Bash : les symboles dans les calculs

- Objet : suite de la série de wiki visant à maîtriser bash via les différents caractère spéciaux.
- Niveau requis :
  débutant, avisé
- Commentaires : scripts
- Débutant, à savoir : Utiliser GNU/Linux en ligne de commande, tout commence là !.
- Suivi :
  en-chantier
    - Création par Hypathie le 08/04/2014
    - Testé par Hypathie Avril 2014
- Commentaires sur le forum : ici [1)]

Page en court de réécriture

Introduction

Dans la page précédente (Les opérateurs de comparaison numérique), nous avons abordé la commande interne **let** et la commande composée **((...))**.
Nous complémentons ici leurs usages, pour réaliser des opérations mathématiques.

Les commandes **let** et **((...))** sont les seules commandes internes que bash dispose pour réaliser des opérations mathématiques.
A travers elles, bash est limité à n'opérer que sur des entiers signés (positifs ou négatifs).
Le résultat de l'évaluation d'une expression sera toujours un entier décimal signé.

Pour réaliser des opérations avec des nombres à virgule ou plus complexes, nous devons nous tourner vers des commandes externes tel que **bc** (non installé par défaut) ou **awk**. Ou alors utiliser d'autres langages interprétés tel que **perl**, **python**, etc (**awk**, **perl** et **python** sont disponibles par défaut sur les systèmes Debian).

Avec **let** et **((...))**, la plage numérique autorisée est : **-9223372036854775808 < 0 > 9223372036854775807**

Après l'évaluation des expressions qu'elles contiennent, **let** et **((...))** renvoient le code de retour :

- **0**, si le résultat est inférieur ou supérieur à 0.
- **1**, si le résultat est égal à 0.

La commande **((...))** peut-être préfixée du caractère de remplacement **$** pour former la commande **$((...))**, afin d'être remplacée par l'évaluation de l'expression qu'elle contient.

La commande **$[...]** équivalente à la commande **$((..))** **ne doit plus être utilisée**.

Elle est délaissée depuis la version 2.0 de bash au profit de **((...))**.

Extrait de la page du manuel de bash

```
L'ancien format $[expression] est obsolète et sera supprimé
dans les prochaines versions de bash.
```

Pour des raisons de rétrocompatibilité, elle est toujours active dans nos bash modernes.
Mais viendra un jour où, **$[...]** ne sera plus.

Les systèmes numériques

Les expressions numériques évaluées par **let** et **((...))** peuvent contenir des nombres de différents systèmes numérique.

L'évaluation des expression (le résultat) sera toujours retourné en entier décimal.

- Sans précision les nombres sont en base 10 (décimal).
- Préfixé d'un **0**, le nombre est un octal (base 8).
- Préfixé de **0x** (ou **0X**), le nombre est un hexadécimal (base 16).

Si non, la syntaxe suivante peut-être utilisée :

- [**base#**]**n**

Avec :

- **base#** où **base** est un nombre en décimal désignant la base utilisée (de 2 à 64).
- **n** Le nombre lui même :
  - Jusqu'à la base 10, le nombre contient que des chiffres.
  - De la base 11 à la base 36, les lettres minuscules ou majuscules peuvent être utilisées indifféremment.
  - De la base 37 à la base 64, les lettres minuscules, majuscules, @ et _ sont à utiliser dans cet ordre.

Exemple :

```
echo $(( 2#101010 ))    # Nombre binaire (base 2)
```

```
42
```

Les opérateurs arithmétiques

Les opérateurs arithmétiques permettent de réaliser des calculs numériques classiques.

Liste des opérateurs arithmétiques		
Opérateurs	**Désignations**	**Résultats**
Opérateurs unaires		
+ *expr*	Signe positif	Valeur de *expr*
- *expr*	Signe négatif	Opposé de *expr*
Opérateurs binaires		
expr1 **+** *expr2*	Addition	*expr2* ajouté à *expr1*
expr1 **-** *expr2*	Soustraction	*expr2* retiré de *expr1*
expr1 ***** *expr2*	Multiplication	*expr1* multiplié par *expr2*
expr1 **/** *expr2*	Division	*expr1* divisé par *expr2*
expr1 **%** *expr2*	Modulo	Reste de la division de *expr1* par *expr2*
expr1 ****** *expr2*	Puissance	*expr1* multiplié par lui-même *expr2* fois
expr, *expr1* et *expr2*, sont sujettes au remplacement des paramètres (et des variables), à la substitution de commandes et à la suppression des protections.		

```
a=42                # Affecte 42 à la variable a
echo $((-a))            # Retourne l'opposé de la valeur contenue dans la
variable a.

echo $((21+025))        # Retourne l’addition de l'octal 25 au décimal 21.

echo 3/2=$((3/2))       # Retourne 3/2
echo 3/2 reste $((3%2))     # Retourne le reste de 3/2

echo a=$a a^2=$((a**2))     # Retourne la valeur de la variable élevée à la
puissance 2

unset a
```

```
-42
42
3/2=1
3/2 reste 1
a=42 a^2=1764
```

La division par 0 retourne une erreur.

```
echo $((42/0))
```

```
bash: 42/0 : division par 0 (le symbole erroné est « 0 »)
```

Les opérateurs d'affectation

Les opérateurs d'affectation, permettent affecter une valeur à une variable ou d'en modifier la valeur.

Liste des opérateurs d'affectation		
Opérateurs	**Désignations**	**Résultats**
Opérateurs unaires		
++ *var*	Affectation par la pré-incrémentation à 1	*var* = *var* + 1 puis retourne *var*
var **++**	Affectation par la post-incrémentation à 1	Retourne *var*, puis *var* = *var* + 1
− *var*	Affectation par la pré-décrémentation à 1	*var* = *var* - 1 puis retourne *var*
var **−**	Affectation par la post-décrémentation à 1	Retourne *var*, puis *var* = *var* - 1
Opérateurs binaires		
var **=** *expr*	Simple affectation	Affecte *epxr* à la variable *var*
var **+=** *expr*	Affectation par l'incrémentation	*var* = *var* + *expr*
var **-=** *expr*	Affectation par la décrémentation	*var* = *var* - *expr*
var ***=** *expr*	Affectation par la multiplication	*var* = *var* * *expr*
var **/=** *expr*	Affectation par la division	*var* = *var* / *expr*
var **%=** *expr*	Affectation par modulo	*var* = *var* % *expr*
var **<<=** *expr*	Ré-affectation par le décalage bit-à-bit à gauche	*var* = *var* << *expr*
var **>>=** *expr*	Ré-affectation par le décalage bit-à-bit à droite	*var* = *var* >> *expr*
var **&=** *expr*	Affectation par le ET binaire	*var* = *var* & *expr*
var **|=** *expr*	Affectation par le OU binaire	*var* = *var* | *expr*
var **^=** *expr*	Affectation par le OU exclusif binaire	*var* = *var* ^ *expr*
expr, est sujette au remplacement des paramètres (et des variables), à la substitution de commandes et à la suppression des protections.		

```
declare -p a
let a=24    ; declare -p a
((a=42))    ; echo a=$a

let 'a += 42'   ; echo "a+=42 -> a=$a"
(( a /= 2 ))    ; echo "a/=2  -> a=$a"

unset a
```

```
bash: declare: a : non trouvé
declare -- a="24"
a=42
a+=42 -> a=84
a/=2  -> a=42
```

Contrairement à la commande **declare -i**, les commandes **let** et **((..))** ne donnent pas l'attribut numérique (**-i**) à une variable.

```
declare -i b=42
declare -p b
echo "Pré-incrémentation à 1  : Avant b=$b ; Pendant b=$((++b)) ; Après :
b=$b"
echo "Post-décrémentation à 1 : Avant b=$b ; Pendant b=$((b--)) ; Après :
b=$b"

unset b
```

```
declare -i b="42"
Pré-incrémentation à 1  : Avant b=42 ; Pendant b=43 ; Après : b=43
Post-décrémentation à 1 : Avant b=43 ; Pendant b=43 ; Après : b=42
```

Voir aussi :

- **typologie de variables**
- **variables numériques et calculs**

L'opérateur virgule

L'opérateur virgule, permet de séparer des opérations.

L'opérateur virgule		
Opérateur	**Désignation**	**Résultat**
Opérateur binaire		
expr1 **,** *expr2*	Virgule	Sépare les opération *expr1* et *expr2* pour les réaliser à la suite.

Exemples :

```
echo $(( 5+5 , 10+5 , 21+21))
```

```
42
```

Seule la dernière opération est retournée.

C'est plus utile avec des affectations :

```
echo $(( a=5+5 , b=10+5 , 21+21))
```

```
echo a=$a b=$b
unset a b
```

```
42
a=10 b=15
```

Les opérateurs logiques

Les opérateurs logiques, réalisent des opérations sur des booléens.
L'algèbre de boole n'énumère que deux états : **vrai** et **faux**.
Les commandes **let** et **((..))** suivent le standard C :

- **0** est compris comme **faux**.
- Toutes autres valeurs (positives ou négatives), souvent **1**, sont comprises comme **vrai**.

... Dans bash, c'est le contraire (voir la note plus bas).

Trois opérateurs logiques sont disponibles : 1 unaire et 2 binaires.


Les opérateurs logiques ne doivent être confondus avec les opérateurs bit-à-bit.

Liste des opérateurs logiques		
Opérateurs	**Désignations**	**Résultats**
Opérateur unaire		
!(*expr* **)**	Négation logique	Retourne l'inverse de *expr* !(1) = 0 (NON vrai = faux) !(0) = 1 (NON faux = vrai)
Opérateurs binaires		
expr1 **&&** *expr2*	ET logique	1 && 1 = 1 (vrai ET vrai = vrai) 1 && 0 = 0 (vrai ET faux = faux) 0 && 1 = 0 (faux ET vrai = faux) 1 && 1 = 0 (faux ET faux = faux)
expr1 **||** *expr2*	OU logique	1 && 1 = 1 (vrai OU vrai = vrai) 1 && 0 = 1 (vrai OU faux = vrai) 0 && 1 = 1 (faux OU vrai = vrai) 0 && 0 = 0 (faux OU faux = faux)
expr, *expr1* et *expr2*, sont sujettes au remplacement des paramètres (et des variables), à la substitution de commandes et à la suppression des protections.		

Exemples :

```
echo "!(0) = $((!(0)))"
echo "!(5+5) = $((!(5+5)))"
echo $((!1))
```

```
!(0) = 1
!(5+5) = 0
```

```
bash: !1: event not found
```

L'opérateur de négation logique **!**) s'utilise uniquement avec des parenthèses : **!(*expr*)**.

Les opérations logiques peuvent se suivre :

```
echo $(( 1 && 1))
echo $(( 1 && 0))
echo $(( 1 && 1 && 0))  # vrai && vrai && faux, retourne faux
```

```
1
0
0
```

Les opérateurs **&&** et **||** n'étudient l'opérande de droite (*expr2*) que si cela est pertinent :

```
a=1
echo $((0 && (a=10) )) a=$a # Le second opérande (a=10) n'est pas étudié car
cela n'est pas pertinent
                # avec 0 (faux) et l'opérateur &&, le résultat sera toujours
faux.

echo $((1 && (a=10) )) a=$a # Le second opérande (a=10) est étudié car cela
est pertinent
                # avec 1 (vrai) et l'opérateur &&, le résultat pourra être
vrai.
unset a
```

```
0 a=1
1 a=10
```

Nous avons placé l'affection **a=10** entre parenthèses **(a=10)**, pour la protéger contre la priorité des opérateurs.

Sinon, l'opérateur **&&** étant prioritaire à l'opérateur d'affectation **=**, les opérations effectuées deviennent **0 && a**, puis **=10**, ce qui génère une erreur.

```
echo $((1 && a=10 )) a=$a
```

```
bash: 0 && a=10  : tentative d'affectation à une non-variable
(le symbole erroné est « =10  »)
```

Voir plus bas les priorités des opérateurs.

S'il n'est pas pertinent d'étudier l'opérande de droite et qu'il existe d'autres opérateurs, alors les

opérations se poursuivent.

```
a=0
echo $((a && (a=10) || (a=20) )) a=$a   # Avec a=0 (faux) et l'opérateur &&,
il n'est pas pertinent d'étudier le second opérande
                    # L'opération se reporte sur l'opérateur ||, étant
pertinent, son opérande de droite est étudié.
a=1
echo $((a && (a=10) || (a=20) )) a=$a
unset a
```

```
1 a=20
1 a=10
```

Les opérations logiques retournent une valeur booléenne.

Les commandes **let** et **((..))** suivent le standard C.
Elles numérisent **faux** avec **0** et **vrai** avec une valeur **non nulle**.

Avec bash c'est l'inverse, **faux** est une valeur **non nulle** et **vrai** correspond à **0**.

Il en résulte que le code retour renvoyé est l'inverse du résultat des opérations logiques effectuées par **let** ou **((..))**

```
echo $(( !(0) ))
(( !(0) )); echo $?
```

```
1
0
```

Les opérateurs relationnels

Les opérateurs relationnels permettent la comparaison entre deux epressions numériques.
Voir Comparaison numérique avec ((...))

Comme avec les opérateurs logiques, le résultat des comparaisons des opérateurs relationnels retourne une valeur booléenne.
(**vrai** en cas de succès **faux** en cas d'échec).

Le code de retour renvoyé est donc contraire au résultat des comparaisons effectuées par **let** ou **((..))**

```
echo $((24<42))
((24<42)); echo $?
```

```
1
0
```

Les opérateurs bit-à-bit

Les opérateurs bit-bit, opèrent sur des nombres binaires (une suite de 0 ou de 1).
Les nombres sont d'abord convertis en binaire, l'opération est appliquée, puis le résultat est reconverti en décimal.

Les opérateurs bit-à-bit ne doivent pas être confondus avec les opérateurs logiques

Liste des opérateurs bit-à-bit		
Opérateurs	**Désignations**	**Résultats**
Opérateur unaire		
~ *expr*	Négation binaire	Renvoie le complément à 1 de *expr* (~1 → 0 ; ~0 → 1)
Opérateurs binaires		
expr1 **&** *expr2*	ET binaire	Renvoie le ET bit-à-bit entre *expr1* et *expr2* (1 ET 1 → 1 ; 0 ET 0 → 0 ; 1 ET 0 → 0 ; 0 ET 1 → 0)
expr1 **|** *expr2*	OU binaire	Renvoie le OU bit-à-bit entre *expr1* et *expr2* (1 OU 1 → 1 ; 0 OU 0 → 0 ; 1 OU 0 → 1 ; 0 OU 1 → 1)
expr1 **^** *expr2*	OU binaire exclusif (XOR)	Renvoie le OU bit-à-bit exclusif entre *expr1* et *expr2* (1 XOR 1 → 0 ; 0 XOR 0 → 0 ; 1 XOR 0 → 1 ; 0 XOR 1 → 1)
expr1 **<<** *expr2*	Décalage binaire à gauche	Renvoie le décalage de *expr2* bit(s) à gauche de *expr1* (*expr2* 0 ajoutés(s) à droite d'*expr1*)
expr1 **>>** *expr2*	Décalage binaire à droite	Renvoie le décalage de *expr2* bit(s) à droite de *expr1* (*expr2* bit(s) tronqué(s) à droite d'*expr1*)
expr, *expr1* et *expr2*, sont sujettes au remplacement des paramètres (et des variables), à la substitution de commandes et à la suppression des protections.		

Exemples

```
echo $((~42))
echo $((42^42))
echo $((42<<2))
```

```
-43
0
```

```
168
```

Les opérateurs conditionnels

Les opérateurs conditionnels permettent de réaliser des opérations sous conditions.
Les opérations sélectionnées sont déterminées selon une structure, **si** ... **alors** ... **sinon** ..., mais en plus succin.

Les opérateurs conditionnels		
opérateurs	**Désignations**	**Explications**
expr1 **?** *expr2* **:** *expr3*	Opérateur conditionnel	*epxr1* est comprise comme un booléen : **Vrai** (**!=0**) ou **faux** (**=0**) Si *expr1* est vrai **?** (alors) *expr2* **:** (sinon) *expr3*.
expr1, *expr2* et *expr3*, sont sujettes au remplacement des paramètres (et des variables), à la substitution de commandes et à la suppression des protections.		

Exemples :

```
echo $((24<42 ? 5 : 10 ))
let "24>42 ? (a=5) : (a=10)"
echo $a

unset a
```

```
5
10
```

Les conditions peuvent s'imbriquer :

```
a=42 b=10
echo $(( a ? b > 0 ? 100 : -100 : 24 ))
b=-10
echo $(( a ? b > 0 ? 100 : -100 : 24 ))
a=0
echo $(( a ? b > 0 ? 100 : -100 : 24 ))

unset a b
```

```
100
-100
24
```

Cela risque rapidement de devenir illisible, pour plus de clarté, nous pouvons utiliser des parenthèses.

```
echo $(( a ? (b > 0 ? 100 :-100) : 24 ))
```

```
24
```

Les opérateurs conditionnels sont deux et ne peuvent s'utiliser seuls.

```
((24<42 ? 5))
echo $?
((24<42 : 5))
echo $?
```

```
bash: ((: 24<42 ? 5 : « : » attendu pour une expression
conditionnelle (le symbole erroné est « 5 »)
1
bash: ((: 24<42 : 5 : erreur de syntaxe dans l'expression (le
symbole erroné est « : 5 »)
1
```

Priorités des opérateurs

La priorité des opérateurs précise l'ordre dans lequel les opérations sont effectuées dans une expression complexe.
L'associativité permet de définir l'ordre des opérations pour les opérateurs de même priorité.
La priorité et l'associativité des opérateurs des commande **let** et **((...))** sont identiques au langage C.

Priorités et associativité des opérateurs		
Opérateurs	**Désignations**	**Associativités**
Les priorités sont classé ci-dessous par ordre décroissant. Sur une même ligne, les opérateurs ont même priorité.		
(…)	Parenthèses	
expr**++** expr**–**	Post-incrément et post-décrément de variables	→
- +	Moins et plus unaires	←
++expr **–**expr	Pré-incrément et pré-décrément de variables	←
! ~	Négations logique et binaire	←
******	Exponentiation (puissance)	
*** / %**	Multiplication, division, modulo	→
+ -	Addition, soustraction	→
<< >>	Décalage arithmétique à gauche et à droite	→
<= >= < >	Relationnels	→
== !=	Egalité et différence	→
&	ET binaire	→
^	OU exclusif binaire	→
|	OU binaire	→
&&	ET logique	→
||	OU logique	→
? :	Opérateur conditionnel	←
= *= /= %= += -= <<= >>= &= ^= |=	Affectations	←
,	Virgule	

Sources : Les priorités proviennent de page du manuel de bash (section ÉVALUATION ARITHMÉTIQUE).
Les associativités du langage C.

Les parenthèses ont la priorité la plus haute, elles permettent d'outre-passer toutes les autres.

Exemples :

```
echo $(( 18 + 3 * 2 ))

echo $(( (18 + 3) * 2 ))
```

```
24
42
```

Écriture utile pour les boucles

Post-incrémentation et pré-incrémentation :

script

```
#!/bin/bash
declare -i x=20 y  # ici les signes = permettent une affection
(( y = x++ ))      # d'abord la valeur de x est conservée dans la
valeur de y (donc $y= 20) puis la valeur
                   #  de x est incrémentée ($x est donc égal à 21)
                   # les espaces autour du signe = ne sont pas
obligatoires
echo "y=$y x=$x"   # réponse : y=20 x=21
```

script

```
#!/bin/bash
declare -i x=20 y
(( y = ++x ))      # d'abord la valeur de x est incrémentée puis la
valeur de y reçoit
                   # la valeur du x incrémenté
                   # les espaces autour du signe = ne sont pas
obligatoires
echo "y=$y x=$x"   # réponse : y=21 x=21
```

Post-décrémentation et pré-décrémentation :

script

```
#!/bin/bash
declare -i x=20 y
(( y = x-- ))      # d'abord la valeur de x est conservée dans la
valeur de y (donc $y= 20)
                   # puis la valeur de x est décrémentée ($x est donc
égal à 19)
                   # les espaces autour du signe = ne sont pas
obligatoires
echo "y=$y x=$x"   # réponse : y=20 x=19
```

script

```
#!/bin/bash
declare -i x=20 y
(( y = --x ))           # d'abord la valeur de x est décrémentée
($x=19), puis la valeur de y
                        # reçoit la valeur du x incrémenté (donc $y=19)
                        # les espaces autour du signe = ne sont pas
obligatoires
echo "y=$y x=$x"        # réponse : y=19 x=19
```

En bref,
Post-incrémentation/décrémentation : Les signes d'incrémentation (**++**) ou de décrémentation (**--**) sont placés **après** une valeur à incrémenter (+1) ou à décrémenter (-1) ; cette valeur est conservée dans “y” puis elle est **incrémentée (+1)** ou **décrémentée (-1)**.

Pré-incrémentation/décrémentation : Les signes d'incrémentation (**++**) ou de décrémentation (**--**) sont placés **avant** une valeur à incrémenter ou à décrémenter ; cette valeur est **incrémentée (+1)** ou **décrémentée (-1)** puis elle est conservée dans “y”.

Tuto précédent

Bash : les opérateurs de comparaison numérique

La suite, c'est ici

Bash : les tableaux

1)

N'hésitez pas à y faire part de vos remarques, succès, améliorations ou échecs !