# Reprenez le contrôle à l'aide de Linux !

30 heures Facile

Mis à jour le 29/06/2021

**Le contenu de ce cours n'est plus à jour**

Nous avons archivé ce cours et n'actualiserons plus son contenu.

Accédez au contenu le plus récent en découvrant ce cours :

VOIR LE NOUVEAU COURS

## Afficher et manipuler des variables

Comme dans tous les langages de programmation, on trouve en bash ce que l'on appelle des **variables**. Elles nous permettent de stocker temporairement des informations en mémoire. C'est en fait la base de la programmation.

Les variables en bash sont assez particulières. Il faut être très rigoureux lorsqu'on les utilise. Si vous avez fait du C ou d'autres langages de programmation, vous allez être un peu surpris par leur mode de fonctionnement ; soyez donc attentifs.
Et si vous n'avez jamais programmé, soyez attentifs aussi.

## Déclarer une variable

Nous allons créer un nouveau script que nous appellerons `variables.sh` :

```
$ vim variables.sh
```

La première ligne de tous nos scripts doit indiquer quel shell est utilisé, comme nous l'avons appris plus tôt. Commencez donc par écrire :

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

Cela indique que nous allons programmer en bash.

Maintenant, définissons une variable. Toute variable possède un nom et une valeur :

```
message='Bonjour tout le monde'
```

Dans le cas présent :

- la variable a pour **nom** `message` ;
- ... et pour **valeur** `Bonjour tout le monde` .

Ne mettez pas d'espaces autour du symbole égal « = » ! Le bash est très pointilleux sur de nombreux points, évitez par conséquent de le vexer.
Je vous signalerai systématiquement les pièges à éviter, car il y en a un certain nombre !

Si vous voulez insérer une apostrophe dans la valeur de la variable, il faut la faire précéder d'un antislash \. En effet, comme les apostrophes servent à délimiter le contenu, on est obligé d'utiliser un **caractère d'échappement** (c'est comme ça que cela s'appelle) pour pouvoir véritablement insérer une apostrophe :

```
message='Bonjour c\'est moi'
```

Bien, reprenons notre script. Il devrait à présent ressembler à ceci :

```
#!/bin/bash

message='Bonjour tout le monde'
```

Exécutez-le pour voir ce qui se passe (après avoir modifié les droits pour le rendre exécutable, bien sûr) :

```
$ ./variables.sh
$
```

Il ne se passe rien !

Que fait le script, alors ?

Il met en mémoire le message `Bonjour tout le monde`, et c'est tout ! Rien ne s'affiche à l'écran !

Pour afficher une variable, il va falloir utiliser une commande dont je ne vous ai pas encore parlé...

## echo : afficher une variable

Avant de commencer à parler de **variables**, il y a une commande que j'aimerais vous présenter : `echo`. J'aurais pu en parler avant que l'on commence à faire des scripts bash, mais vous n'en auriez pas vu l'utilité avant d'aborder ce chapitre.

Son principe est très simple : elle affiche dans la console le message demandé. Un exemple :

```
$ echo Salut tout le monde
Salut tout le monde
```

Comme vous le voyez, c'est simple comme bonjour. Les guillemets ne sont pas requis.

Mais... comment est-ce que cela fonctionne ?

En fait, la commande `echo` affiche dans la console tous les paramètres qu'elle reçoit. Ici, nous avons envoyé quatre paramètres :

- `Salut` ;
- `tout` ;
- `le` ;
- `monde` .

Chacun des mots était considéré comme un paramètre que `echo` a affiché. Si vous mettez des guillemets autour de votre message, celui-ci sera considéré comme étant un seul et même paramètre (le résultat sera visuellement le même) :

```
$ echo "Salut tout le monde"
Salut tout le monde
```

Si vous voulez insérer des retours à la ligne, il faudra activer le paramètre `-e` et utiliser le symbole `\n` :

```
$ echo -e "Message\nAutre ligne"
Message
Autre ligne
```

### Afficher une variable

Pour afficher une variable, nous allons de nouveau utiliser son nom précédé du symbole dollar `$` :

```
#!/bin/bash

message='Bonjour tout le monde'
echo $message
```

> Comparez les lignes 3 et 4 : lorsque l'on **déclare** la variable à la ligne 3, on ne doit pas mettre de `$` devant. En revanche, lorsqu'on l'**affiche** à la ligne 4, on doit cette fois mettre un `$` !

Résultat :

```
Bonjour tout le monde
```

Maintenant, supposons que l'on veuille afficher à la fois du texte et la variable. Nous serions tentés d'écrire :

```
#!/bin/bash

message='Bonjour tout le monde'
echo 'Le message est : $message'
```

Le problème est que cela ne fonctionne pas comme on le souhaite car cela affiche :

```
Le message est : $message
```

Pour bien comprendre ce qui se passe, intéressons-nous au fonctionnement de ce que l'on appelle les « quotes ».

## Les quotes

Il est possible d'utiliser des **quotes** pour délimiter un paramètre contenant des espaces. Il existe trois types de quotes :

- les apostrophes ' ' (simples quotes) ;
- les guillemets " " (doubles quotes) ;
- les accents graves ` ` (back quotes), qui s'insèrent avec `Alt Gr + 7` sur un clavier AZERTY français.

Selon le type de quotes que vous utilisez, la réaction de bash ne sera pas la même.

### Les simples quotes ' '

Commençons par les simples quotes :

```
message='Bonjour tout le monde'
echo 'Le message est : $message'
```

```
Le message est : $message
```

Avec de simples quotes, la variable n'est pas analysée et le `$` est affiché tel quel.

### Les doubles quotes " "

Avec des doubles quotes :

```
message='Bonjour tout le monde'
echo "Le message est : $message"
```

```
Le message est : Bonjour tout le monde
```

... ça fonctionne ! Cette fois, la variable est analysée et son contenu affiché.

En fait, les doubles quotes demandent à bash d'analyser le contenu du message. S'il trouve des symboles spéciaux (comme des variables), il les interprète.
Avec de simples quotes, le contenu était affiché tel quel.

### Les back quotes ` `

Un peu particulières, les back quotes demandent à bash d'**exécuter** ce qui se trouve à l'intérieur.

Un exemple valant mieux qu'un long discours, **regardez la première ligne** :

```
message=`pwd`
echo "Vous êtes dans le dossier $message"
```

```
Vous êtes dans le dossier /home/mateo21/bin
```

La commande `pwd` a été exécutée et son contenu inséré dans la variable `message` ! Nous avons ensuite affiché le contenu de la variable.

Cela peut paraître un peu tordu, mais c'est réellement utile. Nous nous en resservirons dans les chapitres suivants.

## read : demander une saisie

Vous pouvez demander à l'utilisateur de saisir du texte avec la commande `read`. Ce texte sera immédiatement stocké dans une variable.

La commande `read` propose plusieurs options intéressantes. La façon la plus simple de l'utiliser est d'indiquer le nom de la variable dans laquelle le message saisi sera stocké :

```
read nomvariable
```

Adaptons notre script pour qu'il nous demande notre nom puis qu'il nous l'affiche :

```
#!/bin/bash

read nom
echo "Bonjour $nom !"
```

Lorsque vous lancez ce script, rien ne s'affiche, mais vous pouvez taper du texte (votre nom, par exemple) :

```
Mathieu
Bonjour Mathieu !
```

Notez que la première ligne correspond au texte que j'ai tapé au clavier.

### Affecter simultanément une valeur à plusieurs variables

On peut demander de saisir autant de variables d'affilée que l'on souhaite. Voici un exemple de ce qu'il est possible de faire :

```
#!/bin/bash

read nom prenom
echo "Bonjour $nom $prenom !"
```

```
Deschamps Mathieu
Bonjour Deschamps Mathieu !
```

`read` lit ce que vous tapez mot par mot (en considérant que les mots sont séparés par des espaces). Il assigne chaque mot à une variable différente, d'où le fait que le nom et le prénom ont été correctement et respectivement assignés à `$nom` et `$prenom` .
Si vous rentrez plus de mots au clavier que vous n'avez prévu de variables pour en stocker, la dernière variable de la liste récupèrera tous les mots restants. En clair, si j'avais tapé pour le programme précédent « Nebra Mathieu Cyril », la variable `$prenom` aurait eu pour valeur « Mathieu Cyril ».

### `-p` : afficher un message de prompt

Bon : notre programme n'est pas très clair et nous devrions afficher un message pour que l'utilisateur sache quoi faire. Avec l'option `-p` de `read` , vous pouvez faire cela :

```
#!/bin/bash

read -p 'Entrez votre nom : ' nom
echo "Bonjour $nom !"
```

Notez que le message `'Entrez votre nom'` a été entouré de quotes. Si on ne l'avait pas fait, le bash aurait considéré que chaque mot était un paramètre différent !

Résultat :

```
Entrez votre nom : Mathieu
Bonjour Mathieu !
```

C'est mieux !

### `-n` : limiter le nombre de caractères

Avec `-n` , vous pouvez au besoin couper au bout de X caractères si vous ne voulez pas que l'utilisateur insère un message trop long.

Exemple :

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

```
read -p 'Entrez votre login (5 caractères max) : ' -n 5 nom
echo "Bonjour $nom !"
```

```
Entrez votre login (5 caractères max) : mathiBonjour mathi !
```

Notez que le bash coupe automatiquement au bout de 5 caractères sans que vous ayez besoin d'appuyer sur la touche `Entrée` . Ce n'est pas très esthétique du coup, parce que le message s'affiche sur la même ligne. Pour éviter cela, vous pouvez faire un `echo` avec des `\n` , comme vous avez appris à le faire plus tôt :

```
#!/bin/bash

read -p 'Entrez votre login (5 caractères max) : ' -n 5 nom
echo -e "\nBonjour $nom !"
```

```
Entrez votre login (5 caractères max) : mathi
Bonjour mathi !
```

### `-t` : limiter le temps autorisé pour saisir un message

Vous pouvez définir un *timeout* avec `-t` , c'est-à-dire un nombre de secondes au bout duquel le `read` s'arrêtera.

```
#!/bin/bash

read -p 'Entrez le code de désamorçage de la bombe (vous avez 5 secondes) : '  -t 5 code
echo -e "\nBoum !"
```

### `-s` : ne pas afficher le texte saisi

Probablement plus utile, le paramètre `-s` masque les caractères que vous saisissez. Cela vous servira notamment si vous souhaitez que l'utilisateur entre un mot de passe :

```
#!/bin/bash

read -p 'Entrez votre mot de passe : ' -s pass
echo -e "\nMerci ! Je vais dire à tout le monde que votre mot de passe est $pass ! :)"
```

```
Entrez votre mot de passe :
Merci ! Je vais dire à tout le monde que votre mot de passe est supertopsecret38 ! :)
```

Comme vous pouvez le constater, le mot de passe que j'ai entré ne s'affiche pas lors de l'instruction `read` .

## Effectuer des opérations mathématiques

En bash, les variables sont toutes des chaînes de caractères. En soi, le bash n'est pas vraiment capable de manipuler des nombres ; il n'est donc pas capable d'effectuer des opérations.

Heureusement, il est possible de passer par des commandes (eh oui, encore). Ici, la commande à connaître est `let` .

```
let "a = 5"
let "b = 2"
let "c = a + b"
```

À la fin de ce script, la variable `$c` vaudra 7. Testons :

```
#!/bin/bash

let "a = 5"
let "b = 2"
let "c = a + b"
echo $c
```

```
7
```

Les opérations utilisables sont :

- l'addition : + ;
- la soustraction : - ;
- la multiplication : * ;
- la division : / ;
- la puissance : ** ;
- le modulo (renvoie le reste de la division entière) : %.

Quelques exemples :

```
let "a = 5 * 3" # $a = 15
let "a = 4 ** 2" # $a = 16 (4 au carré)
let "a = 8 / 2" # $a = 4
let "a = 10 / 3" # $a = 3
let "a = 10 % 3" # $a = 1
```

Une petite explication pour les deux dernières lignes :

- 10 / 3 = 3 car la division est entière (la commande ne renvoie pas de nombres décimaux) ;
- 10 % 3 renvoie 1 car le reste de la division de 10 par 3 est 1. En effet, 3 « rentre » 3 fois dans 10 (ça fait 9), et il reste 1 pour aller à 10.

Notez qu'il est possible aussi de contracter les commandes, comme cela se fait en langage C.
Ainsi :

```
let "a = a * 3"
```

... équivaut à écrire :

```
let "a *= 3"
```

Actuellement, les résultats renvoyés sont des nombres entiers et non des nombres décimaux. Si vous voulez travailler avec des nombres décimaux, renseignez-vous sur le fonctionnement de la commande `bc` .

## Les variables d'environnement

Actuellement, les variables que vous créez dans vos scripts bash n'existent que dans ces scripts. En clair, une variable définie dans un programme A ne sera pas utilisable dans un programme B.

Les variables d'environnement sont des variables que l'on peut utiliser dans n'importe quel programme. On parle aussi parfois de **variables globales**. Vous pouvez afficher toutes celles que vous avez actuellement en mémoire avec la commande `env` :

```
$ env
ORBIT_SOCKETDIR=/tmp/orbit-mateo21
GLADE_PIXMAP_PATH=:/usr/share/glade3/pixmaps
TERM=xterm
SHELL=/bin/bash
GTK_MODULES=canberra-gtk-module
USER=mateo21
PATH=/home/mateo21/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin: /usr/bin:/sbin:/bin:/usr/games
GDM_XSERVER_LOCATION=local
PWD=/home/mateo21/bin
EDITOR=nano
SHLVL=1
HOME=/home/mateo21
OLDPWD=/home/mateo21

[ ... ]
```

Il y en a beaucoup. Certaines sont très utiles, d'autres moins. Parmi celles que je peux vous commenter et qui peuvent s'avérer utiles, on trouve :

- `SHELL` : indique quel type de shell est en cours d'utilisation (sh, bash, ksh...) ;
- `PATH` : une liste des répertoires qui contiennent des exécutables que vous souhaitez pouvoir lancer sans indiquer leur répertoire. Nous en avons parlé un peu plus tôt. Si un programme se trouve dans un de ces dossiers, vous pourrez l'invoquer quel que soit le dossier dans lequel vous vous trouvez ;
- `EDITOR` : l'éditeur de texte par défaut qui s'ouvre lorsque cela est nécessaire ;
- `HOME` : la position de votre dossier `home` ;
- `PWD` : le dossier dans lequel vous vous trouvez ;
- `OLDPWD` : le dossier dans lequel vous vous trouviez auparavant.

Notez que les noms de ces variables sont, par convention, écrits en majuscules.

Comment utiliser ces variables dans vos scripts ? C'est très simple, il suffit de les appeler par leur nom !

Exemple :

```
#!/bin/bash

echo "Votre éditeur par défaut est $EDITOR"
```

```
Votre éditeur par défaut est nano
```

Plus rarement, vous pourriez avoir besoin de définir votre propre variable d'environnement. Pour cela, on utilise la commande `export` que vous avez pu voir dans votre `.bashrc` .

## Les variables des paramètres

Comme toutes les commandes, vos scripts bash peuvent eux aussi accepter des paramètres. Ainsi, on pourrait appeler notre script comme ceci :

```
./variables.sh param1 param2 param3
```

Le problème, c'est que nous n'avons toujours pas vu comment récupérer ces paramètres dans notre script. Pourtant, c'est très simple à réaliser !

En effet, des variables sont automatiquement créées :

- `$#` : contient le nombre de paramètres ;
- `$0` : contient le nom du script exécuté (ici `./variables.sh` ) ;
- `$1` : contient le premier paramètre ;
- `$2` : contient le second paramètre ;
- … ;
- **$9** : contient le 9e paramètre.

Essayons :

```
#!/bin/bash

echo "Vous avez lancé $0, il y a $# paramètres"
echo "Le paramètre 1 est $1"
```

```
$ ./variables.sh param1 param2 param3
Vous avez lancé ./variables.sh, il y a 3 paramètres
Le paramètre 1 est param1
```

Et si on utilise plus de neuf paramètres ? J'ai cru voir que les variables s'arrêtaient à `$9` …

Là, ça va un peu loin, mais ça peut arriver. On peut imaginer un script qui accepte une liste de fichiers en paramètre. Rien ne nous empêcherait de lui envoyer quinze paramètres dans ce cas :

```
./script.sh fichier1 fichier2 fichier3 fichier4 ... fichier14 fichier15
```

En général, pour traiter autant de paramètres, on s'occupera d'eux un par un... On peut « décaler » les paramètres dans les variables `$1` , `$2` , etc. à l'aide de la commande `shift` .

Reprenons notre script :

```
#!/bin/bash

echo "Le paramètre 1 est $1"
shift
echo "Le paramètre 1 est maintenant $1"
```

```
$ ./variables.sh param1 param2 param3
Le paramètre 1 est param1
Le paramètre 1 est maintenant param2
```

Comme vous le voyez, les paramètres ont été décalés : `$1` correspond après le `shift` au second paramètre, `$2` au troisième paramètre, etc.

Bien sûr, `shift` est généralement utilisé dans une boucle qui permet de traiter les paramètres un par un. Nous verrons d'ailleurs comment faire des boucles dans peu de temps.

## Les tableaux

Le bash gère également les variables « tableaux ». Ce sont des variables qui contiennent plusieurs cases, comme un tableau. Vous en aurez probablement besoin un jour ; voyons comment cela fonctionne.

Pour définir un tableau, on peut faire comme ceci :

```
tableau=('valeur0' 'valeur1' 'valeur2')
```

Cela crée une variable `tableau` qui contient trois valeurs ( `valeur0` , `valeur1` , `valeur2` ).

Pour accéder à une case du tableau, il faut utiliser la syntaxe suivante :

```
echo ${tableau[2]}
```

... ceci affichera le contenu de la case n° 2 (donc `valeur2` ).

Les cases sont numérotées à partir de 0 ! La première case a donc le numéro 0.
Notez par ailleurs que pour afficher le contenu d'une case du tableau, vous devez entourer votre variable d'accolades comme je l'ai fait pour `${tableau[2]}` .

Vous pouvez aussi définir manuellement le contenu d'une case :

```
tableau[2]='valeur2'
```

Essayons tout ceci dans un script :

```
#!/bin/bash

tableau=('valeur0' 'valeur1' 'valeur2')
tableau[5]='valeur5'
echo ${tableau[1]}
```

À votre avis, que va afficher ce script ?

Réponse :

```
valeur1
```

Comme vous pouvez le constater, le tableau peut avoir autant de cases que vous le désirez. La numérotation n'a pas besoin d'être continue, vous pouvez sauter des cases sans aucun problème (la preuve, il n'y a pas de case n° 3 ni de case n° 4 dans mon script précédent).

Vous pouvez afficher l'ensemble du contenu du tableau d'un seul coup en utilisant `${tableau[*]}` :

```
#!/bin/bash

tableau=('valeur0' 'valeur1' 'valeur2')
tableau[5]='valeur5'
echo ${tableau[*]}
```

```
valeur0 valeur1 valeur2 valeur5
```

## En résumé

- Comme dans la plupart des langages de programmation, on peut créer des variables en shell qui stockent temporairement des valeurs en mémoire. Une variable nommée `variable` est accessible en écrivant `$variable` .
- La commande `echo` affiche un texte ou le contenu d'une variable dans la console.
- `read` attend une saisie au clavier de la part de l'utilisateur et stocke le résultat dans une variable.
- On peut effectuer des opérations mathématiques sur des nombres à l'aide de la commande `let` .
- Certaines variables sont accessibles partout, dans tous les scripts : ce sont les variables d'environnement. On peut les lister avec la commande `env` .
- Les paramètres envoyés à notre script (comme `./script -p` ) sont transmis dans des variables numérotées : `$1` , `$2` , `$3` ... Le nombre de paramètres envoyés est indiqué dans la variable `$#` .

**Que pensez-vous de ce cours ?**

J'AI TERMINÉ CE CHAPITRE ET JE PASSE AU SUIVANT

INTRODUCTION AUX SCRIPTS SHELL

LES CONDITIONS

## Le professeur

**Mathieu Nebra**

Entrepreneur à plein temps, auteur à plein temps et co-fondateur d'OpenClassrooms :o)

## Découvrez aussi ce cours en...

Livre PDF

FAQ

POUR LES ENTREPRISES

Former et recruter

EN PLUS

Boutique

Mentions légales

Conditions générales d'utilisation

Politique de protection des données personnelles

Cookies

Accessibilité

Français

Télécharger dans l'App Store