



# TOP 10 : les erreurs les plus fréquentes en programmation de scripts shell

Formations Logilin : <http://www.logilin.fr/>

Voici ci-dessous les dix erreurs les plus fréquemment commises lors de l'écriture de scripts shell (de type Bourne). Certaines peuvent être diagnostiquées dès le début de l'exécution, d'autres sont plus complexes à retrouver.

Saurez-vous deviner – sans lire l'explication - où se cache l'erreur en voyant le script et le résultat d'exécution ?

## Erreur numéro 1

La plus fréquente, l'incontournable, celle que tout le monde commet de temps à autres. Notre script est censé compter de 0 à 4.

### Script (erreur\_1.sh)

```
#!/bin/sh
i=0
while [ $i -le 5 ]; do
    echo $i
    i = (($i + 1))
done
```

### Résultat d'exécution

```
$ ./erreur_1.sh
./erreur_1.sh: line 6: i: command not found
0
./erreur_1.sh: line 6: i: command not found
0
./erreur_1.sh: line 6: i: command not found
0
[...]
(ctrl-c)
$
```

### Explication

Lors de l'affectation d'une variable il ne faut jamais laisser d'espace autour du signe « = »  
Sur la ligne 6 : `i = (($i + 1))` le shell croit qu'il doit exécuter la commande « i » en lui passant en arguments = et le résultat de `(($i + 1))`. Évidemment il ne trouve pas de commande « i » sur le système et échoue. En outre la variable n'étant pas incrémentée la boucle se poursuit à l'infini.

### Solution

Au vu du message d'erreur ci-dessus (`i: command not found`) on aura le réflexe de chercher sur la ligne erronée l'endroit où un espace est accolé à un signe « égal ».

## Erreur numéro 2

Cette erreur apparaît aussi assez fréquemment. Elle guette plus particulièrement les programmeurs Awk, Perl ou PHP, mais tout un chacun est susceptible de la commettre un jour ou l'autre.

Ici nous voulons afficher les carrés des nombres 1 à 4 :

### Script (erreur\_2.sh)

```
#!/bin/sh
for i in 1 2 3 4 ; do
    $j=$(( $i * $i ))
    echo "$i2 = $j"
done
```

### Résultat d'exécution

```
$ ./erreur_2.sh
./erreur_2.sh: line 4: =1: command not found
12 =
./erreur_2.sh: line 4: =4: command not found
22 =
./erreur_2.sh: line 4: =9: command not found
32 =
./erreur_2.sh: line 4: =16: command not found
42 =
$
```

### Explication

Lors de l'affectation de « j », on a préfixé par erreur la variable par l'opérateur « \$ ». Elle est donc remplacée par son contenu (vide) avant l'affectation.

Cette ligne devient « =1 », « =4 », « =9 »... et le shell la rejette tout naturellement.

Rappelons qu'avec le shell, le caractère « \$ » est opérateur qui sert à prendre le contenu d'une variable. Contrairement aux langages comme Perl ou PHP, le shell considère « \$n » comme le *contenu* de la variable n, et non comme une variable à part entière.

### Solution

Il ne doit donc jamais y avoir de caractère « \$ » au début d'une ligne d'affectation.

On peut même considérer, de manière plus générale, qu'il ne doit pas y avoir de caractère « \$ » à gauche d'un signe « = » (sauf s'il s'agit d'un indice de tableau dont on remplit une case).

## Erreur numéro 3

Encore une erreur très fréquente. Elle est due à une contrainte syntaxique du shell, dont la rigueur n'est souvent pas très intuitive.

Notre script affiche un message, lit la réponse de l'utilisateur et agit en conséquence.

### Script (erreur 3.sh)

```
#!/bin/sh

printf "Voulez-vous quitter (O/N) "
read reponse
if [ "$reponse" = "0"]
then
    exit
fi
printf "suite du script...."
```

### Résultat d'exécution

```
$ ./erreur_3.sh
Voulez-vous quitter (O/N) N
./erreur_3.sh: line 5: [: missing `]'
suite du script....
$
```

### Explication

Le crochet fermant du test est collé à l'apostrophe encadrant le 0. Le shell pense donc que ce crochet fait partie de la chaîne de caractères testée.

En effet, le shell ne considère pas que les crochets soient des caractères spéciaux. Pour lui le crochet ouvrant « [ » est une commande (sur certains systèmes Unix, il existe même un fichier `/usr/bin/[` ). Une commande doit donc être distinguée du reste de la ligne par des espaces.

En outre, le crochet fermant « ] » doit apparaître isolé, c'est un argument indépendant de la commande « [ ».

### Solution

À retenir : toujours des espaces autour des crochets de test (pour `if`, `while`, etc.)

Ceci doit devenir un réflexe. Jamais d'espaces autour du signe « = » et toujours des espaces autour des crochets.

## Erreur numéro 4

Une erreur courante qui peut rester dissimulée pendant longtemps, et n'apparaître que dans certains cas. Notre script vérifie si le fichier indiqué est un fichier normal.

### Script (erreur\_4.sh)

```
#!/bin/sh

printf "Indiquez un nom de fichier : "
read fichier
if [ ! -e $fichier ]; then
    printf "Il n'existe pas\n"
    exit
fi
if [ -d $fichier ]; then
    printf "C'est un répertoire\n"
    exit
fi
if [ ! -f $fichier ]; then
    printf "Ce n'est pas un fichier normal\n"
    exit
fi

printf "Ok !\n"
```

### Résultat d'exécution

```
$ ./erreur_4.sh
Indiquez un nom de fichier : /etc
C'est un répertoire
$ ./erreur_4.sh
Indiquez un nom de fichier : /etc/passwd
Ok !
```

Ces deux premières exécutions ne montrent aucun problème. Continuons...

```
$ ./erreur_4.sh
Indiquez un nom de fichier : Mes Documents
./erreur_4.sh: line 5: [: Mes: binary operator expected
./erreur_4.sh: line 9: [: Mes: binary operator expected
./erreur_4.sh: line 13: [: Mes: binary operator expected
Ok !
$
```

Non seulement il y a un problème qui se traduit par des messages d'erreur, mais en plus le résultat est faux, « Mes Documents » est ici un répertoire et non un fichier classique.

### Explication

Le problème vient de l'espace entre « Mes » et « Documents ». La ligne 9 se développe en « if [ -d Mes Documents ] », or le shell attend un seul nom de fichier après le « -d » mais trouve deux mots « Mes » et « Documents ». Il faut les conserver groupés.

### Solution

Une règle d'or : lors de la consultation d'une variable, toujours l'encadrer par des guillemets si elle peut contenir un espace ou d'être vide. Ceci concerne toutes les variables qui sont remplies « en-dehors » du script (arguments de ligne de commande, saisie d'utilisateur, etc.) y compris les noms de fichiers.

## Erreur numéro 5

Cette erreur est liée à une faiblesse de conception du langage de programmation du shell.

Elle touche généralement les scripts longs, qui s'étendent sur plusieurs pages de listings.

### Script (erreur\_5.sh)

Dans une partie du script, nous définissons une fonction qui sert à interroger l'utilisateur et à renvoyer 0 (vrai) s'il répond Y et 1 (faux) s'il répond N.

```
function confirmation()
{
    while true; do
        printf "(Y/N) ?"
        read rep
        if [ "$rep" = "Y" ]; then return 0; fi
        if [ "$rep" = "N" ]; then return 1; fi
    done
}
```

Beaucoup plus loin dans le script, nous parcourons une liste de répertoire en demandant confirmation avant de les sauvegarder (la sauvegarde proprement dit est faite par une fonction non écrite ici) :

```
# parcours des repertoires
for rep in Documents Donnees Scripts
do
    printf "Sauvegarder $rep ?"
    if confirmation; then
        sauvegarder_repertoire $rep
    fi
done
```

Voyez-vous le problème ?

### Explication

Le problème qui se pose est que le même nom de variable (rep) est employé pour stocker la réponse de l'utilisateur et le nom du répertoire. Lorsqu'on appellera la fonction sauvegarder\_repertoire, ce sera toujours avec « 0 » en argument à la place du nom du répertoire à sauver.

### Solution

Les variables qui sont destinées à être utilisées uniquement dans une fonction doivent y être déclarées en les précédant du mot-clé « local ».

Dans de nombreux langages, les variables sont locales par défaut, et si l'une d'elle doit être globale au programme, il faut l'indiquer avec un mot-clé spécifique. Le shell adopte le comportement inverse et c'est regrettable.

## Erreur numéro 6

Une erreur que l'on peut souvent éviter avec de bonnes habitudes.

Le script reçoit des nombres sur sa ligne de commande et affiche le plus petit d'entre-eux.

### Script (erreur\_6.sh)

```
#!/bin/sh
minimum="$1"
for i in "$@"
do
    if [ "$i" -lt "$minimum" ]
    then
        mimumum="$i"
    fi
done
echo "La plus petite valeur est $mimumum"
```

### Résultat d'exécution

```
$ ./erreur_6.sh 16 23 4 8 24 15
La plus petite valeur est
$
```

### Explication

Une faute de frappe s'est dissimulée dans la ligne d'affichage. On consulte le contenu de la variable « mimumum » (inexistante) au lieu de « minimum ».

### Solution

Cette erreur peut être détectée par la commande « set -u » à glisser en début de script, sur la seconde ligne par exemple. Elle demande au shell d'échouer avec un message d'erreur si on essaye de lire le contenu d'une variable inexistante.

Toutefois le problème reste entier si l'erreur arrive au moment d'une affectation de variable déjà existante (par exemple sur la ligne « minimum=\$i »), et il n'y a pas de solution absolue.

## Erreur numero 7

Petite erreur de syntaxe liée à l'aspect peu intuitif du shell.

### Script (erreur\_7.sh)

```
#!/bin/sh

jour[0]="Lundi"
jour[1]="Mardi"
jour[2]="Mercredi"
jour[3]="Jeudi"
jour[4]="Vendredi"
jour[5]="Samedi"
jour[6]="Dimanche"

for i in 0 1 2 3 4 5 6
do
    echo "Jour $i -> $jour[$i]"
done
```

### Résultat d'exécution

Le problème apparaît facilement à l'exécution, bien qu'il ne soit pas si facile à voir dans le listing.

```
$ ./erreur_7.sh
Jour 0 -> Lundi[0]
Jour 1 -> Lundi[1]
Jour 2 -> Lundi[2]
Jour 3 -> Lundi[3]
Jour 4 -> Lundi[4]
Jour 5 -> Lundi[5]
Jour 6 -> Lundi[6]
$
```

### Explication

Lorsque le shell voit \$jour[5], il l'interprète comme le contenu de la variable jour, suivi d'un 5 entre crochet. Le contenu de jour et de jour[0] étant identique, le script répète les « lundi ».

### Solution

Pour consulter la  $i^{\text{eme}}$  case du tableau jour, il faut écrire `${jour[$i]}`.

## Erreur numéro 8

Une erreur de syntaxe déroutante.

Supposons que nous écrivions un script d'installation d'un logiciel, et que nous voulions tester systématiquement toutes les étapes de l'installation.

Voici un fragment de script contenant deux fois la même erreur.

### Script (erreur\_8.sh)

```
#!/bin/sh
echo "Inserez le CD d'installation"
read
mount /mnt/cdrom || { echo "Impossible de monter le CD"; exit 1 }
tar -xf /mnt/cdrom/install.tar || { echo "Erreur d'install"; exit 1 }
```

### Résultat d'exécution

Il est inutile d'insérer un CD pour faire le test, l'erreur se produit toujours :

```
$ ./erreur_8.sh
Inserez le CD d'installation
(Entrée)
./erreur_8.sh: line 7: syntax error: unexpected end of file
$
```

### Explication

L'erreur vient de la syntaxe des commandes composées. Le shell ne trouve pas l'accolade fermante (il pense que c'est le second argument de « exit »), et la recherche en vain jusqu'à la fin du fichier.

Pourquoi ne la voit-il pas ? Parce que l'accolade fermante n'est interprétée comme fin de commande composée que si elle se trouve en début de ligne.

### Solution

Lorsque vous encadrez des commandes par des accolades, faites toujours précéder l'accolade fermante par un point-virgule (pour le shell c'est l'équivalent d'un retour à la ligne).

## Erreur numéro 9

Voici une erreur subtile, le script peut très bien marcher sur certaines implémentations du shell Bourne (par exemple sur les shells Korn) et pas sur d'autres (par exemple Bash).

### Script (erreur\_9.sh)

```
#!/bin/sh
find /etc -type f 2>/dev/null | wc -l | read nb
echo "il y a $nb fichiers normaux dans l'arborescence /etc"
```

### Résultat d'exécution

```
$ ./erreur_9.sh
il y a      fichiers normaux dans l'arborescence /etc
$
```

### Explication

Le problème vient de la fin du pipeline, la partie « | read nb ». L'interprétation de cette opération dépend du shell.

Avec un shell Korn, le résultat est bien envoyé dans la variable nb, et le script affiche alors :

```
$ ./erreur_9.sh
il y a 1619 fichiers normaux dans l'arborescence /etc
$
```

Avec un shell Bash, la dernière commande (read) est exécutée dans un processus indépendant du shell – comme les autres commandes. Et la variable nb remplie n'est pas la même que celle du shell, c'est une variable appartenant uniquement à la mémoire de la commande read, et à la ligne suivante du script, c'est le contenu d'une autre variable nb, celle appartenant à la mémoire du shell qui est affichée.

### Solution

La seule solution est d'éviter cette construction non portable. A la place on peut écrire :

```
nb=$( find /etc -type f 2>/dev/null | wc -l)
```

## Erreur numéro 10

Pour finir, cette petite capture de session interactive, où l'utilisateur débutant a bien du mal à comprendre ce qui se passe. Les erreurs ne viennent pas toujours des scripts !

### Résultat d'exécution

```
$ a=test
$ echo $a
test
$
```

Ok, ça marche, j'ai compris le principe, essayons un autre test :

```
$ a=test 2
bash: 2: command not found
$
```

Tiens ? Ah oui ! Il faut mettre des apostrophes autour des chaînes de caractères

```
$ a=`test 2`
$ echo $a
$
```

M'enfin ?

### Explication

Lorsque le débutant en programmation shell est confronté à plusieurs erreurs successives, le comportement du système peut lui paraître bien surprenant. Dans le son premier essai il remplit une variable a avec un mot et la consulte à nouveau : tout va bien.

Ensuite, il s'enhardit à écrire deux mots. Alors le shell pense avoir à faire à une première affectation temporaire de variable suivie d'un appel de commande, il essaye de lancer la commande « 2 » et échoue.

L'utilisateur se souvient alors qu'il faut encadrer la chaîne par des apostrophes, mais – malédiction – il utilise les mauvaises quotes, les *backquotes*, qui servaient autrefois à lancer les commandes. Aujourd'hui on leur préfère la syntaxe `$( ... )` plus lisible. Le shell exécute alors la commande « test » (qui existe et n'affiche rien) et capture son résultat (vide) pour le placer dans la variable a.

### Solution

Bien comprendre le rôle des différentes *quotes*.

Les apostrophes (*quotes* simples) servent à encadrer des chaînes qui ne doivent pas être interprétées. Les guillemets (*doubles quotes*) encadrent des chaînes dans lesquelles des noms de variables doivent être remplacés par leurs valeurs. Les apostrophes inverses (*backquotes*) encadrent une commande qui est invoquée et dont le résultat est capturé et remplace la chaîne initiale.

*Des remarques ? Des questions ?*

Contactez-nous sur Formations Logilin - <http://www.logilin.fr>