nous étudierons les opérateurs bits à bits, et les opérateurs de décalage de bits.
Pour finir, nous résumerons les règles de priorité entre les différents opérateurs que nous aurons ici étudié.

I:\ Opérateurs arithmétiques

Puisque l'on dispose de variables à même de mémoriser des valeurs, il apparait comme nécessaire de pouvoir manipuler ces valeurs : les additionner, les soustraire, ou leur faire subir des traitements plus... exotiques.

A: Les cinq opérations

Le C utilise les mêmes opérateurs que les mathématiques pour les quatre opérations standards :

- Addition : +
- Soustraction : - (tiret ; touche 6 du clavier azerty)
- Multiplication : * (étoile, à gauche de la touche entrée, sur un clavier azerty)
- Division : / (slash)

En plus de ceux-ci, un cinquième opérateur est défini, qui permet de calculer le modulo, c'est-à-dire le reste de la division entière entre deux nombres. Le symbole utilisé pour cela est le pourcent : %

Ces cinq opérateurs sont des opérateurs binaires. Cela signifie qu'ils travaillent avec deux éléments, un à leur gauche, et un à leur droite.

Voici un petit exemple d'utilisation de ces opérateurs :

```
// C Source File
// Created 09/10/2003; 12:33:35

#include <tigcclib.h>

// Main Function
void _main(void)
{
    clrscr();

    short a = 10;
    short b = 20;
    printf("\n%d+%d=%d", a, b, a+b);
    printf("\n%d-%d=%d", a, b, a-b);
    printf("\n%d*%d=%d", a, b, a*b);
    printf("\n%d/%d=%d", a, b, a/b);
    printf("\n%d%%%d=%d", a, b, a%b); // Pour afficher un symbole '%',
                                      // il faut en mettre deux,
                                      // puisque '%' indique,
                                      // normalement, l'option
                                      // de formattage

    printf("\n%d+%d-50/2=%d", a, b, a+b-50/2);

    ngetchx();
}
```

Exemple Complet

Comme nous pouvons le remarquer, on peut utiliser ces opérateurs avec des variables, ou directement sur des valeurs ; cela revient au même. Cela dit, on utilise la plupart du temps des variables, contenant le résultat d'appels de fonctions, par exemple : un code statique, faisant toujours la même chose, ne servirait pas à grand chose, en dehors des exemples !

Naturellement, ces opérateurs peuvent être séparé des variables par des espaces, des tabulations, des retours à la ligne... enfin, tout ce qui vous semble améliorer la lisibilité, en particulier dans le cas d'expressions longues et complexes !

Les quatre opérateurs "standard" peuvent travailler aussi bien avec des entiers qu'avec des flottants (notez que la division de deux entiers est en fait une division entière : elle renvoie le quotient, arrondi à la valeur inférieure (par exemple, 7/4 renverra un entier valant 1, et non pas un flottant valant 1.75)).
L'opérateur modulo, lui, ne peut travailler qu'avec des nombres entiers.

Notez qu'une division, ou un modulo, par 0 a un résultat non défini par la norme ; sur nos calculatrices, une division par 0 entraîne un plantage.

B: Altération de la priorité des opérateurs

Comme en mathématiques, encore une fois, il est possible d'altérer la priorité des opérateurs, en utilisant des parenthèses. En premier sera évalué ce qui est dans les parenthèses les plus internes.
Par exemple, 3+2*2 donnera 3+(2*2), soit 3+4, soit 7
Alors que (3+2)*2 donnera 5*2, soit 10.
Il est, naturellement, possible d'imbriquer plusieurs niveaux de parenthèses.

D'ailleurs, lorsque vous travaillez avec des expressions complexes, je vous conseille d'utiliser des parenthèses pour clarifier votre code ; elles permettront qu'au premier coup d'oeil, on comprenne l'ordre d'évaluation, sans avoir à réfléchir sur les priorités d'opérateurs !

C: Forme concise des opérateurs

Il nous arrive parfois d'effectuer une opération sur une variable, et d'affecter le résultat de ce calcul dans cette variable...
Par exemple, nous pourrions penser à une écriture de ce type :

```
a = a+20;
```

Ceci peut être écrit de façon plus concise, en utilisant la syntaxe présentée ci-dessous :

```
a += 20;
```

En fait, la plupart des opérateurs arithmétiques admettent une syntaxe de ce genre :
Une expression de la forme "A = A opérateur (B);" équivaut à "A opérateur= B", mais à part le fait que A ne sera évalué qu'une fois.
Les cinqs opérateurs binaires +, -, *, / et % que nous avons ici étudié admettent cette forme, et les cinq opérateurs binaires de manipulation de bits que nous verrons plus bas au cours de ce chapitre l'admettent aussi. Notez que les opérateurs unaires, ceux ne travaillant que sur une seule variable, que nous allons voir juste au-dessous, ne peuvent pas utiliser une syntaxe concise de la sorte !

Encore une fois, j'insiste sur le fait que le C est un langage concis, et que cette particularité est particulièrement appréciée par ses utilisateurs ; il est donc certain que vous rencontrerez ces formes si vous parcourez des codes sources, et, donc, il serait utile que vous les reteniez...

D: Opérateurs arithmétiques unaires

Il existe deux opérateurs, en C, permettant de forcer le signe d'une valeur. Ce sont tous deux des opérateurs unaire, ce qui signifie qu'ils ne travaillent que sur une donnée.

Pour obtenir l'opposé d'une valeur, on utilise celle-ci :

```
-a;
```

Notez que cela revient à soustraire la valeur à 0... mais en l'écrivant de façon plus propre.

De façon symétrique, l'opérateur + unaire a été défini. Il renvoie la valeur de l'opérande sur laquelle il travaille.
Notez qu'il ne renvoie pas la valeur absolue ! Utiliser l'opérateur + unaire sur une valeur négative renverra... une valeur négative !

Voici un petit exemple illustrant l'utilisation de ces deux opérateurs, sur une valeur positive, et sur une donnée négative :

```
// C Source File
// Created 08/10/2003; 14:17:20

#include <tigcclib.h>

// Main Function
void _main(void)
{
    clrscr();

    short a = 10;
    short b = -20;

    printf("+a=%d", +a);
    printf("\n+b=%d", +b);

    printf("\n-a=%d", -a);
    printf("\n-b=%d", -b);

    ngetchx();
}
```

Exemple Complet

Si vous exécutez cet exemple, prêtez tout particulièrement attention à la seconde ligne affichée ! Et retenez en mémoire la remarque que j'ai fait juste avant de présenter ce code source !

E: Incrémentation, et Décrémentation

Lorsqu'il s'agit d'ajouter un à une valeur, c'est-à-dire de l'incrémenter, ou de lui retrancher un, c'est-à-dire la décrémenter, il est possible d'utiliser respectivement les opérateurs ++, ou -- (deux plus à la suite, ou deux moins à la suite).
Ces opérateurs peuvent être placés avant, ou après, leur opérande. Dans le premier cas, on parlera d'incrémentation préfixée, et, dans le second, d'incrémentation postfixée.

Lorsque l'on utilise un opérateur postfixé, la valeur de la variable nous est renvoyée, et, seulement ensuite, elle est incrémentée (ou décrémentée, selon le cas).
Lorsque l'on travaille avec un opérateur préfixé, la variable est incrémentée (ou décrémentée), et, ensuite, sa valeur nous est renvoyée.

Pour que vous compreniez bien ce qui se passe, et comment, voici un exemple :

```
// C Source File
// Created 08/10/2003; 14:17:20

#include <tigcclib.h>

// Main Function
void _main(void)
{
    clrscr();

    short a;

    a = 10;
    printf("1: a++ => %d", a++);
    printf("\n2: a = %d", a);

    a = 10;
    printf("\n3: a-- => %d", a--);
    printf("\n4: a = %d", a);

    a = 10;
    printf("\n5: a = %d", a);
    printf("\n6: ++a => %d", ++a);

    a = 10;
    printf("\n7: a = %d", a);
    printf("\n8: --a => %d", --a);

    ngetchx();
}
```

Exemple Complet

Ce que je vous recommande de faire est de l'exécuter, et de suivre en parallèle le résultat de son exécution et son code source.

F: A ne pas faire !

Il est tout à fait possible d'utiliser des écritures telles que celle-ci sans que le compilateur ne s'en offusque :

```
c = a++ + ++b;
```

Ou que celles-là :

```
c = a++ + +(++b);
c = a++ + + ++b;
c = a-- - -(--b);
```

Ces écritures nous montrent bien que l'on peut mettre un paquet de plus ou de moins à la suite... tout en sachant qu'il faut, par endroit, utiliser des espaces ou des parenthèses, pour que le compilateur soit à même de faire la différence entre les opérateurs d'incrémentation, d'addition, et de signe...
Cela dit, c'est assez peu lisible ! Je vous conseille donc de bannir totalement ce genre de syntaxe, et de préférer étaler ceci sur plusieurs lignes, que vous n'aurez pas de mal à relire ! Parce que pour comprendre ces expressions, c'est quelque peu complexe, et on ne peut s'empêcher d'hésiter quand à l'ordre dans lequel les opérations seront menées (notamment, a sera incrémenté avant, ou après de faire la somme ? Je dois reconnaître que je suis incapable de le dire avec certitude... si j'ai bonne mémoire, la norme ne le dit même pas, et on se retrouve dans la même situation que celle que je présente en dessous)

Pour l'écriture suivante, je suis sûr de moi : elle est totalement indéfinie par la norme (ce n'est pas la seule, mais c'est la plus évidente) :

```
c = a++ + a++;
```

L'ordre d'évaluation n'est pas clairement défini, et on ne sait pas dans quel ordre les incrémentations et la somme seront effectuées...
D'ailleurs, GCC vous préviendra que L'opération concernant a peut être indéfinie.
Dans un programme, c peut au final valoir une certaine valeur... et dans un autre programme, une autre valeur !
Ce type d'écriture, présentant ce genre d'effets de bords est à bannir !

II:\ Opérateurs travaillant sur les bits

En plus des opérateurs arithmétiques usuels, que nous utilisons tout le temps, le C définit des opérateurs travaillant sur les bits. Certains de ces opérateurs permettent de travailler bit à bit sur des valeurs ; d'autres permettent de décaler, vers la droite ou la gauche, les bits d'une donnée.
Nous commencerons cette partie par un bref rappel de logique booléenne, puis nous présenterons les opérateurs correspondants.

A: Rappels de logique booléenne

La logique booléenne n'utilise que deux valeurs, 0, et 1, qui correspondent tout à fait à ce que l'on emploie lorsque l'on travaille en binaire.
Trois opérations, voire quatre, d'algèbre de Boole nous seront utiles lorsque nous programmerons en C : le OU (or, en anglais), le OU exclusif (xor, ou eor, selon les habitudes du programmeur ; en C, on utilise généralement xor, alors qu'en Assembleur, on aura plus tendance à utiliser eor, par exemple ; mais ce sont deux appellations qui renvoient à la même chose), et le ET (and). En plus de cela, il est possible de définir le NON (not).

Voici les tables de ces opérations :

a	b		a OR b		a XOR b		a AND b		NOT a
0	0		0		0		0		1
0	1		1		1		0		1
1	0		1		1		0		0
1	1		1		0		1		0

a OR b vaut 1 si a, ou b, ou les deux, valent 1.
a XOR b vaut si a ou b, mais pas les deux à la fois, valent 1.
a AND b vaut 1 si a et b valent tous les deux 1.
Et NOT a vaut 1 si a vaut 0, et, inversement, 0 si a vaut 1.

B: Opérateurs bits à bits

Le C propose des opérateurs bits à bits permettant de faire ceci directement sur les bits composant une, ou deux, valeurs, selon l'opérateur. Notez que ces opérateurs ne travaillent qu'avec des valeurs, on des variables, de type entier : ils ne peuvent pas être utilisés sur des flottants !

Pour effectuer un OR bit à bit, vous utiliserez l'opérateur | (le pipe sous les systèmes UNIX, accessible par Alt-Gr 6 sur un clavier azerty).
Pour le XOR, il convient d'utiliser le ^ (accent ciconflexe).
Pour le AND, c'est un & qu'il revient d'écrire (é commercial, touche 1 du clavier azerty).
Et enfin, pour le NOT, c'est le ~ (tilde, soit Alt-Gr 2).

Notez que |, ^, et & sont des opérateurs binaires, alors que le ~ est un opérateur unaire. Les trois premiers, en tant qu'opérateurs binaires, disposent d'une écriture abrégée pour les affectations, telle &=, par exemple.

Ci-dessous, un extrait de code source présentant quelques emplois de ces quatre opérateurs :

```
short a=10,
      b=20,
      c;
c = a | b;
c = a & b;
c = a ^ b;
c = ~a;
```

Je reconnais qu'il est assez rare pour un débutant d'employer ces opérateurs, mais ils sont souvent fort utiles pour certains types de programmes ; ne voulant pas avoir à revenir dessus plus tard, au risque de les oublier, j'ai préféré les présenter dans ce chapitre.

C: Opérateurs de décalage de bits

Un autre type permet de manipuler directement les bits d'une variable, ou d'une donnée. Ils s'agit des deux opérateurs de décalage de bits.

Le premier, >> (deux fois de suite le signe supérieur) permet de décaler les bits d'un nombre vers la droite, et le second, << (deux signes inférieur successifs), est sa réciproque, c'est-à-dire qu'il décale vers la gauche.

Ces opérateurs s'utilisent de la façon suivante :
valeur OP N
Où valeur est une donnée, une variable, une valeur, ..., et N le nombre de bits dont on souhaite décaler les bits de la donnée.
Notez que décaler de N bits vers la droite revient à diviser par 2 puissance N, et décaler de N bits vers la gauche correspond à une multiplication par 2 puissance N.

Voici un petit exemple, pour vous donner la syntaxe d'utilisation, en clair :

```
short a = 0b00011100;
short b;
b = a << 2;
// b vaut maintenant 0b01110000
b = a >> 4;
// b vaut maintenant 0b00000001
a <<= 3;
// a vaut maintenant 0b11100000
```

La remarque faite plus haut pour les opérateurs bits à bits est ici aussi valable.

III:\ Résumé des priorités d'opérateurs

Pour terminer ce chapitre, je pense que récapituler les priorités des opérateurs que nous avons ici vu peut être une très bonne chose...
C'est pour ce genre de choses qu'un livre utilisé comme référence peut être une bonne chose : même si aucun livre ne traite de la programmation en C pour TI, le C pour PC et le C pour TI sont le même langage ! Moi-même, malgré plusieurs années d'expérience en programmation en langage C, j'ai préféré me référer à un livre pour écrire cette partie, afin de ne pas écrire quoi que ce soit d'erroné.
Dans le doute, il demeure possible d'utiliser des parenthèses pour forcer un ordre de priorité ; je vous conseille d'en utiliser assez souvent, dès que les choses ne sont pas tout à fait certaines dans votre esprit, afin de rendre votre source plus clair, et plus facile à comprendre si vous êtes un jour amené à le relire ou à le distribuer...

Le tableau ci-dessous présente les différents opérateurs que nous avons vu (j'espère ne pas en oublier, ni en rajouter que nous n'ayions pas encore étudié). La ligne la plus haute du tableau regroupe les opérateurs les plus prioritaire, et les niveaux de priorité diminuent au fur et à mesure que l'on descend dans le tableau. Lorsque plusieurs opérateurs sont sur la même ligne, cela signifie qu'ils ont le même niveau de priorité.

Opérateurs, du plus prioritaire au moins prioritaire

```
()
~ ++ -- + - sizeof
* / %
+ -
<< >>
&
^
|
= += -= *= /= %= &= ^= |= <<= >>=
```

Notez que les opérateurs + et - unaires sont plus prioritaires que leurs formes binaires.
N'oubliez pas que le C n'impose pas l'ordre dans lequel les opérandes d'un opérateur sont évaluées, du moins, pour les opérateurs que nous avons jusqu'à présent vu ; le compilateur est libre de choisir ce qu'il considère comme apportant la meilleure optimisation. Il en va de même pour l'ordre d'évaluation des arguments passés à une fonction.

Maintenant que nous en avons terminé avec tous ces opérateurs, nous allons passer à un chapitre qui nous permettra d'étudier ce que sont les structures de contrôles, lesquelles sont proposées en C, et comment les utiliser.
Cependant, avant de pouvoir nous lancer dans le vif du sujet, il nous faudra étudier encore quelques nouveaux opérateurs, qui nous seront indispensables pour la suite...