

# Chapitre 8 : Scripts

## Construction et maintenance de logiciels

Guy Francoeur

basé sur les travaux d'Alexandre Blondin Massé, professeur

29 avril 2019

**UQÀM** | **Département d'informatique**

# Table des matières

1. La programmation Shell
2. Éléments de base du Shell
3. Les tests Shell
4. Structures de contrôle
5. Fonctions
6. Automatiser les tâches de développement

# Table des matières

1. La programmation Shell
2. Éléments de base du Shell
3. Les tests Shell
4. Structures de contrôle
5. Fonctions
6. Automatiser les tâches de développement

- ▶ Un **interpréteur** de commandes pour Unix;
- ▶ Fait le lien entre les **commandes tapées** et le **noyau Unix**;
- ▶ Très utile pour
  - ▶ **Interagir** avec le système;
  - ▶ **Automatiser** des tâches;
  - ▶ **Planifier** des tâches, etc.

# Implémentations

- ▶ Il existe une **multitude** d'implémentations
  - ▶ Bourne shell (sh)
  - ▶ Bourne-Again shell (bash)
  - ▶ C shell (csh)
  - ▶ Korn shell (ksh)
  - ▶ Z Shell (zsh)
  - ▶ ...
- ▶ **Debian/Ubuntu**: Bourne-Again shell (/bin/bash)
- ▶ **Centos/RedHat**: Bourne-Again shell (/bin/bash)

- ▶ Commandes saisies **directement** dans la console:
  - ▶ date
  - ▶ ls
  - ▶ git init
  - ▶ ...
- ▶ Possible de saisir **plusieurs commandes** en une ligne:
  - ▶ date; pwd; ls
  - ▶ make && make install
- ▶ **Différence** entre ; et &&?

# Script Shell

- Un **script** est un fichier contenant une suite de commandes à exécuter:

```
#!/bin/bash
# Exemple de script
mkdir -p tmp
cd tmp
echo -ne "#include<stdio.h>\nint main(void) { puts(\"Hello ,
        World!\"); }" > a.c
gcc a.c
./a.out
cd ..
rm -rf tmp
```

- Pour le lancer:

```
$ /bin/bash script.sh
```

# Sha-bang

- Le **sha-bang** (`#!/`) placé en début de fichier indique le shell à utiliser:

```
#!/bin/bash
# Exemple de script
mkdir -p tmp
cd tmp
echo -ne "#include<stdio.h>\nint main(void) { puts(\"Hello ,
      World!\"); }" > a.c
gcc a.c
./a.out
cd ..
rm -rf tmp
```

- Pour le lancer:

```
$ chmod +x script.sh
$ ./script.sh
```



- ▶ On peut utiliser n'importe quel programme dans le **sha-bang**:
  - ▶ `#!/bin/bash`
  - ▶ `#!/bin/sh`
  - ▶ `#!/usr/bin/perl`
  - ▶ `#!/usr/bin/python`
  - ▶ ...

# Table des matières

1. La programmation Shell
2. Éléments de base du Shell
3. Les tests Shell
4. Structures de contrôle
5. Fonctions
6. Automatiser les tâches de développement

# Variables

- ▶ **Définition** d'une variable:

```
nom=user1  
home=/home/user1
```

- ▶ Nom de variable: **même syntaxe** que les identifiants C.
- ▶ Attention: **pas d'espaces** dans la déclaration!

```
nom = user2 # Invalide
```

- ▶ Accès au **contenu** d'une variable:

```
echo $nom  
echo "Bonjour, $nom!"  
ls $home
```

# Variables

- ▶ Les variables sont considérées comme du **texte**;
- ▶ Pas d'opération **arithmétique** directement sur les variable:

```
i=1  
echo $i + 1 # Affiche "1 + 1"
```

- ▶ Il faut utiliser la commande **expr**:

```
echo `expr $i + 1`
```

- ▶ Ou les **expressions arithmétiques**:

```
echo $((i + 1))
```

- ▶ Bourne Again Shell et Korn Shell **seulement**.

- ▶ Déclaration:

```
tab[0]=pomme  
tab[1]=poire  
tab[2]=fraise
```

# ou

```
tab=(pomme poire fraise)
```

- ▶ Utilisation:

```
echo $tab # Affiche tab[0]  
echo ${tab[1]} # Affiche tab[1]  
echo ${tab[*]} # Affiche toutes les valeurs  
echo $#tab[*]} # Affiche la taille du tableau
```

- ▶ Variables définies au niveau du système:
  - ▶ \$HOME: le répertoire de l'**utilisateur**;
  - ▶ \$PWD: le répertoire **courant**;
  - ▶ \$PATH: les répertoires vers les **binaires** utiles;
  - ▶ \$\$: le numéro du **processus courant**;
  - ▶ \$?: l'**état** (*status*) retourné par la **dernière** commande exécutée.
- ▶ Voir aussi ~/.**bashrc** et ~/.**profile**.

- Qu'affiche le script suivant?

```
#!/bin/bash  
# exercice1.sh  
kill $$  
echo "Hello"
```

## Trois types de guillemets (1/2)

- ▶ Plusieurs types de **guillemets**:
  - ▶ **simples** (');
    - ▶ **doubles** (");
    - ▶ **inversés** (‘);
  - ▶ Les **guillemets simples** permettent de protéger une chaîne utilisant des **caractères spéciaux** :

```
fichier='nom$fichier$avec$dollar$'  
back='meme le caractere \ est preserve'
```

- ▶ Les **guillemets doubles** protègent certains caractères, mais en **interprètent** aussi.



## Trois types de guillemets (2/2)

- ▶ Les **guillemets inversés** (*backticks*) permettent d'exécuter une commande et de la **substituer** :

```
os=`uname`  
srcs=`ls *.c`  
objs=`ls *.o`  
echo "Il est `date +%H:%m`"
```

- ▶ Alternative aux guillemets inversés: utiliser  $\$(...)$ .

```
os=$(uname)  
srcs=$(ls *.c)  
objs=$(ls *.o)  
echo "Il est $(date +%H:%m)"
```

- ▶ Utile si on souhaite **conserver** le résultat d'une commande:

```
path=`pwd`  
echo "Le répertoire courant est $path"
```

# Table des matières

1. La programmation Shell
2. Éléments de base du Shell
3. Les tests Shell
4. Structures de contrôle
5. Fonctions
6. Automatiser les tâches de développement

# La commande test

- ▶ La commande test permet de vérifier le type des fichiers et de comparer des valeurs

test [OPTION] EXPRESSION

- ▶ **Attention!** Si l'expression est **vraie** alors la commande retourne **0** (succès), sinon elle retourne **1**.
- ▶ Exemples:

```
test 1 -lt 2; echo $?
test 'echo "Alex"' = "Alex"; echo $?
test Prof == "Prof" ; echo $?
V1=Vinh ; test $V1 == "vinh" ; echo $?
test -f Makefile; echo $?
test -d bin/; echo $?
```

# Tests sur chaînes de caractères

## ► Syntaxe:

```
test <CHAINE1> <OPERATEUR> <CHAINE2>
```

Option	Exemple	Description
= ou ==	test "Alex" = "Alex"	Identique à
!=	test "Alex" != "alex"	Différent de
-z	test -z ""	Chaîne vide
-n	test -n	Chaîne non-vide

# Tests sur les valeurs numériques

## ► Syntaxe:

```
test <NUM> <OPERATEUR> <NUM2>
```

Option	Exemple	Description
-eq	test 1 -eq 1	Égal à
-ne	test 2 -ne 1	Différent de
-lt	test 1 -lt 2	Strictement inférieur
-le	test 1 -le 2	Inférieur ou égal
-gt	test 2 -gt 1	Strictement supérieur
-ge	test 2 -ge 1	Supérieur ou égal

## ► Attention aux pièges:

```
test "01" = 1 # Retourne faux  
test "01" -eq 1 # Retourne vrai
```

► Syntaxe:

```
test <OPTION> <CHEMIN>
```

Opt.	Exemple	Description
-e	test -e chemin	chemin existe?
-f	test -f chemin	chemin est un fichier?
-d	test -d chemin	chemin est un répertoire?
-s	test -s chemin	chemin est un fichier non vide?
-r	test -r chemin	chemin accessible en lecture?
-w	test -w chemin	chemin accessible en écriture?
-x	test -x chemin	chemin exécutable?

# Opérateurs logiques

## ► Syntaxe:

```
test <EXPRESSION1> <OPERATEUR> <EXPRESSION2>
```

Opt.	Exemple	Description
-a	test \$exp1 -a \$exp2	ET logique
-o	test \$exp1 -o \$exp2	OU logique
!	test ! \$exp	NOT logique

## ► Exemple:

```
# Retourne vrai si foo.sh est un fichier vide  
test -f foo.sh -a ! -s foo.sh
```

- ▶ La syntaxe

```
test EXPRESSION
```

- ▶ est équivalente à:

```
[ EXPRESSION ]
```

- ▶ Exemple:

```
[ -f foo.sh ]  
echo $?
```

- ▶ Attention aux **espaces**!



# Table des matières

1. La programmation Shell
2. Éléments de base du Shell
3. Les tests Shell
4. Structures de contrôle
5. Fonctions
6. Automatiser les tâches de développement

# L'alternative if

- La syntaxe de base:

```
if EXPRESSION
then
  # commandes si EXPRESSION vaut 0
[ else
  # commandes sinon
]
fi
```

- Exemple:

```
fichier=a.out
if test -f $fichier
then
  echo "$fichier ' existe.'"
else
  echo "$fichier ' n'existe pas ou n'est pas un fichier.'"
fi
```

## L'alternative if (suite)

- Il est de fréquent de mettre le if et le then sur la même ligne:

```
if test -f $fichier; then
    echo "$fichier ' existe.'"
else
    echo "$fichier ' n'existe pas ou n'est pas un fichier.'"
fi
```

- Encore plus court:

```
if [ -f $fichier ]; then
    echo "$fichier ' existe.'"
else
    echo "$fichier ' n'existe pas ou n'est pas un fichier.'"
fi
```

# L'alternative multiple avec if

## ► Syntaxe:

```
if EXPRESSION1
then
  # commandes si EXPRESSION1 vaut 0
elif EXPRESSION2
then
  # commandes si EXPRESSION2 vaut 0
[ else
  # commandes sinon
]
fi
```

## ► Exemple:

```
if [ -f $fichier ]; then
  echo "$fichier ' est un fichier"
elif [ -d $fichier ]; then
  echo "$fichier ' est un répertoire"
else
  echo "$fichier ' n'existe pas"
fi
```

# L'alternative multiple avec case

## ► Syntaxe:

```
case EXPRESSION in
X)
  # commandes si EXPRESSION vaut X
  ;;
Y)
  # commandes si EXPRESSION vaut Y
  ;;
*)
  # commandes dans les autres cas
  ;;
esac
```

- Ici, X et Y peuvent être des entiers, des chaînes, des sous-commandes, etc.

# Boucles while et until

## ► Boucle *tant que*:

```
while EXPRESSION; do  
  # commandes tant que EXPRESSION est vraie  
done
```

## ► Boucle *tant que* inverse:

```
until EXPRESSION; do  
  # commandes tant que EXPRESSION est faux  
done
```

## ► Exemple:

```
echo "Alex" | while read -n 1 c; do  
  echo $c # Affiche "A", "l", "e" puis "x"  
done
```

# Boucles for

- Syntaxe:

```
for VARIABLE in EXPRESSIONS; do  
    # commandes pour chaque valeur dans EXPRESSION  
done
```

- Permet d'itérer sur chaque valeur d'une liste

```
for fruit in pomme poire banane; do  
    echo $fruit  
done
```

- Itération sur un tableau:

```
fruits=(pomme poire banane)  
for fruit in ${fruits[*]}; do echo $fruit; done
```

- Itération sur le résultat d'une commande:

```
for fichier in `ls`; do echo $fichier; done
```

# Instructions break et continue

- ▶ Même **comportement** qu'en C
- ▶ Utilisables sur les **boucles** while, until et for
- ▶ Exemple:

```
for i in `seq 0 100`; do
  if [  $$(i \% 2)$  -ne 0 ]; then
    continue
  elif [ $i -gt 10 ]; then
    break
  else
    echo $i
  fi
done
```



# Table des matières

1. La programmation Shell
2. Éléments de base du Shell
3. Les tests Shell
4. Structures de contrôle
5. Fonctions
6. Automatiser les tâches de développement

# Déclarer une fonction

## ► Syntaxe:

```
ma_fonction()  
{  
  # corps de la fonction  
  echo "Hello, fonction!"  
}
```

► Nom de fonction: **même syntaxe** que les identifiants C.

► **Appel** d'une fonction:

```
ma_fonction # Affiche "Hello, fonction"
```

► On appelle une fonction comme on appelle une commande.

# Lire les arguments

- ▶ Les **arguments** d'un script sont disponibles dans des **variables prédéfinies**:
  - ▶ \$0: **nom du script** ou de la **fonction**;
  - ▶ \$1 .. \$9: valeurs des **neuf premiers** arguments;
  - ▶ \$#: **nombre** d'arguments;
  - ▶ \$\*: **tous** les arguments.
- ▶ Accéder aux **autres arguments** avec ksh et bash:

```
echo ${10}
```

- ▶ Sinon utiliser **shift**:

```
echo $1 # Affiche le premier argument  
shift 1  
echo $1 # Affiche le second argument
```

# Exemple avec une fonction

- Fonction acceptant **un argument**:

```
hello_fonction()  
{  
    if [ $# -ne 1 ]; then  
        echo "Usage: hello_fonction <nom>"  
        return 1  
    fi  
    echo "Hello, $1!"  
    return 0  
}
```

- **Appel** de la fonction:

```
hello_fonction Alex # Affiche "Hello, Alex!"  
hello_fonction # Affiche "Usage: hello_fonction <nom>"
```

# Argument d'un script

- ▶ Les **fonctions** et les **scripts** se comportent de la même façon.
- ▶ **Script** acceptant un argument:

```
if [ $# -ne 1 ]; then
    echo "Usage: hello_script.sh <nom>"
    exit 1
fi
echo "Hello , $1!"
```

- ▶ Appeler le script:

```
$ ./hello_script.sh Alex # Affiche "Hello , Alex!"
$ ./hello_script.sh # Affiche "Usage: hello_script.sh <nom>"
```

# Retourner une valeur d'état (*status*)

- ▶ Les fonctions (comme les commandes) peuvent retourner une valeur indiquant leur **état** (en anglais, *status*).
- ▶ On utilise le mot-clé `return`:

```
est_positif()  
{  
    if [ $1 -ge 0 ]; then  
        return 0 # true  
    fi  
    return 1 # false  
}
```

- ▶ **Utiliser** la valeur retournée:

```
if est_positif $1; then  
    echo "$1 est positif (ou égal à 0)"  
else  
    echo "$1 est strictement négatif"  
fi
```

# Retourner une chaîne de caractères

- Rappel: les fonctions sont des **commandes**:

```
hello_fonction()  
{  
    if [ $# -ne 1 ]; then  
        echo "Usage: hello_fonction <nom>"  
        exit 1  
    fi  
    echo "Hello , $1!"  
}  
  
hello='hello_fonction $1'  
echo $hello
```

# Table des matières

1. La programmation Shell
2. Éléments de base du Shell
3. Les tests Shell
4. Structures de contrôle
5. Fonctions
6. Automatiser les tâches de développement



**Automatiser** les choses que l'on fait souvent:

- ▶ sauvegardes
- ▶ tests
- ▶ déploiements
- ▶ ... les possibilités sont infinies ...

# Commandes utiles (1/2)

Pour la **sauvegarde** et le **déploiement**:

- ▶ git: pour **versionner** les sauvegardes;
- ▶ tar et zip: pour créer des **archives**;
- ▶ ftp: pour **échanger** des fichiers via FTP;
- ▶ scp: pour **échanger** des fichiers via SSH;
- ▶ cron: pour **planifier** le lancement des commandes.

## Commandes utiles (2/2)

Pour écrire des **suites de tests**:

- ▶ make: pour **compiler** et **lancer** des programmes;
- ▶ diff: pour **comparer** deux fichiers;
- ▶ sed et awk: pour apporter des **modifications textuelles** à des fichiers;
- ▶ timeout: pour **tuer** une commande après un certain délai.

# Exemple de script de sauvegarde

- **Sauvegarder** un répertoire sur un **serveur**:

```
#!/bin/bash
#sauvegarde.sh
dossier=mon_projet/
timestamp='date +%s'
sauvegarde=sauvegarde_${timestamp}.tar.gz

echo "Création de la sauvegarde $timestamp"

tar -zcvf $sauvegarde $dossier
scp $sauvegarde usager@java.labunix.uqam.ca:~/
```

# Exemple dans crontab

- ▶ Automatiser l'exécution de sauvegarde.sh :
- ▶ Lancer l'édition :

```
$ crontab -e
```

- ▶ Exemples :

```
# --> une heure du matin, du lundi au vendredi  
0 1 * * 1-5 /chemin/vers/sauvegarde.sh  
#ou  
# --> toutes les 10 minutes  
*/10 * * * * /chemin/vers/sauvegarde.sh
```

- ▶ <http://frederic-lang.developpez.com/tutoriels/linux/prog-shell/>
- ▶ <http://www.freeos.com/guides/lsst/>
- ▶ <https://openclassrooms.com/courses/reprenez-le-controle-a-l-aide-de-linux/introduction-aux-scripts-shell>
- ▶ <http://overthewire.org/wargames/bandit/>