Les opérations mathématiques

c

Dernière mise à jour : dimanche 22 septembre 2019 à 14h47

Auteurs : Lucas-84 Taurre informaticienzero

Catégorie : Programmation et algorithmique

Manipulations basiques des entrées/sorties Tests et conditions

Nous savons désormais déclarer, affecter et initialiser une variable, mais que diriez-vous d'apprendre à réaliser des opérations dessus ? Il est en effet possible de réaliser des calculs sur nos variables, comme les additionner, les diviser voire des opérations plus complexes. C'est le but de cette sous-partie. Nous allons donc enfin transformer notre ordinateur en grosse calculette programmable !

Les opérations mathématiques de base

Jusqu'à présent, nous nous sommes contenté d'afficher du texte et de manipuler très légèrement les variables. Voyons à présent comment nous pouvons réaliser quelques opérations de base. Le langage C nous permet d'en réaliser cinq :

- l'addition (opérateur `+`) ;
- la soustraction (opérateur `-`) ;
- la multiplication (opérateur `*`) ;
- la division (opérateur `/`) ;
- le modulo (opérateur `%`).

> Le langage C fournit bien entendu d'autres fonctions mathématiques et d'autres opérateurs, mais il est encore trop tôt pour vous les présenter.

Division et modulo

Les quatre premières opérations vous sont connues depuis l'école primaire. Cependant, une chose importante doit être précisée concernant la division : quand les deux nombres manipulés sont des entiers, il s'agit d'une division entière. Autrement dit, le quotient sera un entier et il peut y avoir un reste. Par exemple, 15 ÷ 6, ne donnera pas 2,5 (division réelle), mais un quotient de 2, avec un reste de 3.

```
1 printf("15 / 6 = %d\n", 15 / 6);
```

Résultat

```
1 15 / 6 = 2
```

Le modulo est un peu le complément de la division entière : au lieu de donner le quotient, il renvoie le reste d'une division euclidienne. Par exemple, le modulo de 15 par 6 est 3, car 15 = 2 × 6 + 3.

```
1 printf("15 % 6 = %d\n", 15 % 6);
```

Résultat

```
1 15 % 6 = 3
```

Avec des flottants, la division se comporte autrement et n'est pas une division avec reste. La division de deux flottants donnera un résultat « exact », avec potentiellement plusieurs chiffres après la virgule.

```
1 printf("15 / 6 = %f\n", 15. / 6.);   /* En C, ce n'est pas la même chose que 15 / 6 
```

Résultat

```
1 15 / 6 = 2.500000
```

Utilisation

Il est possible d'affecter le résultat d'une expression contenant des calculs à une variable, comme lorsque

nous les utilisons comme argument de `printf()`.

```c
#include <stdio.h>

int main(void)
{
    int somme = 5 + 3;

    printf("5 + 3 = %d\n", somme);
    return 0;
}
```

Résultat

```
5 + 3 = 8
```

Toute opération peut manipuler :

- des constantes ;
- des variables ;
- les deux à la fois.

Voici un exemple avec des constantes.

```c
#include <stdio.h>

int main(void)
{
    printf("2 + 3 = %d\n", 2 + 3);
    printf("8 - 12 = %d\n", 8 - 12);
    printf("6 x 7 = %d\n", 6 * 7);
    printf("11 % 4 = %d\n", 11 % 4);
    return 0;
}
```

Résultat

```
2 + 3 = 5
8 - 12 = -4
6 x 7 = 42
11 % 4 = 3
```

Un autre avec des variables.

```c
int a = 5;
int b = 3;
int somme = a + b;

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

Résultat

```
5 + 3 = 8
```

Et enfin, un exemple qui mélange variables et constantes.

```c
int a = 5;
int b = 65;

printf("%d\n", b / a * 2 + 7 % 2);
```

Résultat

```
27
```

La priorité des opérateurs

Dans l'exemple précédent, nous avons utilisé plusieurs opérations dans une même ligne de code, une même expression. Dans ces cas-là, faites attention à la **priorité des opérateurs** ! Comme en mathématiques, certains opérateurs passent avant d'autres : les opérateurs `* / %` ont une priorité supérieure par rapport aux opérateurs `+ -`.

Dans le code ci-dessous, c'est `c * 4` qui sera exécuté d'abord, puis `b` sera ajouté au résultat. Faites donc attention sous peine d'avoir de mauvaises surprises. Dans le doute, ajoutez des parenthèses.

```c
a = b + c * 4;
```

Les expressions

Il est possible de combiner opérateur, variables et constantes pour former des **expressions**, des lignes de code qui sont évaluées et produisent un résultat. Les lignes de code suivantes sont toutes des expressions.

```c
"Bonjour !"
2 + 3
10 > 2
```

Généralement, une expression ne peut être écrite seule et doit faire partie d'une **instruction**. La frontière entre instruction et expression est assez floue puisqu'une instruction peut être composée de nombreuses expressions. Le code ci-dessous est un exemple d'instruction qui est *quasi* une expression (on parle d'**expression-instruction**).

```c
x = 2 + 3;
```

Nous donnons en effet un ordre à l'ordinateur (« affecte la valeur 2 + 3 à `x` »), mais c'est aussi une expression qui produit la valeur 5 comme résultat. Vous verrez qu'en C, la majorité des lignes de code sont des instructions-expressions. C'est ce qui est appelé la **programmation impérative**. C'est le choix des concepteurs du langage, mais ce n'est pas la seule possibilité (il en existe d'autres, mais ça ne nous concerne pas en tant qu'utilisateurs du C).

Type d'une expression

Vous avez sans doute remarqué que nous avons utilisé directement des expressions (`2 + 3` par exemple) comme argument de la fonction `printf()`. Rien de bien surprenant me direz-vous… À un détail près : quel indicateur de format doit-on utiliser ? Autrement dit, quel est le type d'une expression ? D'ailleurs, ont-elles un type ?

Hé bien, oui. Tout comme les variables, les expressions ont un type. Ce dernier dépend toutefois du type des éléments qui la compose. En l'occurrence, une constante entière comme `2` ou `3` est par défaut de type `int`. Le résultat d'une somme, par exemple, sera donc également de type `int`. Les constantes flottantes comme `5.` ou `78.0` sont, elles, de type `double` et le résultat d'une opération sera alors de type `double`.

> D'accord, mais si j'additionne un `int` avec un `double`, cela me donne quoi ?

Heureusement pour nous, la norme[1] a prévu ces différents cas et a fixé des règles :

1. si un opérande est de type `long double`, le résultat est de type `long double` ;
2. si un opérande est de type `double`, le résultat est de type `double` ;
3. si un opérande est de type `float`, le résultat est de type `float` ;
4. si un opérande est de type `unsigned long long`, le résultat est de type `unsigned long long` ;
5. si un opérande est de type `long long`, le résultat est de type `long long` ;
6. si un opérande est de type `unsigned long`, le résultat est de type `unsigned long` ;
7. si un opérande est de type `long`, le résultat est de type `long` ;
8. si un opérande est de type `unsigned int`, le résultat est de type `unsigned int` ;
9. le résultat est de type `int`.

Type d'une constante

> *Heu…* D'accord, mais vous venez de dire que les constantes entières étaient de type `int` et que les constantes flottantes étaient de type `double`. Du coup, je fais comment pour obtenir une constante d'un autre type ?

À l'aide d'un suffixe. Celui-ci se place à la fin de la constante et permet de modifier son type. En voici la liste complète.

Type	Suffixe

Type	Suffixe
u ou U	**unsigned**
l ou L	**long**
ll ou LL	**long long**
f ou F	**float**
l ou L	**long double**

Notez que les suffixes `L` (ou `l`) et `U` (ou `u`) peuvent être combinés.

Faites attention que le suffixe `L` n'a pas le même effet suivant qu'il est apposé à une constante entière ou réelle. Dans le premier cas la constante sera de type `long` alors que dans le second elle sera de type `long double`.

Exemple

Allez, un petit récapitulatif.

```c
#include <stdio.h>


int main(void)
{
    /* long double + int = long double */
    printf("78.56 + 5 = %Lf\n", 78.56L + 5);

    /* long + double = double */
    printf("5678 + 2.2 = %f\n", 5678L + 2.2);

    /* long + unsigned long = unsigned long */
    printf("2 + 5 = %lu\n", 2L + 5UL);

    /* long long + int = long long */
    printf("1 + 1 = %lld\n", 1LL + 1);
    return 0;
}
```

Résultat

```
78.56 + 5 = 83.560000
5678 + 2.2 = 5680.200000
2 + 5 = 7
1 + 1 = 2
```

Nous vous conseillons d'opter pour les lettres majuscules qui ont l'avantage d'être plus lisibles.

Conversions de types

La **conversion de type** est une opération qui consiste à changer le type de la valeur d'une expression. Ainsi, il vous est par exemple possible de convertir une valeur de type `float` en type `int`.

Perte d'informations

Une perte d'informations survient quand le type d'une variable est converti vers un autre type ayant une capacité plus faible *et* que celui-ci ne peut pas contenir la valeur d'origine. Si, par exemple, nous convertissons un `double` de cent chiffres en un `int`, il y a perte d'informations, car le type `int` ne peut pas contenir un nombre de cent chiffres. Retenez donc bien cette assertion : une conversion d'un type T vers un type S de plus faible capacité entraîne une perte d'informations (une perte de précision pour les nombres).

Les conversions peuvent être vicieuses et doivent être manipulées avec précaution, au risque de tomber sur des valeurs fausses en cas de perte d'informations. Nous découvrirons d'ici quelques chapitres comment connaître la taille d'un type pour éviter ces pertes d'informations.

Deux types de conversions

Il existe deux types de conversions : les conversions explicites et les conversions implicites.

Les conversions explicites

Les **conversions explicites** sont des conversions demandées par le programmeur. Elles s'utilisent suivant ce modèle.

```
(<Type>)<Expression>
```

Voici par exemple un code où nous demandons explicitement la conversion d'un `double` en `int`.

```
int a;
const double pi = 3.14;

a = (int)pi;
```

La valeur de `pi` reste inchangée, elle vaudra toujours 3.14 dans la suite du programme. Par contre, `a` vaut maintenant 3, puisque la valeur de `pi` a été convertie en `int`.

Conversions implicites

Les **conversions implicites** sont des conversions spécifiées par la norme et réalisées automatiquement par le compilateur. En général, cela ne pose pas de problèmes et cela est même désirable. Par exemple, il y a toujours une conversion implicite dans le cadre d'une affectation.

Ainsi, la conversion explicite du code précédent n'est en fait pas nécessaire.

```
int a;
```

```
2 const double pi = 3.14;
3
4 /* Il y a conversion implicite du type double vers le type int. */
5 a = pi;
```

1. ISO/IEC 9899:201x, doc. N1570, § 6.3.1.8, Usual arithmetic conversions, p. 52. ↩

Sucre syntaxique

Dans les expressions vues au-dessus, nous avons utilisé des affectations pour sauvegarder le résultat de l'opération dans une variable. Les expressions obtenues ainsi sont assez longues et on peut se demander s'il existe des moyens pour écrire moins de code. *Hé* bien, le langage C fournit des écritures pour se simplifier la vie. Certains cas particuliers peuvent s'écrire avec des raccourcis, du « **sucre syntaxique** ».

Les opérateurs combinés

Comment vous y prendriez-vous pour multiplier une variable par trois ? La solution qui devrait vous venir à l'esprit serait d'affecter à la variable son ancienne valeur multipliée par trois.

```
1 int variable = 3;
2
3 variable = variable * 3;
4 printf("variable * 3 = %d\n", variable);
```

Résultat

```
1 variable * 3 = 9
```

Ce qui est parfaitement correct. Cependant, cela implique de devoir écrire deux fois le nom de la variable, ce qui est quelque peu pénible et source d'erreurs. Aussi, il existe des opérateurs combinés qui réalisent une affectation et une opération en même temps.

Opérateur combiné	Équivalent à
variable += nombre	variable = variable + nombre
variable -= nombre	variable = variable - nombre
variable *= nombre	variable = variable * nombre
variable /= nombre	variable = variable / nombre
variable %= nombre	variable = variable % nombre

Avec le code précédent, nous obtenons ceci.

```
1 int variable = 3;
```

```
int variable = 3;

variable *= 3;
printf("variable * 3 = %d\n", variable);
```

Résultat

```
variable * 3 = 9
```

L'incrémentation et la décrémentation

L'**incrémentation** et la **décrémentation** sont deux opérations qui, respectivement, ajoute ou enlève une unité à une variable. Avec les opérateurs vu précédemment, cela se traduit par le code ci-dessous.

```
variable += 1; /* Incrémentation */
variable -= 1; /* Décrémentation */
```

Cependant, ces deux opérations étant très souvent utilisées, aussi elles ont droit chacune à un opérateur spécifique disponible sous deux formes :

- une forme **préfixée** ;
- une forme **suffixée**.

La forme préfixée s'écrit comme ceci.

```
++variable; /* Incrémentation */
--variable; /* Décrémentation */
```

La forme suffixée s'écrit comme cela.

```
variable++; /* Incrémentation */
variable--; /* Décrémentation */
```

Le résultat des deux paires d'opérateurs est le même : la variable `variable` est incrémentée ou décrémentée, *à une différence près* : le résultat de l'opération.

1. Dans le cas de l'opérateur préfixé (`--variable` ou `++variable`), le résultat sera la valeur de la variable augmentée ou diminuée d'une unité.
2. Dans le cas de l'opérateur suffixé (`variable--` ou `variable++`), le résultat sera la valeur de la variable.

Illustration !

```
#include <stdio.h>

int main(void)
{
    int x = 1;

    int y = 1;
```

```
    int a = x++;
    int b = ++y;

    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    return 0;
}
```

Résultat

```
a = 1
b = 2
x = 2
y = 2
```

Comme vous pouvez le constater, la valeur de l'expression `x++` est `1` alors que la valeur de l'expression `++y` est 2. Cela étant, dans les deux cas, les variables `x` et `y` ont bien été incrémentées.

Exercices

Vous êtes prêts pour un exercice ?

Essayez de réaliser une minicalculatrice qui :

- dit « bonjour » ;
- demande deux nombres entiers à l'utilisateur ;
- les additionne, les soustrait, les multiplie et les divise (au millième près) ;
- dit « au revoir ».

Un exemple d'utilisation pourrait être celui-ci.

```
Bonjour !
Veuillez saisir le premier nombre : 4
Veuillez saisir le deuxième nombre : 7
Calculs :
        4 + 7 = 11
        4 - 7 = -3
        4 * 7 = 28
        4 / 7 = 0.571
Au revoir !
```

Bien, vous avez maintenant toutes les cartes en main, donc : au boulot ! 

Afficher/Masquer le contenu masqué

Vous y êtes arrivé sans problèmes ? Bravo ! Dans le cas contraire, ne vous inquiétiez pas, ce n'est pas grave. Relisez bien tous les points qui ne vous semblent pas clairs et ça devrait aller mieux.

Dans le chapitre suivant, nous nous pencherons sur les **conditions**.

En résumé

1. Le C fournit des opérateurs pour réaliser les opérations mathématiques de base ;
2. Dans le cas où une division est effectuée entre des nombres entiers, le quotient sera également un entier ;
3. La priorité des opérateurs est identique à celle décrite en mathématiques ;
4. En cas de mélange des types dans une expression, le C prévoit des règles de conversions ;
5. Le type d'une constante peut-être modifié à l'aide de suffixes ;
6. Il est possible de convertir une expression d'un type vers un autre, certaines conversions sont implicites ;
7. Le C fourni des opérateurs combinés et des opérateurs pour l'incrémentation et la décrémentation afin de gagner en concision.

Manipulations basiques des entrées/sorties Tests et conditions

mmaire

I Les bases du langage C

rtager

Twitter

Facebook

Diaspora*

Envoyer par mail

écharger

HTML (578,4 Kio)

LaTeX (737,2 Kio)

EPUB (398,2 Kio)

Archive (2,2 Mio)