ECRITURE DE SCRIPTS SHELL SOUS UNIX/LINUX II

Plan

- Structures conditionnelles
- Structures itératives
- Fonctions
- Tableaux

Les structures conditionnelles

- La structure `if`
- La structure `if..else:`
- La structure `if..elif..else`
- Les structures `if…else` imbriquées
- La structure (choix multiples) `case...esac` :

Structure conditionnelle

La structure if

```
if   commande test - vraies
then
Instructions…
fi
```

Exemple:

```
#!/bin/bash
echo -n "Donner un nombre: " ;  read VAR
if   [ $VAR -ge 10 ]
then
echo "Ce nombre est supèrieur à 10 !"
fi
```

Structure conditionnelle

La structure if..else:

```
if    commande test –vraies
then
Instructions…
else
Instructions…
fi
```

Exemple

```
#!/bin/bash
echo -n "Donner un nombre: "  read   VAR
if [ $VAR -gt 10 ]
then
    echo "Ce nombre est supèrieur à 10 !
else
    echo "Ce nombre est inférieur ou égal à 10 !"
fi
```

Structure conditionnelle


```
if [ $# = 0 ]
then
  echo "$0 : Aucun argument recu !"
fi

if cp "$1" "$1%"
then
  echo "sauvegarde de $1 reussie"
else
  echo "sauvegarde du fichier $1 impossible"
fi
```

Structures conditionnelles

La structure if..elif..else

```
if   commande test – conditions vraies
then
     Instructions …
elif  commande test – conditions vraies
then
     Instructions …
else
     Instructions …
fi
```

Les blocs **elif** et **else** sont facultatifs, la commande test peut réussir ou échouer,

Si elle réussi le bloc **then** est exécuté

Sinon le bloc **else** est exécuté

Structures conditionnelles

Exemple

```
#!/bin/bash
echo -n "Donner un nombre" read VAR
if [ $VAR -gt 10 ]
then
     echo "Ce nombre est supérieur à 10 !"

elif [ $VAR -eq 10 ]
then
 echo "Ce nombre est égal à 10 !"
else
 echo "Ce nombre est inférieur à 10 !"

fi
```

Structures conditionnelles

Les structures if…else imbriquées

(Nested if Statements)

- Possibilité de faire des if..else à l'intérieur d'un autre if..else

```
if    commande test – conditions vraies
then
   if  conditions
   then
   ….
   else
   …
   fi…
elif  commande test – conditions vraies
then
Instructions…
else
Instructions…
fi
```

Structures conditionnelles

La structure `case...esac` : (choix multiples)

```
case variable in
expression_reg1)
Commande1
Commande2 …
;;#executés s'il y a correspondence entre variable et l'expression_reg
expression_reg2)
Commande1
Commande2 …
;;
*)
commandes … #Default, executés dans le cas échéant
;;
esac
```

Structures conditionnelles


```
case $# in
0)
 echo "aucun parametre"
 echo "Syntaxe : $0 "
;;
1)
 echo "1 parametre passe au programme : $1"
;;
2)
echo "2 parametres passes au programme : $1 et $2 "
;;
*)
echo "TROP DE PARAMETRES !"
esac
```

Les structures itératives

- Boucle for
- Boucle while
- Boucle until

Structures itératives

- **Structure itérative ou boucle:**

  Un bloc de code qui répète une liste de commandes aussi longtemps que la condition de contrôle de la boucle est vraie

- On distingue 3 possibilités
  - Boucle for
  - Boucle while
  - Boucles until

Structures itératives

Boucle for

```
for variable in liste
do
commande(s)...
done
```

```
for variable in liste; do
commande(s)...
done
```

- `Variable`: c'est un nom de variable
- `Liste` :chaine de caractères avec des espaces ou des caractères de tabulation

Structures itératives

Boucle for

```
for variable; do
commande(s)...
Done
```

```
for variable in "$@"; do
commande(s)...
Done
```

- Sans arguments

Structures itératives

Boucle for

Exemple

Liste explicite

```
#!/bin/bash
for i in 1 2 3 4 5
do
echo "Message $i"
done
```

```
#!/bin/bash
for i in Rouge Vert Bleu Blanc Noir
do
echo "Message $i"
done
```

Structures itératives

Boucle for

Exemple

intervalle

```
#!/bin/bash
for i in in {1..5}
do
echo “Message $i”
done
```

```
#!/bin/bash
for i in {1..10..2}
do
echo “Message $i”
done
```

Structures itératives

Boucle for

Exemple

substitution

```
#!/bin/bash
for i in $(seq 1 2 20)
do
echo “Message $i”
done
```

```
#!/bin/bash
echo -n "Donner un nombre:”; read $Var
for i in $(seq $Var)
do
echo “Message $i”
done
```

Structures itératives

Boucle for

Exemple

C style

```
#!/bin/bash
for (( c=1; c<=5; c++ ))  do
echo “c=  $c ”
done
```

```
#!/bin/bash
#Boucle infinie
for (( ; ; ))
do
echo "infinie [ CTRL-C pour arrêter]"
done
```

Structures itératives


```
#!/bin/bash
for file in /etc/*  do
if [ "${file}" == "/etc/resolv.conf" ]  then
countNameservers=$(grep -c nameserver /etc/resolv.conf)
echo "Total ${countNameservers} nameservers defined in ${file}"
break
fi
done
```

Structures itératives

Boucle while

```
while condition
do
commande(s)...
done
```

- La boucle **`while`** teste une condition au début de la boucle et continue à boucler tant que la condition est vraie c.à.d. (renvoie un 0 comme code de sortie).
- Par opposition à la boucle for, la boucle **`while`** est utile lorsque le nombre de répétitions n'est pas connu à l'avance

Structures itératives

Boucle while

```
#!/bin/bash
x=1
while [ $x -le 5 ]
do
echo "Hello World $x"
x=$(( $x + 1 ))
done
```

```
#!/bin/bash
while :
do
echo "infinite loops [CTRL+C to stop] "
sleep 1
done
```

Structures itératives

Boucle while

```
#!/bin/bash
counter=$1
factorial=1
while  test $counter -gt 0
do
factorial=$((factorial * counter ))
counter=$((counter - 1 ))
done
echo $factorial
```

Structures itératives

Boucle until

```
until condition
do
commande(s)...
done
```

- Cette construction teste une condition au début de la boucle et continue à boucler tant que la condition est fausse

  (A l'opposé de la boucle **while**)

Structures itératives

Boucle until

```
#!/bin/bash  x=20
until [ $x -lt 10 ];  do
echo "The counter is $x"
let x-=1
done
```

```
#!/bin/bash
i=1
until [ $i -gt 6 ]  do
echo "i = $i"  #i=$(( i+1 ))
(( i++ ))
done
```

Fonctions

- Le Shell est un langage procédural dans lequel on peut utiliser des fonctions semblables aux fonctions des autres langages, mais relativement limitées

- Unefonction peut être utilisée:
  - **En dehors d'un script** – définition avec l' invite de commande Shell, on peut créer des fonctions pour regrouper des séries d'instructions couramment exécutées et effectuer le même travail qu'un script Shell
  - **Au sein d' un script,** sous-routines implémentant un certain nombre d'opérations pour remplacer des suites d'opérations répétées plusieurs fois au cours des scripts

Fonctions

- Une fonction ne peut pas être vide
- Elle peut prendre des arguments de la même manière qu'un script mais pas entre les parenthèses, accessibles via $1 $2 etc … au sein de la fonction
- Les fonctions peuvent réaliser différentes fonctionnalités comme les itérations, inclure la déclaration des variables, …
- Comme tout programme, elle renvoie également un code de sortie, une valeur numérique que l'on peut utiliser dans les tests

Fonction

Création des fonctions : bash

- La syntaxe permettant de définir une fonction sous Bash est la suivante :

```
function      nomf ()
{
    instruction1 ;
    instruction2 ;
    …
}
```

- Les parenthèses sont optionnelles.

Exemple

```
function     fonction1
{
    pwd ;
    mkdir rep1 ;
    echo "répertoire rep1 crée" ;
}
```

Fonctions

Création des fonctions :

- C-like : les parenthèses obligatoires

```
nom_fonction()
{
        Instruction1
        Instruction2
        ...;
}
```

Exemple

Fonctions

- Pour afficher les fonctions définies dans l'environnement de l'utilisateur courant, on utilise la commande **declare** comme suit :

```
$ declare -F
```

- Pour afficher le contenu d'une fonction particulière, par exemple notre fonction fonction1, on peut faire :

```
$declare -f fonction1
$type -all fonction1
```

- Pour supprimer une fonction on utilise la commande **unset**, cette commande prend en paramètre le nom de la fonction à supprimer

Supprimer la fonction **fonction1** : `$unset fonction1`

Fonctions

Déclaration des fonctions d'une façon permanente

- Les fonctions ainsi définis ne sont pas conservés d'une session à l'autre.

- Pour qu'ils soient initialisés à chaque ouverture du Shell Bash, on peut les définir :
  - Dans le fichier **`/etc/bashrc`** : dans ce cas, ils seront définis dans toutes les sessions de tous les utilisateurs
  - Dans le fichier **`~/.bashrc`** qui se chargera au lancement de la session de l'utilisateur intéressé : dans ce cas, les fonctions auront une portée restreinte ; ils seront définis uniquement dans l'environnement de l'utilisateur en question

Tableaux

Déclaration des tableaux: Deux méthodes

- Première méthode:

```
tabx=("Alice" "Bob" "Sam" )
```

- Deuxième méthode:

```
tabx[0]="Alice"
tabx[1]="Bob"
tabx[2]="Sam"
```

Tableaux

- Nombre d'éléments d'un tableau :

```
echo    ${#tabx[*]}
ou echo ${#tabx[@]}
```

- Le premier élément

```
    echo ${tabx[0]}
    Ou bien echo ${tabx}
```

- Pour afficher un élément en 3ème position:

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

Tableaux

- Pour afficher tous les éléments :
  - `echo ${tabx[@]}`
- Ou bien avec une boucle for :
  - Bash style:
```
for i in ${!tabx[@]}; do
    echo ${tabx[i]};
done
```
  - C style:
```
for (( i=0; i < ${#tabx[@]}; i++ ));
do
    echo ${tabx[i]};
done
```

Tableaux

- Note: Toutes les variables sont des tableaux. Par défaut, c'est le premier élément qui est appelé

```
varz="dog"
echo $varz
echo ${varz}
#!/bin/bash
# Mettre les arguments du script dans un tableau
tabxx=( "$@" );
echo ${tabxx[1]}
echo "Nombre d'éléments: ${#tabxx[@]} "  echo
"Les éléments sont: ${tabxx[@]}
```

Exemple tableau


```
#!/bin/bash
tab=("Alice" " Bob" " Sam" " Rim" )

#Bash Style
for i in ${!tab[@]}; do
 echo ${tab[i]};
done

#C style
for (( i=0; i < ${#tab[@]}; i++ )); do
echo ${tab[i]};
done
```

Tableaux

Exemple:

```
#!/bin/bash
# Mettre les arguments du script dans un tableau

  tabxx=( "$@" );
 echo ${tabxx[1]}
 echo "Nombre d'éléments: ${#tabxx[@]} "
echo "Les éléments sont: ${tabxx[@]}
```

La commande getopts


```
#!/bin/bash
while getopts 'abc' OPTION; do
  case "$OPTION" in
    a)        echo "Option a used "
     ;;
    b)       echo "Option b used"
      ;;
    c)       echo "Option c used"
      ;;
    ?)       echo "Usage: $(basename $0) [-a] [-b] [-c]"
      exit 1
      ;;
  esac
done
```

La commande getopts


```
#!/bin/bash
while getopts ':abc' OPTION; do
  case "$OPTION" in
    a)  echo "Option a used"
      ;;
    b) echo "Option b used"
      ;;
    c) echo "Option c used"
      ;;
    ?)   echo "Usage: $(basename $0) [-a] [-b] [-c]"
      exit 1
      ;;
  esac
done
```

La commande getopts


```
#!/bin/bash
while getopts ':ab:c:' OPTION; do
  case "$OPTION" in
    a)echo "Option a used"
      ;;
    b)argB="$OPTARG"
      echo "Option b used with: $argB"
      ;;
    c) argC="$OPTARG"
      echo "Option c used with: $argC"
      ;;
    ?) echo "Usage: $(basename $0) [-a] [-b argument] [-c argument]"
      exit 1
      ;;
  esac
done
```

La commande getopts


```
echo "Before - variable one is: $1"
shift "$(($OPTIND -1))"
echo "After - variable one is: $1"
echo "The rest of the arguments (operands)"

for x in "$@"
do
  echo $x
done
```

Déboguer un script bash

- **Avec l'option "-x«**

```
bash -x myscript.sh
```

Ou bien

```
$ bash --debug myscript.sh
```

- **On peut activer et désactiver**

```
#!/bin/bash
set -x    # Enable debugging
# some code here
set +x    # Disable debugging output.
```

Déboguer un script bash

- L'option -n vous permet de vérifier la syntaxe d'un script sans avoir à l'exécuter:
- `$ bash -n testscript.sh`