

# Unix : shell scripts

ASRI - Département d'informatique  
IUT Bordeaux 1

2010-2011

## 1 Quelques rappels

### 1.1 Commandes

Vous connaissez déjà `emacs`, `vi`, `g++`, `cp`, `rm`, `mv`, `ls`, `cat`, `less`, `more`, `mkdir`, `rmdir`, `cd`, `pwd`, `echo`, `clear`, `man`, ...

### 1.2 Redirections

- De la *sortie standard* (cout en C++) vers un fichier

```
echo "bonjour" > message.txt
date "+%H:%M" >> message.txt
```
- de la *sortie d'erreurs* (cerr en C++) vers un fichier

```
g++ mon-programme.cc 2> erreurs.txt
dialog --inputbox "Votre nom ?" \
      8 40 2> /tmp/nom
```
- de l'*entrée standard* depuis un fichier

```
tr '[a-z]' '[A-Z]' < texte.txt
```
- de l'*entrée standard* (cin) depuis le script

```
$ tr '[a-z]' '[A-Z]' <<XXX
ceci Est un
exemple
XXX
```
- *pipe* entre commandes

```
w | cat -n
anytopnm dscn3214.jpg |
  pnmscale -width 100 |
  pnmtopng > statue-hcm-100px.png
```

## 2 Encore des commandes

Soit le fichier `villes.txt`

```
france:paris
vietnam:ho chi minh
italie:roma
france:bordeaux
vietnam:hanoi
inde:delhi
```

et `pays.txt`

```
europa:france
europa:italie
asie:vietnam
asie:chine
```

### 2.1 grep : sélection de lignes

Sélectionne les lignes qui contiennent un certain *motif*.

- `^` : marque de début de ligne.
- `$` : fin de la ligne.
- `.` : caractère quelconque
- `.*` : chaîne quelconque
- `c*` : répétition du caractère c (0-n fois)

```
grep italie villes.txt
grep '^i' villes.txt
grep 'i$' villes.txt
grep '^f.*:p' villes.txt
```

### 2.2 cut : sélection de colonnes

- `-c` : position des caractères
- `-d, -f` : délimiteur et numéros de champs

```
cut -c 1-3 villes.txt
cut -d: -f1 villes.txt
grep vietnam villes.txt | cut -d: -f2
```

### 2.3 sort : tri

Ordonne les lignes selon un critère

```
sort pays.txt
sort -t: -k2 villes.txt
```

Exercice : produire des fichiers `villes.p.txt` et `pays.p.txt` triés par pays.

## 2.4 join : rapprochement

Rapproche deux fichiers sur une *clé commune*. Les fichiers doivent être triés.

```
join -t: -1 2 -2 1 pays.p.txt villes.p.txt
```

## 3 Scripts

*Shell-script* : fichier texte qui contient une suite de commandes.

### 3.1 Shell script

```
#!/bin/bash
# Mon premier essai

clear
echo -n "Nous sommes le_"
date
echo "et c'est mon premier script"
```

### 3.2 Variables

Contiennent des chaînes de caractères.

- Liste des variables affichées par *set*
- Variables système définies automatiquement : HOME, PWD, SHELL, USERNAME, PATH, LANG etc.
- Affectation de variable : `NOM=[CHAINE]`
- Affectation numérique : `let NOM=EXPRESSION`
- Expansion par “\$NOM” ou “\${NOM}”

```
A=2
B=3
C=$A+$B
let D=A+B
echo $C vaut $D
```

### 3.3 Environnement, export

Environnement : valeurs de variables d'un processus.

Visibles à l'aide de *printenv*.

Appel de scripts : les variables système sont (*exportées*) automatiquement.

Export explicite : `export NOM[=VALEUR]`

Voir la liste des variables exportées : `export`.

### 3.4 Paramètres positionnels

Invocation :

```
mon-script Paris Bordeaux "Ho Chi Minh City"
```

Pendant l'exécution

- `$1="Paris"`,
- `$2="Bordeaux"`,

- `$3="Ho Chi Minh City"`
- `$# = 3` – le nombre de paramètres
- `$* = Paris Bordeaux "Ho Chi Minh City"`
- `$0 = "mon-script"` – le nom du script

Exercice : Écrire un script qui indique les villes d'un pays passé en paramètre.

### 3.5 Lecture de variables

`read v1 v2 ...` lit une ligne au terminal, et affecte les mots dans les variables citées.

Exemple : `read nom prenom`

La variable IFS (*input field separator*) indique le séparateur reconnu par `read`

```
$ IFS=, read NOM PRENOM
einstein,albert
$ echo $PRENOM
albert
```

### 3.6 Expansions

*Expansion* : remplacement d'une expression par sa *valeur*

```
echo bonjour $USER
echo perimetre = $((2*(longueur+hauteur)))
echo il y a $(who | wc -l) connexions
echo il y a 'who | wc -l' connexions
```

la dernière forme est déconseillée. Moins lisible, risque de confusion avec les apostrophes, pas d'imbrication possible.

### 3.7 Chaînes et expansion

L'expansion

- se fait dans les chaînes délimitées par `"..."`
- pas dans les chaînes délimitées par `'...'`

```
echo 'la variable $USER ' "contient $USER"
```

### 3.8 Fonctions

Exemple

```
#
destination=/var/svgd

function archiver
{
    local nom=$(basename $1)
    tar czf $destination/$nom.tgz
}

archiver /home/billaud/photos
archiver /home/billaud/musique
```

### 3.9 Structure de contrôle case

```
#!/bin/bash
# Usage: archiver nom-de-répertoire

echo "Format=_normal_gz_?"
read format
case "$format" in
gz)
    option=z
    suffixe=tgz
    ;;
normal | "" )
    option= ;
    suffixe=tar ;;
*)
    echo "format '$format' inconnu" >&2
    exit 1
    ;;
esac

prefixe=$(basename $1)
tar -c${option}f $prefixe.$suffixe $1
```

```
#
# usage:
# tel ajouter num nom
# tel chercher nom
# tel voir
#
nomFichier="telephone.dat"

case "$1" in
ajouter | add )
    shift
    echo "$*" >> $nomFichier
    ;;
chercher | search )
    grep "$2" $nomFichier
    ;;
voir | view )
    cat $nomFichier
    ;;
*)
    echo "Erreur"
    exit 1
    ;;
esac
```

Utilisation des "jokers" de bash

```
case $response of
[00]* )
    echo "d'accord"
    ;;
*)
    echo "tant pis"
    ;;
esac
```

### 3.10 Processus

**CTRL-Z** stoppe la commande en avant-plan.

**fg** ramène la commande en avant-plan.

**bg** relance en avant-plan.

Si il y a plusieurs commandes en arrière-plan, commandes **jobs**, **fg %n**, **bg %n**.

Liste de processus : commande **ps**, options intéressantes : **a,x,u,l,e...**

Voir aussi **top**, **pstree**

La commande **kill** envoie un *signal* à un/des processus

```
$ kill -TERM 4734
$ kill -9 1234 1235
$ kill -1
```

Signaux :

- TERM (9) termine le programme,
- STOP arrête un processus,
- CONT le relance.

commande **&** lance une commande en arrière-plan.

La variable **#!** contient son numéro de processus.

La variable **\$\$** = numero du shell courant.

```
#
mplayer funny-music.mp3 >/dev/null &
music=$!

# sauvegardes
tar czf .....

# arrêter la musique à la fin
kill -9 $music
```

**wait nnn** attend un processus. Exemple :

```
#
# sauvegardes en parallèle

tar czf archive1.tar ..... &
svgd1=$!
tar czf archive2.tar ..... &
svgd2=$!

wait $svgd1
wait $svgd2

echo fini
```

### 3.11 Boucle for

```
for f in *.cc
do
    n=$(wc -l $f)
    echo "le fichier $f contient $n lignes"
done
```

### 3.12 Code de retour : *exit status*

Le code de retour de la dernière commande est dans la variable  **\$?** .

Par convention : 0 = OK.

```

#
# usage:
#  compiler.sh prefixe
#
if g++ -o $1 $1.cc
then
  echo "compilation réussie"
else
  echo "il y a eu un problème"
fi

```

### 3.13 La commande test

Le code de retour dépend d'une *condition*.

```

#!/bin/bash

if test -f $1.cc
then
  g++ -Wall -o $1 $1.cc
  echo Compilation terminée
else
  echo "Erreur: _pas_de_fichier_$1.cc"
  exit 1
fi

```

Autre notation : [ condition ]

Autres tests :

```

[ -d nom ]           # nom est répertoire
[ chaine1 = chaine2 ] # comp. de chaînes
[ chaine1 != chaine2 ]
[ chaine1 \

```

### 3.14 Enchaînement conditionnel

Exécute la seconde commande seulement si la première a réussi (&&) ou échoué (!)

```

#
max=$1
[ $2 -ge $max ] && max=$2
echo $max

```

### 3.15 if-then-elif-else

```

if [ -d $nom ]
then

```

```

  echo $nom est un répertoire
elif [ -f $nom ]
  echo $nom est un fichier
elif [ -H $nom ]
  echo $nom est un lien symb.
else
  echo je ne sais pas
fi

```

### 3.16 Boucle while

```

#
# fac nombre
# calcule la factorielle d'un nombre
#
# exemple:  fac 5
i=1
f=1
while [ $i -le $1 ]
do
  let f=f*i
  let i++
done
echo $f

```

### 3.17 Boucle while-read

```

while read numero nom
do
  printf "| %12s | %-30s |\n" $numero $nom
done < $nomFichier

```

### 3.18 L'instruction break

```

while true
do
  ....
  echo "voulez-vous arrêter ?"
  read reponse
  [ $reponse = oui ] && break
  ...
done

```

### 3.19 Autres...

```

select NAME [in WORDS ... ]
do
  COMMANDS
done

until COMMANDS
do
  COMMANDS
done

```