Programmation Unix 1 – cours n°3

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Septembre 2016

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/unix/

Lien court : http://j.mp/progunix

Plan du cours n°3

1. Opérateurs de variables
2. Fonctions
3. Substitution de commandes
4. Débogage

1 - Opérateurs de variables

Opérateurs de bash sur les variables et les arguments $\rightarrow$ éviter de faire appel à des commandes externes (`tr`, `sed`, `awk`, etc)

`${var-val}` $\rightarrow$ `$var` si définie, sinon `val`

Exemples :

```
$ a="ga"
$ echo "${a-meu}"
ga
$ unset a
$ echo "${a-meu}"
meu
```

```
$ set bu zo
$ echo "${2-meu}"
zo
$ set bu
$ echo "${2-meu}"
meu
```

Opérateurs, suite

`${var?mess}`	→	`$var` si définie, sinon affiche `mess` sur `stderr` + `exit 1`
`${var+val}`	→	`val` si `var` définie, rien sinon
`${var=val}`	→	`$var` si définie, sinon `val` + affectation

Exemples :

```
$ a="ga"
$ echo "${a?Non définie}"
ga
$ echo "${a+toto}"
toto
$ echo "${a=meu}"
ga
```

```
$ unset a
$ echo "${a?Non définie}"
bash: a: Non définie
$ echo "${a+toto}"

$ echo "${a=meu}"
meu
```

Sous-chaînes

`${var:i}` → sous-chaîne depuis `i` $\geqslant 0$
`${var:i:k}` → sous-chaîne depuis `i` $\geqslant 0$ de longueur `k`
`${#var}` → longueur de `$var`

Exemples :

```
$ a="bonjour"
$ echo ${#a}
7
$ echo ${a:3}
jour
$ echo ${a:3:1}
j
```

Suppression de motifs au début

`${var#motif}` → supprime le plus court début qui correspond
`${var##motif}` → supprime le plus long début qui correspond

Exemples :

```
$ p="ens/unix/tp1/hop.tar.gz"
$ echo "${p#ens/un}"
ix/tp1/hop.tar.gz
$ echo "${p#toto}"
ens/unix/tp1/hop.tar.gz
$ echo "${p#*/}"
unix/tp1/hop.tar.gz
$ echo "${p##*/}"
hop.tar.gz
```

Suppression de motifs à la fin

`${var%motif}` → supprime la plus courte fin qui correspond
`${var%%motif}` → supprime la plus longue fin qui correspond

Exemples :

```
$ p="ens/unix/tp1/hop.tar.gz"
$ echo "${p%tar.gz}"
ens/unix/tp1/hop.
$ echo "${p%toto}"
ens/unix/tp1/hop.tar.gz
$ echo "${p%.*}"
ens/unix/tp1/hop.tar
$ echo "${p%%.*}"
ens/unix/tp1/hop
```

Remplacement de motifs

`${var/motif/txt}` → remplace la plus longue correspondance au motif par le texte `txt`

`${var//motif/txt}` → remplace toutes les correspondances

Exemples :

```
$ s="bonjour mon bon ami"
$ echo "${s/bon/ga}"
gajour mon bon ami
$ echo "${s//bon/ga}"
gajour mon ga ami
$ echo "${s/o*o/ga}"
bgan ami
```

Complément : `read` à plusieurs variables

`read` v_1 $v_2 \cdots v_n$

- lit une ligne sur l'entrée standard,
- découpe en mots (séparateur blanc, selon `$IFS`),
- affecte les variables :

$$\begin{aligned} \text{mot}_1 &\to v_1 \\ \text{mot}_2 &\to v_2 \\ &\vdots \\ \text{mot}_{n-1} &\to v_{n-1} \\ \text{reste ligne} &\to v_n \end{aligned}$$

Donc `read` v_1 met toute la ligne dans v_1 sans la découper

2 - Fonctions

Déclaration :

```
nom_fonction ()
{
    instructions
}
```

ou encore sur une ligne :

```
nom_fonction () { instructions ;}
```

Tapée dans un terminal, ou placé n'importe où dans un script, avant l'appel.

⚠ `()` toujours accolées : signifie *déclaration de fonction*

Appel et paramètres

Appel : `nom_fonction arguments`

→ Une fonction s'utilise comme un script :

Script `titi` :

```
#! /bin/bash
echo "Reçu $1"
exit 0
```

Appel :

```
$ ./titi foo
Reçu foo
```

Avec une fonction :

```
$ toto () {
    echo "Reçu $1"
    return 0
  }

$ toto foo
Reçu foo
```

Masquage des arguments

Les arguments courants sont masqués pendant l'appel :

```
$ set foo
$ echo "$1"
foo
$ hop () { echo "$1" ;}
$ hop bar
bar
$ echo "$1"
foo
```

Argument $0

L'argument $0 est immuable :

```
$ echo "$0"
bash
$ shift
$ echo "$0"
bash
$ pouet () { echo "$0" ;}
$ pouet
bash
```

Idem dans un script.

Sortie de fonction

Il ne faut pas confondre :

`return [x]` : sortie immédiate de la fonction
`exit [x]` : sortie immédiate du script (ou du shell)

- `x` est le code de terminaison $\rightarrow$ `$?`
- par défaut : code de terminaison de la dernière commande exécutée

```
foo () { ... ; return 0  ;}    # réussite
foo () { ... ; echo "ga" ;}    # réussite
foo () { ... ; false     ;}    # échec
foo () { ... ; test ...  ;}    # selon test
```

Pas de renvoi de valeur

Comme les scripts, les fonctions réussissent ou échouent mais ne renvoient pas de valeur.

Usage typique :

```
myfunc () { ..... ; return x ;}

if myfunc ga bu ; then ... ; fi

while myfunc zo meu ; do ... ; done
```

Redirections

- On peut rediriger les E/S d'une fonction :

```
hop () { ls ;}

hop > toto
hop | sort
```

- Bloc entre accolades : permet aussi de rediriger des instructions

```
{ echo "ga" ; echo "bu" ;} > toto
```

Portée des variables

Les variables sont globales par défaut :

```
$ hop () { a="foo" ;}
$ a="bar" ; hop ; echo "$a"
foo
```

On peut déclarer des variables locales à une fonction :

```
local var1 var2 ... varx=valeur ...
```

```
$ hop () { local a="foo" ;}
$ a="bar" ; hop ; echo "$a"
bar
```

→ Indispensable pour éviter les effets de bord !

Affectations avec `local`

```
$ a=hips
$ g() {
      local a="$1" b="$a"
      echo "$b"
  }
$ g bu
hips
```

⚠ Dans `local`, les substitutions sont faites en premier

Solution : faire en 2 temps

```
$ g() {                          # ou encore :
      local a="$1"               # local a="$1" b
      local b="$a"               # b="$a"
      echo "$b"
  }
```

Indication des paramètres

Bon usage :

- indiquer les paramètres en commentaire
- récupérer les paramètres dans des variables locales.

```
afficher_age () # nom prenom age
{
    local nom="$1" prenom="$2" age="$3"

    echo "$prenom $nom a $age ans."
}
```

Exemple

Une fonction qui prend en argument un chemin de fichier et une extension, et qui réussit s'il se termine par cette extension.

```
possede_extension () # fichier extension
{
    local fichier="$1" ext="$2"
    local fichier_sans_ext="${fichier%$ext}"
    test "$fichier_sans_ext$ext" = "$fichier"
}
```

Exemple d'usage :

```
fic="Mes images/IMG234.jpg"
if possede_extension "$fic" ".jpg" ; then
    echo "Le fichier $fic possède l'extension .jpg"
fi
```

3 - Substitution de commandes

Permet de substituer l'appel d'une commande par ce qu'elle affiche.
Deux formes :

`$(commande)`	bash, recommandée
`` `commande` ``	sh, ancienne

Bash exécute la commande dans un sous-shell, puis la substitue par sa sortie standard.

Exemple :

```
x=$(expr 5 + 7)
```

a le même effet que :

```
expr 5 + 7 >| tmp
read x < tmp
rm -f tmp
```

Substitution dans des chaînes

```
f="ga.c"
echo "Le fichier $f a $(wc -l < $f) lignes"
                          ↓
echo "Le fichier ga.c a $(wc -l < ga.c) lignes"
                          ↓
echo "Le fichier ga.c a 25 lignes"
```

Suppression de lignes

Lors de la substitution de commandes, bash supprime les dernières lignes vides :

```
$ echo -e "\n\n  bonjour  \n\n"


  bonjour


$ a=$(echo -e "\n\n  bonjour  \n\n")
$ echo "'$a'"
'

  bonjour  '
$
```

Substitution par un fichier

Pour substituer par le contenu d'un fichier : `$(cat < fichier)`

Raccourci (plus efficace) : `$(< fichier)`

```
$ cat hello.txt
bonjour buongiorno
good morning guten Tag

$ texte=$(< hello.txt)
$ echo "$texte"
bonjour buongiorno
good morning guten Tag
$
```


Imbrication et blocs

- On peut imbriquer les substitutions de commandes :

```
msg="taille = $(wc -c < $(which test)) octets"
                ↓
msg="taille = $(wc -c < /usr/bin/test) octets"
                ↓
msg="taille = 30272 octets"
```

- On peut substituer des blocs de commandes :

```
meu=$(echo "ga" ; echo "bu" ; echo "zo")
case $(ls -t | head -1) in ... esac
```

Résultat d'une fonction

`return` est réservé pour réussir/échouer.

Les fonctions peuvent fournir le résultat par la sortie standard :

```
ma_fonction ()  # arguments
{
  ...
  echo "le résultat"
  return 0   # on peut omettre
}
```

On récupère le résultat par substitution de commandes :

```
res=$(ma_fonction arguments)
```

La fonction ne doit rien afficher d'autre que le résultat

Exemple 1

```
plus () # x y
{
  local x="$1" y="$2"
  expr "$x" + "$y"
}

mult () # x y
{
  local x="$1" y="$2"
  expr "$x" '*' "$y"    # Attention joker
}

norme_carre () # x y
{
  local x="$1" y="$2"
  plus $(mult "$x" "$x") $(mult "$y" "$y")
}

x=3 ; y=4 ; z=$(norme_carre "$x" "$y")  # 25
```

Exemple 2 : maximum

```
max () # liste d'entiers
{
  if test $# -lt 1 ; then echo "" ; return ; fi
  local i max="$1"
  shift
  for i ; do
    if test "$max" -lt "$i" ; then max="$i" ; fi
  done
  echo "$max"
}

m=$(max 5 7 20 6 8)   # 20
```

Exemple 3 : mot le plus long

Longueur d'une variable : `${#nom_variable}`

```
plus_long () # liste de mots
{
  local i max=""
  for i ; do
    if test ${#max} -lt ${#i} ; then max="$i" ; fi
  done
  echo "$max"
}

m=$(plus_long a bra cada bra)   # cada
```

4 - Débogage

- Afficher la liste des fonctions :

  ```
  declare -F
  ```

- Afficher le corps d'une fonction :

  ```
  declare -f nom_fonction
  ```

  réussit si la fonction existe.

- Supprimer une fonction :

  ```
  unset -f nom_fonction
  ```

  on peut à tout moment redéfinir une fonction,
  `unset` non obligatoire.

Informations sur les variables

- Afficher la liste des variables :

  ```
  declare -p
  ```

- Afficher plus d'informations sur une variable :

  ```
  declare -p nom_variable
  ```

  réussit si la variable existe.

- Supprimer une variable :

  ```
  unset -v nom_variable
  ```

Conversion automatique

```
declare -l nom_variable
declare -u nom_variable
```

La variable sera convertie en minuscules ou majuscules à l'affectation.

```
$ declare -u ga
$ read ga
bonJOUr
$ echo "$ga"
BONJOUR
```

Désactiver : `declare +l|+u` ou `unset`

Trace automatique

Pour demander à bash d'afficher toutes les commandes telles qu'elles sont exécutées : `set -x` désactiver : `set +x`

```
$ ga="home"; bu="thiel"
$ set -x
$ if test "$ga" = "$bu" ; then echo "ok" ; fi
+ test home = thiel
$ toto(){ local a="/bin"; cd "$1/$a";}
$ toto "/$ga/$bu"
+ toto /home/thiel
+ local a=/bin
+ cd /home/thiel//bin
$ ls *.sh
+ ls -F numsauv.sh sudoku.sh
numsauv.sh sudoku.sh
$ set +x
```

Trace automatique (2)

- Activer/désactiver dans la console ou dans un script :

`set -x ... set +x`

- Configurer le script pour qu'il s'affiche en mode trace :

```
#! /bin/bash -x
```

- Exécuter le script en mode trace sans le modifier :

```
$ bash -x ./monscript
```

- Voir le chapitre 32 de l'ABSG.