Système utilisateur
.:: Système UNIX ::

semaine 6 – les scripts shell, la suite

La boucle for

=> Quelques exemples divers :

- for i in ours éléphants castors ; do echo $i ; done
- for i in * ; do echo $i ; done
- var='a b c d e '; for i in $var; do echo $i ; done
- for i in `ls /bin`; do echo $i ; done
- for i in `seq 1 10`; do echo $i; done
- IFS=':'; var='ab:b c:d'; for i in $var; do echo $i ; done
  - => IFS indique la liste des séparateurs
  - => i prend donc pour valeurs : « ab » puis « b c » puis « d »

break : Interrompre une boucle

=> Permet de quitter une boucle sans en attendre la fin normale.

=> S'applique à for, while, until, select

=> ex : analyser les arguments et quitter en cas d'argument non reconnu :

```
while [ $# -gt 0 ] ; do
      if [ $1 == "-v" ] ; then
         echo "verbose"
       elif [ $1 == "-m" ] ; then
          echo "mode m"
      else
          echo "erreur de syntaxe, option inconnue !"
          break;
      fi
      shift
done
```

continue : aller directement à la suite

=> Permet de passer à l'itération suivante sans exécuter le suite

=> S'applique à for, while, until, select

=> ex : afficher les nombres pairs passés en arguements

```
for i ; do
        reste= $(( $i % 2 ))
        if [ reste -ne 0 ] ; then
                continue
        fi
        echo $i
done
```

Gestion des cas

=> l'instruction **case** permet de remplacer certains enchaînements de if, elif, elif ...

=> Syntaxe :

```
case variable in
    'val1')
            action
            ;;
    'val2' | 'val3')
            action
            ;;
    '*')
            action par défaut
            ;;
esac
```

Gestion des cas

=> Exemple : gestion des arguments d'un programme

```
while [ $# -ne 0 ] ; do
    case $1 in
        '-v'|'-V')
            echo "mode verbose"
            ;;
        '-m')
            echo "option m avec pour valeur : $2" ;  shift
            ;;
        '-n')
            echo "option n avec pour valeur : $2" ; shift
            ;;
        '*')
            echo "option inconnue !"
            ;;
    esac
    shift
done
```

creation de menus : select

=> Permet de créer un menu utilisateur en mode texte et de récupérer sa réponse ; celle-ci entraîne l'exécution d'un traitement.

=> syntaxe : **select** *variable* [**in** *liste*] ; **do** *action* **; done**

=> exemple : un dé numérique

```
select choix in 'Lancer le dé' 'Quitter'; do
    case $choix in
        'Lancer le dé') val=$(( 1 +  $RANDOM % 6 )) ; echo $val ;;
        'Quitter') break;;
    esac
done
```

=> Résultat :

```
1) Lancer le dé
2) Quitter
#? 1
2
#? 1
6
#? 2
```

=> select est un boucle !!

TD

=> Commande menu1 : en utilisant la structure while puis la structure select, écrire deux scripts qui, tant que l'utilisateur n'a pas tapé 4, affiche le menu et exécute l'opération associée.

Le menu étant :

1) Afficher la date (date)
2) Afficher le nombre de personnes connectées (who)
3) Afficher la liste des processus (ps)
4) Quitter

TD

=> menu1 avec **while**

```
choix=0
while [ $choix -ne 4 ] ; do
        echo "1)Afficher la date (date)"
        echo "2)Afficher le nombre de personnes connectées (who)"
        echo "3)Afficher la liste des processus (ps)"
        echo "4)Quitter"
        read choix
        case $choix in
                '1') date ;;
                '2') who ;;
                '3') ps ;;
        esac
done
```

TD

=> menu1 avec **select**

```
select choix in 'Afficher la date (date)' 'Afficher le nombre de personnes connectées (who)' 'Afficher la liste des processus (ps)' ; do
        case $choix in
                '1') date ;;
                '2') who ;;
                '3') ps ;;
                '*') break;;
        esac
done
```

Les fonctions

=> Exemple

```
maFonction() {
        echo -n 'Voici mes arguments : '
        while [ $# -gt 1 ] ; do
                echo -n $1'#'
                shift
        done
        echo $1
        return 0
}

> maFonction 1 2 3 4
Voici mes arguments : 1#2#3#4
```

Les fonctions

=> Fonctionne comme un script shell indépendant
=> Reçoit ses paramètres de la même façon
=> Retourne un entier

=> Attention : pas de prototype ni vérification des arguments !
=> Le code de retour d'une fonction est lu dans $?
=> L'appel se fait comme celui d'une commande shell

=> Une fonction connaît toutes les variables de la fonction appelante !
    -> ne pas les utiliser pour des raison de portabilité
=> Hérite des E/S du père
=> Ne crée pas de processus sauf si utilisée avec un |

Les fonctions

=> Comme dans un shell, toutes les variables sont globales à tout le shell, elles sont visibles par toutes les fonctions !! danger !!

-> risque de redéfinition

-> mauvaise réutilisabilité des fonctions

=> Préférez l'utilisation de variables locales :

local *maVariable=valeur*

-> Définissable dans une fonction

-> Prioritaire sur les variables globales

=> Toutefois, les variables locale restent visibles dans toutes les sous-fonctions appelées !! danger !!

Les fonctions

=> Exemple :

```
fonctionFille() {
    echo Dans fille: $varMere $varLocMere
    local varLocFille="varLocFille";
    varFille="varFille";
}
fonctionMere() {
    local varLocMere="varLocMere";
    varMere="varMere";
    fonctionFille;
    echo Dans mere:$varLocFille $varFille;
}
fonctionMere
```

=> Resultat

```
Dans fille: varMere varLocMere
Dans mere: __ varFille
```

Les fonctions

=> Retourner une valeur

-> Les fonctions retournent une valeur numérique entre 0 et 127

-> Pour retourner d'autre données : texte, entiers ... utiliser la sortie standard

-> exemple : fonction hazard retourne un nombre entre 0 et *n*

```
hazard() {
        echo $(( 1 + $RANDOM % $1 ))
        return 0
}
```

-> récupération de la donnée :

ret=`hazard *1000*`

TD fonctions

=> écrire la fonction min qui retourne le minimum des arguments passés

=> écrire la fonction getNumLignes qui retourne le nombre de ligne du fichier f passé en paramètre ; retourne 0 si tout va bien, 1 si le fichier n'existe pas ou n'est pas lisible.

=> écrire la fonction extractLigne qui retourne la ligne « l » du fichier « f » les arguments étant passé dans cet ordre ; retourne 0 si tout s'est bien passé, 1 si la ligne n'existe pas dans le fichier

=> écrire la commande 'cat' utilisant les fonctions ci-dessus

TD fonctions

=> écrire la fonction min qui retourne le minimum des arguments passés

```
min() {
    if [ $# -ge 1 ] ; then
        mini=$1 ; shift
        for i ; do
            if [ $mini -gt $i ] ; then
                mini=$i
            fi
        done
        echo $mini
        ret=0
    else
        ret=1
    fi
    return $ret
}
```

TD fonctions

=> écrire la fonction getNumLignes qui retourne le nombre de ligne du fichier f passé en paramètre ; retourne 0 si tout va bien, 1 si le fichier n'existe pas ou n'est pas lisible.

```
getNumLignes() {
    if [ $# -eq 1 ] && [ -f $1 ] && [ -r $1 ] ; then
        echo `wc -l $1 | cut -d " " -f 1`
        return 0
    else
        return 1
    fi
}
```

TD fonctions

=> écrire la fonction extractLigne qui retourne la ligne « l » du fichier « f » les arguments étant passé dans cet ordre ; retourne 0 si tout s'est bien passé, 1 si la ligne n'existe pas dans le fichier

```
extractLigne() {
    if [ $# -eq 2 ] ; then
        local maxl=`getNumLigne $2`
        if [ $? -eq 0] && [ $1 -gt 0 ] && [ $1 -le $maxl ] ; then
            echo `head -n $1 | tail -n 1`
            return 0
        else
            return 1
        fi
    fi
    return 1
}
```

TD fonctions

=> écrire la commande 'cat' utilisant les fonctions ci-dessus

```
#!/bin/bash
# fonction getNumLigne
getNumLigne() { ... }
# fonction extractLigne
extractLigne() { ... }

for file ; do
        c=1
        ligne=`extractLigne $c $file`
        while [ $? -eq 0  ]; do
                echo $ligne
                c=$(( $c + 1 ))
                ligne=`extractLigne $c $file`
        done
done
```

TD

=> Commande waitfor : scrute les processus et se termine lorsque l'utilisateur ident se connecte (si l'option utilisée est -user) ou lorsque le process ident est détecté (option -process)

=> utilisation de deux fonctions : detectProcess et detectUser

TD

```
#!/bin/bash

detectUser() {
        # retourne 0 lorsqu'un process de l'utilisateur $1 est présent
        ps -u $1 > /dev/null
        return $?
}

detectProcess() {
        # retourne 0 lorsqu'un process $1 est présent
        ps aux | tr -s " " | cut -d " " -f 11 | grep $1
        return $?
}

...
```

TD

...

```
commande='detectUser'
if [ $# -eq 2 ] ; then
  if  [ $1 == '-process' ] ; then
      commande='detectProcess'
  fi
   shift
fi
while [ 1 -eq 1 ] ; do
      if eval "$commande $1" ; then
        break
      fi
      sleep 1
done
```