Scripting Shell GNU/Linux

1. Les paramètres d'un script

1.1. Paramètres de position

Les **paramètres de position** sont aussi des variables spéciales utilisées lors d'un **passage de paramètres** à un script.

Variable	Contenu
$0	Nom de la commande (du script).
$1-9	$1,$2,$3… Les neuf premiers paramètres passés au script.
$#	Nombre total de paramètres passés au script.
$*	Liste de tous les paramètres au format « $1 $2 $3 … ».
$@	Liste des paramètres sous forme d'éléments distincts « $1 » « $2 » « $3 » …

- Saisissez et testez le script **param.sh** en l'appelant avec les arguments suivants : # **param.sh riri fifi loulou** et identifiez les paramètres $0, $#, …

```
#!/bin/bash
echo "Nom : $0"
echo "Nombre de parametres : $#"
echo "Parametres : 1=$1 2=$2 3=$3"
echo "Liste : $*"
echo "Elements : $@"
```

Variable	**1. Complétez** le résultat
$0	Nom :
$#	Nombre de parametres :
$1, $2, $3	Parametres :
$*	Liste :
$@	Elements :

1.2. Redéfinition des paramètres

L'instruction **set** permet de **redéfinir le contenu des variables de position** pour par exemple affecter des valeurs par défaut lorsque le passage de paramètre est facultatif.

Avec : **set valeur1 valeur2 valeur3 …,** $1 prendra comme contenu valeur1, $2 valeur2 et ainsi de suite.

2. Saisissez le script et testez-le avec les paramètres suivants : **# param2.sh riri fifi loulou**

param2.sh	Notez, **complétez** le résultat
#!/bin/bash echo "Avant :" echo "Nombre de parametres : $#" echo "Parametres : 1=$1 2=$2 3=$3 4=$4" echo "Liste : $*" set alfred oscar romeo zoulou echo "apres set alfred oscar romeo zoulou" echo "Nombre de parametres : $#" echo "Parametres : 1=$1 2=$2 3=$3 4=$4" echo "Liste : $*"	Avant : Nombre de parametres : Parametres : Liste : apres set alfred oscar romeo zoulou Nombre de parametres : Parametres : Liste :

2. Tests de conditions

La commande **test** permet d'effectuer des tests de conditions. Le résultat est récupérable par la variable système **$?** (code retour). Si ce résultat est 0 alors la condition est réalisée.

2.1. Tests sur une chaîne

- **test -z "variable"** : teste si la variable est vide (ex : test -z "$a").
- **test -n "variable"** : l'inverse, teste si la variable n'est pas vide.
- **test "variable" = chaîne** : OK si les deux chaînes sont identiques.
- **test "variable" != chaîne** : OK si les deux chaînes sont différentes.

3. Saisissez le script **testCond.sh**.
Attention de bien placer les espaces autour des opérateurs (« **=** » et « **!=** »):

```
#!/bin/bash
a=$1
test -z "$a" && echo "var. vide" || echo "var. pas vide"
test -n "$a" && echo "var. pas vide" || echo "var. vide"
test "$a" = "btsio" && echo "égal à btsio" || echo "different de btsio"
```

4. Appelez 3 fois votre script comme suit et expliquez le résultat :
1. # **testCond.sh**
2. # **testCond.sh bts**
3. # **testCond.sh btsio**

5. Créez le script **testCond2.sh** qui affecte aux variables a et b les 2 valeurs passés en paramètres et qui compare ces variables :
- A l'aide de l'opérateur « = »
- A l'aide de l'opérateur « != ».

Pour chaque test, affichage de « variables identiques » ou « variables différentes » selon le cas.
Exemple d'appel : # **testCond2.sh toto titi** #Testez tous les cas possibles

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

2.2. Tests sur les valeurs numériques

La syntaxe est : **test valeur1 opérateur valeur2** et les opérateurs possibles sont les suivants :

Opérateur	Rôle
-eq	Equal : égal
-ne	Not Equal : différent
-lt	Less than : inférieur
-gt	Greater than : supérieur
-le	Less or equal : inférieur ou égal
-ge	Greater or equal : supérieur ou égal

6. Créez le script **testCond3.sh** qui affecte aux variables a et b les 2 valeurs numériques passés en paramètre et qui affiche ensuite si $a est inférieur, supérieur ou égal à $b. Testez les 3 cas.

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

2.3. Syntaxe allégée

La commande **test** peut être remplacée par les crochets ouverts et fermés **[...]**.
Il faut respecter un **espace avant et après les crochets**.

Cette syntaxe est souvent utilisée dans les structures suivantes (if […] then … else… fi) que l'on voit ensuite.

Par exemple le premier script **testCond.sh** s'écrit avec la syntaxe allégée :

```
#!/bin/bash
a=$1
[ -z "$a" ] && echo "var. vide" || echo "var. pas vide"
[ -n "$a" ] && echo "var. pas vide" || echo "var. vide"
[ "$a" = "btssio" ] && echo "égal à btssio" || echo "different de btssio"
```

7. Réécrivez testCond2.sh en **testCond2-v2.sh** avec la syntaxe allégée

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

8. Réécrivez testCond3.sh en testCond3-v2.sh avec la syntaxe allégée

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

3. if ... then ... else … fi

La structure **if then else fi** est une structure de contrôle conditionnelle dont la syntaxe est :

```
if [ condition ]
then
        <commandes exécutées si condition réalisée>
else
        <commandes exécutées si dernière condition pas réalisée>
fi
```

Vous pouvez aussi préciser plusieurs **elif** (signifie else if). Si la dernière condition n'est pas réalisée, on en teste une nouvelle.

```
if [ condition1 ]
then
        action1
elif [ condition2 ]
then
        action2
else
        action3
fi
```

9. Saisissez, testez et commentez le script testIf.sh suivant.

	Exécutez-le et notez les résultats
#!/bin/bash **if** [$# -ne 0] **then** echo "$# parametres en ligne de commande" **else** *echo "Aucun parametre; paramètres créés"* *set alfred oscar romeo zoulou* ***fi*** echo "Nombre de parametres : $#" echo "Parametres : 1=$1 2=$2 3=$3 4=$4" echo "Liste : $*"	**# testIf.sh titi toto** **# testIf.sh**

10. Réécrivez le script testCond3.sh (**testCond3-if.sh**) en utilisant la structure if… then… else… fi.
Exécutez-le en lui passant les 3 cas de paramètres, nécessaires pour bien tester.

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

A vous de jouer : **Affichage du chemin courant**.
La variable $PWD contient le répertoire d'où un script est lancé.
La variable $HOME contient le répertoire de l'utilisateur actif.

11. Ecrivez un script nommé **chemin.sh** qui affiche un message suivant le répertoire où on se trouve au moment de son lancement :

- Lancé depuis le répertoire personnel, il affiche "Répertoire personnel".
- Lancé depuis la racine /, il affiche "Racine".
- Lancé depuis un autre endroit, il affiche "Répertoire quelconque".

```
root@debianRef:~/scripting# chemin.sh
Vous êtes dans un répertoire quelconque
root@debianRef:~/scripting# cd
root@debianRef:~# ./scripting/chemin.sh
Vous êtes dans votre répertoire personnel
root@debianRef:~# cd /
root@debianRef:/# ./root/scripting/chemin.sh
Vous êtes à la racine
root@debianRef:/#
```

Attention, pour tester ce script, vous devez le lancer de plusieurs répertoires de travail. Il faudra donc utiliser un chemin relatif ou absolu lorsqu'il ne sera pas présent dans le répertoire ou vous le lancez.

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

4. Les boucles

4.1. La boucle while

La commande **while** permet une boucle conditionnelle « tant que ». Tant que la condition est réalisée, le bloc de commande est exécuté. On sort si la condition n'est plus valable.

```
while condition
do
   commandes
done
```

ou

```
while
bloc d'instructions formant la condition
do
   commandes
done
```

12. Saisissez, testez et commentez le script **while1.sh** : lecture d'un fichier.
Créez auparavant un **fichier toto.txt** avec une dizaine de lignes quelconques.

while1.sh	**Testez et répondez aux questions suivantes**
#!/bin/bash typeset -i cpt cpt=1 cat toto.txt | while read line do echo "$cpt /" "$line" cpt=$cpt+1 done	Que fait ce programme ?

13. Saisissez, testez et commentez le script **while2.sh** : lecture d'un fichier.
Il travaillera sur le même fichier toto.txt

while2.sh	
#!/bin/bash typeset -i cpt cpt=1 while read line do echo "$cpt / $line" cpt=$cpt+1 done < toto.txt	Et celui-ci ? … voir la note ci-dessous

Il y a une **énorme différence entre les deux versions malgré les apparences !** Dans la première, notez la présence du tube (pipe) : la boucle est exécutée dans un second processus. Aussi toute variable modifiée au sein de cette boucle perd sa valeur en sortie, ce qui n'est pas le cas dans la seconde version.

Pour chaque programme, rajoutez l'instruction « echo "i=$i" » après la boucle et testez-les.

14. Que constatez-vous ?

En effet, la version avec le tube modifie la variable que dans la boucle.

4.2. true et false

La commande **true** ne fait rien d'autre que de renvoyer 0. La commande **false** renvoie toujours 1. De cette manière il est possible de réaliser des boucles sans fin. La seule manière de sortir de ces boucles est un exit ou un break (ctrl C).

Par convention, tout programme qui ne retourne pas d'erreur retourne 0, tandis que tout programme retournant une erreur, ou un résultat à interpréter, retourne autre chose que 0. C'est l'inverse en logique booléenne.

15. Saisissez, testez le script while3.sh.

While3.sh	**Que constatez-vous ?**
#!/bin/bash typeset –i cpt=0 while true do cpt=$cpt+1 echo "Ligne $cpt" done	

5. Application guidée : Programmation Shell Niveau 1

Objectif :
Dans un fichier texte (appelons le **ficNum**) nous avons les lignes suivantes :
1 3
5 7
12 19
...

Vous devez écrire un script (**calcFicNum.sh**) qui lit ce fichier et pour chacune de ses lignes calcule la somme des deux nombres et l'affiche sous la forme suivante :
1 + 3 = 4
5 + 7 = 12
12 + 19 = 31

Résolution en 5 étapes de A à E :

A. Vérifier en début de script que le nombre de paramètres passé au script est égal à 1 et que ce paramètre est bien un fichier.

```
1. if [ $# -ne 1 ]
2. then
3.       echo "un paramètre unique est obligatoire, fin du script"
4.       exit 1
5. else
6.       if [ ! -f $1 ] # équivalent à "test ! -f $1"
7.       then
8.                echo "Ce n'est pas un fichier, fin du script"
9.                exit 2
10.      fi
11. fi
```

Vous pouvez dès maintenant tester ce premier morceau de code et répondre aux questions suivantes :

➔ Quelle est l'utilité de la variable $# ?
➔ A quoi servent les [ ] après le if ?typeset –i
➔ Pourquoi y-a-t'il deux tests ?
➔ Que faut-il saisir pour que ce script ne sorte pas en « fin du script » ?
➔ Que se passe-t-il lorsque tout est correctement saisi ?
➔ Que faut-il tester pour vérifier qu'il fonctionne ? Voyez-vous d'autres tests à effectuer ?

B. Initialiser une variable à 0 qui contiendra le total de chacune des lignes.

```
12. typeset -i nb1 nb2 total=0
```

➔ Quelle est l'utilité de la commande typeset ?
➔ Que représentent nb1, nb2 et total ? Quelle utilité ?
➔ Utilité de « =0 » ?

C. Le fichier doit être lu ligne à ligne ; écrire une boucle qui lit une ligne tant que la fin du fichier n'est pas atteinte.

```
13. while read ligne
14. do
...
15. done < $1
```

Ici, il est important de bien comprendre ces lignes

➔ Que représente « ligne » ?
➔ A quoi sert $1 ?
➔ Et l'opérateur « < », quelle utilité ?
➔ Il y a une structure de boucle (while-do-done), combien de fois le programme va boucler ?

D. Dans la boucle, récupérer les deux valeurs des lignes, le séparateur est l'espace. Placer les deux valeurs dans les variables nb1 et nb2nb1

```
13. while read ligne
14. do
15.      nb1=$(echo $ligne | cut -d" " -f1)
16.      nb2=$(echo $ligne | cut -d" " -f2)
         ...
17. done < $1
```

Pour comprendre cette ligne, vous pouvez la tester par 'morceau' en ligne de commande. La commande « cut » étant nouvelle, il est nécessaire d'aller voir « man cut » et de faire des tests…

➔ Que fait echo $ligne ?

➔ Quelle est l'utilité du pipe (|)

➔ Que fait la commande cut –d…

➔ Que contiendra nb1 et nb2 ?

E. Additionnez ces deux valeurs et placez le résultat dans result. Affichez result.

```
13. while read ligne
14. do
15.     nb1=$(echo $ligne | cut -d" " -f1)
16.     nb2=$(echo $ligne | cut -d" " -f2)
17.     total=$nb1+$nb2
18.     echo $nb1 + $nb2 = $total
19. done < $1
```

Vous avez appris ici à lire le contenu d'un fichier et à utiliser chaque champ de chaque ligne.

6. Application en autonomie

Vous allez maintenant créer votre propre script qui va afficher pour un utilisateur donné, son UID et son nom. Le fichier qui contient ces informations est le **fichier des utilisateurs /etc/passwd** → Examinez son contenu.

16. Ecrire un script qui affiche le détail d'un compte utilisateur contenu dans le fichier /etc/passwd.
Le nom du compte utilisateur est passé en argument au script

Résultat attendu pour l'utilisateur sio par exemple :

Nom de connexion	sio
Caractère	x
Numéro de l'utilisateur	1000
Numéro du groupe	1000
Détail	sio,,,,
Répertoire d'accueil	/home/sio
Programme	/bin/bash

Aide importante pour effectuer ce script : A lire avant de scripter

1. Les parties du script

- ✓ Vérification qu'un et un seul paramètre est passé au script
- ✓ Initialisation des variables
- ✓ Boucle de lecture et de recherche de l'utilisateur dans le fichier passwd
- ✓ Affichage du résultat ou message d'erreur sir l'utilisateur n'existe pas

2. Pour distinguer des chaînes de caractères séparées par un séparateur (comme dans chaque ligne du fichier passwd), deux méthodes :

a) La commande « cut ».

Cette commande a été utilisée dans l'exercice précédent : nb1=$(echo $ligne | cut -d" " -f1)

Explication:

- ➔ $ligne contient 2 nombres séparés par un espace
- ➔ -d" " determine le séparateur, ici l'espace
- ➔ -f1 signifie qu'on veut extraire le 1er champ (f pour field)

b) La commande « set » et la variable IFS

Si une variable $toto contient une liste de valeur séparée par le caractère « : » (par exemple aa:bb:cc), les commandes suivantes permettent d'identifier les champs de la variable $toto et de redéfinir le contenu des variables de position ($1, $2, ...) :

IFS=: # Internal File Separator. Permet à la commande suivant d'identifier les champs d'une variable et de redéfinir le contenu des variables de position ($1, $2, ...)
Set $toto

- ➔ Regardez maintenant le contenu des variables $1, $2, …

3. Mise en forme des messages écran à l'aide de la commande « echo »

Testez la suite de commandes « **$ echo -e "\n\t Salut. Je m'appelle Seb\b\b\bPersonne\n\\" ; echo "Nouvelle ligne ?\n" ; echo -e "\n\n"** » et complétez le tableau suivant.

```
root@debianRef:~# echo -e "\n\t Salut.  Je m'appelle Seb\b\b\bPersonne\n\\" ; echo "Nouvelle ligne ?\n" ; echo -e "\n\n"

        Salut.  Je m'appelle Personne
\
Nouvelle ligne ?\n


root@debianRef:~#
```

	Description
-e	Permet d'interpréter les caractères suivants dé-spécifiés par un backslash (contre-oblique)
\n	Passage à la ligne
\t	Tabulation horizontale
\b	Retour d'un caractère en arrière
****	Afficher le backslash (barre oblique inverse)