

# Script bash : variables, arguments, paramètres

## Création, suppression, exportation de variables

Le nom d'une variable est un simple pointeur vers l'emplacement mémoire où sont conservées les données qu'elle contient.

Les variables qu'on crée dans un script (ou dans le terminal) sont localisées dans ce script (ou à l'ouverture d'un terminal) c'est-à-dire qu'elles ne sont utilisables que lorsqu'on exécute son script, (ou que l'on appelle la valeur d'une variable qu'on vient de déclarer dans un terminal). Et il s'agit du script d'un utilisateur, il faut les distinguer [des variables de substitution prédéfinies](#) et [des variables d'environnement prédéfinies](#).

Mais comment enregistrer une valeur en mémoire ?

C'est par exemple, l'affectation d'une valeur au nom d'une variable qui va permettre d'enregistrer en mémoire cette variable avec sa valeur



Avant tout prenez bien conscience que la déclaration d'une variable n'est pas confinée au script, mais qu'il est possible de déclarer une variable dans le shell courant (dans le terminal). Voir absolument : [détail sur le caractère \\$](#).

**Voyons d'abord comment créer une variable de cette manière et comment utiliser sa valeur.** 😊

### Affectation directe :

La déclaration d'une variable se fait lors de son affectation, c'est-à-dire lorsqu'on assigne au nom de la variable une valeur au moyen du caractère = (sans espace avant et après).

```
#!/bin/bash  
nom_de_la_variable=ValeurDeLaVariable
```

La valeur ValeurDeLaVariable a été mémorisée.

### **\$nom\_de\_la\_variable**

Pour "utiliser" une variable, on se sert de sa valeur : il faut donc appeler sa valeur et cela se fait avec le caractère spécial \$ accolé au nom de la variable :

[script](#)

```
#!/bin/bash
nx_fichier=les-fonctions
touch ~/ $nx_fichier
ls -la ~/ $nx_fichier
```

Les programmes (ou commandes) touch et ls ont utilisé la valeur de la fonction nommée nx\_fichier, dont la valeur correspond à la chaîne de caractères les\_fonctions.

## Une variable n'est pas typée

La valeur d'une variable peut être un nombre, un ou plusieurs caractères, du texte espacé, une commande, la valeur d'une variable.

### script

```
#!/bin/bash
var1=a
var2=texte
var3="texte avec espaces"
var4=55
var5=$var1          #ici on affecte à la variable var5, la
                    #valeur de la variable var1
var6=$0             #ici on affecte à la variable var6, la
                    #valeur de la variable pré-définie $0 (1)
echo -e "valeur de var1: $var1\nvaleur de var2: $var2\nvaleur de var3:
$var3\nvaleur de var3: $var4\nvaleur de var5: $var5\nvar6: $var6"
```

(1) \$0 a pour valeur pré-définie le nom du programme

```
valeur de var1: a
valeur de var2: texte
valeur de var3: texte avec espaces
valeur de var3: 55
valeur de var5: a
var6: /home/hypathie/MesScripts/mon-script
```

## Déclaration de plusieurs variables sur une ligne

On peut déclarer plusieurs variables sur une même ligne, il suffit pour cela de mettre un espace entre chacune :

### script

```
#!/bin/bash
set -o posix
var1=a var2=texte var3="texte avec espaces" var4=55 var5=$var1
var6=$0
/bin/echo -e "valeur de var1: $var1\nvaleur de var2: $var2\nvaleur de
var3: $var3\nvaleur de var3: $var4\nvaleur de var5: $var5\nvar6: $var6"
# même retour que précédemment
```

Le nom d'une variable peut être composé :

- par des lettres de a-z ou de A-Z ;
- par des chiffres de 0-9 .
- Il peut contenir un underscore \_.



Mais il ne doit jamais :

- commencer par un nombre ;
- par underscore ;
- un caractère spécial ;
- ni être un mélange de lettres minuscules et majuscules.

- Voir [l'exemple 4.3. Affectation de variable, basique et plus élaborée](#)
- Voir [Variable nulle et variable non-déclarée dans l'exemple](#)

Pour concaténer les valeurs deux variables :

script



```
#!/bin/bash
var1=lala
var2=li
var3=$var1$var2
echo $var3
```

## Affectation par la lecture : read

On peut créer des variables au moyen de commandes, comme par exemple la commande read qui est une commande interne (ou primitive) au shell.

- Syntaxe :

```
read nom_de_la_variable
```

- La valeur est enregistrée par l'utilisateur sur l'entrée standard (i.e. ce qu'on écrit à l'invite de commande).
- Le nom de la variable s'écrit juste après read, ce n'est qu'un pointeur vers la valeur que vous avez rentrée.

Par exemple dans un script :

script

```
#!/bin/bash
echo "Bonjour : qui êtes-vous ?"
read nom
echo "Enchanté $nom !"
```



De même dans le terminal on peut tout à fait entrer tour à tour chacune des lignes de ce script, essayez ! 😊

- L'option -p permet d'insérer un message avant l'attente de la valeur que l'utilisateur doit entrer.

script

```
#!/bin/bash
read -p "entrez votre prénom: " prenom
echo "bonjour $prenom !"
```

Ici prenom est le nom de la variable, et sa valeur est entrée par l'utilisateur depuis le terminal.

Là aussi ces deux commandes peuvent être entrées dans le terminal.

- read permet de déclarer plusieurs variables successivement (dans terminal ou script):

```
read -p "entrez votre nom et prénom: " nom prenom
```

retour de la commande

```
entrez votre nom et prénom:
```

On entre par exemple les deux chaînes de caractères, debian et facile, puis on peut récupérer la valeur de chacune des deux variables nom et prenom.

```
echo $prenom $nom
```

retour de la commande

```
facile debian
```

Mais dans un script c'est plus rapide !



- read et variable non-déclarée

script

```
#!/bin/bash
read -p "entrez votre nom d'utilisateur: "
echo "bonjour $USER !"
```

Ci-dessus, on n'a pas mis le nom de la variable, parce qu'on ne cherchera pas à utiliser la valeur de cette variable. Cela permet au programme de se poursuivre. Quand on entre dans le terminal la réponse attendu par read, le retour de la commande est  $\emptyset$ , donc elle est considérée comme exécutée, et le shell passe à l'exécution de la commande suivante.

Quelques options utiles de la commande read.

- -p : afficher un message
- -n : limiter le nombre de caractères
- -t : limiter le temps autorisé pour saisir un message
- -s : ne pas afficher le texte saisi



On peut utiliser plusieurs options.  
Par exemple:

```
read -p "entrez votre année de naissance (deux derniers chiffres): " -n 2 annee
```

## Modification de la valeur d'une variable

Pour modifier la valeur d'une variable, il suffit de l'affecter d'une nouvelle valeur.

script

```
#!/bin/bash
var1=bonjour
echo $var1
var1=23
```

```
echo $var1
var1=
echo $var1
var1=Bonjour
echo $var1
```

```
bonjour
23

Bonjour
```

## Protection de variable : "readonly"

On protège une variable avec la commande `readonly`. La variable devient alors une variable en lecture seule. Cela lui donne la caractéristique d'être figée : on ne peut plus alors, au cours du même script, réaffectée par une nouvelle valeur une variable en lecture seule, y compris si cette variable a été déclarée nulle.

script

```
#!/bin/bash
var1=toto
var2=
echo "$var1 $var2"
readonly var1 var2
var1=titi
var2=titi
unset var1 var2
```

```
toto
ligne6: var1 : variable en lecture seule
ligne7: var2 : variable en lecture seule
ligne 8 : unset: var1 : « unset » impossible : variable est en lecture
seule
ligne 8 : unset: var2 : « unset » impossible : variable est en lecture
seule
```

## Suppression de variable : unset

Soit le script "essai.sh" : passons à ce script l'argument vous

script `essai.sh`

```
#!/bin/bash
var1=coucou
```

```
var2=$1
echo "$var1 $var2"
unset var2
echo "$var1 $var2"
var1=$1
var2=vous
echo "$var1 $var2"
unset var2
echo "$var1 $var2"
```

script `essai.sh`

```
#!/bin/bash
var1=yep
var2=coucou
echo "$var1 $var2"
unset var2
echo "$var1 $var2"
echo " "
echo "mais pour un paramètre :"
echo " "
var1=yep
var2=coucou
var3=$1
echo "$var1 $var2 $var3"
unset var2 var3
echo "$var1 $var2 $var3"
var2=
var3=$1
echo "$var1 $var2 $var3"
```

```
yep coucou
yep

mais pour un paramètre :

yep coucou
yep
yep
```

## Exportation de la valeur d'une variable

### Définition

Exporter la valeur d'une variable signifie que l'on envoie à un processus fils, la valeur d'une variable depuis un processus père.



Un processus, c'est un programme en cours d'exécution, mais aussi son environnement d'exécution.

Ainsi le shell qui lance un script est le processus père du script lancé. Sur la notion de processus voir [Notion de processus](#)

## Exemple

On peut exporter depuis le terminal, une valeur pour remplacer provisoirement une variable d'un script.

```
Coucou="Bonjour"
```

On déclare la variable Coucou, affectée de la valeur Bonjour depuis le terminal.

Le shell courant l'a enregistré.

```
echo 'echo "Coucou=$Coucou" ' > test.sh
```

Là, on crée le fichier "test.sh" contenant la ligne : echo "Coucou=\$Coucou" .

Il contient une variable, de même nom que celle, précédemment déclarée et affectée de la valeur Bonjour depuis le terminal.

```
chmod u+x test.sh
```

Le fichier "test.sh" devient exécutable pour l'utilisateur principal.

```
export Coucou
```

On exporte la variable Coucou<sup>1)</sup>.

Attention, ce n'est jamais la valeur d'une variable que l'on exporte !

```
./test.sh
```

[retour de la commande](#)

```
Coucou="Bonjour"
```

Il faut que le script déclare une variable de même nom (Coucou= ) ; qu'il récupère celle exportée depuis le terminal (\$Coucou) ; et bien évidemment, qu'il affiche (echo) tout cela :( echo "Coucou=\$Coucou" ).

Dans ces conditions, la valeur de la variable Coucou une fois exportée, peut valoir pour la variable du script "test.sh" qui est le processus fils du shell courant.

Mais une fois le terminal réinitialisé, si on lance ./test.sh, ce script est le

processus fils d'un "nouveau" processus père (= le nouveau terminal) qui n'a plus en mémoire la valeur Bonjour pour la variable nulle Coucou du script.  
Et dans ce cas :

```
./test.sh
```

[retour de la commande](#)

```
Coucou=
```

Pour réinitialiser son terminal,  
il suffit de fermer et de le ré-ouvrir  
ou de recharger son fichier ~/.bashrc :



```
source ~/.bashrc
```

ou

```
./~/.bashrc
```

## Quand les valeurs sont des paramètres



On peut considérer que les termes paramètre et argument sont synonymes.  
Le terme paramètre de position, renvoie à l'appel de la valeur des paramètres (ou arguments) passés au script.

### Utiliser des paramètres de positions

Lorsqu'on ajoute un argument au script avant son exécution, on peut alors récupérer la valeur de ce paramètre.

- Pour récupérer chaque paramètre : \$1 ; \$2 ; \$3 etc.
- Pour récupérer tous les paramètres : \$@
- Pour récupérer le nombre de paramètres passés au script : \$#
- Pour récupérer la dernière commande : \$?

(Par défaut, 0 quand tout s'est bien passé, 1 quand il y a une erreur, sinon on fait exit xx, \$? affiche xx)

EXEMPLES : soit le script "essai.sh"

## script essai.sh

```
#!/bin/bash
var=Bonjour
echo $var
echo "$1"
echo "$2"
echo "$3"
echo "ou le paramètre 1 est: $1, le deuxième est: $2, le troisième est
: $3"
echo " "
echo "tous les paramètres $@"
```

```
./essai.sh a b c
```

## retour de la commande

```
Bonjour # on peut récupérer une valeur (ou des valeurs) déclarée(s) et
les paramètres de position
a
b
c
ou le paramètre 1 est: a, le deuxième est: b, le troisième est : c

tous les paramètres a b c
```

- Tous les arguments passés au scripts \$\* et \$@ sont synonymes

## script

```
#!/bin/bash
echo $1
echo $*
echo $@
echo $#
```

```
./essai.sh bonjour à tous
```

## retour de la commande

```
bonjour
bonjour à tous
bonjour à tous
3
```

- Là de même pour \$\* et \$@ :

## script

```
#!/bin/bash
echo $1
echo $*
echo $@
echo $#
```

```
./essai.sh "bonjour à tous"
```

## retour de la commande

```
bonjour à tous
bonjour à tous
bonjour à tous
1
```

- Mais avec la commande set qui modifie provisoirement les paramètres :

Pour plus de détails sur la commande set voir : [script-bash-detail-sur-les-parametres-et-les-boucles](#)

## script essai.sh

```
#!/bin/bash
set "bonjour à tous"
echo $*
echo $@
echo $1
echo $#
```

```
./essai.sh
```

## retour de la commande

```
bonjour à tous
bonjour à tous
bonjour à tous
1
```

- Ou encore :

## script essai.sh

```
#!/bin/bash
set bonjour à tous
echo $*
echo $@
echo $1
echo $#
```

```
./essai.sh
```

[retour de la commande](#)

```
bonjour à tous
bonjour à tous
bonjour
3
```

[script essai.sh](#)

```
#!/bin/bash

Nombre_arguments_attendus=1

if [ $# -ne $Nombre_arguments_attendus ]; then
  echo "Le nombre d'arguments est invalide : $#"
```

```
  echo "Nombre argument attendu : ${Nombre_arguments_attendus} "
```

```
fi
```

```
#if [ "$#" -ne 1 ]; then
```

```
# echo "Le nombre d'arguments est invalide"
```

```
#fi
```

```
echo "Script Started !"
```

```
./essai.sh
```

[retour de la commande](#)

```
Script Started !
```

## Récupérer la longueur d'une valeur de variable

- Pour obtenir la longueur d'une chaîne stockée dans une variable, on écrit `${#VAR}`.

Exemples :

script

```
#!/bin/bash
var="j'aime debian-facile"
echo ${#var}
```

- Pour récupérer la longueur d'un paramètre de position :



## Substitutions de commande

### Utilisation

Permet de se servir de la sortie d'une commande dans un autre contexte pour ;

1. affecter cette sortie à une variable ;
2. utiliser cette sortie comme argument d'une autre commande
3. etc.

### Deux syntaxes :

```
`commande`
```

OU

```
$(commande)
```

### Substitution simple : \$(commande)

script

```
#!/bin/bash
dir=$(pwd)
echo "mon répertoire est : $dir"
```

```
mon répertoire est : /home/hypathie
```

- plusieurs commandes:

script

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

```
echo $(pwd ; ls)
```

## Imbrication de commandes : \$(cmd \$(cmd))

```
echo $( ls $(pwd)/Documents )
```

## Imbrication avec "set"

script

```
#!/bin/bash
set $(pwd ; whoami)
echo "$1 : $2"
echo $#
```

Ou

script

```
#!/bin/bash
set -- $(ls -l $(pwd)/.bashrc)
echo $*
```

## Typologie des variables

Comme nous l'avons vu on peut affecter une variable par différents types de valeurs ; des chaînes de caractères, des nombres, des valeurs d'autres variables, des substitutions de commandes.

On dit pour cela qu'en bash les variables ne sont pas typées.

Mais il peut être intéressant de typer une variable. Pour ce faire, il faut utiliser des commandes internes à bash qui permettent de déclarer une variable typée.

## declare et typeset

commandes	options
declare/typeset	- r : lecture seule
declare/typeset	- i : entier
declare/typeset	- a : tableau (array)
declare/typeset	- f : fonction(s)
declare/typeset	- x : export
declare/typeset	- x : var=\$valeur

Voir : [Guide avancé d'écriture des scripts Bash: 9.4. Typage des variables : declare ou typeset](#)

## Remarques sur la commande "declare" et les calculs

### Méthode non POSIX

- La valeur d'une variable peut être une expression arithmétique, pour initialiser une variable de type entier on utilise l'option `-i` de la commande `declare` :  
`declare -i nom[=expression] nom[=expression] ...`

script

```
#!/bin/bash
declare -i x=35*2
echo $x
```

retour

70



- Pour que la valeur d'une variable ne soit pas accidentellement modifiée, il faut ajouter l'attribut `-r`.

script

```
#!/bin/bash
declare -ir a=35*2
declare -ir b=5+5
echo $((a+b))
```

80

**Méthode POSIX** : Les commandes `let` et `(( ... ))` permettent de typer les variables pour effectuer des calculs sur les variables (synonyme de `declare -i`).

## Variables numériques et calculs

### Les variables typées pour les calculs : `let` ou `(( ... ))`

Voir : [les opérateurs arithmétiques](#)

## Syntaxe

```
let 'var = 5 + 5'  
    OU LE SHELL ARITHMÉTIQUE :  
$(( 5 * 3 ))
```

## Exemples

### script

```
#!/bin/bash  
let "a = 10"  
let "b = 2"  
let "c = a+b"  
echo $c  
let "e = 10*2"  
echo $e  
let "f = 15"  
let "f *=2"  
echo $f  
echo " "  
let 'var = 5 + 5'  
echo "$var"  
echo " "  
echo $(( 20 + 20 ))  
var1="2"  
var2="5"  
echo $(( $var2 % $var1 ))
```

```
12  
20  
30  
  
10  
  
40  
1
```

## L'affectation arithmétique

Voir [les opérateurs d'affectation arithmétique](#)

Cela consiste à affecter à une variable le résultat d'un calcul arithmétique, par la constante (donc avec `let`) qu'on lui a donné.

Soit une variable `var` de valeur `x`, si l'on fait `var +=2` alors la variable `var` sera `x + 2`.

cela permet de faire des incrémentation par autre chose que 1.)  
Il en va de même pour les autres opérateurs.

- Exemples

```
let "a = 5"
let "b = 10"
let "c = a *= 3"
let "d = a += 3"      # valeur précédente de a conservée pour calculer d :
15+3 =18
let "e = b /= 3"
let "f = b /= 3"      # valeur précédente de b conservée pour calculer f : 3/3
=1
echo "c=$c d=$d e=$e f=$f" # réponse : c=15 d=18 e=3 f=1
i=1
let "i += 7"
echo "i=$i"          # réponse : i=8
j=4
let "j *= 5"
echo "j=$j"          # réponse : j=20
```

## Incrémentation, décrémentation

- incrémentation, décrémentation de la valeur 1 : (( var++ )) ; (( ++var )) ; (( var-- )) , etc.

### script

```
#!/bin/bash
let "var = 5"
echo "$var"
(( var++ ))
echo "$var"
(( var-- ))
echo "$var"
```

Retour :

```
5
6
5
```

- L'incrémentacion se fait aussi sur une boucle !



script

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

```
i=1 # on initialise le compteur
while [ $i -le 10 ]; do
  echo $i
  let $[ i+=1 ] # incremente i de 1 a chaque boucle
done
```



voir aussi ici

- Ou encore de cette manière :

```
N=$(( $N+1 ))
```

voir ici le code complet de cet extrait

## Changements de bases

Bash permet de changer de base (Il est par défaut en base 10)

- base 8 (octal) : un chiffre n précédé de zéro : 0n

exemple : 02 ⇒ 2 en base 8

- base 16(hexadécimal): un chiffre n précédé de zéro+x : 0xn

exemple : 0x3 ⇒ 3 en base 16

- autres bases : base#nombre
- base maximale : base 64

## Références

Le shell pour tous : "Variables et environnement"

variables

Guide avancé d'écriture des scripts Bash : Introduction aux variables et aux paramètres

1)

Il s'agit ben sûr, de celle déclarée dans le terminal au début de l'exemple

From:

<https://chanterie37.fr/fablab37110/> - Castel'Lab le Fablab MJC de Château-Renault

Permanent link:

<https://chanterie37.fr/fablab37110/doku.php?id=start:raspberry:bash4>

Last update: **2023/01/27 16:08**



