Chapitre 4 : Expressions

Informatique de base
2013-2014

Sup Galilée

Qu'est-ce qu'une expression ?

Une expression est une phrase formée à l'aide de constantes littérales, de noms de variables et d'opérateurs. Une expression est destinée à être évaluée (autrement dit une expression possède une valeur).

Par exemple, dans un environnement où la variable `x` est visible et a la valeur 8, les phrases suivantes :

```
12
x
(x+12)-3
(x > 4) && (x < 10)
```

sont des expressions qui ont pour valeur respective 12, 8, 17 et 1 ("vrai"). Les deux premières expressions sont des expressions atomiques et les deux dernières sont des expressions composées.

En C toute expression a obligatoirement :

- un type,
- une valeur

et éventuellement

- une adresse qui est celle de la case mémoire contenant la valeur de cette expression,
- un effet de bord qui change l'état du programme.

Valeurs gauches et droites

On distingue deux catégories d'expressions :

- les valeurs gauches qui sont des noms de donnée modifiable. Une valeur gauche est donc associée à une case de la mémoire et a une adresse (de cette case mémoire).

- les valeurs droites qui sont des expressions qui ne sont pas associées à une case de la mémoire et qui n'ont donc pas d'adresse. Une valeur gauche peut être utilisée comme une valeur droite.

Notations : type et valeur

Dans ce cours, si `exp` désigne une expression du langage C, alors nous noterons :

- type(`exp`), le type de `exp`,
- val(`exp`), la valeur de `exp`.

Une expression peut être mise entre parenthèses. Si `exp` est une expression, alors `(exp)` est également une expression de même type et de même valeur que `exp`.

Nous allons voir comment on détermine le type et la valeur d'une expression.

Expressions atomiques : constantes littérales

Une constante littérale c est une expression telle que :

- type(c) = type de la valeur représentée par c,
- val(c) = valeur représentée par c.

Par exemple, type(12) est `int` et val(12) est bien sûr 12.

Expressions atomiques : variables

Un nom de variable `v` est une expression telle que :

- type(`v`) = type déclaré pour `v`,
- val(`v`) = valeur de `v` obtenue comme on l'a vu au chapitre 2.

Par exemple, si on est dans un environnement où `x` est une variable de type `float` et de valeur 8.5, alors `x` est une expression de type `float` et de valeur 8.5.

Expressions composées

Les expressions sont composées à l'aide d'opérateurs.

On distingue les

- opérateurs arithmétiques (addition, multiplication, division, etc.),
- opérateurs booléens (ou logique, ...),
- l'affectation $=$,
- le changement de type.

Les opérateurs de manipulation des valeurs composées (structures et tableaux) seront étudiés ultérieurement, comme ceux permettant la manipulation des pointeurs.

L'évaluation d'une expression et le calcul de son type peuvent entraîner des conversions de type. Il faut en plus savoir que l'ensemble des types entiers est muni d'une relation d'ordre et qu'il en est de même des types flottants (nous verrons cela dans la section concernant la conversion de type).

Expressions composées : le moins unaire

L'opérateur - calcule l'opposé d'un nombre.

Si `exp` ext une expression de type numérique, alors `-exp` est une expression telle que :

- type(`-exp`) = type(`exp`),
- val(`-exp`) = -val(`exp`).

Par exemple si `x` est une variable de type `float` et de valeur 3.2, alors `-x` est de type `float` et de valeur -3.2.

Expressions composées : opérateurs arithmétiques binaires

Les opérateurs $+$ $-$ $*$ $/$ calculent respectivement la somme, la différence, le produit et le quotient de deux nombres. L'opérateur % calcule le reste de la division de deux nombres **entiers**.

Si `exp1` et `exp2` sont des expressions de type numérique, alors

```
exp1 + exp2
exp1 - exp2
exp1 * exp2
exp1 % exp2
```

sont des expressions telles que (où op est l'un des opérateurs $+, -, *, /$) type(`exp1 op exp2`)=T où T est le plus grand des types `int`, type(`exp1`) et type(`exp2`), et val(`exp1 op exp2`)= v1 op v2 où v1 et v2 sont les résultats des conversions de val(`exp1`) et de val(`exp2`) en T.

Expressions composées : opérateurs arithmétiques binaires, exemples

val(((4.0 + 2.25)*2.0))=12.5

val(15/2)=7 de type `int`

val(15%2)=1 de type `int`.

Expressions composées : opérateurs de comparaison

Les opérateurs de comparaison == != < <= > >= testent respectivement l'égalité, la différence, l'infériorité, l'infériorité ou égalité, la supériorité et la supériorité ou égalité de deux nombres.

Notons op l'un de ces opérateurs de comparaison, et soient `exp1`, `exp2` des expressions numériques. Alors type(`exp1 op exp2`) = `int`, et val(`exp1 op exp2`)=0 si la comparaison `exp1 op exp2` est fausse et val(`exp1 op exp2`)=1 si la comparaison `exp1 op exp2` est vraie.

Par exemple, val(1 == 9)=0 alors que val(1 != 9)=1.

Expressions composées : négation

L'opérateur ! calcule la négation d'un booléen.

Si `exp` est une expression, alors `!exp` est une expression telle que

- type(`exp`)=`int`.
- val(`!exp`)=0 si val(`exp`)=1, 1 sinon.

Expressions composées : conjonction et disjonction

Les opérateurs || et && calculent respectivement la conjonction (et logique) et la disjonction (ou logique) de deux expressions booléennes.

Si `exp1` et `exp2` sont deux expressions, alors type(`exp1 || exp2`)=`int`=type(`exp1 && exp2`).

Si val(`exp1`)$\neq 0$, alors val(`exp1 || exp2`)=1, sinon (val(`exp1`)=0) si val(`exp2`)$\neq 0$, alors val(`exp1 || exp2`)=1, sinon (val(`exp1`)=0 et val(`exp2`)$= 0$), alors val(`exp1 || exp2`)=0.

Si val(`exp1`)=0, alors val(`exp1 && exp2`)=0, sinon (val(`exp1`)$\neq 0$) si val(`exp2`)=0, alors val(`exp1 && exp2`)=0, sinon (val(`exp1`)$\neq 0$ et val(`exp2`)$\neq 0$), alors val(`exp1 && exp2`)=1.

Expressions composées : conjonction et disjonction

Le second opérande d'une expression de la forme "`exp1` II `exp2`" ou "`exp1` && `exp2`" n'est pas évalué si ce n'est pas nécessaire, c'est-à-dire si le premier opérande a la valeur $\neq 0$ dans le cas de l'opérateur II, ou la valeur 0 dans le cas de l'opérateur &&.

Par exemple, val(!((1 II 0) && (0 II 1)))=0.

Expressions composées : affectation

L'opérateur = affecte une valeur à une donnée modifiable. Plus exactement il enregistre cette valeur dans la case mémoire associée à cette donnée.

Si `exp1` est une expression qui est le nom d'une valeur gauche, et si `exp2` est une expression, alors `exp1 = exp2` est une expression telle que :

- type(`exp1 = exp2`) = type(`exp1`),
- val(`exp1 = exp2`)=val(`exp2`) convertie en type(`exp1`),
- il y a un effet de bord : val(`exp2`) est stockée dans la case mémoire désignée par `exp1`.

Par exemple si `x` et `y` sont des variables de type `int` et si val(`y`)=5, alors l'expression `(x = y) > 2` a la valeur `1` et a pour effet de bord d'affecter la valeur 5 à `x`.

Attention : Il ne faut bien sûr pas confondre l'opérateur d'affectation = et celui d'égalité ==.

Expressions composées : affectation

Signalons deux abréviations classique : si ident est le nom d'une variable numérique, alors

- ident++ correspond à ident = ident+1,
- ident−− correspond à ident = ident-1.

Conversion de type

On distingue :

- les conversions explicites réalisées par l'opérateur de cast,
- les conversions implicites réalisées avant l'application d'opérateurs dont les opérandes ne sont pas du type attendu.

Conversion explicitie : opérateur de cast

Si T est un nom de type et si `exp` est une expression dont la valeur est convertible en T, alors `(T) exp` est une expression telle que :

- type(`(T) exp`)=T,
- val(`(T) exp`)=val(`exp`) convertie en T.

Conversion entier - entier

Les types entiers sont ordonnés de la façon suivante :

`char < unsigned char < short < unsigned short < long < unsigned long`.

Soient deux types entiers T1 et T2 et une valeur v de type T1. La conversion de v en une valeur de type T2 est réalisée :

- sans perte, si $T1 \leq T2$ ou si $T1 > T2$ et v est inférieure ou égale à la plus grande valeur représentable en T2.
- avec troncature, si $T1 > T2$ et v est supérieure à la plus grande valeur représentable en T2.

Conversions entier - flottant et flottant - entier

- La conversion d'une valeur v de type entier en une valeur de type flottant produit le flottant le plus proche de v.
- La conversion d'une valeur v de type flottant en une valeur de type entier produit la partie entière de v.

Conversion implicite : opérations arithmétiques

Si op est un opérateur arithmétique, alors l'expression `exp1 op exp2` est évaluée de la façon suivante :

- Les opérandes sont évalués.
- Si l'un des deux opérandes est de type `char`, `unsigned char`, `short` ou `unsigned short`, alors il est tout d'abord converti en `int` : c'est la promotion entière.
- Les types restants sont ordonnés de la façon suivante : `int` < `long` < `unsigned long` < `float` < `double`. Si l'un des opérandes est de type T1 et l'autre de type T2 et que T1 < T2, alors l'opérande de type T1 est converti en T2, puis l'opérateur est appliqué en fournissant une valeur de type T2.

Conversion implicite : affectation

L'expression `exp1 = exp2`, où `exp1` est une expression de type T1 et `exp2` est une expression de type T2, est évaluée de la façon suivante :

- L'expression `exp2` est évaluée, puis sa valeur est convertie en T1 si T1$\neq$T2.
- Cette valeur est affectée à la case mémoire désignée par `exp1`.

Conversion implicite : affectation, exemple

Par exemple, l'évaluation de l'expression `y= 5.0 + 3*x` dans l'état

```
variable    float    x    2.5
variable    int      y    0
```

produit la valeur 12 de type `int` et le nouvel état :

```
variable    float    x    2.5
variable    int      y    12
```

Priorité et associativité des opérateurs

La priorité des opérateurs est utilisée lors de l'évaluation d'expressions dans lesquelles l'ordre des opérations n'a pas été explicitement indiqué par des paires de parenthèses.

Ci-dessous les opérateurs classés en fonction de leur priorité (plus celle-ci est élevée plus l'opérateur est prioritaire) avec entre parenthèses le type d'associativité.

() [ ] . priorité de 15
! - (unaire) & (unaire) *(unaire) (T) priorité de 14 (à gauche)
/ % priorité de 13 (à gauche)
+ - (binaire) priorité de 12 (à gauche)
< <= > >= priorité de 10 (à gauche)
== != priorité de 9 (à gauche)
&& priorité de 5 (à gauche)
II priorité de 4 (à gauche)
= priorité de 2 (à droite)

Priorité et associativité des opérateurs

On dit qu'un opérateur binaire op est infixe s'il est utilisé sous la forme `exp1 op exp2`. Un opérateur préfixe op est utilisé comme suit `op exp`. Enfin un opérateur postfixe : `exp op`.

Voyons comment on détermine la signification d'expressions en fonction des priorités des opérateurs et du type d'associativité :

- Si op1 et op2 sont deux opérateurs infixes, alors
  - x op1 y op2 z est l'expression (x op1 y) op2 z si la priorité de op1 est supérieure à celle de op2, ou bien si les deux priorités sont égales et op1 est associatif à gauche.
  - x op1 y op2 z est l'expression x op1 (y op2 z) sinon.
- Si op1 est un opérateur préfixe et op2 est un opérateur infixe, alors
  - op1 x op2 y est l'expression (op1 x) op2 y si la priorité de op1 est supérieure à celle de op2, ou bien si les deux priorités sont égales et op1 est associatif à gauche.
  - op1 x op2 y est l'expression op1 (x op2 y) sinon.

Priorité et associativité des opérateurs : exemples

- 6+3*4 est 6+(3*4), car la priorité de * est supérieure à celle de +.
- 6*3+4 est (6*3)+4, pour la même raison.
- !a && b II x > 3 est (!a) && b II x > 3 car la priorité de ! est supérieure à celle de &&, qui est (((!a) && b) II (x > 3)) car la priorité de > est supérieure à celle de II.
- 3+4-5 est (3+4)-5 car + et - ont la même priorité et + est associatif à gauche.
- x = y = 5 est x = (y = 5) car = est associatif à droite.