Introduction aux scripts shell

Un script shell permet d'automatiser une série d'opérations. Il se présente sous la forme d'un fichier contenant une ou plusieurs commandes qui seront exécutées de manière séquentielle.

```
#!/bin/bash
# This script will take an animated GIF and delete every other frame
# Accepts two parameters: input file and output file
# Usage: ./<scriptfilename> input.gif output.gif

# Make a copy of the file
cp "$1" "$2"

# Get the number of frames
numframes=$(gifsicle --info "$1" \
              | grep --perl-regexp --only-matching '\d+ images' \
              | grep --perl-regexp --only-matching '\d+')

# Deletion
let i=0
while test $i -lt $numframes
do
    rem=$(( $i % 2 ))

    if test $rem -eq 0
    then
        gifsicle "$2" --delete "#"$(($i/2)) -o "$2"
    fi

    let i=i+1
done
```

Pour faire qu'un script soit exécutable

Méthode graphique

Votre script est un simple fichier texte, par défaut il s'ouvre donc avec l'éditeur de texte défini par défaut (ex : Gedit dans une session Unity ou Gnome).
Pour qu'il soit autorisé à se lancer en tant que programme, il faut modifier ses propriétés. Pour cela faites un clic droit sur son icône, et dans l'onglet “Permissions” des “Propriétés”, cocher la case *“autoriser l'exécution du fichier comme un programme”*.

Par la suite, un double-clic sur l’icône vous laissera le choix entre afficher le fichier (dans un éditeur de texte) et le lancer (directement ou dans un terminal pour voir d'éventuels messages d'erreurs)

Par ailleurs Nautilus ne propose pas de lancer le script par simple clic avec les réglages de bases. Il faut aller dans **Menu→ Édition → Préférences → Onglet comportement → fichier texte et**

exécutable et cocher pour fichiers exécutables **Demander à chaque fois**.

Problème connu

Sous Lubuntu, si cette méthode ne fonctionne pas, vous devez d'abord effectuer l'opération suivante :

1. Dans le menu principal, allez sur *Outils système* et faites un *clic droit* → *Propriétés* sur le raccourci vers le terminal. Notez le contenu du champ *Commande* et annulez.
2. Ouvrez votre gestionnaire de fichier PCManFM et allez dans le menu supérieur sur *éditer* → *Préférences* puis dans la fenêtre qui s'ouvre sélectionnez *Avancé*.
3. Remplacez le contenu du champ *Terminal emulator* par le contenu du champ *Commande* que vous avez pris soin de noter à la première étape.
4. Vous pouvez ensuite suivre la méthode graphique indiquée ci-dessus pour exécuter vos scripts shell.

Méthode dans un terminal

Il suffit de se placer dans le dossier où est le script, et de lancer :

```
bash nom_du_script
```

mais pas toujours bash (dépend du langage du script)

ou si vous voulez l'exécuter par son nom , il faut le rendre exécutable avec `chmod`. Pour ceci tapez la commande qui suit :

```
chmod +x nom_du_script
```

Puis vous pouvez exécuter le script en faisant :

```
./nom_du_script
```

mais pourquoi le ./ ?

Le chemin ./

Il peut être intéressant d'ajouter un répertoire au “PATH” pour pouvoir exécuter ses scripts sans avoir à se placer dans le bon dossier. Je m'explique, quand vous tapez une commande (“ls” par exemple), le shell regarde dans le PATH qui lui indique où chercher le code de la commande.

Pour voir à quoi ressemble votre PATH, tapez dans votre console:

```
echo $PATH
```

Cette commande chez moi donnait initialement :

```
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
```

C'est à dire que le shell va aller voir si la définition de la commande tapée (“ls” pour continuer sur le même exemple) se trouve dans **/usr/local/bin** puis dans **/usr/bin**... jusqu'à ce qu'il la trouve.

Ajouter un répertoire au PATH peut donc être très pratique. Par convention, ce répertoire s'appelle **bin** et se place dans votre répertoire personnel. Si votre répertoire personnel est /home/toto, ce répertoire sera donc **/home/toto/bin**. Pour pouvoir utiliser vos scripts en tapant directement leur nom (sans le “./”) depuis n'importe quel répertoire de votre ordinateur, il vous suffit d'indiquer au shell de chercher aussi dans ce nouveau dossier en l'ajoutant au PATH. Pour ceci, il suffit de faire :

```
export PATH=$PATH:$HOME/bin
```

La commande

```
echo $PATH
```

retourne maintenant

```
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:/home/toto/bin
```

et je peux lancer le script appelé “monScript” situé dans “/home/toto/bin” en tapant directement : monScript

Cette procédure est pour une modification temporaire du PATH et qui sera donc effacée à la fin de la session. Pour rendre la modification permanente, ajouter la commande dans le fichier texte caché .bashrc se trouvant dans votre dossier personnel ainsi que dans le dossier /root.

Dans les dernières versions de ubuntu (12.04 +) si le dossier $HOME/bin existe il est automatiquement ajouté au PATH. La commande est incluse dans le fichier ~/.profile lancé lors de toutes sessions (graphique ou console).

Les différents types de shells

Comme vous avez sûrement dû l'entendre, il existe différents types de shells ou en bon français, interpréteurs de commandes :

- dash (*Debian Almquist shell*) : shell plus léger que bash, installé par défaut sur Ubuntu ;
- bash (*Bourne Again SHell*) : conçu par le projet GNU, shell linux ; le shell par défaut sur Ubuntu ;
- rbash : un shell restreint basé sur bash. Il existe de nombreuses variantes de bash ;
- csh, tcsh : shells C, créés par Bill Joy de Berkeley ;
- zsh, shell C écrit par Paul Falstad ;
- ksh (⇔ ksh88 sur Solaris et équivaut à ksh93 sur les autres UNIX/Linux cf.Korn shell History): shells korn écrits par David Korn, pdksh (Public Domain Korn Shell ⇔ ksh88) ;
- rc : shell C, lui aussi conçu par le projet GNU ;
- tclsh : shell utilisant Tcl ;
- wish : shell utilisant Tk .

Il existe bien entendu beaucoup d'autres types de shells.
Pour savoir quel type de shell est présent sur une machine, aller dans un terminal et taper la commande **ps**.

La commande **sh** est en fait un lien symbolique vers l'interpréteur de commandes par défaut : **/bin/dash**.

Les variables

Il faut savoir que en bash les variables sont toutes des chaînes de caractères.
Cela dépendra de son USAGE, pour une opération arithmétique prochaine voir : let ma_variable sinon pour conserver une valeur : il suffit de lui donner un nom et une valeur avec l'affectation égale :

```
ma_variable=unmot
```

Ici la valeur est affectée à la variable ma_variable .
Attention: pas d'espace ni avant ni après le signe “=” .
Autre exemple avec une commande avec arguments :

```
nbre_lignes=$(wc -l < fichier.ext)
```

nbre_lignes contiendra le nombre de lignes contenu dans *fichier.ext* .

Pour voir le contenu d'une variable, on utilisera echo (par exemple) :

```
echo $ma_variable
```

renverra : unmot .

Pour gérer les espaces et autres caractères spéciaux du shell, on utilisera les guillemets ou bien une notation avec des apostrophes :

```
echo $ma_variable
```

```
echo "$ma_variable"
```

```
echo ${ma_variable}
```

```
echo "${ma_variable}"
```

renverront toutes la même réponse : unmot .

Et avec des chemins de répertoires :

```
chemin_de_base="/home/username/un repertoire avec espaces"
chemin_complet="$chemin_de_base/repertoire"
```

Comme on le voit ci-dessus si on met une chaîne de caractères avec des espaces

entre guillemets, la variable la prend bien mais attention à l'utiliser aussi avec des guillemets...

```
rsync -av "$chemin_complet" …
```

sinon les espaces reprennent leurs rôles de séparateur!

Des variables système permettent d'accélérer la saisie et la compréhension. Pour voir les variables d'environnement de votre système tapez simplement :

```
env
```

Quelques variables d'environnement à connaître : HOME, USER, PATH, IFS,...

Pour appeler ou voir une variable, par exemple HOME, il suffit de mettre un $ devant, par exemple :

```
echo $HOME
```

Ce petit code va afficher la variable HOME à l'écran.

Il existe des variables un peu spéciales :

Nom	fonction
$*	contient tous les arguments passés à la fonction
$#	contient le nombre d'arguments
$?	contient le code de retour de la dernière opération
$0	contient le nom du script
$n	contient l'argument n, n étant un nombre
$!	contient le PID de la dernière commande lancée

Exemple : créer le fichier arg.sh avec le contenu qui suit :

```
#!/bin/bash
echo "Nombre d'arguments ... :  "$#
echo "Les arguments sont ... :  "$*
echo "Le second argument est :  "$2

echo "Et le code de retour du dernier echo est :  "$?
```

Lancez ce script avec un ou plusieurs arguments et vous aurez :

```
./arg.sh 1 2 3
Nombre d'arguments ... :  3
Les arguments sont ... :  1 2 3
Le second argument est :  2
Et le code de retour du dernier echo est :  0
```

Exemple: un sleep interactif pour illustrer $! (Cf. les fonctions).
Pour déclarer un tableau, plusieurs méthodes : première méthode (compatible bash, zsh, et ksh93

mais pas ksh88, ni avec dash, qui est lancé par “sh”) :

```
tab=("John Smith" "Jane Doe")
```

ou bien :

```
tab[0]='John Smith'
tab[1]='Jane Doe'
```

Pour compter le nombre d'éléments du tableau :

```
len=${#tab[*]} ou echo ${#tab[@]}
```

Pour afficher un élément :

```
echo ${tab[1]}
```

Pour afficher tous les éléments :

```
echo ${tab[@]}
```

ou bien (en bash ou en ksh93 mais pas en ksh88) :

```
for i in ${!tab[@]}; do echo ${tab[i]}; done
```

ou encore (C style) :

```
for (( i=0; i < ${#tab[@]}; i++ )); do echo ${tab[i]}; done
```

NB : toutes les variables sont des tableaux. Par défaut, c'est le premier élément qui est appelé :

```
echo ${tab[0]}
```

et :

```
echo ${tab}
```

renverront la même réponse.

NB2 : les tableaux sont séparés par un séparateur défini : l'IFS. Par défaut l'IFS est composé des trois caractères : $' \t\n' soit espace, tabulation, saut de ligne. Il peut être forcé sur un autre caractère.

```
IFS=$SEPARATEUR
```

$SEPARATEUR pourra être :

- une lettre (pe : n, i,...)
- une ponctuation (pe : ',', '.', '-'...)
- un caractère spécial : ($'\t' : tabulation, $'\n' : saut de ligne,...)

Les arguments en ligne de commande

Pour passer des arguments en ligne de commande c'est encore une fois très simple. Chaque argument est numéroté et ensuite on l'appelle par son numéro :

```
./test.sh powa noplay
```

Voici notre test.sh

```
#!/bin/sh
echo $3
echo $2
```

Notez que $0 est le nom du fichier.

`shift` est une commande très pratique lorsque vous traitez des arguments en ligne de commande. Elle permet de faire “défiler” les arguments ($0, $1, $2, ...). C'est à dire que le contenu de $1 passe dans $0, celui de $2 dans $1 et ainsi de suite. Il est tout à fait possible de traiter les arguments avec `for i in $*; do` mais lorsque vous aurez des options du style `–title “mon_titre”` il sera très laborieux de récupérer la valeur “`mon_titre`”.

Voici un exemple de script où vous devez vous souvenir de ce que vous avez écrit (un petit jeu de mémoire, quoi) :

```
#!/bin/sh
clear # Un peu facile si la commande reste au dessus :-)
until [ $# = 0 ]
do
  echo -n "Taper l'option suivante : "
  read Reslt
  if [ "$Reslt" = "$1" ]; then
    echo "Bien joué !"
  else
    echo "Non mais quand même !!! C'ÉTAIT $1 ET NON PAS $Reslt PETIT FRIPON !!!"
    sleep 3 # Juste pour le fun du script qui rage ;-p
    echo "Donc je te bannis de ubuntu-fr.org ! Et toc !! Tu ne peux rien contre moi !!!"
    exit 1
  fi
  shift # On défile
done
echo "Vous avez réussi !"
```

L'arithmétique

```
(( variable = 2 + $autre_var * 5 ))
```

Exemple: besoin de définir des plages de valeurs (1 à 500 puis 501 à 1000 puis 1001 à 1500...)

```
id_per_step = 500
for (( i=0; i<8; i++ )); do
    (( min_step_id = 1 + $i * $id_per_step ))
    (( max_step_id = (( $i + 1 )) * $id_per_step ))
    echo "$min_step_id to $max_step_id "
done
```

Vocabulaire

La commande test

La commande test existe sous tous les Unix, elle permet de faire un test et de renvoyer 0 si tout s'est bien passé ou 1 en cas d'erreur.

En mode console, faites man test pour connaître tous les opérateurs, en voici quelques-uns :

Opérateurs de test sur fichiers

Syntaxe	Fonction réalisée
-e fichier	renvoie 0 si fichier existe.
-d fichier	renvoie 0 si fichier existe et est un répertoire.
-f fichier	renvoie 0 si fichier existe et est un fichier 'normal'.
-w fichier	renvoie 0 si fichier existe et est en écriture.
-x fichier	renvoie 0 si fichier existe et est exécutable.
f1 -nt f2	renvoie 0 si f1 est plus récent que f2.
f1 -ot f2	renvoie 0 si f1 est plus vieux que f2.

Opérateurs de comparaison numérique

Syntaxe	Fonction réalisée
$A -lt 5	renvoie 0 si $A est strictement inférieur à 5
$A -le 5	renvoie 0 si $A est inférieur ou égal à 5
$A -gt 5	renvoie 0 si $A est strictement supérieur à 5
$A -ge 5	renvoie 0 si $A est supérieur ou égal à 5
$A -eq 5	renvoie 0 si $A est égal à 5
$A -ne 5	renvoie 0 si $A est différent de 5

Les crochets

On peut raccourcir la commande test par des crochets. Exemple :

```
test -f /etc/passwd
echo $?
0
[ -f /etc/passwd ]
echo $?
```

```
0
```

Affichera la valeur 0 : ce fichier existe, 1 dans le cas où le fichier /etc/passwd n'existe pas. Sous Unix, le code de retour est par convention et en général 0 s'il n'y a aucune erreur et différent de 0 dans les autres cas.

La syntaxe la plus appropriée dans de la programmation shell moderne est le double crochet :

```
[[ -f /etc/passwd ]]
```

Cela gère bien mieux les problèmes d'espaces dans les noms de fichiers, les erreurs etc... C'est une structure ~~propre à~~ bash (ksh, ?) qui est le shell par défaut dans la plupart des distributions Linux, et de Ubuntu en particulier. On garde en général des simples crochets pour les scripts shell qui doivent être à tout prix POSIX (utilisation sur des Unix sans installation préalable de bash, comme BSD, Solaris...) .

Les opérateurs logiques

Il y a en 3 :

- le **et** logique : -a
- le **ou** logique : -o
- le **non** logique : !

Exemple :

```
echo "renverra 0 si les deux expressions sont vraies"
test expr1 -a expr2
[ expr1 -a expr2 ]
```

Table de vérité de « -a »

Comparaison	Résultat	Calcul
0 et 0	0	0 × 0 = 0
0 et 1	0	0 × 1 = 0
1 et 0	0	1 × 0 = 0
1 et 1	1	1 × 1 = 1

Les deux assertions doivent être vérifiées pour que la condition le soit aussi.

Table de vérité de « -o »

Comparaison	Résultat	Calcul
0 ou 0	0	0 + 0 = 0
0 ou 1	1	0 + 1 = 1
1 ou 0	1	1 + 0 = 1
1 ou 1	1	1 + 1 = 1

Dès que l'une des deux assertions est vérifiée, la condition globale l'est aussi.

Exemple plus complet :

```
#!/bin/sh

echo -n "Entrez un nom de fichier: "
read file
if [ -e "$file" ]; then
        echo "Le fichier existe!"
else
        echo "Le fichier n'existe pas, du moins n'est pas dans le répertoire
d'exécution du script"
fi
```

La seule chose qui prête à confusion est que l'on vérifie seulement si le fichier « file » est dans le répertoire où le script a été exécuté.

La structure : `if`

Avant de commencer à faire des scripts de 1000 lignes, il serait intéressant de voir comment se servir des variables, et des instructions `if`, `then`, `elif`, `else`, `fi`. Cela permet par exemple de faire réagir le script de manière différente, selon la réponse de l'utilisateur à une question.

En bash, les variables ne se déclarent généralement pas avant leur utilisation, on les utilise directement et elles sont créées lors de sa première mise en œuvre.

Pour pouvoir voir la valeur d'une variable il faut faire précéder son nom du caractère « $ ».

```
#!/bin/sh
echo -n "Voulez-vous voir la liste des fichiers Y/N : "
read ouinon
if [ "$ouinon" = "y" ] || [ "$ouinon" = "Y" ]; then
    echo "Liste des fichiers :"
    ls -la
elif [ "$ouinon" = "n" ] || [ "$ouinon" = "N" ]; then
    echo "Ok, bye! "
else
    echo "Il faut taper Y ou N!! Pas $ouinon"
fi
```

Explication

Ce script peut paraître simple à première vue mais certaines choses prêtent à confusion et ont besoin d'être expliquées en détail.

Tout abord, le `echo -n` permet de laisser le curseur sur la même ligne, ce qui permet à l'utilisateur de taper la réponse après la question (question d'esthétique).

L'instruction `read` permet d'affecter une valeur ou un caractère à une variable quelconque, en la demandant à l'utilisateur.

En bash, la variable est considérée comme une chaîne même si celle-ci contient une valeur numérique, et les majuscules sont considérées différentes des minuscules, $M ≠ $m.

Ensuite vient l'instruction conditionnelle `if`. Elle est suivie d'un « [» pour délimiter la condition. La condition doit bien être séparée des crochets par un espace ! Attention, la variable est mise entre guillemets car dans le cas où la variable est vide, le shell ne retourne pas d'erreur, mais en cas contraire, l'erreur produite ressemble à :

```
[: =: unaryoperator expected
```

L'opérateur `||` signifie exécuter la commande suivante si la commande précédente n'a pas renvoyé 0. Il existe aussi l'opérateur && qui exécute la commande suivante si la commande précédente a renvoyé 0, et enfin `;` qui exécute l'opération suivante dans tous les cas.

Exemple : créer le répertoire toto s'il n' existe pas

```
[ ! -d /tmp/toto ] && mkdir /tmp/toto
[ -d /tmp/toto ] || mkdir /tmp/toto
test ! -d /tmp/toto && mkdir  /tmp/toto
rm -rf /tmp/toto;mkdir /tmp/toto
```

Les « { » servent à bien délimiter le bloc d'instructions suivant le `then`, est une commande et donc si elle est sur la même ligne que le `if` les deux commandes doivent être séparées par un `;`

Ensuite, `elif` sert à exécuter une autre série d'instructions, si la condition décrite par `if` n'est pas respectée, et si celle fournie après ce `elif` l'est.

Enfin, `else` sert à exécuter un bloc si les conditions précédentes ne sont pas respectées (ah les jeunes, ils respectent plus rien de nos jours 🙂).

`fi` indique la fin de notre bloc d'instructions `if`. Cela permet de voir où se termine la portion de code soumise à une condition.

Quelques petites commandes pratiques :

```
sh -n nom_du_fichier
```

ou

```
bash -x chemin_du_fichier
```

Cette commande vérifie la syntaxe de toutes les commandes du script, pratique quand on débute et pour les codes volumineux.

```
sh -u nom_du_fichier
```

Celle-ci sert à montrer les variables qui n'ont pas été utilisées pendant l'exécution du programme.

Voici le tableau des opérateurs de comparaison, ceux-ci peuvent s'avérer utiles pour diverses raisons, nous verrons un peu plus loin un exemple.

```
$A = $B # Vérifie si les deux chaînes sont égales.

$A != $B # Vérifie si les deux chaînes sont différentes.

-z $A # Vérifie si A n'existe pas (ne contient pas de chaîne).

-n $A # Vérifie si A existe (contient une chaîne).
```

Les structures while et until

La commande `while` exécute ce qu'il y a dans son bloc tant que la condition est respectée :

```
#!/bin/sh

cmpt=1
cm=3
echo -n "Mot de passe : "
read mdp

while [ "$mdp" != "ubuntu" ] && [ "$cmpt" != 4 ]
do
     echo -n "Mauvais mot de passe, plus que "$cm" chance(s): "
     read mdp
     cmpt=$(($cmpt+1))
     cm=$(($cm-1))
done
echo "Non mais, le brute-force est interdit en France !!"
```

On retrouve des choses déjà abordées avec `if`. Le `&&` sert à symboliser un “et”, cela implique que deux conditions sont à respecter. Le `do` sert à exécuter ce qui suit si la condition est respectée. Si elle ne l'est pas, cela saute tout le bloc (jusqu'à `done`). Vous allez dire :

Mais qu'est-ce que c'est ce truc avec cette syntaxe bizarre au milieu ?

Cette partie du code sert tout simplement à réaliser une opération arithmétique. A chaque passage, 'cmpt = cmpt+1' et 'cm = cm-1'.

`while` permet de faire exécuter la portion de code un nombre indéterminé de fois. La commande `until` fait la même chose que la commande `while` mais en inversant. C'est-à-dire qu'elle exécute le bloc jusqu'à ce que la condition soit vraie, donc elle s'emploie exactement comme la commande `while`.

Par exemple, si on a besoin d'attendre le démarrage de notre window manager pour exécuter des commandes dans notre Xsession il sera plus intéressant d'utiliser le `until`:

```
#!/bin/sh
until pidof wmaker
do
  sleep 1
done
xmessage "Session loaded" -buttons "Continue":0,"That all":1;
[ $? -eq 0 ] && xmessage "Load more..."
```

Mais on aurait pu aussi faire:

```
#!/bin/sh
while [ -z $(pidof wmaker) ]
do
 sleep 1
done
#(...)
```

La structure case

Regardons la syntaxe de cette commande, qui n'est pas une des plus simples :

```
case variable in
    modèle [ | modèle] ...) instructions;;
    modèle [ | modèle] ...) instructions;;
      ...
esac
```

Cela peut paraître complexe mais on s'y habitue quand on l'utilise.

> Mais à quoi sert cette commande ?

Elle sert à comparer le contenu d'une variable à des modèles différents. Les ;; sont indipensables car il est possible de placer plusieurs instructions entre un modèle et le suivant. Les ;; servent donc à identifier clairement la fin d'une instruction et le début du modèle suivant.

Exemple :

```
#!/bin/sh

echo -n "Êtes-vous fatigué ? "
read on

case "$on" in
    oui | o | O | Oui | OUI ) echo "Allez faire du café !";;
    non | n | N | Non | NON ) echo "Programmez !";;
    * ) echo "Ah bon ?";;
```

```
esac
```

La seule chose qui mérite vraiment d'être expliquée est sans doute `* )`. Cela indique tout simplement l'action à exécuter si la réponse donnée n'est aucune de celles données précédemment.

Il existe aussi plusieurs structures pour les modèles, telles que :

```
case "$truc....." in
    [nN] *) echo "Blablabla...";;
    n* | N* ) echo "Bla....";;
```

Et plein d'autres encore...

On mélange tout ça

Pour vous donner une idée précise de ce que peuvent réaliser toutes ces instructions, voici un petit script censé refaire un prompt avec quelques commandes basiques :

```
#!/bin/bash

clear
echo
echo "#################### Script ############################"
echo
echo "############################"
echo -n "LOGIN: "
read login
echo -n "Hôte: "
read hote
echo "############################"
echo
echo "### Pour l'aide tapez help ###"
echo
while [ 1 ]; do                                  # permet une boucle infinie
echo -n ""$login"@"$hote"$ "                     # qui s'arrête avec break
read reps

case $reps in
  help | hlp )
     echo "À propos de TS --> about"
     echo "ls --> liste les fichiers"
     echo "rm --> détruit un fichier (guidé)"
     echo "rmd --> efface un dossier (guidé)"
     echo "noyau --> version du noyau Linux"
     echo "connect --> savoir qui s'est connecté dernièrement";;
  ls )
     ls -la;;
  rm )
     echo -n "Quel fichier voulez-vous effacer : "
```

```
      read eff
      rm -f $eff;;
   rmd | rmdir )
      echo -n "Quel répertoire voulez-vous effacer : "
      read eff
      rm -r $eff;;
   noyau | "uname -r" )
      uname -r;;
   connect )
      last;;
   about | --v | vers )
      echo "Script simple pour l'initiation aux scripts shell";;
   quit | "exit" )
      echo Au revoir!!
      break;;
   * )
     echo "Commande inconnue";;
esac
done
```

Remarque

Comme vous l'avez remarqué, l'indentation a une place importante dans ce programme. En effet, celui-ci est plus lisible et cela évite aussi de faire des erreurs. C'est pourquoi il est préférable de bien structurer le code que vous écrivez.

La structure for

L'instruction `for` exécute ce qui est dans son bloc un nombre de fois prédéfini. Sa syntaxe est la suivante :

```
for variable in valeurs; do
    instructions
done
```

ou le classique:

```
for (( i=$min; i<=$max; i++ )); do
    instructions_avec_i # ou pas
done
```

Comme vous l'aurez sans doute remarqué, on assigne une valeur différente à *variable* à chaque itération. On peut aussi très facilement utiliser des fichiers comme “valeur”.
Rien ne vaut un exemple :

```
#!/bin/sh
for var in *.txt; do
    echo "$var"
```

```
done
```

On peut voir une syntaxe un peu particulière :

```
$(sort *.txt)
```

Ceci sert à indiquer que ce qui est entre les parenthèses est une commande à exécuter.

On peut aussi utiliser cette instruction simplement avec des nombres, cela permet de connaître le nombre d'itérations :

```
#!/bin/sh
for var in 1 2 3 4 5 6 7 8 9; do
    echo $var
done
```

On peut très bien aussi utiliser d'autres types de variables, comme par exemple des chaînes de caractères :

```
#!/bin/sh
for var in Ubuntu Breezy 5.10; do
    echo $var
done
```

Il faut quand même faire attention au fait que *Ubuntu Breezy 5.10* est différent de *“Ubuntu Breezy 5.10”* dans ce cas. En effet, tous les mots placés entre “” sont considérés comme faisant partie de la même chaîne de caractères. Sans les “”, sh considèrera qu'il y a une liste de trois chaînes de caractères.

Les fonctions

Les fonctions sont indispensables pour bien structurer un programme mais aussi pouvoir le simplifier, créer une tâche, la rappeler... Voici la syntaxe générale de 'déclaration' d'une fonction :

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

Cette partie ne fait rien en elle même, elle dit juste que quand on appellera nom_fonction, elle fera instruction. Pour appeler une fonction (qui ne possède pas d'argument, voir plus loin) rien de plus simple :

```
nom_fonction
```

Rien ne vaut un petit exemple :

```
#!/bin/sh

#Definition de ma fonction
```

```
mafonction(){
  echo 'La liste des fichiers de ce répertoire'
  ls -l
}
#fin de la définition de ma fonction

echo 'Vous allez voir la liste des fichiers de ce répertoire:'
mafonction       #appel de ma fonction
```

Comme vous l'avez sans doute remarqué, quand on appelle la fonction, on exécute simplement ce qu'on lui a défini au début, dans notre exemple, echo... et ls -l, on peut donc faire exécuter n'importe quoi à une fonction.

Les fonctions peuvent être définies n'importe où dans le code du moment qu'elles sont définies avant d'être utilisées. Même si en bash les variables sont globales, il est possible de les déclarer comme locales au sein d'une fonction en la précédant du mot clé local: local ma_fonction .

Exemple: un sleep interactif :

```
#!/bin/bash
info(){
        echo -e "$1\nBye"
        exit
}
test -z "$1" && info "requiert 1 argument pour le temps d'attente..." ||
PRINT=$(($1*500))
test -z $(echo "$1" | grep -e "^[0-9]*$") && info "'$1' est un mauvais
argument"
test $1 -gt 0 || info "Je ne prends que les entiers > 0"
print_until_sleep(){
         local COUNT=0
         while [ -d /proc/$1 ]; do

                  test $(($COUNT%$2)) -eq 0 && echo -n "*"
                  COUNT=$(($COUNT+1))
         done
}
sleep $1 & print_until_sleep $! $PRINT
echo -e "\nBye"
```

Extraire des sous-chaînes

Pour extraire une chaîne d'une chaîne on utilise : **${ chaîne : position : nombre de caractères }** (n'oubliez pas le **:** qui sépare les “paramètres”).

Dans la partie chaîne pour faire référence à une variable **on ne met pas de $** ! Tandis que dans les autres options le $ est nécessaire (sauf si vous n'utilisez pas de variable). Il y a de quoi s’emmêler les pinceaux. Si vous n'avez pas compris (ce n'est pas étonnant), les exemples de cette partie vous aideront beaucoup.

Par exemple pour savoir ce que l'on aime manger en fonction de sa langue (vous êtes alors vraiment ultra geek  !) :

```
#!/bin/bash
#favoritefood
if [ ${LANG:0:2} = "fr" ]; then
    echo "Vous aimez les moules frites !"
elif [ ${LANG:0:2} = "en" ]; then
    echo "You love the... pudding !"
elif [ ${LANG:0:2} = "es" ]; then
    echo "Te gusta el jamón !"
else
    echo ":'-("
fi
#Noter que $LANG n'a pas le préfixe '$'. ${$LANG:0:2} ne fonctionne pas !
```

puis :

```
$ ./favoritefood
Vous aimez les moules frites !
$ env LANG=en ./favoritefood
You love the... pudding !
$ env LANG=es ./favoritefood
Te gusta el jamón !
$ env LANG=it ./favoritefood
:'-(
```

Ce code illustre un moyen de faire des scripts multilingues .

Une variante permet de tronquer uniquement le début de la chaîne. C'est **${ chaîne : nombre de caractères}** .
Le tout peut s'illustrer par un (vraiment) petit exemple :

```
#!/bin/bash
#truncbegin <chaîne> <nombre>
echo ${1:$2}
#Noter bien que echo ${1:2} tronquerait les 2 premiers caractères (et non le nombre indiqué par le 2e paramètre).
```

puis :

```
$ ./truncbegin "Hello world !" 5
world !
```

La couleur

Qui n’a jamais voulu faire un script avec des couleurs pour pouvoir différencier les titres des paramètres et les paramètres de leur valeur par exemple...

Présentation de la syntaxe

Comme toute commande sous Linux, il faut utiliser une syntaxe par défaut et y passer quelques paramètres. Pour les couleurs au sein de scripts shell, c'est le même principe.

```
echo -e '\033[A;B;Cm toto \033[0m'
```

Dans la commande passée ci-dessus, nous pouvons constater qu'il y a 3 paramètres présents: A, B et C.
A : correspond à un effet affecté au texte affiché B : correspond à la couleur du texte C : identifie la couleur du fond du texte affiché

Et enfin on termine notre affichage avec « \033[0m », qui spécifie au terminal de revenir aux couleurs définies par défaut.

Présentation des différentes valeurs Effet

Nous allons commencer par les différents effets possibles :

Code	Effet
0	Normal
1	**Gras**
21	Non-gras
2	Sombre
22	Non-sombre
3	*Italique*
23	Non-italique
4	Souligné
24	Non-souligné
5	Clignotant
25	Non-clignotant
7	Inversé
27	Non-inversé
8	Invisible
28	Non-invisible
9	~~Barré~~
29	Non-barré

Présentation des différentes valeurs des couleurs

Maintenant que nous avons présenté les différents effets possibles d'attribuer à du texte, nous allons nous attaquer aux couleurs.
Chaque couleur a 2 valeurs, la première utilisée pour la couleur du texte, et la seconde pour la couleur du fond.

Couleur	Couleur texte	Couleur fond
Noir	30	40
Rouge	31	41
Vert	32	42
Jaune	33	43
Bleu	34	44
Magenta	35	45
Cyan	36	46
Blanc	37	47

Exemple

```
echo -e '\033[1;30;47m toto \033[0;32m est sur \033[1;33m un bateau \033[0m'
```

Exemples et exercices

Comme indiqué dans la section liens de cette page, de très bon exemples et exercices illustrent le cours disponible sur cette page : Guide avancé d'écriture des scripts Bash - Une exploration en profondeur de l'art de la programmation shell

Aux structures décrites ci-dessus, il est nécessaire, pour réaliser des scripts poussés, de connaître les commandes shell les plus usitées.
Vous en trouverez une présentation sur cette autre page du wiki : initiation_au_shell .

La programmation de script shell étant ouverte à tous, cela permet de bénéficier de nombreux scripts pour des applications très variées ; cependant, **la plupart sont proposés sans aucune garantie**.
Vous pourrez trouver une liste de scripts pouvant servir d'exemples sur la page scripts utiles du wiki.

Une fois vos armes faites, proposez vos contributions sur le topic du forum [VOS SCRIPTS UTILES] et rajoutez un lien dans la page du wiki ci-dessus.

L'art d'écrire un script

- Des vérifications approfondies doivent être effectuées sur TOUTES les commandes utilisées.
- Des commentaires détaillés doivent apparaître lors de chaque étape. De même, chaque étape doit être suivie d'un “echo <voici ce que je fais>” (particulièrement utile notamment lors du débogage).
- Lors d'une mise à jour, un fil de discussion doit être précisé pour tracer les bugs éventuels.
- Avertir les utilisateurs des dégâts que peuvent causer les commandes utilisées. (Ces deux dernières remarques ne concernent bien sûr que les scripts que l'on souhaite diffuser.)
- Commencer par :

```
#!/bin/bash
# Version du script
```

- Créer des fonctions pour des actions précises :

```
nom_de_la_fonction()
{
...

}
```

- Utiliser des chemins absolus pour les dossiers et des chemins relatifs pour les noms de fichiers :

```
$CHEMIN_DU_DOSSIER/$NOM_DU_FICHIER
```

- Utiliser les entrées de commandes pour les fonctions :

```
nom_de_la_fonction $1 $2 $3 ....
```

- Si votre script doit s'arrêter à cause d'une erreur, d'une variable qui ne correspond pas à vos attentes utiliser des numéros exit différents :

```
exit 100;
exit 101;
exit 102;
....
```

Ça permettra d'identifier d'où vient l'erreur.

- Utiliser le tableau ${PIPESTATUS[@]} pour récupérer les états des autres commandes.
- On peut écrire une fonction d'erreur du type :

```
erreur()
{
    tab=( ${PIPESTATUS[@]} )

    for (( i=0; i < ${#tab[@]}; i++ )); do ((i+=i)); done

    if ((i > 0)); then
        zenity --error --title="Une erreur est survenue" --text="Une
erreur est survenue "
        exit 100
    fi
}
```

ainsi après chaque commande vous pouvez donner des codes d'exécutions différents.

Astuce : le plus important dans tout programme est l'algorithme utilisé.

Exemple : supposons que vous ayez une base de données, avec 3 catégories d'enregistrements possibles : éléphant bleu, éléphant blanc, éléphant rose ayant chacun 30 individus. Votre script doit compter le nombre d'éléphants bleus et blancs. Deux possibilités s'offrent à vous :

- calculer le nombre d'éléphants bleus + éléphants blancs

ou

- calculer le nombre total d'éléphants - nombre d'éléphants roses

Quel algorithme choisissez-vous ?

Résultat : le premier car dans le deuxième il faut d'abord calculer le nombre total d'éléphants, donc un calcul en plus 😛 .

Liens

- (fr) https://marcg.developpez.com/ksh/ : Pour ceux qui souhaitent aller plus loin dans la conception de script shell.
- (fr) Guide avancé d'écriture des scripts Bash : Un très bon tutoriel concernant la réalisation du script shell. C'est l'un des plus complets et les mieux détaillés disponibles en français. Il contient également des exemples de script complets, une carte de référence (variables, tests...). Ce site est un site qui vaut réellement le détour pour tous ceux qui cherchent à créer des scripts complets en utilisant au mieux les performances du shell.
- (fr) https://openclassrooms.com/courses/reprenez-le-controle-a-l-aide-de-linux : Un tutoriel très complet pour linux qui comporte quelques parties sur la réalisation de scripts bash.
- (en) Bash parameters and parameter expansions. En anglais mais contient de nombreux exemples concernant la gestion et l'analyse des paramètres.
- (fr) Introduction à Bash
- (fr) http://www.scotchlinux.tuxfamily.org/ exemples de scripts bash, quelques trucs utiles (fonctions, fonctions comme paramètres...)
- (en) https://www.shellcheck.net/ Permet de corriger la syntaxe du script (parenthèse oubliée, graphie incorrecte d'une commande, un “if” sans son “fi”, un “while sans son “do” ou son “done”, etc...).

Contributeurs: Gapz, Gloubiboulga ,sparky et deax_one