Opérateurs arithmétiques

Entiers et expressions arithmétiques

Variables de type entier

Pour définir et initialiser une ou plusieurs variables de type entier, on utilise la syntaxe suivante :

declare -i *nom*[**=***expr_arith*] [*nom*[**=***expr_arith*] …]

```
$ declare -i x=35   => définition et initialisation de la variable entière
x
$
$ declare -i v w    => définition des variables entières v et w
$
$ v=12              => initialisation de v par affectation
$
$ read w
34                  => initialisation de w par lecture
$
```

Rappel : Il n'est pas nécessaire de définir une variable avant de l'utiliser !

Pour que la valeur d'une variable entière ne soit pas accidentellement modifiée après qu'elle ait été initialisée, il suffit d'ajouter l'attribut **r**.

```
$ declare -ir a=-6
$
$ a=7
-bash: a: readonly variable   => seule la consultation est autorisée !
$
```

Enfin, pour connaître toutes les variables entières définies, il suffit d'utiliser la commande **declare –i**.

```
$ declare -i
declare -ir EUID ="1007"
declare -ir PPID="19620"
declare -ir UID="1007"
declare -ir a="-6"
declare -i v="12"
declare -i w="14"
declare -i x="35"
$
```

Commande interne ((

Cette commande interne est utilisée pour effectuer des opérations arithmétiques.

Syntaxe : **((** *expr_arith* **))**

Son fonctionnement est le suivant : *expr_arith* est évaluée ; si cette évaluation donne une valeur différente de **0**, alors le code de retour de la commande interne **((** est égal à **0** sinon le code de retour est égal à **1**.

Il est donc important de distinguer deux aspects de la commande interne **((** *expr_arith* **))** :

la valeur de *expr_arith* issue de son évaluation et

le code de retour de **((** *expr_arith* **))**.

Attention : la valeur de **expr_arith** n'est pas affichée sur la sortie standard.

```
$ (( -5 ))    => la valeur de l'expression arithmétique est égale à -5 (c.-
à-d.
$             => différente de 0), donc le code de retour de (( est égal à 0
$ echo $?
0
$
$ (( 0 ))     => la valeur de l'expression arithmétique est 0, donc le code
de retour
$             => de (( est égal à 1
$ echo $?
1
$
```

Les opérateurs permettant de construire des expressions arithmétiques évaluables par **bash** sont issus du langage C (ex : = + - > <=).

```
$ declare -i a=2 b
$ (( b = a + 7 ))
$
```

Le format d'écriture est libre à l'intérieur de la commande **((**. En particulier, plusieurs caractères **espace** ou **tabulation** peuvent séparer les deux membres d'une affectation.

Il est inutile d'utiliser le caractère de substitution **$** devant le nom d'une variable car il n'y a pas d'ambiguïté dans l'interprétation ; par contre, lorsqu'une expression contient des paramètres de position, le caractère $ doit être utilisé.

```
$ date
jeudi 21 décembre 2006, 19:41:42 (UTC+0100)
$
$ set $(date)
$
$ (( b = $2 +1 ))    => incrémentation du jour courant
$
$ echo $b
22
$
```

Code de retour de **((** et structures de contrôle

Le code de retour d'une commande interne ((est souvent exploité dans une structure de contrôle if ou while.

Le programme shell dix ci-dessous affiche les dix chiffres :

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#!/bin/bash

declare -i i=0

while (( i < 10 ))
do
  echo $i
  (( i = i + 1 )) # ou(( i++ ))
done
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Son fonctionnement est le suivant : l'expression *i < 10* est évaluée, sa valeur est vraie, le code de retour de *((i < 10))* est égal à **0**, le corps de l'itération est exécuté. Après affichage de la valeur **9**, la valeur devient égale à **10**. L'expression *i < 10* est évaluée, sa valeur est maintenant fausse, le code de retour de *((i < 10))* est égal à **1**, l'itération se termine.

```
$ dix
0
1
2
3
4
5
6
7
8
9
$
```

Valeur d'une expression arithmétique

La commande interne **((** *expr_arith* **))** n'affiche pas sur la sortie standard la valeur de l'expression arithmétique *expr_arith*.

Pour obtenir la valeur de l'expression arithmétique, on utilise la syntaxe : **$((** *expr_arith* **))**

Exemples :

```
echo $(( 7 * 2 ))
echo $(( a= 12*8 ))
echo $(( 7 < 10 ))
```

Substitutions de commandes et substitutions de paramètres peuvent être présentes dans une commande interne **((** ou bien dans une substitution d'expression arithmétique **$(( ))** si le résultat aboutit à un nombre entier.

Par exemple, dans l'expression **$(($(ls -l | wc -l) -1))** le résultat du pipeline **ls -l | wc –l** peut être interprété comme un nombre.

Attention : lors de l'évaluation d'une expression arithmétique, les débordements éventuels ne sont pas détectés.

```
$ echo $((897655*785409*56789*6789999999999999999999 ))
-848393034087410691     => nombre négatif !
$
```

Si l'on souhaite utiliser de grands entiers (ou des nombres réels), il est préférable d'utiliser la commande unix **bc**.

```
$ bc -q
897655*785409*56789*6789999999999999999999
271856250888322242449959962260546638845
$
```

L'option **–q** de **bc** évite l'affichage du message de bienvenue lorsqu'on utilise cette commande de manière interactive.

Basée sur le contexte, l'interprétation des expressions arithmétiques est particulièrement souple en **bash**.

```
$ declare -i x=35
$
$ z=x+5           => affectation de la chaîne x+5 à la variable z
$ echo $z
x+5               => non évaluée car z de type chaîne de caractères par
défaut
$
$ echo $((z))     => par le contexte, z est évaluée comme une variable de
type entier
40
$ (( z = z+1))
$ echo $z
41
$
```

Opérateurs

Une expression arithmétique contenue dans une commande interne **((** peut être une valeur, un paramètre ou bien être construite avec un ou plusieurs opérateurs. Ces derniers proviennent du langage C. La syntaxe des opérateurs, leur signification, leur ordre de priorité et les règles d'associativité s'inspirent également du langage C.

Les opérateurs traités par la commande **((** sont nombreux. C'est pourquoi, seuls les principaux opérateurs sont mentionnés dans le tableau ci-dessous, munis de leur associativité (*g* : de gauche à droite, *d* : de droite à gauche). Ne sont pas présentés dans ce tableau les opérateurs portant sur les représentations binaires (ex : **et bit à bit, ou bit à bit**, etc.).

Les opérateurs ayant même priorité sont regroupés dans une même classe et les différentes classes sont citées par ordre de priorité décroissante.

1. *a***++** *a***--** post-incrémentation, post-décrémentation (**g**)
2. **++***a* **--***a* pré-incrémentation, pré-décrémentation(**d**)
3. **-***a* **+***a* moins unaire, plus unaire(**d**)
4. **!***a* négation logique (**d**)
5. *a********b* exponentiation : a puissance b (**d**)
6. *a******b* *a***/***b* *a***%***b* multiplication, division entière, reste (**g**)

7. *a***+***b* *a***-***b* addition, soustraction (**g**)
8. *a***<***b* *a***<=***b* *a***>***b* *a***>=***b* comparaisons (**g**)
9. *a***==***b* *a***!=***b* égalité, différence (**g**)
10. *a***&&***b* ET logique (**g**)
11. *a***||***b* OU logique (**g**)
12. *a***?***b***:***b* opérateur conditionnel (**d**)
13. *a***)***b* *a****=***b* *a***%=***b* *a***+=***b* *a***-=***b* opérateurs d'affectation (**d**)
14. *a***,***b* opérateur virgule (**g**)

Ces opérateurs se décomposent en :

- opérateurs arithmétiques
- opérateurs d'affectations
- opérateurs relationnels
- opérateurs logiques
- opérateurs divers.

1. Opérateurs arithmétiques :

   Les quatre opérateurs ci-dessous s'appliquent à une *variable*.

   - Post-incrémentation : *var***++** la valeur de *var* est d'abord utilisée, puis est incrémentée

```
$ declare -i x=9 y
$ (( y = x++ ))    => y reçoit la valeur 9 ; x vaut 10
$ echo "y=$y x=$x"
y=9 x=10
$
```

   - Post-décrémentation : *var***--** la valeur de *var* est d'abord utilisée, puis est décrémentée

```
$ declare -i x=9 y
$ (( y = x-- ))    => y reçoit la valeur 9 ; x vaut 8
$ echo "y=$y x=$x"
y=9 x=8
$
```

   - Pré-incrémentation : **++***var* la valeur de *var* est d'abord incrémentée, puis est utilisée

```
$ declare -i x=9 y
$ (( y=++x))      => x vaut 10 ; y reçoit la valeur 10
$ echo "y=$y x=$x"
y=10 x=10
$
```

   - Pré-décrémentation : **--***var* la valeur de *var* est d'abord décrémentée, puis est utilisée

```
$ declare -i x=9 y
$ (( y= --x ))      => x vaut 8 ; y reçoit la valeur 8
$ echo "y=$y x=$x"
y=8 x=8
$
```

Les autres opérateurs s'appliquent à des expressions arithmétiques.

- Moins unaire : **-** *expr_arith*
- Addition : *expr_arith* **+** *expr_arith*
- Soustraction : *expr_arith* **-** *expr_arith*
- Multiplication : *expr_arith* ***** *expr_arith*
- Division entière : *expr_arith* **/** *expr_arith*
- Reste de division entière : *expr_arith* **%** *expr_arith*

```
$ (( a = b/3 +c ))            => division entière et addition
$
$ echo $(( RANDOM%49 +1 ))    => reste et addition
23
$
```

La variable prédéfinie du shell **RANDOM** renvoie une valeur pseudo-aléatoire dès qu'elle est utilisée.

L'utilisation de **parenthèses** permet de modifier l'ordre d'évaluation des composantes d'une expression arithmétique.

```
$ (( a = ( b+c )*2 ))
$
```

1. Opérateurs d'affectations :
   - Affectation simple : *nom* = *expr_arith*
   - Affectation composée : *nom opérateur*= *expr_arith*

Comme en langage C, l'affectation n'est pas une instruction (on dirait *commande* dans la terminologie du shell) mais une expression, et comme toute expression elle possède une valeur. Celle-ci est la valeur de son membre gauche après évaluation de son membre droit.

```
$ ((a=b=c=5))   => la valeur de l'expression a=b=c=5 est égale
à 5
$ echo $a $b $c
5 5 5
$
```

Dans l'exemple ci-dessous, l'expression *a = b + 5* est évaluée de la manière suivante : *b* est évaluée (sa valeur est **2**), puis c'est l'expression *b+5* qui est évaluée (sa valeur vaut**7**), enfin cette valeur est affectée à la variable *a*. La valeur de l'expression *a = b + 5* est égale à celle de *a*, c'est à dire 7. Ceci permet d'écrire :

```
$ declare -i b=2
```

```
$
$ echo $(( a = b+5 ))
7
$
$ echo $a
7
$
```

La syntaxe *nom opérateur*= *expr_arith* est un raccourci d'écriture provenant du langage C et signifiant : *nom* = *nom opérateur expr_arith*nom = nom opérateur expr_arith

Opérateur pourra être : *** / + -**

```
$ (( a += 1 ))   => ceci est équivalent à (( a = a + 1 ))
$
```

2. Opérateurs relationnels :

Lorsqu'une expression relationnelle (ex : *a < 3*) est vraie, la valeur de l'expression est égale à **1**.

Lorsqu'elle est *fausse*, sa valeur est égale à **0**.

- Egalité : *expr_arith* **==** *expr_arith* (Attention aux deux caractères égal)
- Différence : *expr_arith* **!=** *expr_arith*
- Inférieur ou égal : *expr_arith* **<=** *expr_arith*
- Supérieur ou égal : *expr_arith* **>=** *expr_arith*
- Strictement inférieur : *expr_arith* **<** *expr_arith*
- Strictement supérieur : *expr_arith* **>** *expr_arith*

```
$ declare -i a=4
$
$ (( a<3 ))    => expression fausse, valeur égale à 0, code de
retour égal à 1
$ echo $?
1
$
$ declare -i a=3 b=2
$
$ if (( a == 7 ))
> then (( a= 2*b ))
> else (( a = 7*b))
> fi
$
$ echo $a
14
$
```

L'expression relationnelle *a == 7* est fausse, sa valeur est **0** et le code de retour de *((a == 7))* est égal à **1** : c'est donc la partie **else** qui est exécutée.

3. Opérateurs logiques :

Comme pour une expression relationnelle, lorsqu'une expression logique (ex : *a > 3 && a < 5*) est *vraie*, la valeur de l'expression est égale à **1**. Lorsqu'elle est *fausse*, sa valeur est égale à **0**.

- Négation logique : **!** *expr_arith*

  Si la valeur de *expr_arith* est différente de **0**, la valeur de **!** *expr_arith* est égale à **0**.

  Si la valeur de *expr_arith* est égale à **0**, la valeur de **!** *expr_arith* est égale à **1**.

  Au moins un caractère **espace** doit être présent entre **!** et *expr_arith*.

```
$ echo $(( ! 12 ))   => echo $(( !12 )) provoque une erreur
0
$
```

- Et logique : *expr_arith1* **&&** *expr_arith2*

  Si la valeur de *expr_arith1* est égale à **0** (*fausse*), alors *expr_arith2* n'est pas évaluée et la valeur de *expr_arith1* **&&** *expr_arith2* est **0**.

  Si la valeur de *expr_arith1* est différente de **0** (*vraie*), alors *expr_arith2* est évaluée : si sa valeur est égale à **0**, alors la valeur de *expr_arith1* **&&** *expr_arith2* est **0**, sinon elle vaut **1**.

```
$ declare -i a=4
$
$ echo $(( a<3 ))                          => expression
fausse
0
$
$ echo $(( a > 3 && a <5 ))     => expression vraie, valeur
égale à 1
1
$ ((a>3 && a<5 ))               => expression vraie, code de
retour égal à 0
$ echo $?
0
$
```

- Ou logique : *expr_arith1* **||** *expr_arith2*

  Si la valeur de *expr_arith1* est différente de **0** (*vraie*), alors *expr_arith2* n'est pas évaluée et la valeur de *expr_arith1* **||** *expr_arith2* est égale à **1**.

Si la valeur de *expr_arith1* est égale à **0** (*fausse*), alors *expr_arith2* est évaluée : si sa valeur est différente de **0**, alors la valeur de *expr_arith1* **||** *expr_arith2* est **1**, sinon elle vaut **0**.

```
$ declare -i x=4
$
$ (( x > 4 || x < 4 ))
$ echo $?
1
$
$ echo $(( x > 4 || x<4 ))
0
$
```

4. Opérateurs divers :

- Exponentiation : *expr_arith1* ****** *expr_arith2*

  Contrairement aux autres opérateurs, l'opérateur d'exponentiation ****** n'est pas issu du langage C. A partir de la version 3.1 de **bash**, son associativité a changé : elle est de droite à gauche (**d**).

```
$ echo $(( 2**3**2 ))     => interprétée comme 2**(3**2) : 29
512
$
```

- Opérateur conditionnel : *expr_arith1* **?** *expr_arith2* **:** *expr_arith3*

Le fonctionnement de cet opérateur ternaire est le suivant : *expr_arith1* est évaluée, si sa valeur est *vraie*, la valeur de l'expression arithmétique est celle de *expr_arith2*, sinon c'est celle de *expr_arith3*.

```
$ declare -i a=4 b=0
$ echo $(( a < 3 ? (b=5) : (b=54) ))
54
$
$ echo $b
54
$
```

Il aurait également été possible d'écrire : `(( b = a < 3 ? 5 : 54 ))`

- Opérateur virgule : *expr_arith1* **,** *expr_arith2*

  expr_arith1 est évaluée, puis *expr_arith2* est évaluée ; la valeur de l'expression est celle de *expr_arith2*.

```
$ declare -i a b
$ echo $(( a=4,b=9 ))
9
$ echo "a=$a b=$b"
a=4 b=9
$
```

Structure for pour les expressions arithmétiques

Bash a introduit une nouvelle structure *for* adaptée aux traitements des expressions arithmétiques, itération issue du langage C. Elle fonctionne comme cette dernière.

Syntaxe :

for ((*expr_arith1* **;** *expr_arith2***;***expr_arith3* **))**

do

suite_cmd

done

expr_arith1 est l'expression arithmétique d'initialisation.

expr_arith2 est la condition d'arrêt de l'itération.

expr_arith3 est l'expression arithmétique qui fixe le pas d'incrémentation ou de décrémentation.

Exemples :

```
declare -i x
for (( x=0 ; x<5 ; x++ ))
do
  echo $(( x*2 ))
done
declare -i x y
for (( x=1,y=10 ; x<4 ; x++,y-- ))
do
  echo $(( x*y ))
done
```