UNIVERSITÉ PARIS-EST Marne-la-Vallée

Shell Scripts Avancés

Licence 2

2012-2013

Sylvain Cherrier

cherrier@univ-mlv.fr

Les fonctions

- Comme les autres langages
- Capitalisation et réutilisabilité du code
- Formalisme un peu limité
- Non typée
- Non « déclarée » (pas de prototype, la fonction doit être décrite avant son utilisation)

Format de la fonction

- Deux formes

```
function nom_fonction {
}
```

- Ou plutôt

```
nom_fonction() { #C like
}
```

- La fonction **main** est implicite

Exemple de fonctions

```
#!/bin/bash
Rebours() {
        echo  -n "Wait :  "
        for ((i=9;i>=0;i--))
        do
                sleep 1
                echo -ne "\b$i"
        done
}

#ici, rien d'indiqué, donc, c'est le main
rebours()
echo "Started"
…
echo "please change CD and hit ENTER"
read rep
rebours()
echo "Burning CD"
….
```

Les échanges avec fonctions

- Habituellement, une fonction attend (ou non) des arguments
- Elle peut renvoyer une valeur
- En shell : Pas d'indication sur les arguments (non typé, non déclaré:-)
- Le retour d'une valeur se fait par **return**

Appel de fonctions

- L'appel se fait en écrivant les arguments après le nom de la fonction (*C unlike*)
- Les arguments sont récupérés sous la forme habituelle en Shell, c'est à dire $0 $1 $2 …

```
#!/bin/bash

affiche_etat() {
        echo "le feu $1 est $2"
}

echo "Etat réseau ferré"
affiche_etat gare rouge
affiche_etat route vert
affiche_etat pont rouge
```

```
sylvain@ilium:/tmp$ ./test.bash
Etat réseau ferré
le feu gare est rouge
le feu route est vert
le feu pont est rouge
```

$1 ou $1 ?

- Même si ils portent le même nom, et même si ils sont manquants dans l'appel de la fonction, le shell ne confond pas les arguments du script et ceux de la fonction

```
#!/bin/bash

affiche_etat() {
        echo "le feu $1 est $2"
}

echo "Etat réseau ferré"
affiche_etat gare rouge
affiche_etat $1 $2
affiche_etat
```

```
sylvain@ilium:/tmp$ ./test.bash tunnel vert
Etat réseau ferré
le feu gare est rouge
le feu tunnel est vert
le feu  est
```

Valeur de retour

- Une fonction peut renvoyer une valeur
- Spécifiquement avec la commande **return**
- Implicitement, elle renvoie la *exit value* de sa dernière commande
- **Return** permet de renvoyer une valeur numérique entre 0 et 255
- Cette valeur est accessible dans **$?**

Exemple de valeur de retour

```
#!/bin/bash

compte_connect() {
        return `who | wc -l`
}

compte_connect
echo -n "$? users connected today : "
date +%D
```

```
sylvain@ilium:/tmp$ ./test.bash
2 users connected today : 02/27/13
```

Et si on veut d'autres types de valeurs ?

- Pour des nombres hors du domaine de définition de exit, on peut passer par une variable globale (mais pas de sous-shell dans ce cas)
- Pour des chaines, ou des résultats plus complexes, on peut utiliser l'astuce des back-quotes (recupérant le stdout d'une commande) avec des echos dans la fonction

Valeurs de retour

```
#!/bin/bash
max() {
        if [ -z "$2" ]
        then
                return 1
        else if [ "$1" -gt "$2" ]
            then
                echo $1
            else
                echo $2
            fi
        fi
}
plus_grand=`max $1 $2`
if [ $? -eq 0 ]
then
        echo "la plus grande valeur est $plus_grand"
else
        echo "Erreur "
fi
```

```
sylvain@ilium:/tmp$ ./test.bash 1544
Erreur
sylvain@ilium:/tmp$ ./test.bash 1544 54
le plus grand pid de 1544 est 1544
```

Plus loin avec les variables

- On peut connaître la longueur d'une chaine avec *${#var}*

  echo ${#var} indique 7 si var=Atchoum

- *Expr index $chaine $rech* rechercher la position d'une ou plusieurs lettres de $rech dans $chaine (index de la première trouvée)

  expr index $var c affiche 3 pour var=Atchoum

Manipulation de variables

- Extraction d'une sous-chaine

  ${var:position}

  ${var:position:longueur}

- Exemple

  ***var**=Atchoum*

  *echo ${**var**:3} ${**var**:1:2} ${**var**:(-3)}*

```
houm tc oum
```

Manipulation de Variables

- Plus utile dans le cas d'un shell, ma suppression d'une partie de la chaine

  ${var#sub} efface la plus petite occurrence de sub

  ${var##sub} efface la plus longue occurrence de sub

  ${var%fin} même chose en partant de la fin

  ${var%%fin} même chose en partant de la fin

Exemple de manipulation de chaine

```
#!/bin/bash

echo "copie de sauvegarde des fichiers .c de $1 vers $2"
if [ -d $1 -a -d $2 ]
then
        for i in `ls $1/*.c`
        do
                newFile=${i#$1/}
                cp      $i      $2/${newFile%c}oldC
        done
else
        echo "$1 et $2 doivent être des répertoires"
fi
```

```
sylvain@ilium:/tmp$ ./test.bash exercices .
copie de sauvegarde des fichiers .c de exercices vers .
copie de exercices/exemple_de_thread.c ./exemple_de_thread.oldC
copie de exercices/OpenGL_main.c ./OpenGL_main.oldC
copie de exercices/OpenGL_rail.c ./OpenGL_rail.oldC
copie de exercices/OpenGL_reseau.c ./OpenGL_reseau.oldC
copie de exercices/OpenGL_train.c ./OpenGL_train.oldC
```

Remplacement de chaines

- Il est possible de modifier directement le contenu des chaines

${var/rech/remp} remplace la 1ere occurrence de rech par remp

${var//rech/remp} remplace toute occurrence de rech par remp

${var/#rech/remp} avec le sens...

${var/%rech/remp}

Manipulation de chaine V 2

```
#!/bin/bash

echo "copie de sauvegarde des fichiers .c de $1 vers $2"
if [ -d $1 -a -d $2 ]
then
        for i in `ls $1/*.c`
        do
                newFile=${i#$1/}
                cp      $i      $2/${newFile/%c/oldC}
        done
else
        echo "$1 et $2 doivent être des répertoires"
fi
```

```
sylvain@ilium:/tmp$ ./test.bash exercices .
copie de sauvegarde des fichiers .c de exercices vers .
copie de exercices/exemple_de_thread.c ./exemple_de_thread.oldC
copie de exercices/OpenGL_main.c ./OpenGL_main.oldC
copie de exercices/OpenGL_rail.c ./OpenGL_rail.oldC
copie de exercices/OpenGL_reseau.c ./OpenGL_reseau.oldC
copie de exercices/OpenGL_train.c ./OpenGL_train.oldC
```

Valeurs par défaut

- Parfois utile dans les langages interprétés
- Permet d'avoir tjs une variable, au pire avec sa valeur par défaut
- ${var-defaut} ou ${var:-defaut}
- La valeur par défaut peut être une autre variable, ou le résultat d'un sous-shell (` `)
- Le : permet de gérer le cas où la variable existe, mais est NULL

Exemple de valeurs par défaut

```
sylvain@ilium:/tmp$ echo ${name-Sylvain}
Sylvain
sylvain@ilium:/tmp$ name=Bob
sylvain@ilium:/tmp$ echo ${name-Sylvain}
Bob
sylvain@ilium:/tmp$ name=
sylvain@ilium:/tmp$ echo ${name-Sylvain}

sylvain@ilium:/tmp$ echo ${name:-Sylvain}
Sylvain
sylvain@ilium:/tmp$ echo ${os:-`uname`}
Linux
```

Listes de commandes

- Écrire une liste de commandes permet d'éviter les tests ou les switchs
- Le ET permet d'obliger l'exécution de toutes les commandes tant que cela fonctionne, ou jusqu'au bout de la ligne
- Le OU exécute les commandes fausses jusqu'à ce qu'une donne vrai

cmd1 && cmd2 && cmd3 && … &&cmdN

cmd1* || *cmd2* || *cmd3* || *…* || *cmdN

Exemple + extrait du script bootlog

```
[ -z $var ] && echo "set" || echo "not set"
```

```
stop)
        PATH=/bin:/sbin:/usr/bin:/usr/sbin
        [ "$VERBOSE" != no ] && log_daemon_msg "Stopping $DESC" "$NAME"
        start-stop-daemon --oknodo --stop --quiet --exec $DAEMON
        ES=$?
        sleep 1
        [ "$VERBOSE" != no ] && log_end_msg $ES
        if [ -f /var/log/boot ] && [ -f /var/log/boot~ ]
        then
                [ "$VERBOSE" = no ] || log_action_begin_msg "Moving boot log file"
                ….   ….
                ES=$?
                [ "$VERBOSE" = no ] || log_action_end_msg $ES
        fi
        ;;
```

Déclaration et typage de variables

- Mensonges ! Il est possible de typer et déclarer une variable

  declare [-ilravu] (créer des variables)

  typeset [-ilravu] var (leur donner un type)

  -i (entier) -l (minuscule) -r (read only)

  -a (array) -u (majuscule)

Typage...

```
root@ilium:/var/log# declare -i brouzoufs
root@ilium:/var/log# echo $brouzoufs

root@ilium:/var/log# brouzoufs=50000
root@ilium:/var/log# echo $brouzoufs
50000
root@ilium:/var/log# brouzoufs=plein
root@ilium:/var/log# echo $brouzoufs
0
```

```
sylvain@ilium:/var/log$ declare -u nom
sylvain@ilium:/var/log$ nom=toto
sylvain@ilium:/var/log$ echo $nom
TOTO
```

Tableaux

- Une dimension
- On indique pas de taille, les « trous » se remplissent tout seul
- On peut assigner les valeurs une à une, ou l'ensemble d'un coup
- Les opérations habituelles sur les variables fonctionnent (substring, longueur, remplacement...)

Exemple de tableaux

```
#!/bin/bash

poeme[1]="Dans le pays où s'étendent les Ombres"
poeme[2]="Un anneau pour les gouverner tous"
poeme[3]="un anneau pour les trouver"
poeme[4]="un anneau pour les amener tous"
poeme[5]="et dans les ténèbres les lier"

for index in 1 2 3 4 5
do
  echo -e "\t\t${poeme[$index]}"
done
```

Tableau à cases vides

```
#!/bin/bash

ALPHABET=ABCDEFGHIJKLMNOPQRSTUVWXYZ

for ((i=0;i<20;i++))
do
        rand=$(($RANDOM%26))
        scrabble[$rand]=` expr ${scrabble[$rand]} + 1 `
done

for((i=0;i<26;i++))
do
        echo ${ALPHABET:$i:1}  ${scrabble[$i]}
done
```

Debugger

- Les messages d'erreurs générés par l'interpréteur ne sont pas toujours clairs
- Pour en savoir plus, utilisez -x (trace) ou -n (verif syntaxe uniquement)
- Soit en écrivant set -x dans le script
- Soit en lançant bash -x script

Mode debug

```
sylvain@ilium:/tmp$ bash -x test.bash
+ ALPHABET=ABCDEFGHIJKLMNOPQRSTUVWXYZ
+ (( i=0 ))
+ (( i<20 ))
+ rand=23
++ expr + 1
+ scrabble[$rand]=1
+ (( i++ ))
+ (( i<20 ))
+ rand=9
++ expr + 1
+ scrabble[$rand]=1
+ (( i++ ))
+ (( i<20 ))
+ rand=12
++ expr + 1
+ scrabble[$rand]=1
```

```
+ (( i++ ))
+ (( i<20 ))
+ rand=0
++ expr + 1
+ scrabble[$rand]=1
+ (( i++ ))
+ (( i<20 ))
+ rand=1
++ expr + 1
+ scrabble[$rand]=1
+ (( i++ ))
+ (( i<20 ))
+ rand=21
++ expr + 1
+ scrabble[$rand]=1
```

Encore plus d'outils

- Whiptail est un outil additionnel (dans Debian) qui permet de faire des affichages (presque) graphiques en Shell

  Il propose des boites de dialogues, des zones de saisies, des listes, des menus, des barres de progression
- Il existe d'autres outils (ncurses, dialog, xdialog)

whiptail

Les actions sont récupérés dans la valeur de retour, et les saisies par des redirections dans des variables.

```
NOM=$(whiptail --inputbox "Votre nom ?" 8 78 --title
"Saisie" 3>&1 1>&2 2>&3)

exitstatus=$?
if [ $exitstatus = 0 ]
then
    echo "l'utilisateur a validé et s'appelle " $NOM
else
    echo "Annulation."
fi
```

Exemples whiptail

```
#!/bin/bash

[ -d SAV ] || mkdir SAV
whiptail --title "Copie de sauvegarde" --yesno "Voulez vous sauvegarder ?" 8 78

exitstatus=$?
if [ $exitstatus = 0 ]; then
    nbFile=`ls *.c | wc -l`
    let step=100/$nbFile
    { for i in *.c
      do
          cp $i SAV/$i
          let state=$state+$step
          sleep 1 ;   echo $state
      done
    } | whiptail --gauge "Please wait" 5 50 0
else
     echo "Annulation de la sauvegarde"
fi
```

Quelques astuces

```
filename=$(basename "$fullfile")
extension="${filename##*.}"
filename="${filename%.*}"
```

- Utilisez ; et autre { } pour compacter / regrouper les instructions
- Syslog est très pratique pour les scripts système (traces de l'activité)
- Les scripts dans /usr/local/bin deviennent accessibles à tous