Unix/Linux V

1^{re} année DUT

Université de Marne La vallée

1. Des scripts Shell
2. premier script
3. Les variables en shell
   - Déclaration
   - Saisie
   - Opérations mathématiques
   - Les variables d'environnements
   - Les variables des paramètres
4. Conditionnelles
   - If
   - If then else
   - Sinon si
5. Les tests
6. Les Boucles
   - `while` : boucler "tant que"
   - `for` : boucler sur une liste de valeurs

la programmation shell. De quoi s'agit-il ?

Imaginez un mini-langage de programmation intégré à Linux. Ce n'est pas un langage aussi complet que peuvent l'être le `C`, le `C++` ou le `Java` par exemple, mais cela permet d'automatiser la plupart de vos tâches. Voici un aperçu de ce qu'on peut faire avec :

- Sauvegarde de vos données
- Surveillance de la charge de votre machine
- Système de gestion personnalisé de vos téléchargements
- ...etc

Pourquoi pas le `C` ?

Le gros avantage des **`scripts shell`**, c'est qu'ils sont totalement intégrés à Linux : il n'y a rien à installer et rien à compiler. Et surtout : vous avez trés peu de nouvelles choses à apprendre. En effet, toutes les commandes que l'on utilise dans les **`scripts shells`** sont des commandes du système que vous connaissez déjà : **`ls, cut, grep, sort`**

`shell`: Un interpréteur de commandes

Les fonctionnalités offertes par l'invite de commande peuvent varier en fonction du `shell` que l'on utilise.
Les principaux sont

- `sh` : Bourne Shell. L'ancêtre de tous les shells.
- `bash` : Bourne Again Shell. Une amélioration du Bourne Shell, disponible par défaut sous Linux et Mac OS X.
- `ksh` : Korn Shell. Un shell puissant présent sur les Unix propriétaires, mais aussi disponible en version libre, compatible avec bash.
- `csh` : C Shell. Un shell utilisant une syntaxe proche du `C`.
- `tcsh` : Tenex C Shell. Amélioration du C Shell.
- `zsh` : Z Shell. Shell assez récent reprenant les meilleures idées de `bash`, `ksh` et `tcsh`.

A quoi sert un shell

`Shell` : programme qui gère l'invite de commandes. C'est donc le programme qui attend que vous rentriez des commandes :

- Se souvenir quelles étaient les dernières commandes tapées
- Auto-complétion d'une commande ou d'un nom de fichier lorsque vous appuyez sur `Tab`
- Gérer les processus (envoi en arrière-plan, mise en pause avec Ctrl + Z ...).
- Rediriger et chaîner les commandes (les fameux symboles $>$, $<$, | ...)

Avec quel shell écrire nos scripts alors ? `bash`

- On le trouve par défaut sous Linux et Mac OS X (cela couvre assez de monde !).
- Il rend l'écriture de scripts plus simple que **`sh`**.
- Il est plus répandu que **`ksh`** et **`zsh`** sous Linux.

En clair, le **`bash`** est un bon compromis entre **`sh`** (le plus compatible) et **`ksh/zsh`** (plus puissants).

- Commençons par créer un nouveau fichier pour notre script : `gedit essai.sh` $\longrightarrow$ fichier vide
- La première chose à faire dans un script shell est d'indiquer... quel shell est utilisé : Rajouter dans `essai.sh` la ligne

  `#!/bin/bash`

  le #! est appelé le sha-bang
- Après le sha-bang, nous pouvons commencer à coder.
  Le principe : Ecrire les commandes que vous souhaitez exécuter. Ce sont les mêmes que celles que vous tapiez dans l'invite de commandes !

Exple :

```
#!/bin/bash
ls
```

- Donner les droits d'exec au script

```
chmod +x essai.sh
```

- Exécuter le script, en tapant " ./" devant le nom du script

```
./essai.sh
```

Que fait le script ? Il fait juste un `ls`, donc il affiche la liste des fichiers dans le répertoire.
On peut vouloir préciser en plus le rep courant :

```
# !/bin/bash
pwd
ls
```

Créer sa propre commande :

Actuellement, le script doit être lancé via `./essai.sh` et vous devez être dans le bon répertoire.
Comment font les autres programmes pour pouvoir être exécutés depuis n'importe quel répertoire sans "./" devant ? Ils sont placés dans un des rep. du `PATH`.
Def : Le `PATH` est une variable système qui indique où sont les programmes exec. Si vous tapez echo $PATH , vous aurez la liste de ces rep.$\longrightarrow$ déplacer ou copier le script dans un de ces rep, (`/bin, ou /usr/bin, ou /usr/local/bin`).
Rq : Il faut être `root` pour pouvoir faire ça.

Comme dans tous les langages de programmation, on trouve en `bash` ce qu'on appelle des variables.
$\longrightarrow$ stocker temporairement des informations en mémoire. C'est en fait la base de la programmation.
Les variables en `bash` sont particulières. Il faut être très rigoureux lorsqu'on les utilise $\longrightarrow$ différent du `C`

Variables :

- Un nom
- Une valeur

Exple

```
message='Bonjour tout le monde'
```

Rq :Pas d'espace autour de "="
Executons!!! `./var.sh`

echo : afficher une variable

Exple

- `echo "Salut tout le monde"`
- `echo -e "Message\n Autre ligne"`

`./varaffich.sh`

Les quotes

- Les apostrophes ' '(simples quotes)

  `./simplequote.sh`
- Les guillemets " " (doubles quotes)

  `./doublequote.sh`
- Les accents graves ` ` (back quotes)

  `./backquote.sh`

read

Demander au user de saisir du texte avec la commande `read`. La façon la plus simple de l'utiliser est d'indiquer le nom de la variable dans laquelle le message saisi sera stocké :

`./read1.sh`

La commande read propose plusieurs options intéressantes.

- `-p` : afficher un message de prompt `./readp.sh`
- `-n` : limiter le nombre de caractères `./readn.sh`
- `-s` : ne pas afficher le texte saisi `./reads.sh`

En `bash`, les var. sont toutes des chaînes de caractères
⇒Incapable de manipuler des nombres ⇒ pas opérations !!

la commande `let`

`./calcul1.sh`

Les opérations :

- L'addition : +
- La soustraction : −
- La multiplication : ∗
- La division : /
- La puissance : ∗∗
- Le modulo : %

Les var. que crées dans scripts `bash` n'existent que dans ces scripts. *ie.* une variable définie dans un `pgme A` ne sera pas utilisable dans un `pgme B`.

Les var. d'environnement :var. utilisables n'importe quel pgme. On parle aussi parfois de var. globales. Afficher toutes celles actuellement en mémoire avec la commande `env`.

Quelques variables d'environnement

- `SHELL` : type de shell est en cours d'utilisation (`sh, bash, ksh...`)
- `PATH` : une liste rep qui contiennent des exec que vous souhaitez pouvoir lancer sans indiquer leur rep.
- `EDITOR` :Editeur de txt par défaut
- `HOME` : position du dossier `home`
- `PWD` : Dossier courant

Rq : En majuscule

Les scripts bash acceptent des paramètres

```
./varparam.sh param1 param2 param3
```

- $# : contient le nombre de param.
- $0 : contient le nom du script exécuté (ici "./varparam.sh")
- $1 : contient 1^r param.
- ...
- $9 : contient 9^m param.

Syntaxe

```
if [ test ]
then
echo "true"
fi
```

Rq : l'espace dans `[ test ]`

`./if1.sh`

Syntaxe

```
if [ test ]
then
echo "true"
else
echo "false"
fi
```

```
./if2.sh param1
```

Syntaxe

```
if [ test ]
then
echo "premier test a été verif"
elif [ autre_test ]
echo "second test a été verif"
elif [ encore_autre_test ]
echo "troisième test a été verif"
else
echo "Aucun des tests prec. n'a été vérifié"
fi
```

```
./if3.sh param1
```

3 types de tests différents en `bash` :

1. Tests sur des chaînes de caractères
2. Tests sur des nombres
3. Tests sur des fichiers

- `$chaine1 = $chaine2`
  teste si 2 chaînes sont identiques. `B` $\neq$ `b` (sensible à la casse...)
- `$chaine1 != $chaine2`
  Teste si 2 chaînes sont $\neq$
- `-z$chaine`
  Teste si 1 chaînes est vide
- `-n$chaine`
  Teste si 1 chaînes est non vide

```
./test1.sh param1 param2
```

Exo :Ecrire un script qui teste l'existence d'un paramètre

- `$num1 -eq $num2` Teste si les nombres sont égaux(equal) "=" compare les caractères.
- `$num1 -ne $num2` Teste si les nombres sont diff(non equal) "!=" compare les caractères.
- `$num1 -lt $num2` Teste si num1 est < num2 (lower than)
- `$num1 -le $num2` Teste si num1 est <= num2 (lower or equal)
- `$num1 -gt $num2` Teste si num1 est > num2 (greater than)
- `$num1 -ge $num2` Teste si num1 est >= num2 (greater or equal)

```
./test2.sh param1
```

- `-e $nomfich` Teste si le fich. existe
- `-d $nomfich` Teste si le fich. est un rep.
- `-f $nomfich` Teste si le fich. est un... fich. Un vrai fich. pas un dossier.
- `-L $nomfich` Teste si fich est un lien symbolique
- `-r $nomfich` Teste si fich est lisible (r)
- `$fich1 -nt $fich2` Teste si fich1 est plus récent que fich2 (newer than) | `$fich1 -ot $fich2` (older than)

EXO Ecrire un script qui demande au user de rentrer le nom d'un rep et de verifier si c'est bien un rep

Effectuer plusieurs tests à la fois

Dans un `if`, il est possible de faire plusieurs tests à la fois.

- Si un test est vrai ET qu'un autre test est vrai : `&&`
- Si un test est vrai OU qu'un autre test est vrai : `||`

Rq :encadrer chaque condition par des crochets
EXO Ecrire un script qui vérifie qu'il a au moins un param et la valeur du 1^r param est `asticot`

Syntaxe

```
while [ test ]
do
echo 'Action en boucle'
done
```

```
while [ test ]; do
echo 'Action en
boucle' done
```

./while1.sh

La boucle `for` permet de parcourir une liste de valeurs, et de boucler autant de fois qu'il y a de valeurs.

Syntaxe

```
for variable in 'valeur1' 'valeur2' 'valeur3'
do
echo "La variable vaut $variable"
done
```

`./for1.sh`

Rq : La liste des valeurs n'a pas besoin d'être définie directement dans le code :

`./for2.sh`

EXO : Script qui renomme tous les fichiers trouvés

Un cas plus classique du `for`

```
for i in ‘seq 1 10‘;
do
echo $i
done
```

`./for3.sh`

Pour faire des sauts de 2 faire `for i in ‘seq 1 2 10‘;`