Next Up Previous contents Index
Next: Les opérateurs **Up:** Les expressions **Previous:** Les expressions

Sous-sections

Les conversions de types

Utilité des conversions

Dans un programme, dans un contexte où l'on attend une valeur d'un certain type, il faut normalement fournir une valeur de ce type. Par exemple, si la partie gauche d'une affectation est de type flottant, la valeur fournie en partie droite doit également être de type flottant. Il est cependant agréable de ne pas être trop strict sur cette règle. Si le type attendu et le type de la valeur fournie sont trop différents, (on attend un flottant et on fournit une structure), il est normal que le compilateur considère qu'il s'agit d'une erreur du programmeur. Si par contre, le type attendu et le type de la valeur fournie sont assez << proches >>, c'est une facilité agréable que le compilateur fasse lui-même la conversion. On peut admettre par exemple, que dans un contexte où on attend un nombre flottant, on puisse fournir un nombre entier.

Autre situation où les conversions sont utiles : les expressions. Les machines physiques sur lesquelles s'exécutent les programmes comportent des instructions différentes pour réaliser de l'arithmétique sur les entiers et sur les flottants. Cette situation se retrouve dans les langages de programmation de bas niveau (les assembleurs) où le programmeur doit utiliser des opérateurs différents pour réaliser la même opération (au sens mathématique du terme) selon qu'elle porte sur des entiers ou des flottants. Les langages de programmation de haut niveau par contre, surchargent les symboles des opérateurs arithmétiques de manière à ce que le même symbole puisse réaliser une opération indifféremment entre entiers ou entre flottants : le symbole + permet de réaliser l'addition de deux entiers ou deux flottants. Ceci est déjà une facilité agréable, mais il est possible d'aller plus loin. Le langage peut autoriser le programmeur à donner aux opérateurs des opérandes de types différents, charge au compilateur de faire une conversion de type sur l'un ou l'autre des opérandes pour les amener à un type commun.

Enfin, il se peut que le langage offre au programmeur la possibilité de demander explicitement une conversion de type : si le langage PASCAL n'offre pas une telle possibilité, le langage C par contre dispose d'un opérateur de conversion de type.

Ce qu'il y a dans une conversion

Pour comprendre ce qui se passe dans une conversion il faut bien distinguer type, valeur et représentation. La représentation d'une valeur est la chaîne de bits qui compose cette valeur dans la mémoire de la machine. La représentation des entiers est une suite de bits en notation binaire simple pour les positifs, généralement en complément à 2 pour les négatifs. La représentation des flottants est plus compliquée, c'est généralement un triplet de chaînes de bits : (signe, mantisse, exposant).

Une conversion a pour but de changer le type d'une valeur, sans changer cette valeur si c'est possible ; elle pourra éventuellement s'accompagner d'un changement de représentation.

Exemple de conversion avec changement de représentation : la conversion d'entier vers flottant ou vice versa. Exemple de conversion

sans changement de représentation : la conversion d'entier non signé vers entier signé ou vice versa, sur une machine où les entiers signés sont représentés en complément à 2.

L'ensemble des conversions possibles

Conversions vers un type entier

- **depuis un type entier** La règle est de préserver, si c'est possible, la valeur mathématique de l'objet. Si ce n'est pas possible :
  - si le type destination est un type signé, on considère qu'il y a dépassement de capacité et la valeur du résultat n'est pas définie.
  - si le type destination est un type non signé, la valeur du résultat doit être égale (modulo *n*) à la valeur originale, où *n* est le nombre de bits utilisés pour représenter les valeur du type destination.

  Dans ce qui suit, on se place précisément dans la cas où la machine représente les nombres signés en complément à 2 (c'est le cas de pratiquement toutes les machines). Une conversion d'un entier signé vers un entier non signé, ou vice versa, se fait sans changement de représentation. Une conversion d'un entier vers un entier plus court se fait par troncature des bits les plus significatifs. Une conversion d'un entier vers un entier plus long se fait par extension du bit de signe si le type originel est signé, par extension de zéros si le type originel est non signé.

- **depuis un type flottant** La règle est de préserver, si c'est possible, la valeur mathématique de l'objet, sachant qu'il peut y avoir une erreur d'arrondi.

- **depuis un pointeur** Un pointeur peut être converti en un type entier. Pour cela il est considéré comme un type entier non signé de la même taille que les pointeurs. Il est ensuite converti dans le type destination selon les règles de conversions d'entiers vers entiers.

Conversions vers un type flottant

Seuls les types entiers et flottants peuvent être convertis en un type flottant. Là aussi, la règle est de préserver la valeur si possible, sinon c'est un cas d'overflow ou d'underflow.

Conversion vers un type pointeur

Les différentes possibilités sont les suivantes :

- Un type pointeur vers *T*1 peut être converti en un type pointeur vers *T*2 quels que soient *T*1 et *T*2.
- La valeur entière 0 peut être convertie en un type pointeur vers *T* quel que soit *T*, et c'est la valeur dite de *pointeur invalide*.
- Une valeur entière non nulle peut être convertie en un type pointeur vers *T* quel que soit *T*, mais cela est explicitement non portable.

Nous avons vu précédemment au paragraphe 4.1 :

- Toute expression de type tableau de x est convertie en type pointeur vers x.

Il y a une règle similaire concernant les fonctions :

- Toute expression de type fonction retournant x est convertie en type pointeur vers fonction retournant x.

Conversion vers le type void

N'importe quelle valeur peut être convertie vers le type void. Cela n'a de sens que si la valeur résultat n'est pas utilisée.

Les situations de conversions

Dans le langage C, les situations où se produisent les conversions sont les suivantes :

1. une valeur d'un certain type est utilisée dans un contexte qui en demande un autre.
   - passage de paramètre : le paramètre effectif n'a pas le type du paramètre formel ;
   - affectation : la valeur à affecter n'a pas le même type que la variable ;
   - valeur rendue par une fonction : l'opérande de *return* n'a pas le type indiqué dans la déclaration de la fonction.
2. opérateur de conversion : le programmeur demande explicitement une conversion.

3.
un opérateur a des opérandes de types différents.

Dans les cas 1 et 2, type de départ et type d'arrivée de la conversion sont donnés. Dans le cas 3, par contre, c'est le compilateur qui choisit le type d'arrivée de la conversion. Il le fait selon des règles soigneusement définies. Il y en a deux dans le langage C qui portent les noms de << promotion des entiers >> et << conversions arithmétiques habituelles >>.

La promotion des entiers

Ce que l'on appelle dans le langage C *promotion des entiers* est une règle de conversion des opérandes dans les expressions. La promotion des entiers a pour but d'amener les << petits entiers >> à la taille des `int`.

Domaine d'application

La promotion des entiers est appliquée à l'opérande des opérateurs unaires `+`, `-` et `~`, ainsi qu'aux deux opérandes des opérateurs de décalage `>>` et `<<`. La promotion des entiers est également utilisée dans la définition des *conversions arithmétiques habituelles*.

La règle

Une valeur de type `char`, un `short int` ou un champ de bits, ou d'une version signée ou non signée des précédents, peut être utilisée dans un contexte où `un int` ou un `unsigned int` est demandé. Cette valeur est convertie en `un int` ou un `unsigned int` d'une manière (hélas) dépendante de l'implémentation :

-
si un `int` peut représenter toutes les valeurs du type de départ, la valeur est convertie en `int` ;
-
sinon, elle est convertie en `unsigned int`.

Les conversions arithmétiques habituelles

Domaine d'application

Les conversions arithmétiques habituelles sont réalisés sur les opérandes de tous les opérateurs arithmétiques binaires sauf les opérateurs de décalage `>>` et `<<` ainsi que sur les second et troisième opérandes de l'opérateur `?:`.

La règle

1.
Si un opérande est de type `long double`, l'autre opérande est converti en `long double`.
2.
Sinon si un opérande est de type `double`, l'autre opérande est converti en `double`.
3.
Sinon si un opérande est de type `float`, l'autre opérande est converti en `float`.
4.
Sinon la promotion des entiers est réalisée sur les deux opérandes. Ensuite :
a.
Si un opérande est de type `unsigned long int`, l'autre opérande est converti en `unsigned long int`.
b.
Sinon, si un opérande est de type `long int` et l'autre de type `unsigned int`, alors :
-
si un `long int` peut représenter toutes les valeurs d'un `unsigned int`, l'opérande de type `unsigned int` est converti en `long int`.
-
sinon, les deux opérandes sont convertis en `unsigned long int`.
c.
Sinon, si un opérande est de type `long int`, l'autre opérande est converti en `long int`.
d.
Sinon, si un opérande est de type `unsigned int`, l'autre opérande est converti en `unsigned int`.
e.
Sinon, les deux opérandes sont de même type, et il n'y a pas de conversion à réaliser.

Discussion

Les points 1, 2, 3 sont faciles à comprendre : si les deux opérandes sont flottants, celui de moindre précision est converti dans le type de l'autre. Si un seul des opérandes est de type flottant, l'autre est converti dans ce type.

On aborde le point 4 si les deux opérandes sont des variétés d'entiers courts, normaux ou longs, signés ou non signés. On applique alors la promotion des entiers, de manière à se débarrasser des entiers courts. À la suite de cela, il n'y plus comme types possibles que `int`, `unsigned int`, `long int` et `unsigned long int`.

Si l'on excepte les cas où les deux types sont identiques, le reste des règles peut se résumer dans le tableau suivant :

opérande	opérande	résultat
`unsigned long int`	*quelconque*	`unsigned long int`

long int	unsigned int	long int unsigned long int
long int	int	long int
unsigned int	int	unsigned int

Les surprises des conversions

D'une manière générale, les conversions sont un mécanisme qui fonctionne à la satisfaction du programmeur. Il y a cependant une situation où cela peut donner des résultats surprenants : quand on réalise une comparaison entre entiers signés et entiers non signés. Par exemple, le programme suivant :

```
int main()
{
unsigned int i = 0;

if (i < -1 )
   printf("Bizarre, bizarre ...\n");
   else printf ("Tout semble normal\n");
}
```

imprimera le message `Bizarre, bizarre ...`, pouvant laisser croire que pour le langage C, 0 est inférieur à -1.

L'explication est la suivante : l'opérateur < a un opérande de type `unsigned int` (la variable `i`), et un autre opérande de type `int` (la constante -1). D'après le tableau des conversions donné ci-dessus, on voit que dans un tel cas, les opérandes sont convertis en `unsigned int`. Le compilateur génère donc une comparaison non signée entre 0 et 4294967295 (puisque -1 = 0xffffffff = 4294967295), d'où le résultat.

Pour que tout rentre dans l'ordre, il suffit d'utiliser l'opérateur de conversion pour prévenir le compilateur de ce qu'on veut faire :

```
int main()
{
unsigned int i = 0;

if ((int) i < -1 )    /*   comparaison entre deux int   */
   printf("Bizarre, bizarre ...\n");
   else printf ("Tout semble normal\n");
}
```

Là où tout se complique c'est qu'on peut utiliser des entiers non signés sans le savoir ! Considérons le programme suivant :

```
int main()
{
if (sizeof(int) < -1)
   printf("Bizarre, bizarre ...\n");
else printf ("Tout semble normal\n");
}
```

le lecteur a sans doute deviné qu'il va imprimer le message `Bizarre, bizarre ...`, et cependant les entiers n'ont pas une longueur négative ! L'explication est la suivante : l'opérateur `sizeof` rend une valeur dont le type est non signé. Voici ce que dit exactement la norme : << *La valeur du résultat* [de `sizeof`] *dépend de l'implémentation, et son type (un type entier non signé) est* `size_t` qui est définit dans le fichier d'include `stddef.h` >>. Dans notre exemple, le compilateur a généré une comparaison non signée entre 4 (`sizeof(int)`) et 4 294 967 295, d'où le résultat.

Recommandations

1. Ne jamais mélanger des entiers signés et non signés dans des comparaisons : utiliser l'opérateur de conversion pour amener l'opérateur de comparaison à avoir des opérandes de même type.
2. Bien noter que l'opérateur `sizeof` rend une valeur de type entier non signé.

Next Up Previous contents Index
Next: Les opérateurs **Up:** Les expressions **Previous:** Les expressions
Bernard Cassagne
1998-12-09