Module 9 : Scripts et automatisation de tâches

Table des matières

Qu'est-ce qu'un "shell" ?

Un "shell" est un programme permettant d'isoler l'utilisateur des commandes internes du système d'exploitation. Nous donnerons une description des principales commandes qui peuvent être utilisées dans le “c-shell” et dans le “bash”.

Variables définies par l'usager

On peut définir une variable de la façon suivante:

Pour une chaine de caractères → Variable=”Voici le contenu de la chaine”
Pour une valeur numérique → Variable=5 Variable=5.2

Affichage à l'écran

L'affichage à l'écran se fait au moyen de la commande « echo ». Celle-ci affiche le contenu du texte qui est borné par des guillemets.

Exemple :

```
Variable=5
Nom="Stef"

echo $Variable          j'obtiens  5
echo $Nom               J'obtiens Stef
```

Je peux faire des combinaisons comme :

```
echo "Votre nom est $Nom"                  J'Obtiens  Votre nom est Stef
echo " La valeur de variable : $Variable"    J'obtiens   La valeur de variable : 5
```

Si je veux faire afficher le signe de dollar, je dois faire précéder ce dernier du caractère d'échappement « backslash » (\). Par exemple, je veux faire afficher la phrase :

La valeur de la variable $Variable vaut 5.

```
echo "La valeur de la variable \$Variable vaut $Variable"
```

Conclusion : Ne jamais oublier le signe de dollar devant la variable lorsqu'on veut afficher ou travailler avec son contenu.

Lecture d'une entrée au clavier

Les données entrées au clavier peuvent être lues par le biais de la commande "**read**" pour le **bash shell** et du caractère "$<" pour le **c-shell**.

Supposons que vous voulez demander à l'usager de confirmer le remplacement d'un fichier. Vous pourrez alors le faire de la façon suivante dans les deux shells suivants:

c-shell	bash
#!/bin/csh echo -n "Voulez-vous remplacer le fichier ? (o/n): " set remplace = $< # ????????????? Quelle instruction va ici ?	#!/bin/bash fichier=allo.txt echo -n "Voulez-vous remplacer le fichier ? (o/n): " **read** remplace if [$remplace = o]; then rm $fichier fi

Les variables pré-définies

Il existe un moyen rapide de connaître le nombre de variables et le contenu des variables qui sont passées en paramètre à la ligne de commande. Comme vous avez pu le constater, les paramètres sont nommés $0 à $9 dans le système d'exploitation DOS. En Unix, les paramètres passées à la ligne de commande ont aussi leurs syntaxes et ont des noms différents dépendemment que vous travaillez dans un shell ou un autre.

Variables pré-définies du "BASH" (Bourne Again Shell)

$0 à $9	Les variables qui contiennent les paramètres passés à la ligne de commande.	$0 : Le nom de la commande $1 : Le premier paramètre $2 : Le deuxième paramètre etc...
$*	Donne la liste des paramètres de la commande.	echo $* Affiche la liste des paramètre $1, $2, $3, etc...
$?	Donne la valeur du code de retour de la dernière commande exécutée.	
$#	Donne le nombre de paramètre de la commande appelée (sauf $0).	ls -al fichier.c echo $# Affiche 2

Exemple:

Supposons le fichier "essai" qui contient les lignes suivantes:

```
echo $#
echo $*
```

Supposons que l'on tape ensuite la commande suivante:

essai Bonjour Allo Salut 2 4

Le résultat sera : 5

echo $# donne le nombre de paramètre excluant la commande elle-même.

Bonjour Allo Salut 2 4

Variables pré-définies du « C-Shell »

$argv	Un tableau qui contient les paramètres passés à la ligne de commande. Cette variable est accessible sous la forme d'un tableau de variable.	$argv[1] : Le premier paramètre $argv[2] : Le deuxième paramètre etc...
$#argv	Donne le nombre de paramètre(s) passé(s) à la commande.	echo $#argv Affiche le nombre de paramètre présent dans la commande.

Exemple:

Soit le fichier qui contient les lignes suivantes:

```
#!/bin/csh
#Fichier script en c-shell qui affiche les parametres passes a la commande
echo "Il y a $#argv parametre(s) a la commande. Les voici: "
echo -n "le premier parametre: $argv[1]"
echo -n "Le deuxieme parametre: $argv[2]"
```

On sauvegarde le fichier sous le nom essai2 et on tape ensuite au prompt:

essai2 Allo Bonjour

Le script affiche:

```
Il y a 2 parametre(s) a la commande. Les voici:
Le premier parametre: Allo
Le deuxieme parametre: Bonjour
```

Les tableaux

Le bash gère également les variables « tableaux ». Ce sont des variables qui contiennent plusieurs cases, comme un tableau. Vous en aurez probablement besoin un jour ; voyons comment cela fonctionne.

Pour définir un tableau, on peut faire comme ceci :

```
tableau=('valeur0' 'valeur1' 'valeur2')
```

Cela crée une variable tableau qui contient trois valeurs (valeur0, valeur1, valeur2).

Pour accéder à une case du tableau, il faut utiliser la syntaxe suivante : ${tableau[2]}

```
${tableau[2]}
```

… ceci affichera le contenu de la case n° 2 (donc valeur2).

Les cases sont numérotées à partir de 0 ! La première case a donc le numéro 0.
Notez par ailleurs que pour afficher le contenu d'une case du tableau, vous devez entourer votre variable d'accolades comme je l'ai fait pour ${tableau[2]}.

Vous pouvez aussi définir manuellement le contenu d'une case : tableau[2]="valeur2"

```
tableau[2]='valeur2'
```

Essayons tout ceci dans un script :

```
#!/bin/bash

tableau=('valeur0' 'valeur1' 'valeur2')
tableau[5]='valeur5'
echo ${tableau[1]}
```

À votre avis, que va afficher ce script ?

Comme vous pouvez le constater, le tableau peut avoir autant de cases que vous le désirez. La numérotation n'a pas besoin d'être continue, vous pouvez sauter des cases sans aucun problème (la preuve, il n'y a pas de case n° 3 ni de case n° 4 dans mon script précédent).

Vous pouvez afficher l'ensemble du contenu du tableau d'un seul coup en utilisant ${tableau[*]} :

```
#!/bin/bash

tableau=('valeur0' 'valeur1' 'valeur2')
tableau[5]='valeur5'
echo ${tableau[*]}
```

Résultat:

```
valeur0 valeur1 valeur2 valeur5
```

Opérations arithmétiques

Il existe plusieurs façons d'effectuer des opérations arithmétiques avec la langage Bash. Vous pouvez effectuer des opérations entières en englobant l'opération entre les caractères "$((" et "))". Le bash évaluera alors l'expression. Voici des exemples:

```
echo $(( 100 / 3 ))
33
myvar=56
echo $(( $myvar + 12 ))
68
echo $(( $myvar - $myvar ))
0
myvar=$(( $myvar + 1 ))
echo $myvar
57
```

On peut aussi utiliser l'instruction “expr” qui permet d'effectuer une opération arithmétique. Voici un exemple:

expr 5 + 6 affiche 11

var=10

expr $var + 5 affiche 15

Par contre, pour incrémenter ou décrémenter une variable, on n'utilise pas le signe de dollar ($) devant la variable, on doit faire:

décrémenter	incrémenter
((var--))	((var++))

dans une boucle:

```
i=0
while [ $i  -lt  5 ] ; do
  (( i++ ))
```

```
done
```

Étape dans la création d'un fichier script

Étape 1:

- Éditez le script avec votre éditeur de texte préféré. Tapez les lignes ci-dessous dans votre fichier.

```
#!/bin/bash
#Ce script affiche la chaine « Mon premier script bash » ainsi que les paramètres passés
# à la ligne de commande à l'aide des variables pré-définies.
#Script en bash
#
echo  "Il y a $# arguments sur la ligne de commande"

if  [ $# -gt 0 ];  then     # Le nombre d'argument est donne par: $#
   echo  "Le premier argument: $1"
fi

if [ ($# -gt 1 ]; then
   echo "Le deuxieme argument: $2"
fi
```

Sauvegardez le fichier sous un nom qui deviendra ainsi le nom de la nouvelle commande. Pour cet exemple, sauvegardez sous le nom "***script1***"

Étape 2:

- Ajoutez l'attribut d'exécution à votre fichier script. (Cette opération ne se fait qu'une seule fois)

chmod u+x script1 ou chmod 750 script1

Étape 3:

- Exécutez le script en tapant le nom du fichier et ses paramètres s'il y a lieu:

./script1 Salut Allo

L'exécution du script précédent avec la ligne de commande suivante:

script1 bonjour allo

donne

Il y a 2 arguments sur la ligne de commande
Le premier argument: bonjour
Le deuxième argument: allo

Structure de contrôle

Dans cette section, il sera question des structures de contrôle principales que l'ont retrouve dans les scripts. Ces structures de contrôles permettent d'ajouter des conditions aux scripts et de tester l'existence de certains paramètres pour ensuite effectuer certaines actions.

Vous allez étudier surtout 3 structures de contrôle. Il s'agit de:

- **if**
- **while**
- **for**

Vous retrouverez la syntaxe de ces commandes autant pour le "c-shell" que pour le "bash" shell.

La structure IF

	Syntaxe en C-shell	Syntaxe en bash

Structure IF simple	**if** (condition(s)) **then**
	Liste de commande(s)
	else
	Liste de commande(s)
	endif
		if [condition(s)]; **then**
		Liste de commande(s)
		else
		Liste de commande(s)
		fi

Structure IF imbriquée	**if** (condition(s)) **then**
	Liste de commande(s)
	else
	if (condition(s)) **then**
	Liste de commande(s)
	else
	Liste de commande(s)
	endif
	endif
		if [condition(s)]; then
		Liste de commande(s)
		else
		if [condition(s)]; then
		Liste de commande(s)
		else
		Liste de commande(s)
		fi
		fi

Exemple:

Le script suivant permet de simuler 3 commandes en unes. Voici les 3 possibilités que cette commande doit permettre:

1. list — Affiche le contenu du répertoire courant
2. list -a fichier en 2e — Affiche le contenu du fichier passé paramètre.

3. list -d fichier — Détruit le fichier passé en 2e paramètre.

c-shell	*bash*

if ($#argv == 0) then
  ls -al
else
  if (“$argv[1]” == “-a”) then
  cat $argv[2]
  else
  rm $argv[2]
  endif
endif
	if [$# = 0]; then
	  ls -al
	else
	  if [$1 = -a]; then
	  cat $2
	  else
	  rm $2
	  fi
	fi

ET logique, OU logique

On doit utiliser l'opérateur -a pour un ET alors que l'opérateur -o permet le OU.

Exemple:
Vérifier si la valeur de la variable se trouve entre 10 et 20.

If [$Var -gt 10 -a $Var -le 20] ; then

Exemple 2:
Vérifier si la réponse d'une question est O ou o
if [$Reponse = “O” -o $Reponse = “o”] ; then

Test de fichiers

Il est souvent pratique de pouvoir tester les attributs d’un fichier pour savoir si ce dernier est exécutable, existe ou n’existe pas, etc...

Tests	*Résultat*
-e fichier	Teste l’existence d’un fichier.
-d répertoire	Teste l’existence d’un répertoire.
-r fichier	Teste si le fichier peut être lu.
-w fichier	Teste si le fichier peut être modifié.
-x fichier	Teste si le fichier peut être exécuté.
-c fichier	Teste si le fichier est un fichier spécial de type “caractère”.
-b fichier	Teste si el fichier est un fichier spécial de type “bloc”.
-s fichier	Teste si le fichier existe et est de taille non nulle.

Exemple:

On veut faire afficher la phrase “C’est un repertoire” si le fichier passé en paramètre est un répertoire.

c-shell

```
if (-d $argv[1] ) then
  echo “$argv[1] est repertoire”
else
  echo “$argv[1] n’est pas un repertoire”
endif
```

bash

```
if [ -d $1 ]; then
   echo “$1 est un repertoire”
else
   echo “$1 n’est pas un repertoire”
fi
```

Opérateurs pour tester les chaines de caractères

Les différents types de tests sur des chaînes

Condition	Signification
$chaine1 = $chaine2	Vérifie si les deux chaînes sont identiques. Notez que bash est sensible à la casse : « b » est donc différent de « B ». Il est aussi possible d'écrire « == » pour les habitués du langage C.
$chaine1 != $chaine2	Vérifie si les deux chaînes sont différentes.
-z $chaine	Vérifie si la chaîne est vide.
-n $chaine	Vérifie si la chaîne est non vide.

Opérateurs pour tester des valeurs numériques

Test	Opérateur bash
<	-lt
>	-gt
<=	-le
>=	-ge
!=	-ne
Égalité ==	-eq

La structure SWITCH-CASE

Cette structure, tout comme en langage C, permet de sélectionner une branche d'action selon la valeur d'une variable. Voici la syntaxe:

c-shell	bash

switch ($variable)
  case valeur**:**
  liste de commande(s)
  breaksw
  case valeur2**:**
  liste de commande(s)
  breaksw
  .
  .
  .
  default:
  liste de commande(s)
endsw
	case $variable **in**
	  valeur**)**
	  liste de commande(s)
	  ;; #breaksw en c-shell
	  valeur2**)**
	  liste de commande(s)
	  ;;
	  .
	  .
	  .
	  ***)**
	  Liste de commande(s) #valeur défaut
	esac

Exemple :

L'exemple suivant montre l'utilisation de ce type de structure: il s'agit d'une commande qui attend au plus 1 paramètre et dans le cas où:
il n'y a pas de paramètre, affiche le répertoire courant (pwd)
un paramètre, affiche le contenu s'il s'agit d'un fichier et affiche le contenu du répertoire si c'est un répertoire.

c-shell	*bash*

#!/bin/csh
switch ($#argv)
  case 0:
  pwd
  breaksw
  case 1:
  if (-f $argv[1]) then
  cat $argv[1]
  else if (-d $argv[1]) then
  ls -al $argv[1]
  else
  echo "Erreur sur le parametre"
  endif
 	
  default:
  echo "$argv[1] n'est ni un rep ni un fichier"
endsw
	#!/bin/bash
	case $# in
	  0)
	  pwd
	  ;;
	  1)
	  if [-f $1]; then
	  cat $1
	  else
	  if [-d $1]; then
	  ls -al $1
	  else
	  echo "erreur sur parametre"
	  fi
	  fi
	  ;;
	  *) echo "$1 n'est ni un rep ni un fichier"
	  ;;
	esac

Structure de répétition

La structure while

Cette instruction permet de répéter un certain nombre de fois, dicté par une condtion, la liste de commandes qui se retrouve à l'intérieur de la boucle.

Syntaxe:

c-shell	bash
while (condition) liste de commande(s) **end**	**while** [condition(s)]; do liste de commande(s) **done**

Exemple:

c-shell	bash
#!/bin/csh set v=5 while ($v > 1) echo "Toujours plus grand que 1 : v = $v" @ v-- #l'espace entre le @ et la variable est #important end	#!/bin/bash v=5 while [$v -gt 1]; do echo "Toujours plus grand que 1 : v = $v" v=$((v – 1)) done

Exemple 2: condition ET (en bash)

```
#!/bin/bash

Var=5

while [ $Var -lt  10  -a  $Var  -ge  5 ] ;  do
  …
done
```

La structure for

Syntaxe:

c-shell	bash

foreach variable (liste de valeur(s))
  liste de commande(s)
end
	for variable **in** chaine chaine ...
	  do liste de commande(s)
	done

Exemple 1 :

L'exemple suivant permet de trouver le fichier donné en paramètre et qui porte l'extension .o, .cpp ou .c et afficher si le fichier existe ou non.

c-shell	bash

#!/bin/csh	#!/bin/bash
	for i in .o .cpp
	  do if [-e 1i]; then
	  echo "Fichier existe"
	  else
	  echo "Fichier n'existe pas!"
	  fi
	done

Les alias

Il s'agit d'un mécanisme qui permet l'écriture de commandes par la définition d'abréviations et évitant de taper des commandes qui pourraient être longue. En ce sens, ce mécanisme ressemble à la notion de macro-instructions dans les langages de programmation. Autrement dit, l'alias que vous créer devient ainsi une sorte de synonyme pour la commande que vous rattachez à cet alias.

Habituellement, les alias permanents seront situés dans le fichier de configuration « .bashrc » dans le répertoire maison de l'usager.

Syntaxe:

c-shell	bash
alias nom_de_l'alias 'commande'	alias nom_de_l'alias = 'commande'

Exemple:
On veut créer des alias pour les commandes suivantes:
find -iname que l'on nommera "f"
rm -r que l'on nommera "deltree"

c-shell	bash
alias f 'find -iname'	alias f='find -iname'
alias deltree 'rm -r'	alias deltree='rm -r'

Exemple #2:

Il arrive souvent que l'on veuille faire afficher le répertoire courant dans le "**prompt**" sur la ligne de commande. Voici les alias à faire:

c-shell	bash
alias cd 'cd \!*;set prompt = "$cwd" '	alias cd = 'cd \!*;set prompt=`pwd`'

Les points-virgules permettent de séparer deux commandes de suite sur une même ligne.
La variable "**prompt**" est un nom réservé pour l'attribution du prompt.
La suite de caractères "\!*" permet d'utiliser un paramètre lorsque cette commande sera tapée. Comme vous le savez, la commande cd a besoin d'un nom de répertoire pour se déplacer dans ce répertoire. Or, les caractères \!* seront remplacés par le nom du répertoire qui sera tapé sur la ligne de commande.

Exemple: cd travail_pratique grâce à l'alias la commande précédente devient:

cd \!*;set prompt="$cwd" ce qui se traduit par la séquence des deux commandes:

cd travail_pratique
set prompt="$cwd"

Le paramètre “travail_pratique” prend directement la place des caractères \!* dans la commande.

Les fonctions

Comme les « vrais » langages de programmation, Bash supporte les fonctions bien qu'il s'agisse d'une implémentation quelque peu limitée. Une fonction est une sous-routine, un bloc de code qui implémente un ensemble d'opérations, une « boîte noire » qui réalise une tâche spécifiée. Quand il y a un code répétitif, lorsqu'une tâche se répète avec quelques légères variations dans la procédure, alors utilisez une fonction.

Syntaxe générale d'une fonction

```
function nom_fonction {
  commande...
}
```

ou

```
nom_fonction ()
{
  commande...
}
```

Cette deuxième forme va réjouir le coeur des programmeurs C (elle est aussi plus portable).

Comme en C, l'accolade ouvrante de la fonction peut apparaître de manière optionnelle sur la deuxième ligne.

Les fonctions sont appelées, *lancées*, simplement en invoquant leur nom. *Un appel de fonction équivaut à une commande.*

Exemple 1: Fonctions simples

```
#!/bin/bash
# Ce script utilise une fonction qui permet de faire attendre 1 seconde

Attendre()
{
  sleep 1
}

Echo "Décompte avant le lancement"

for (( i=0; i<10; i++))
do
  echo "$i..."
  Attendre
done
```

La définition de la fonction doit précéder son premier appel. Il n'existe pas de méthode pour « déclarer » la fonction, comme en C par exemple.

Exemple 2: Fonction utilisée avant sa déclaration

```
#!/bin/bash
# Utilisation de la fonction f1 avant de l'avoir définie

f1

f1()
{
  Echo "Je suis dans la fonction f1"
}
```

L'exemple précédent montre qu'en bash, on ne peut utiliser la fonction avant de l'avoir définie

Passage de paramètres aux fonctions

Les fonctions peuvent récupérer des arguments qui leur sont passés et renvoyer un code de sortie au script pour utilisation ultérieure.

Les arguments d'une fonction sont référencés dans son corps de la même manière que les arguments d'un programme shell le sont : **$1** référence le premier argument, **$2** le deuxième, etc., **$#** le nombre d'arguments passés lors de l'appel de la fonction.

Le paramètre spécial **$0** n'est pas modifié : il contient le nom du programme shell.

Pour éviter toute confusion avec les paramètres de position qui seraient éventuellement initialisés dans le code appelant la fonction, la valeur de ces derniers est sauvegardée avant l'appel à la fonction puis restituée après exécution de la fonction.

Le script suivant illustre les propos :

```
 #!/bin/bash

f()
{
    echo " --- Dans f : \$0 : $0"
    echo " --- Dans f : \$# : $#"
    echo " --- Dans f : \$1 : $1"
}

 echo "Avant f : \$0 : $0"
echo "Avant f : \$# : $#"
echo "Avant f : \$1 : $1"

f pierre paul jacques
echo "Apres f : \$1 : $1"
```

À l'exécution, cela donne:

```
script un deux trois quatre

Avant f : $0 : ./args
Avant f : $# : 4
Avant f : $1 : un

--- Dans f : $0 : ./args
--- Dans f : $# : 3
--- Dans f : $1 : pierre

Apres f : $1 : un
```

Utilisée dans le corps d'une fonction, la commande interne **shift** décale la numérotation des paramètres de position internes à la fonction.

```
#!/bin/bash

f()
{
    echo " --- Dans f : Avant shift : \$* : $*"
    shift
    echo " --- Dans f : Apres shift : \$* : $*"
}

echo "Appel : f un deux trois quatre"
f un deux trois quatre
```

À l'exécution, cela donne:

```
Appel : f un deux trois quatre
--- Dans f : Avant shift : $* : un deux trois quatre
--- Dans f : Apres shift : $* : deux trois quatre
```

Code de retour des fonctions

L'instruction return

Syntaxe : **return [** *n* **]**

La commande interne **return** permet de sortir d'une fonction avec comme code de retour la valeur *n* (**0** à **255**). Celle-ci est mémorisée dans le paramètre spécial **?**.

Si *n* n'est pas précisé, le code de retour fourni est celui de la dernière commande exécutée.

Dans le script ci-dessous, la fonction *f* retourne le code de retour **1** au programme appelant.

```
#!/bin/bash

f()
{
    echo coucou
    return 1
    echo “a demain” # jamais execute
}

f
echo “code de retour de f : $?”
```

```
À l'exécution, cela donne:

coucou
code de retour de f : 1
```

Remarque :il ne faut confondre **return** et **exit**. Cette dernière arrête l’exécution du programme shell qui la contient.

Lecture et écriture dans des fichiers

Nous apprendrons ici comment lire et écrire dans des fichiers à même notre scripts. Ce sont des opérations fort courantes dans le monde Unix, où toutes les configurations sont généralement stockées dans des fichiers textes. Plusieurs applications utilisent également des fichiers texte à la manière de bases de données, avec un enregistrement par ligne et plusieurs champs par enregistrement.

Lire dans un fichier avec read

La commande read, qui nous permet de lire une ligne tapée au clavier, peut également aller lire dans un fichier. Le fonctionnement est assez simple: on utilise la redirection (<) directement dans le script (notez que l'effet est alors exactement le même que si on ne l'utilisait pas dans le script mais que l'usager l'utilisait en appelant notre script).
Par exemple, supposons un fichier file.txt qui contient les lignes suivantes:

```
Bonjour les petits amis
Comment ça va
Je m'appelle
Stephane
```

Écrivons ensuite le script tout simple myread:

```
read LIGNE < file.txt
echo $LIGNE
```

Lorsque l'on exécute ce script, le fichier file.txt est passé à la commande read, qui en lit la première ligne et la place dans la variable LIGNE.

Remarquez que l'effet est le même que si on avait créé myread ainsi:

```
read LIGNE
echo $LIGNE
```

et que l'on l'appelait de cette façon:

```
myread < file.txt
```

À ce moment, le fichier file.txt est utilisé au lieu de l'entrée standard (le clavier). C'est comme si on tapait le contenu du fichier au clavier.

Tout ceci est bien joli, mais qu'arrive-t-il si on veut lire plusieurs lignes dans notre fichier? Si on modifie myread pour y ajouter plusieurs reads, comme ceci:

```
read LIGNE1
echo $LIGNE1
read LIGNE2
echo $LIGNE2
read LIGNE3
echo $LIGNE3
```

on peut ensuite l'appeler avec la redirection:

```
myread < file.txt
```

À ce moment, comme notre *input* est remplacé par le contenu du fichier, chaque read lira une ligne du fichier, comme si on les avaient tapées dans le même ordre, ce qui est fort logique.

Est-ce qu'on peut faire la même chose en utilisant la redirection à l'intérieur du script? Essayons-le. Modifions myread de la façon suivante:

```
read LIGNE < file.txt
echo $LIGNE
read LIGNE < file.txt
echo $LIGNE
read LIGNE < file.txt
echo $LIGNE
```

Remarquez que j'utilise toujours la même variable - en effet, pourquoi en utiliser d'autres puisqu'une fois qu'une ligne est écrite on ne l'utilisera plus? Maintenant, en théorie, on pourrait appeler notre script simplement comme ceci, puisque que notre redirection est à l'intérieur du fichier:

```
myread
```

Malheureusement le résultat n'est pas celui escompté. Pourquoi? Parce que la ligne read LIGNE < file.txt lit la première ligne du fichier, peu importe combien de fois on l'appelle dans un même script. D'un read à l'autre, la commande read ne sait pas qu'il y en a eu d'autres avant elle... Comment régler ce problème, autrement que de forcer l'utilisateur à appeler notre script avec une redirection (solution fort peu élégante dans le cas d'un fichier de configuration)? La solution dans la section suivante...

Lire dans un fichier avec read et une boucle

Eh oui, il suffit d'utiliser une boucle, qui utilise une commande comme condition. Il faut savoir que la commande read retourne 0 (qui sera interprété comme "vrai" dans une boucle) si elle arrive à lire une ligne du fichier et autre chose sinon (entre autres si elle ne peut plus lire de lignes parce qu'elle est arrivée à la fin du fichier).

Pour lire le contenu entier d'un fichier dans une boucle, on redirigera donc le fichier **à la boucle** plutôt qu'au read. Ça se fait (assez curieusement) en redirigeant **au done** qui ferme la boucle:

```
while read LIGNE
do
   echo $LIGNE
done < file.txt
```

Notez bien la particularité de la redirection au done. Si on tentait plutôt de la mettre au read, on aurait des résultats bien différents. Lesquels et pourquoi, selon vous?

Voici donc le fonctionnement de notre lecture en boucle: à l'entrée de la boucle, bash exécute la commande read LIGNE, qui lira une ligne du fichier file.txt puisqu'il est redirigé à la boucle. Si cette lecture fonctionne, read retourne 0, la condition est "vraie", donc on peut entrer dans la boucle (puisque le while roule tant que la condition est vraie).

Le echo sera donc exécuté, affichant la ligne que l'on vient de lire, puis on re-testera la condition en exécutant de nouveau le read. Lorsque la dernière ligne aura été affichée, l'exécution suivante du read retournera un code différent de 0 (erreur), puisqu'il n'y a plus rien à lire dans le fichier et que le read ne fonctionne plus. Dans ce cas, la condition n'est plus vraie, on n'entre donc plus dans la boucle et le tout se termine.

Décomposer les lignes en mots

Il est possible de passer plus qu'une variable au read. Dans ce cas, la phrase qui est lue sera décomposée en différents mots, qui seront placés, dans l'ordre, dans chacune des variables. Par exemple, modifions notre script précédent pour obtenir:

```
while read MOT1 MOT2 MOT3
do
   echo "Mot 1:  $MOT1; Mot 2:  $MOT2; Mot 3:  $MOT3"
done < file.txt
```

Le script fonctionne de la même façon pour lire une ligne à la fois du fichier file.txt. La seule différence ici est qu'au lieu de placer toute la ligne dans la variable LIGNE, il placera le premier mot dans MOT1, le deuxième dans MOT2 et le reste de la ligne dans MOT3. Si jamais une ligne contient moins de trois mots, les dernières variables seront simplement vides.

La décomposition sera faite simplement en fonction des espaces. Notez qu'il peut y avoir plusieurs espaces entre deux mots, il peut même y avoir des tabulations, et la séparation se fera tout de même correctement. Remarquez également que cet usage du read est aussi valide pour la lecture au clavier.

Ce sont là les bases de la décomposition d'un fichier en ses lignes et mots, ce qui permet d'utiliser un fichier de configuration pour faire fonctionner un script.

Évidemment, il y a toujours moyen de faire plus compliqué... Par exemple: comment faire pour aller lire le fichier /etc/passwd afin d'aller décomposer en différents "mots" tout son contenu (c'est à dire, si je veux avoir le nom de l'usager, son groupe, son répertoire home, etc, dans des variables séparées?).

Testons avec notre script précédent en remplaçant la dernière ligne par:

```
done < /etc/passwd
```

Qu'est-ce qui se passe? Pourquoi? Comment remédier à la situation?

Internal Field Separator

Le problème avec le fichier /etc/passwd, c'est que les différents mots sont séparés non pas par des espaces mais par des deux-points (:). Lorsque le read lit une ligne, il ne voit pas d'espaces alors il place la ligne au complet dans la variable MOT1, et laisse les deux autres variables vides puisqu'il n'y a plus rien à y mettre.

La façon d'arranger le tout est toute simple: il existe une variable (appelée IFS pour *Internal Field Separator*), qui contient le caractère qui doit être utilisé pour séparer les mots dans une phrase. Par défaut, cette variable contient un espace, comme on peut le voir en tapant au *prompt*:

```
echo .$IFS.
```

(les points sont nécessaires si on veut voir l'espace...)

On pourra donc simplement modifier le contenu d'IFS avant de faire nos reads pour lui dire de séparer les mots avec des deux-points. On aurait donc la structure suivante :

```
IFS=':'  # Pour indiquer le caractère de séparation

while read -a Liste; do  # -a indique de mettre les entrées lues dans Liste
  echo ${Liste[0]}  # ici, on affiche le nom de l'usager à chaque tour de boucle

done < userlist
```

- IFS=':' permet d'indiquer que le séparateur de champs est le caractère :
- L'option « -a » de la commande read permet de mettre les entrées lues dans une liste (tableau accessible par la notation de [indice]).

À titre de rappel :

En powershell, on écrivait une structure similaire à ceci pour réaliser la même chose :

```
$fichier = get-content -path a.txt
$compteurLigne = 1
foreach ($ligne in $fichier)
{
  write-host "Ligne $compteurLigne :"
  $mots = $ligne.split(" ")
  $compteurMot = 1
  foreach ($mot in $mots)
  {
    write-host "  Mot $compteurMot : $mot"
    $compteurMot++
  }
  $compteurLigne++
}
```

Écrire dans un fichier

De la même façon qu'on peut rediriger un fichier à l'entrée d'un read, on peut rediriger la sortie d'un echo dans un fichier.

Il suffit d'utiliser le ">" ou le ">>", comme on pourrait le faire à la ligne de commandes. N'oubliez pas que le ">" envoie la ligne dans un fichier donné, détruisant ce fichier s'il existe déjà, tandis que le ">>" envoie la ligne dans un fichier donné, l'ajoutant à la fin du fichier s'il existe déjà.

On voudra donc souvent utiliser le ">" pour la première ligne puis des ">>" pour les autres. Par exemple, si on voulait que notre script de lecture de /etc/passwd crée un fichier avec les informations au lieu d'afficher le tout à l'écran, on n'aurait qu'à le modifier de la façon suivante:

```
IFS=":"
echo "Liste des usagers enregistres " > Liste_Usager
echo "-----------------------------------------" >> Liste_Usager

while read -a Liste
do
   echo "Usager:                ${Liste[0]}"      >> mot_passe
          echo   "-----------------------------------------"   >>
/etc/clear_passwd
done < /etc/passwd
```

À ce moment le script créerait un fichier appelé mot_passe dans votre compte, détruisant toute copie qui existerait déjà, pour y inscrire la ligne de titre. Toutes les lignes suivantes seraient ajoutés à la suite du fichier, pour le remplir de tout notre *output*. Remarquez que cela revient au même que de laisser le script comme il était et de l'appeler en faisant par exemple:

```
readpasswd > mot_passe
```

La seule différence est que dans notre nouvelle version, le script est conçu pour écrire automatiquement dans un fichier prédéfini et de ne rien afficher à l'écran – c'est quelque chose que l'on veut souvent faire lorsque l'on créé par exemple un log sur l'exécution de notre script.

Notez que dans le cas d'un script qui sera exécuté de façon automatisée à intervalles réguliers, on voudra ajouter constamment au même log et on évitera d'utiliser la redirection ">", même pour la première ligne.