Objectifs

Dans ce laboratoire, vous effectuerez les tâches suivantes:

Partie 1 : Créer un script shell simple.
Partie 2 : créer des variables shell
partie 3 : Utiliser les structures de contrôle
Partie 4 : Utiliser les fonctions et les alias

Ressources requises

Un ordinateur exécutant une VM Ubuntu 14.04 sous Oracle VirtualBox.

Contexte/scénario

En tant qu'administrateur, vous devez savoir comment transformer les commandes shell en scripts.

Ouvrez une fenêtre de terminal dans Ubuntu.

Partie 1 :Créer un script shell simple.

Au niveau le plus simple, un script shell n'est qu'un ensemble de commandes placées dans un fichier qui, lorsqu'il est exécuté en tant que programme, exécutera chaque commande individuelle dans l'ordre.

Principes de base des scripts shell

Pour créer un script shell, procédez comme suit:

Utilisez un éditeur de texte pour créer un fichier avec les commandes que vous souhaitez exécuter.

Placez la ligne suivante au début du fichier pour indiquer que le script doit être exécuté par le shell bash.

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

La combinaison de caractères `#!` est utilisée pour indiquer au système quel shell utiliser pour exécuter le script (dans notre cas c'est le shell bash).

le `#` est appelé «hachage» et le caractère `!` est appelé «bang». La contraction des deux donne un "hash-bang" . On trouve aussi d'autres dénominations: **sh-bang, sha-bang, she-bang**

Utilisez la commande chmod pour rendre le fichier exécutable:

```
chmod a+x file_name
```

GNU/Linux a de nombreux éditeurs de texte, les mérites de l'un par rapport à l'autre sont souvent vivement débattus. Deux sont spécifiquement mentionnés dans le programme LPIC:

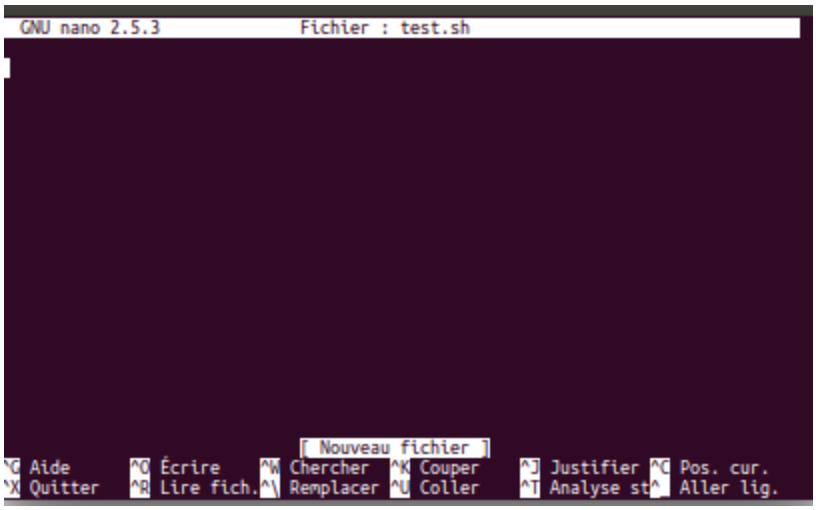
L'éditeur `GNU nano` est un éditeur très simple bien adapté à l'édition de petits fichiers texte.

Le Visual Editor `vi`, ou sa version plus récente (`vim`), est un éditeur remarquablement puissant mais a une courbe d'apprentissage abrupte.

Nous utiliserons L'éditeur GNU `nano.`

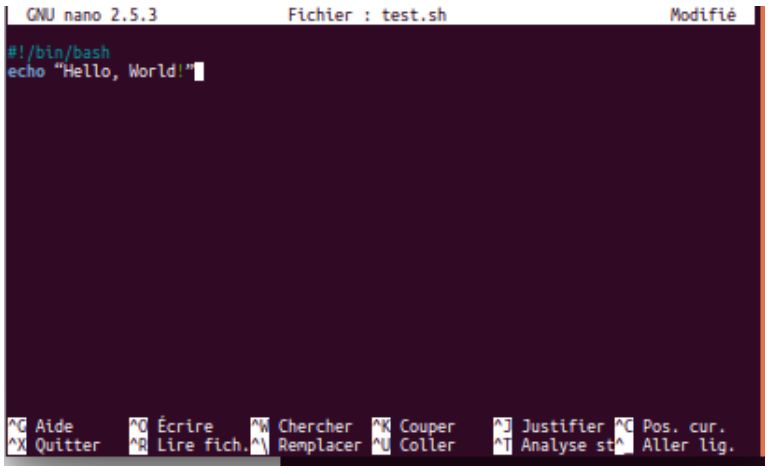
Tapez `nano test.sh` et vous verrez un écran similaire à celui-ci:

```
admin@localhost:~$ nano test.sh
```



Tapez `#!/bin/bash` sur la première ligne et appuyez sur Entrée.

Puis saisissez la commande `echo "Hello, World!"` et appuyez sur Entrée.



Pour sauvegarder le fichier `test.sh` et quitter `nano`, appuyez simultanément sur `Ctrl-o` et appuyez sur Entrée puis appuyez simultanément sur `Ctrl-x` et appuyez sur Entrée)

L'exécution du script `test.sh` peut se faire soit en le passant comme un argument de la commande `bash:`

```
admin@localhost:~$bash test.sh

Hello, World!
```

Soit en l'exécutant directement:

```
admin@localhost:~$ ./test.sh
-bash: ./test.sh: Permission denied
```

L'erreur `-bash: ./test.sh: Permission denied` signifie que le script n'a pas été marqué comme exécutable.

Modifiez les autorisations sur le script, afin qu'il puisse être exécuté directement.

Un rapide `chmod.`

```
admin@localhost:~$ chmod +x test.sh
```

Le script fonctionne maintenant:

```
admin@localhost:~$ ./test.sh
Hello, World!
```

Le `./` indique au shell qu'il doit exécuter le script `test.sh` à partir du répertoire courant.

Partie 2 : Les Variables shell

Les Variables

Les variables sont un élément clé de tout langage de programmation. Une utilisation très simple des variables est présentée ici:

Créez le script `testvar.sh` avec les instructions indiquées ci-dessous:

```
#!/bin/bash

ANIMAL="penguin"
echo "Mon animal préféré est un $ANIMAL"
```

Après la ligne shebang est une directive pour affecter du texte à une variable. Le nom de la variable est `ANIMAL` et le signe égal permet de lui affecter la chaîne `pingouin.` Pensez à une variable comme une boîte dans laquelle vous pouvez stocker des choses. Après avoir exécuté cette ligne, la boîte appelée `ANIMAL` contient le mot `pingouin.`

Il est important qu'il n'y ait pas d'espaces entre le nom de la variable, le signe égal et valeur à affecter à la variable. La mise en majuscule du nom de la variable n'est pas nécessaire mais c'est une convention utile pour différencier les variables des commandes shell.

Ensuite, le script renvoie une chaîne à la console. La chaîne contient le nom de la variable précédée d'un signe dollar `$`. Lorsque l'interprète voit ce signe dollar, il remplacera la variable `ANIMAL` par son contenu, qui est `penguin`. La sortie du script est alors `Mon animal préféré est un pingouin.`

Il est possible de rendre votre script interactif et d'affecter une valeur à une variable via la commande `read:`

Créez le script `testread.sh :`

```
admin@localhost:~$ nano testread.sh

#!/bin/bash

echo –n "Quel est votre nom?"

read NOM
echo "Bonjour $NOM!"
```

La commande `read` permet de récupérer une chaîne de caractères directement depuis l'entrée standard le clavier et la stocker dans la variable NOM

Si vous créez une variable et que vous ne souhaitez plus que cette variable soit définie, utilisez la commande `unset` pour la supprimer:

```
root@localhost ~ $unset ANIMAL
```

Les variables locales et les variables d'environnement

Une variable locale n'est disponible que pour le shell dans lequel elle a été créée. Une variable d'environnement est disponible pour le shell dans lequel elle a été créée et elle est passée dans toutes les autres commandes / programmes démarrés par le shell.

Par convention, les caractères minuscules sont utilisés pour créer des variables locales et les caractères majuscules sont utilisés lors du nommage d'une variable d'environnement. Par exemple, une variable locale peut être appelée `test` tandis qu'une variable d'environnement peut être appelée `TEST`. Bien que ce soit une convention et pas une règle.

Il existe plusieurs façons d'afficher les valeurs des variables. La commande `set` seule affichera toutes les variables (locales et environnementales):

```
admin@localhost ~ $set

PS1='[\u@\h \W]\$ '
PWD=/home/sysadmin
SHELL=/bin/bash
TERM=xterm
UID=500
USER=admin
```

Pour afficher uniquement les variables d'environnement, plusieurs commandes fournissent à peu près la même sortie: `env, declare –x, typeset –x ou export –p:`

Exemples de variables prédéfinies

a- Les variables de position `0 1 2 3....:`

Créez le script `testvarposition.sh :`

```
admin@localhost:~$ nano testvarposition.sh
```

```
#!/bin/bash

echo "Bonjour $0"

echo "Mon 1er Argument est $1"

echo "Mon 2eme Argument est $2"

echo "La commande $0 contient $# arguments $1 et $2 "
```

Le caractère dollar `$` suivi d'un nombre N correspond au Nième argument passé au script. Si vous exécutez le script `testvarposition.sh` avec 2 arguments `poz1` et `poz2`, la sortie sera :

```
admin@localhost:~$bash testvarposition.sh poz1 poz2

 Bonjour testvarposition.sh.

Mon 1er Argument est poz1

Mon 2eme Argument est poz2

La commande testvarposition.sh contient 2 arguments poz1 et poz2
```

La variable `$0` contient le nom du script lui-même.

La variable `$1` contient le premier argument de la commande `poz1`.

La variable `$2` contient le deuxième argument de la commande `poz2`.

La variable `$#` contient le nombre d'arguments utilisés.

b- La variable code retour « ? »

Après l'exécution d'un programme, que ce soit un binaire ou un script, il renvoie un code retour qui est un entier compris entre 0 et 255. Vous pouvez afficher son contenu via la commande `echo $?` pour voir si la commande précédente s'est terminée avec succès ou pas.

```
admin@localhost:~$ grep -q root /etc/passwd

admin@localhost:~$ echo $?

0

admin@localhost:~$ grep -q bobo /etc/passwd

admin@localhost:~$ echo $?

1
```

L'option `-q` de la commande `grep` ne revois pas de sortie (mode silencieux), la variable $? renvoie 0 si la chaîne a été trouvée et 1 sinon. Ces informations peuvent être utilisées pour conditionnée une action basée sur la sortie d'une autre commande.

Par convention, un code de sortie de 0 signifie «tout va bien». Tout code de sortie supérieur à 0 signifie qu'une sorte d'erreur s'est produite, qui est spécifique au programme. Ci-dessus, vous avez vu que grep utilise 1 pour signifier que la chaîne n'a pas été trouvée.

c- La variable invite de commande « PS1 »

La variable PS1 définie l'invite de commande(prompt) d'un compte shell.

Pour remplacer l'invité de commande (prompt) actuel par un compteur d'historique de commandes exécutées suivi du caractère € exécutez la commande suivante:

```
admin@localhost ~]$PS1=' \# €'

20€ls /

bin  etc   init  lib64       media  opt   root  sbin     srv  tmp  var
dev  home  lib   lost+found  mnt    proc  sbin  selinux  sys  usr
21€
```

Redéfinissez la variable locale `PS1` afin qu'elle affiche le nom d'utilisateur actuel, le nom d'hôte, le répertoire de travail et le caractère d'invite $ en exécutant la commande suivante:

21€ PS1='\u@\h \w $'

```
admin@localhost ~ $
```

d- La variable PATH chemin de recherche des commande shell

Affichez la variable `PATH`, qui détermine quels répertoires seront recherchés par le shell pour les commandes exécutées à partir de la ligne de commande, en exécutant la commande suivante :

```
admin@localhost ~ $ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/admin/bin
```

Pour tester la variable PATH, créez d'abord un nouveau répertoire dans votre répertoire personnel nommé scripts en exécutant la commande suivante:

```
admin@localhost ~ $ mkdir scripts
```

Créez un script nommé `today.sh` en exécutant les commandes suivantes:

```
admin@localhost ~ $ echo '#!/bin/bash' > scripts/today.sh

admin@localhost ~ $echo 'echo aujourdhui on est:' >> scripts/today.sh
admin@localhost ~ $echo 'date +%D' >> scripts/today.sh
admin@localhost ~ $more scripts/today.sh
```

```
#!/bin/bash
echo aujourd'hui on est le:

date +%D
```

```
admin@localhost ~ $ cd scripts

admin@localhost ~/scripts $ chmod +x today.sh

admin@localhost ~/scripts $ ./today.sh  #chemin relatif

 aujourd'hui on est le:

02/11/2023

admin@localhost ~/scripts $ /home/admin/scripts/today.sh   #chemin absolu

 aujourd'hui on est le :

02/11/2023
```

Bien que les chemins d'accès relatifs et absolus fonctionnent pour exécuter le script, l'utilisation du nom de script uniquement échouera car le répertoire des scripts n'est pas répertorié en tant que valeur dans la variable `PATH`. Essayez d'exécuter le script en utilisant uniquement le nom du script, il échouera car le répertoire actuel n'est pas recherché par le shell :

```
admin@localhost ~/scripts $ today.sh

-bash: today: command not found
```

Affichez à nouveau la variable `PATH` pour confirmer que le répertoire des scripts n'y figure pas. Modifiez la variable `PATH` afin qu'elle contienne le répertoire des scripts et affichez la variable `PATH` pour vérifier la modification:

```
admin@localhost ~/scripts $ echo $PATH

/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/admin/bin
```

Ajoutez le répertoire scripts dans le `PATH`:

```
admin@localhost ~/scripts $ PATH=$PATH:$HOME/scripts

admin@localhost ~/scripts $ echo $PATH

/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/admin/scripts
```

Une fois que la variable `PATH` contient le chemin d'accès au répertoire où se trouve votre script, vous pouvez taper uniquement le nom du fichier de script pour l'exécuter, quel que soit votre emplacement dans

le système de fichiers. Exécutez le script à partir de votre répertoire actuel. Ensuite, passez à votre répertoire personnel en exécutant la commande `cd` et exécutez à nouveau le script `today.sh` pour voir le résultat:

```
sadmin@localhost ~/scripts $ today.sh

 aujourd'hui on est le :

02/11/2023

admin@localhost ~/scripts $ cd

admin@localhost ~ $ today.sh
 aujourd'hui on est le :

02/11/2023
```

De par leur conception, les modifications apportées aux variables apportées dans le shell ne persistent pas après la fermeture du terminal. Tapez exit pour ferme la session shell.

```
admin@localhost ~ $ exit
```

Ouvrez un autre terminal shell et affichez la variable PATH pour confirmer que sa valeur est revenue à la liste d'origine:

```
admin@localhost ~$ echo $PATH

/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/admin/bin
```

Pour un utilisateur non privilégié, le fichier /`home/admynov/.bashrc` peut être utilisé pour personnaliser les variables telles que PATH. Il suffit d'éditer le fichier `/home/admynov/.bashrc` et rajouter à la fin du fichier le nouveau `PATH`.

Partie 3 : Les structures de contrôle

a- La structure `if then else fi`

La syntaxe de base d'une instruction `if` **est:**

```
if condition

then

   suite_commandes
fi
```

Dans l'exemple ci-dessus, si l'instruction `suite_commandes` renvoie false, aucune commande ne sera exécutée. Pour que les commandes soient exécutées si l'instruction `commandes` renvoie false, incluez une clause `else:`

```
if condition

then
   suite_commandes2
else

 suite_commandes2

fi
```

Exemple 1 :

```
admin@localhost ~ /scripts$nano testif1.sh

if grep root  /etc/passwd

then

echo  "root possede un compte utilisateur"

fi
```

Exemple 2 :

```
admin@localhost ~ /scripts$nano testif2.sh

if cmp $1 $2

then

echo  "fichiers identiques"

else

echo "fichiers différents"

fi

admin@localhost ~ /scripts$chmod u+x testif2.sh

admin@localhost ~ /scripts$echo bonjour >test1

admin@localhost ~ /scripts$echo bonjour >test2
```

```
admin@localhost ~ /scripts$./testif2.sh test1  test2

fichiers identiques

admin@localhost ~ /scripts$.echo hello >test1

admin@localhost ~ /scripts$./testif2.sh test1  test2

fichiers différents
```

La commande `test`

Lors de l'écriture de scripts shell, il y aura des moments où un administrateur voudra exécuter certaines commandes selon si une instruction conditionnelle est vraie ou fausse. Une instruction conditionnelle peut être utilisée pour déterminer les éléments suivants:

-Si deux variables ou valeurs de chaîne correspondent (ou ne correspondent pas)
-Si deux variables ou valeurs numériques correspondent (ou ne correspondent pas)
-L'état des fichiers (si le fichier existe, si le fichier est un répertoire, etc.)
-Si une commande se termine avec succès.

En règle générale, les programmeurs utilisent la commande `test` comme une instruction conditionnelle. La commande test peut effectuer des comparaisons numériques, des comparaisons de chaînes et des opérations de test de fichiers.En règle générale, les programmeurs utilisent la commande test comme une instruction conditionnelle. La commande test peut effectuer des comparaisons numériques, des comparaisons de chaînes et des opérations de test de fichiers.

Le tableau suivant illustre certaines des comparaisons possibles:

Type	Symbole	Exemple
Vrai si la longueur de la chaîne est nulle	-z	–z *file_name*
Vrai si la longueur de la chaîne n'est pas nulle	-n	–n *file_name*
Vrai si les chaînes sont égales	=	*string1* = *string2*
Vrai si les chaînes ne sont pas égales	!=	*string1* != *string2*
Vrai si les entiers sont égaux	-eq	*int1* -eq *int2*
Vrai si les entiers ne sont pas égaux	-ne	*int1* -ne *int2*
Vrai si le premier entier est supérieur au deuxième entier	-gt	*int1* -gt *int2*
Vrai si le premier entier est supérieur ou égal au deuxième entier	-ge	*int1* -ge *int2*
Vrai si le premier entier est inférieur au deuxième entier	-lt	*int1* -lt *int2*
Vrai si le premier entier est inférieur ou égal au deuxième entier	-le	*int1* -le *int2*
Vrai si le fichier est un répertoire	-d	-d *file*
Vrai si le fichier est un fichier ordinaire	-f	-f *file*
Vrai si le fichier existe	-e	-e *file*

Bien que vous puissiez exécuter la commande `test` spécifiquement, dans une instruction if, vous pouvez implicitement appeler l'instruction `test` en plaçant les arguments de l'instruction `test` entre crochets.

Alors, au lieu d'écrire…

```
admin@localhost ~ /scripts$if test $var -eq 7
```

… Vous pourriez écrire:

```
admin@localhost ~ /scripts$ if [ $var -eq 7 ]
```

Remarque: Lorsque vous utilisez la technique des crochets, assurez-vous de placer un espace devant et après les crochets. Sinon, une erreur se produira.

Exemple :

```
admin@localhost ~ $ grep -q  root /etc/passwd

admin@localhost ~ $ if [ $? -eq 0 ];then echo  "root possede un compte utilisateur";fi
```

```
admin@localhost ~ $grep –q  pierre /etc/passwd

admin@localhost ~ $ if [ $? –ne 0 ];then echo  " pierre ne  possede pas un compte utilisateur";fi
```

b- La structure `case in esac`

Lorsqu'il y a un nombre important de choix, la structure **case** est plus appropriée que la structure **if**.

Syntaxe :

```
case $variable in
modele1) commande1;;
modele2) commande2;;
modele3 | modele4 | modele5 ) commande3;;
esac
```

`$variable` est comparé successivement à chaque modèles, dés qu'un modèles correspond, on exécute la commande correspondante.

Exemple :

```
admin@localhost ~ /scripts$nano testcase.sh

#!/bin/bash

echo "–1–francais"

echo "–2– anglais"

echo "–3– espagnol"



echo –n "choisissez une langue : "

read langue

case $langue in

1) echo Bonjour ;;

2) echo Hello ;;

3) echo Buenos Dias ;;

esac
```

Sauvegardez, ajoutez le doit `x` puis exécutez le script !!

c- La structure `for do done`

La boucle `for` permet d'exécuter le corps de la boucle un nombre déterminé de fois.

Syntaxe :

```
for variable in liste_valeurs

    do instruction(s)
done
```

Les commandes comprises entre les mots-clés **do** et **done**, le corps de la boucle, sont exécutées autant de fois qu'il y a de mots dans `list_valeurs.`

Exemple :

```
admin@localhost ~ /scripts$nano testfor.sh

for auto in megane zafira laguna

do

echo Voiture:$auto

done
```

Sauvegardez, ajoutez le doit `x` puis exécutez le script !!

d- La structure `while do done`

La structure `while` est utilisée pour déterminer si la condition est vraie (?=0) ou fausse(?>0); si elle est est vrai, une série d'actions a lieu et la condition est vérifiée à nouveau. Si la condition est fausse, aucune action n'a lieu et le programme continue.

La série d'instructions à exécuter lorsque le test conditionnel while est vrai sera contenue dans un bloc qui commence après le mot clé `do` et se termine au mot clé `done.`

Syntaxe :

```
while  condition

    do instruction(s)
done
```

Exemple ;

```
admin@localhost ~ /scripts$nano testwhile.sh

#!/bin/bash

echo –n "Entrez un mot :"

read mot

while [ "$mot" != "fin" ]

do

echo  "Ce n’est pas le bon mot!"

echo –n  "Entrez un mot :"

read mot

done

echo  "Vous avez trouvé. BRAVO !!!"

admin@localhost ~ /scripts$chmod u+x testwhile.sh

admin@localhost ~ /scripts$./testwhile.sh

Entrez un mot:bonjour

Ce n’est pas le bon mot!

Entrez un mot:hello

Ce n’est pas le bon mot !

Entrez un mot:fin

Vous avez trouvé. BRAVO!!!

admin@localhost ~ /scripts$
```

e- La commande `exit`

La commande `exit` termine l'exécution du script.

Syntaxe :

```
exit [n]
```

La commande `exit` met fin au script immédiatement. Il est possible de préciser le code de retour du script en le précisant en argument (*de 0 à 255*). Sans argument précisé, c'est le code de retou de la dernière commande du script qui sera transmise à la variable `$?`.

Exemple :

```
admin@localhost ~ /scripts$bash

admin@localhost ~ /scripts$exit 10

admin@localhost ~ /scripts$echo $?

10
```

f- La commande `break`

La commande `break` interrompt l'exécution des commandes d'une boucle `while` et `for` entre autres et donne le contrôle à la commande qui suit le mot-clé `done`. L'execution de la commande `break` dans une boucle n'a de sens qui si elle est conditionnée par une structure `if` ou commandée par l'opérateur logique `&& ou ||`.

Exemple :

```
admin@localhost ~ /scripts$testbreak.sh

while true # l'instruction true est toujours vraie

 do

 if [ $# -eq 0 ]

then

echo "Vous n'avez saisi aucun argument"

#break #on désactive la commande break pour provoquer une boucle infinie

fi

done
```

```
admin@localhost ~ /scripts$chmod u+x testbreak.sh

admin@localhost ~ /scripts$./testbreak.sh

Vous n'avez saisi aucun argument

Vous n'avez saisi aucun argument

Vous n'avez saisi aucun argument
```

Appuyez sur `Ctrl-c` pour arrêter le script

Il suffit de de-commenter la commande break pour sortir de boucle infinie (effacez le diese `#` à coté de la commande `break`).

Exécutez à nouveau le script `testbreak.sh` :

```
admin@localhost ~ /scripts$./testbreak.sh

Vous n'avez saisi aucun argument

admin@localhost ~ /scripts$
```

Partie 4 : Les fonctions et les alias

a- Les fonctions

Une fonction shell est un ensemble de commandes identifiées par un nom de fonction et résidant en mémoire principale(RAM), ce qui la rend plus rapide d'accès qu'un script qui réside sur disque.

Syntaxe :

```
nom_foction (){ liste de commandes ;}
```

Exemple :

```
admin@localhost ~ /scripts$nano testfonction.sh

##################Fonction lecture#################

lire_ligne () { echo -n "Entrez une ligne de texte ou <fin>pour finir: "

read ligne

}

##################Fonction affichage#################
```

```
ecrire_ligne () { echo "$ligne" >>ligne.txt

}

##################Programme principal#################

while [ "$ligne" != "fin" ]

do

lire_ligne

ecrire_ligne

done

more  ligne.text
```

admin@localhost ~ /scripts$testfonction.sh

```
Entrez une ligne de texte ou <fin>pour finir: Ligne N 1

Entrez une ligne de texte ou <fin>pour finir: Ligne N 2

Entrez une ligne de texte ou <fin>pour finir: fin

Ligne N 1

Ligne N 1
```

admin@localhost ~ /scripts$

b- Les alias

L'alias est un mécanisme qui permet de designer une commande ou une suite de commandes par un nouveau nom, le nom d'alias.

Syntaxe :

```
alias nouvelle_commande="suite de commande"
```

Exemple :

```
alias list="ls   -ail"
```

```
admin@localhost ~ /scripts$list  /

total 124

      2 drwxr-xr-x  24 root root  4096 janv.  8 18:45 .

      2 drwxr-xr-x  24 root root  4096 janv.  8 18:45 ..

8912897 drwxr-xr-x   2 root root  4096 nov.   8 17:12 bin

3538945 drwxr-xr-x   4 root root  4096 janv.  8 18:46 boot

4849665 drwxrwxr-x   2 root root  4096 mai   24  2018 cdrom

      2 drwxr-xr-x  20 root root  4340 janv. 26 09:31 dev

1310721 drwxr-xr-x 163 root root 12288 janv. 25 12:35 etc

 524289 drwxr-xr-x   5 root root  4096 janv. 25 12:35 home
```