Les Commandes Structurées sous Bash

Boucles for, while et until

Ilias TOUGUI

L'École Supérieure d'Informatique et du Numérique

INF2132 - Systèmes d'Exploitation

PLAN DU COURS

1. **Introduction aux Boucles**
2. **Paramètres de Ligne de Commande**

Partie I: Introduction aux Boucles

Contexte

Rappel du chapitre précédent

Dans le chapitre précédent, nous avons étudié comment manipuler le flux d'un script shell en :

- Vérifiant la sortie des commandes
- Testant les valeurs des variables
- Utilisant les structures conditionnelles (if-then-else)

Dans ce chapitre, nous allons approfondir les **commandes structurées** qui contrôlent le flux des scripts shell.

Objectifs du Chapitre

Les structures de boucles

Ce chapitre couvre les commandes de boucles du shell bash qui permettent de :

- Effectuer des processus répétitifs
- Parcourir un ensemble de commandes jusqu'à ce qu'une condition soit remplie
- Traiter des fichiers, utilisateurs ou lignes de texte de manière itérative

Nous allons étudier trois commandes principales :

- La boucle **for**
- La boucle **while**
- La boucle **until**

La Commande for

Introduction et utilité

L'itération à travers une série de commandes est une pratique courante en programmation.

Cas d'usage typiques :

- Traiter tous les fichiers d'un répertoire
- Parcourir tous les utilisateurs d'un système
- Lire toutes les lignes d'un fichier texte

Le shell bash fournit la commande **for** pour créer une boucle qui itère à travers une série de valeurs.

Syntaxe de Base

Format de la commande for

Format général :

```
for var in list
do
    commands
done
```

Fonctionnement :

- À chaque itération, la variable `var` contient la valeur courante de la liste
- Les commandes entre `do` et `done` sont exécutées pour chaque valeur
- La variable `$var` est accessible dans les commandes

Note : Vous pouvez écrire `for var in list; do` sur une seule ligne en utilisant un point-virgule.

Lecture de Valeurs dans une Liste

Exemple de base

Exemple simple - itération sur des états américains :

```
$ cat test1
#!/bin/bash
# basic for command

for test in Alabama Alaska Arizona Arkansas California Colorado
do
    echo The next state is $test
done
```

Résultat de l'Exécution

Sortie du script test1

```
$ ./test1
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
$
```

La boucle **for** assigne successivement chaque valeur de la liste à la variable `$test` et exécute les commandes pour chaque valeur.

Persistance de la Variable

Valeur après la boucle

La variable de boucle **reste valide** après la fin de la boucle et conserve la dernière valeur.

```
$ cat test1b
#!/bin/bash
# testing the for variable after the looping
for test in Alabama Alaska Arizona Arkansas California Colorado
do
    echo "The next state is $test"
done

echo "The last state we visited was $test"

test=Connecticut
echo "Wait, now we're visiting $test"
```

Résultat - Persistance

Sortie de test1b

```
$ ./test1b
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado

# Variable conserve la dernière valeur
The last state we visited was Colorado
# On peut modifier la variable
Wait, now we're visiting Connecticut
$
```

Valeurs Complexes - Problème

Apostrophes et guillemets

Les choses se compliquent avec certaines données. Voici un problème classique :

```
$ cat badtest1
#!/bin/bash
# another example of how not to use the for command

for test in I don't know if this'll work
do
    echo "word:$test"
done
```

Sortie problématique :

```
$ ./badtest1
word:I
word:dont know if thisll
word:work
```

Valeurs Complexes - Solutions

Échappement et guillemets

Deux solutions possibles :

- Utiliser le caractère d'échappement (backslash) : \
- Utiliser des guillemets doubles pour les valeurs avec apostrophes

Exemple corrigé :

```
$ cat test2
#!/bin/bash
for test in I don\'t know if "this'll" work
do
    echo "word:$test"
done
$ ./test2
word:I
word:don't
word:know
word:if
word:this'll
word:work
```

Valeurs avec Espaces - Problème

États américains composés

Un autre problème courant : les valeurs contenant des espaces.

```
$ cat badtest2
#!/bin/bash
for test in Nevada New Hampshire New Mexico New York North Carolina
do
    echo "Now going to $test"
done
```

Sortie incorrecte :

```
$ ./badtest2
Now going to Nevada
Now going to New      # Séparation incorrecte !
Now going to Hampshire
Now going to New
Now going to Mexico
```

Valeurs avec Espaces - Solution

Utilisation des guillemets doubles

Solution : Encadrer les valeurs contenant des espaces avec des guillemets doubles.

```
$ cat test3
#!/bin/bash
# an example of how to properly define values

for test in Nevada "New Hampshire" "New Mexico" "New York"
do
    echo "Now going to $test"
done
$ ./test3
Now going to Nevada
Now going to New Hampshire
Now going to New Mexico
Now going to New York
$
```

Note : Les guillemets ne font pas partie de la valeur.

Liste depuis une Variable

Accumulation de valeurs

On accumule souvent une liste de valeurs dans une variable.

```
$ cat test4
#!/bin/bash
# using a variable to hold the list

list="Alabama Alaska Arizona Arkansas Colorado"
list=$list" Connecticut"    # Concaténation

for state in $list
do
    echo "Have you ever visited $state?"
done
```

Résultat - Variable Liste

Sortie de test4

```
$ ./test4
Have you ever visited Alabama?
Have you ever visited Alaska?
Have you ever visited Arizona?
Have you ever visited Arkansas?
Have you ever visited Colorado?
Have you ever visited Connecticut?
$
```

La variable `$list` contient la liste complète des valeurs. On peut ajouter des éléments par concaténation.

Valeurs depuis une Commande

Substitution de commande

On peut générer des valeurs en utilisant la sortie d'une commande avec la substitution de commande.

```
$ cat test5
#!/bin/bash
# reading values from a file

file="states"

for state in $(cat $file)
do
    echo "Visit beautiful $state"
done
```

Note : Le fichier doit être dans le même répertoire ou utiliser un chemin absolu/relatif.

Contenu du Fichier et Résultat

Lecture ligne par ligne

Contenu du fichier states :

```
$ cat states
Alabama
Alaska
Arizona
Arkansas
Colorado
Connecticut
Delaware
Florida
Georgia
New York
New Hampshire
North Calorina
```

Sortie du Script test5

Itération sur le fichier

```
$ ./test5
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
...
Visit beautiful Georgia
Visit beautiful New
Visit beautiful York
Visit beautiful New
Visit beautiful Hampshire
Visit beautiful North
Visit beautiful Carolina
$
```

Le fichier `states` contient chaque état sur une ligne séparée. La boucle **for** itère ligne par ligne.

Changement du Séparateur IFS

Variable d'environnement interne

Problème : La variable d'environnement spéciale **IFS** (Internal Field Separator) définit les séparateurs de champs.

Séparateurs par défaut :

- Espace
- Tabulation
- Retour à la ligne

Solution : Modifier temporairement IFS pour reconnaître uniquement le retour à la ligne :

```
IFS=$'\n'
```

Exemple avec IFS Modifié

Gestion des espaces dans les données

```
$ cat test5b
#!/bin/bash
# reading values from a file

file="states"

IFS=$'\n'       # Modification du séparateur
for state in $(cat $file)
do
    echo "Visit beautiful $state"
done
```

Maintenant, le script peut gérer des valeurs contenant des espaces comme "New York" ou "New Hampshire".

Résultat avec IFS Modifié

États avec espaces traités correctement

```
$ ./test5b
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
Visit beautiful New York
Visit beautiful New Hampshire
Visit beautiful North Carolina
$
```

Les noms d'états avec espaces sont maintenant correctement traités.

Bonne Pratique avec IFS

Sauvegarde et restauration

Pratique recommandée : Sauvegarder l'ancienne valeur de IFS avant de la modifier, puis la restaurer.

```
IFS.OLD=$IFS            # Sauvegarde
IFS=$'\n'              # Modification
<use the new IFS value in code>
IFS=$IFS.OLD            # Restauration
```

Autres usages d'IFS :

- Fichiers séparés par deux-points : `IFS=:`
- Multiples séparateurs : `IFS=$'\n':;"`

Cette technique assure que IFS retrouve sa valeur par défaut pour les opérations futures.

Parcours de Répertoire

Utilisation des caractères génériques (wildcards)

File globbing : Processus de génération de noms de fichiers ou chemins qui correspondent à un caractère générique.

Exemple - itération sur tous les fichiers d'un répertoire :

```
$ cat test6
#!/bin/bash
# iterate through all the files in a directory

for file in ./test/*
do
    if [ -d "$file" ]
    then
       echo "$file is a directory"
    elif [ -f "$file" ]
    then
       echo "$file is a file"
    fi
done
```

Résultat - Parcours de Répertoire

Distinction fichiers/répertoires

```
$ ./test6
./test/dir1 is a directory
./test/myprog.c is a file
./test/myscript.sh is a file
./test/newdir is a directory
./test/setup.py is a file
./test/testdir is a directory
$
```

Important : Encadrez `$file` de guillemets doubles pour gérer les noms avec espaces.

Multiples Répertoires

Combinaison de wildcards

On peut combiner plusieurs patterns de wildcards dans la même boucle **for**.

```
$ cat test7
#!/bin/bash
# iterating through multiple directories

for file in /home/ilias/.b* /home/ilias/badtest
do
    if [ -d "$file" ]
    then
       echo "$file is a directory"
    elif [ -f "$file" ]
    then
       echo "$file is a file"
    else
      echo "$file doesn't exist"
    fi
done
```

Résultat - Multiples Wildcards

Sortie de test7

```
$ ./test7
/home/ilias/.backup.timestamp is a file
/home/ilias/.bash_history is a file
/home/ilias/.bash_logout is a file
/home/ilias/.bash_profile is a file
/home/ilias/.bashrc is a file
/home/ilias/badtest doesn't exist
$
```

Attention : La boucle tente de traiter tout ce qui est dans la liste, même si le fichier n'existe pas. Testez toujours avant de traiter !

Boucle for Style C

Introduction

Si vous avez programmé en langage C, vous connaissez probablement cette syntaxe.

En C : Une boucle `for` définit généralement :

- Une variable initialisée
- Une condition qui doit rester vraie
- Une méthode pour altérer la variable à chaque itération

Exemple en C :

```
for (i = 0; i < 10; i++)
{
    printf("The next number is %d\n", i);
}
```

Format Bash Style C

Syntaxe et différences

Format bash de la boucle for style C :

```
for (( variable assignment ; condition ; iteration ))
```

Différences importantes avec le shell classique :

- L'assignation peut contenir des espaces
- La variable dans la condition n'est **pas** précédée de `$`
- L'équation d'itération n'utilise **pas** la commande `expr`

Exemple :

```
for (( a = 1; a < 10; a++ ))
```

Exemple Pratique

Compteur simple

Script utilisant la boucle for style C :

```
$ cat test8
#!/bin/bash
# testing the C-style for loop

for (( i=1; i <= 10; i++ ))
do
    echo "The next number is $i"
done
```

Résultat - Style C

Sortie de test8

```
$ ./test8
The next number is 1
The next number is 2
The next number is 3
The next number is 4
The next number is 5
The next number is 6
The next number is 7
The next number is 8
The next number is 9
The next number is 10
```

La boucle itère en utilisant la variable `i`, qui est incrémentée de 1 à chaque itération.

Variables Multiples

Plusieurs compteurs

La boucle for style C permet d'utiliser **plusieurs variables** avec des processus d'itération différents.

```
$ cat test9
#!/bin/bash
# multiple variables

for (( a=1, b=10; a <= 10; a++, b-- ))
do
    echo "$a - $b"
done
```

Note : Une seule condition peut être définie, mais chaque variable peut avoir son propre processus d'itération.

Résultat - Variables Multiples

Sortie de test9

```
$ ./test9
1 - 10
2 - 9
3 - 8
4 - 7
5 - 6
6 - 5
7 - 4
8 - 3
9 - 2
10 - 1
```

La variable `a` est incrémentée tandis que la variable `b` est décrémentée à chaque itération.

La Commande while

Introduction et concept

La commande **while** est un croisement entre l'instruction **if-then** et la boucle **for**.

Fonctionnement :

- Définit une commande de test
- Boucle tant que la commande retourne un code de sortie zéro
- Teste la condition au **début** de chaque itération
- Arrête quand la commande retourne un code non-zéro

Format de base :

```
while test command
do
  other commands
done
```

Utilisation Courante

Test avec crochets

L'usage le plus courant est d'utiliser des crochets pour vérifier une valeur de variable shell.

```
$ cat test10
#!/bin/bash
# while command test

var1=10
while [ $var1 -gt 0 ]
do
    echo $var1
    var1=$[ $var1 - 1 ]
done
```

Important : La variable utilisée dans le test doit être modifiée dans la boucle, sinon vous obtenez une boucle infinie !

Résultat - while Basique

Sortie de test10

```
$ ./test10
10
9
8
7
6
5
4
3
2
1
```

La condition `[ $var1 -gt 0 ]` est testée à chaque itération. La boucle s'arrête quand `var1` n'est plus supérieur à 0.

Commandes de Test Multiples

Plusieurs tests dans while

La commande **while** permet de définir plusieurs commandes de test. Seul le code de sortie de la **dernière** commande détermine l'arrêt de la boucle.

```
$ cat test11
#!/bin/bash
# testing a multicommand while loop

var1=10

while echo $var1
    [ $var1 -ge 0 ]
do
    echo "This is inside the loop"
    var1=$[ $var1 - 1 ]
done
```

Résultat - Tests Multiples (2/2)

Sortie de test11

```
$ ./test11
10
This is inside the loop
9
This is inside the loop
...
...
2
This is inside the loop
1
This is inside the loop
0
This is inside the loop
-1          # echo s'exécute même quand le test échoue
```

Attention : Toutes les commandes de test sont exécutées à chaque itération, y compris la dernière où le test final échoue !

La Commande until

Fonctionnement inverse du while

La commande **until** fonctionne exactement à l'opposé de la commande while.

Principe :

- Requiert une commande de test qui produit normalement un code de sortie **non-zéro**
- Tant que le code est non-zéro, bash exécute les commandes de la boucle
- Arrête quand la commande retourne un code de sortie **zéro**

Format :

```
until test commands
do
    other commands
done
```

Exemple until

Décrémentation jusqu'à zéro

Script utilisant until :

```
$ cat test12
#!/bin/bash
# using the until command

var1=100

until [ $var1 -eq 0 ]
do
    echo $var1
    var1=$[ $var1 - 25 ]
done
```

La boucle s'arrête dès que `var1` est égal à 0.

Résultat - until Basique

Sortie de test12

```
$ ./test12
100
75
50
25
$
```

Le test vérifie si `var1` est égal à zéro. Dès que c'est le cas, la commande **until** arrête la boucle.

until avec Commandes Multiples

Même comportement que while

Comme pour **while**, on peut utiliser plusieurs commandes de test avec **until**.

```
$ cat test13
#!/bin/bash
# using the until command

var1=100

until echo $var1
      [ $var1 -eq 0 ]
do
    echo Inside the loop: $var1
    var1=$[ $var1 - 25 ]
done
```

Résultat - until Multiple

Sortie de test13

```
$ ./test13
100
Inside the loop: 100
75
Inside the loop: 75
50
Inside the loop: 50
25
Inside the loop: 25
0           # echo s'exécute avant l'arrêt final
$
```

Le shell exécute toutes les commandes de test et s'arrête uniquement quand la dernière commande devient vraie (retourne 0).

Boucles Imbriquées

Concept et précautions

Une boucle peut utiliser n'importe quelle commande, y compris d'autres boucles. C'est ce qu'on appelle une **boucle imbriquée**.

Attention : Les boucles imbriquées effectuent une itération dans une itération, ce qui multiplie le nombre d'exécutions des commandes.

```
$ cat test14
#!/bin/bash
# nesting for loops

for (( a = 1; a <= 3; a++ ))
do
    echo "Starting loop $a:"
    for (( b = 1; b <= 3; b++ ))
    do
       echo "   Inside loop: $b"
    done
done
```

Résultat - Boucles Imbriquées

Sortie de test14

```
$ ./test14
Starting loop 1:
   Inside loop: 1
   Inside loop: 2
   Inside loop: 3
Starting loop 2:
   Inside loop: 1
   Inside loop: 2
   Inside loop: 3
Starting loop 3:
   Inside loop: 1
   Inside loop: 2
   Inside loop: 3
$
```

La boucle interne itère complètement pour chaque itération de la boucle externe.

Mélange de Types de Boucles

for dans while

On peut imbriquer différents types de boucles.

```
$ cat test15
#!/bin/bash
# placing a for loop inside a while loop

var1=5

while [ $var1 -ge 0 ]
do
    echo "Outer loop: $var1"
    for (( var2 = 1; $var2 < 3; var2++ ))
    do
       var3=$[ $var1 * $var2 ]
       echo "  Inner loop: $var1 * $var2 = $var3"
    done
    var1=$[ $var1 - 1 ]
done
```

Résultat - for dans while (1/2)

Sortie de test15

```
$ ./test15
Outer loop: 5
  Inner loop: 5 * 1 = 5
  Inner loop: 5 * 2 = 10
Outer loop: 4
  Inner loop: 4 * 1 = 4
  Inner loop: 4 * 2 = 8
Outer loop: 3
  Inner loop: 3 * 1 = 3
  Inner loop: 3 * 2 = 6
```

Résultat - for dans while (2/2)

Suite de la sortie

```
Outer loop: 2
  Inner loop: 2 * 1 = 2
  Inner loop: 2 * 2 = 4
Outer loop: 1
  Inner loop: 1 * 1 = 1
  Inner loop: 1 * 2 = 2
Outer loop: 0
  Inner loop: 0 * 1 = 0
  Inner loop: 0 * 2 = 0
$
```

Le shell distingue correctement les commandes `do` et `done` de chaque boucle.

Combinaison until et while

Calculs avec bc

On peut même combiner les boucles **until** et **while**.

```
$ cat test16
#!/bin/bash
# using until and while loops

var1=3
until [ $var1 -eq 0 ]
do
    echo "Outer loop: $var1"
    var2=1
    while [ $var2 -lt 5 ]
    do
       var3=$(echo "scale=4; $var1 / $var2" | bc)
       echo "   Inner loop: $var1 / $var2 = $var3"
       var2=$[ $var2 + 1 ]
    done
    var1=$[ $var1 - 1 ]
done
```

Résultat - until/while (1/2)

Sortie de test16

```
$ ./test16
Outer loop: 3
   Inner loop: 3 / 1 = 3.0000
   Inner loop: 3 / 2 = 1.5000
   Inner loop: 3 / 3 = 1.0000
   Inner loop: 3 / 4 = .7500
Outer loop: 2
   Inner loop: 2 / 1 = 2.0000
   Inner loop: 2 / 2 = 1.0000
   Inner loop: 2 / 3 = .6666
   Inner loop: 2 / 4 = .5000
Outer loop: 1
   Inner loop: 1 / 1 = 1.0000
   Inner loop: 1 / 2 = .5000
   Inner loop: 1 / 3 = .3333
   Inner loop: 1 / 4 = .2500
$
```

Important : Chaque boucle doit modifier la variable testée, sinon vous obtenez une boucle infinie !

Itération sur Fichiers

Combinaison de techniques

Souvent, il faut itérer sur des éléments stockés dans un fichier. Cela nécessite de combiner deux techniques :

- Utilisation de boucles imbriquées
- Changement de la variable d'environnement IFS

Exemple classique : Traitement du fichier `/etc/passwd`

Stratégie :

1. Itérer ligne par ligne (IFS=`\n`)
2. Pour chaque ligne, changer IFS en deux-points (:)
3. Extraire les composants individuels

Script de Parsing /etc/passwd

Double boucle avec IFS

```
#!/bin/bash
# changing the IFS value

IFS.OLD=$IFS
IFS=$'\n'           # Lecture ligne par ligne
for entry in $(cat /etc/passwd)
do
    echo "Values in $entry -"
    IFS=:           # Séparation par deux-points
    for value in $entry
    do
       echo "   $value"
    done
done
```

Résultat - Parsing passwd

Extraction des champs

```
Values in ilias:x:1000:1000:Ilias Tougui:/home/ilias:/bin/bash -
   ilias
   x
   1000
   1000
   Ilias Tougui
   /home/ilias
   /bin/bash
Values in amine:x:1002:1002:Amine Dev:/home/amine:/bin/zsh -
   amine
   x
   1002
   1002
   Amine Dev
   /home/amine
   /bin/zsh
```

Cette technique est excellente pour traiter des données CSV (séparées par virgules).

Contrôle des Boucles

Commandes break et continue

Une fois une boucle démarrée, vous n'êtes pas obligé de la laisser terminer toutes ses itérations.

Deux commandes de contrôle :

- **break** : Sortir complètement d'une boucle
- **continue** : Sauter le reste de l'itération courante, mais continuer la boucle

Ces commandes permettent de contrôler finement l'exécution de vos boucles.

La Commande break

Sortie simple d'une boucle

La commande break permet de sortir d'une boucle en cours, quel que soit son type (for, while, until).

```
$ cat test17
#!/bin/bash
# breaking out of a for loop

for var1 in 1 2 3 4 5 6 7 8 9 10
do
    if [ $var1 -eq 5 ]
    then
       break
    fi
    echo "Iteration number: $var1"
done
echo "The for loop is completed"
```

Résultat - break Simple

Sortie de test17

```
$ ./test17
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
The for loop is completed
$
```

Normalement, la boucle aurait dû itérer sur toutes les valeurs. Mais quand la condition if-then est remplie, **break** arrête la boucle.

break dans while

Fonctionne avec toutes les boucles

La commande **break** fonctionne aussi avec les boucles while et until.

```
$ cat test18
#!/bin/bash
# breaking out of a while loop

var1=1

while [ $var1 -lt 10 ]
do
    if [ $var1 -eq 5 ]
    then
       break
    fi
    echo "Iteration: $var1"
    var1=$[ $var1 + 1 ]
done
echo "The while loop is completed"
```

Résultat - break while

Sortie de test18

```
$ ./test18
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
The while loop is completed
$
```

La boucle while se termine quand la condition if-then est remplie et que **break** est exécuté.

break dans Boucle Interne

Sortie de la boucle la plus interne

Avec plusieurs boucles, **break** termine automatiquement la boucle la plus interne dans laquelle vous vous trouvez.

```
$ cat test19
#!/bin/bash
# breaking out of an inner loop

for (( a = 1; a < 4; a++ ))
do
    echo "Outer loop: $a"
    for (( b = 1; b < 100; b++ ))
    do
       if [ $b -eq 5 ]
       then
          break
       fi
       echo "   Inner loop: $b"
    done
done
```

Résultat - break Interne

Sortie de test19

```
$ ./test19
Outer loop: 1
   Inner loop: 1
   Inner loop: 2
   Inner loop: 3
   Inner loop: 4
Outer loop: 2
   Inner loop: 1
   Inner loop: 2
   Inner loop: 3
   Inner loop: 4
Outer loop: 3
   Inner loop: 1
   Inner loop: 2
   Inner loop: 3
   Inner loop: 4
$
```

Même si la boucle interne s'arrête, la boucle externe continue normalement.

break avec Niveau

Sortie de boucles multiples

La commande **break** accepte un paramètre pour spécifier le niveau de boucle à quitter. où `n` indique le niveau de la boucle à quitter (par défaut `n = 1`).

```
break n.
```

Exemple - sortie de la boucle externe :

```
$ cat test20
#!/bin/bash
# breaking out of an outer loop

for (( a = 1; a < 4; a++ ))
do
    echo "Outer loop: $a"
    for (( b = 1; b < 100; b++ ))
    do
       if [ $b -gt 4 ]
       then
          break 2      # Sort des 2 niveaux
       fi
       echo "   Inner loop: $b"
    done
done
```

Résultat - break 2

Sortie de test20

```
$ ./test20
Outer loop: 1
   Inner loop: 1
   Inner loop: 2
   Inner loop: 3
   Inner loop: 4
$
```

Quand le shell exécute la commande `break 2`, la boucle externe s'arrête aussi.

La Commande continue

Sauter une itération

La commande **continue** permet d'arrêter prématurément le traitement des commandes dans une boucle, mais sans terminer complètement la boucle.

Exemple - utilisation dans for :

```
$ cat test21
#!/bin/bash
# using the continue command

for (( var1 = 1; var1 < 15; var1++ ))
do
    if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
    then
       continue
    fi
    echo "Iteration number: $var1"
done
```

Résultat - continue

Sortie de test21

```
$ ./test21
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
Iteration number: 10
Iteration number: 11
Iteration number: 12
Iteration number: 13
Iteration number: 14
$
```

Quand la condition est remplie (valeur > 5 et < 10), le shell exécute **continue**, qui saute les commandes restantes mais poursuit la boucle.

Danger avec continue et while

Attention à l'incrémentation

Soyez extrêmement prudent avec **continue** dans les boucles while et until !

```
$ cat badtest3
#!/bin/bash
var1=0

while echo "while iteration: $var1"
      [ $var1 -lt 15 ]
do
    if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
    then
       continue    # PROBLÈME : saute l'incrémentation !
    fi
    echo "   Inside iteration number: $var1"
    var1=$[ $var1 + 1 ]    # Jamais exécuté si continue
done
```

Résultat - Boucle Infinie

Sortie de badtest3

```
$ ./badtest3 | less
while iteration: 0
   Inside iteration number: 0
while iteration: 1
   Inside iteration number: 1
...
while iteration: 5
   Inside iteration number: 5
while iteration: 6
while iteration: 6   # Boucle infinie !
while iteration: 6
while iteration: 6
while iteration: 6
...
```

La variable `var1` n'est jamais incrémentée quand **continue** est exécuté, créant une boucle infinie !

continue avec Niveau

Continuer une boucle externe

Comme **break**, la commande **continue** accepte un paramètre de niveau :

```
continue n
```

Exemple - continuer la boucle externe :

```
$ cat test22
#!/bin/bash
for (( a = 1; a <= 5; a++ )); do
    echo "Iteration $a:"
    for (( b = 1; b < 3; b++ )); do
       if [ $a -gt 2 ] && [ $a -lt 4 ]
       then
          continue 2    # Continue la boucle externe
       fi
       var3=$[ $a * $b ]
       echo "   The result of $a * $b is $var3"
    done
done
```

Résultat - continue 2

Sortie de test22

```
$ ./test22
Iteration 1:
   The result of 1 * 1 is 1
   The result of 1 * 2 is 2
Iteration 2:
   The result of 2 * 1 is 2
   The result of 2 * 2 is 4
Iteration 3:    # Aucune commande interne exécutée
Iteration 4:
   The result of 4 * 1 is 4
   The result of 4 * 2 is 8
Iteration 5:
   The result of 5 * 1 is 5
   The result of 5 * 2 is 10
$
```

L'itération 3 ne traite aucune commande de la boucle interne car `continue 2` saute directement à la prochaine itération de la boucle externe.

Redirection de Sortie

Traitement de la sortie d'une boucle

Vous pouvez rediriger ou envoyer par pipe la sortie d'une boucle en ajoutant la commande de traitement à la fin de la commande `done`.

Format général :

```
for file in /home/$USER/*
do
    if [ -d "$file" ]
    then
       echo "$file is a directory"
    elif
       echo "$file is a file"
    fi
done > output.txt
```

Au lieu d'afficher sur le moniteur, les résultats sont redirigés vers le fichier.

Exemple - Redirection vers Fichier

Script test23

Script avec redirection :

```
$ cat test23
#!/bin/bash
# redirecting the for output to a file

for (( a = 1; a < 10; a++ ))
do
    echo "The number is $a"
done > test23.txt
echo "The command is finished."
```

Résultat - Redirection

Sortie de test23

```
$ ./test23
The command is finished.
$ cat test23.txt
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
$
```

Le shell crée le fichier `test23.txt` et y redirige uniquement la sortie de la boucle for. L'echo après la boucle s'affiche normalement.

Pipe de Sortie de Boucle

Envoi vers une autre commande

Cette même technique fonctionne pour envoyer par pipe la sortie d'une boucle vers une autre commande.

```
$ cat test24
#!/bin/bash
# piping a loop to another command

for city in Marrakech "Casa Blanca" Fes Oujda Dakhla
do
    echo "$city is the next place to go"
done | sort
echo "This completes our travels"
```

Résultat - Pipe vers sort

Sortie de test24

```
$ ./test24
Casa Blanca is the next place to go
Dakhla is the next place to go
Fes is the next place to go
Marrakech is the next place to go
Oujda is the next place to go
This completes our travels
$
```

Les valeurs ne sont pas dans un ordre particulier dans la liste de la boucle for. La sortie est envoyée à `sort`, qui change l'ordre de la sortie.

Résumé du Chapitre

Points clés

Les boucles sont une partie intégrante de la programmation. Le shell bash fournit trois commandes de boucles :

- **for** : Itère sur une liste de valeurs (liste, variable, fichiers, commande)
- **while** : Boucle tant qu'une condition produit un code de sortie zéro
- **until** : Boucle tant qu'une condition produit un code de sortie non-zéro

Les boucles peuvent être combinées (imbriquées) et contrôlées avec **break** et **continue**.

Fonctionnalités Avancées

Ce que nous avons appris

Techniques importantes :

- La boucle for style C offre plus de flexibilité
- La variable IFS permet de contrôler les séparateurs de champs
- Les boucles peuvent être redirigées ou envoyées par pipe
- Les commandes break et continue contrôlent le flux des boucles
- Les boucles imbriquées multiplient les itérations

Chapitre suivant : Interaction avec l'utilisateur du script shell - comment fournir des données en temps réel à vos scripts.

Partie II: Paramètres de Ligne de Commande

Contexte

Interaction avec l'utilisateur

Jusqu'à présent, vous avez vu comment les scripts interagissent avec les données, les variables et les fichiers sur le système Linux.

Nouveau besoin : Parfois, votre script doit interagir avec la personne qui l'exécute.

Méthodes d'entrée en Bash :

- **Paramètres de ligne de commande** : données ajoutées après la commande
- **Options de ligne de commande** : lettres simples qui modifient le comportement
- **Entrée au clavier** : capacité à lire l'entrée directement

Ce chapitre couvre comment incorporer les **paramètres de ligne de commande** dans vos scripts bash.

Paramètres de Ligne de Commande

Introduction et concept

Méthode la plus basique : Passer des données à un script en utilisant des paramètres de ligne de commande.

Exemple :

```
$ ./adder.sh 10 30
```

Cet exemple passe deux paramètres de ligne de commande (10 et 30) au script `adder.sh`.

Mécanisme : Le script traite les paramètres en utilisant des variables spéciales, appelées **paramètres positionnels**.

Variables Positionnelles

Accès aux paramètres

Assignation automatique par le shell :

- `$0` = le nom du script
- `$1` = premier paramètre
- `$2` = deuxième paramètre
- ... et ainsi de suite, jusqu'à `$9`

Important : Le shell assigne automatiquement les paramètres aux variables. Vous n'avez rien à faire !

Exemple simple - un seul paramètre :

```
#!/bin/bash
factorial=1
for (( number = 1; number <= $1; number++ )); do
   factorial=$[ $factorial * $number ]
done
echo The factorial of $1 is $factorial
```

Résultat - Paramètre Unique

Utilisation de $1

```
$ ./test1.sh 5
The factorial of 5 is 120
$
```

Vous pouvez utiliser la variable `$1` exactement comme n'importe quelle autre variable dans le script.

Paramètres Multiples

Utilisation de $1 et $2

Exemple avec deux paramètres :

```
$ cat test2.sh
#!/bin/bash
# testing two command line parameters

total=$[ $1 * $2 ]
echo The first parameter is $1.
echo The second parameter is $2.
echo The total value is $total.

$ ./test2.sh 2 5
The first parameter is 2.
The second parameter is 5.
The total value is 10.
```

Chaque paramètre doit être **séparé par un espace** sur la ligne de commande.

Paramètres Textes

Utilisation de chaînes de caractères

Vous pouvez aussi utiliser des chaînes de caractères :

```
$ cat test3.sh
#!/bin/bash
# testing string parameters

echo Hello $1, glad to meet you.
$ ./test3.sh Students
Hello Students, glad to meet you.
$
```

Le shell passe la valeur texte entrée sur la ligne de commande au script.

Paramètres avec Espaces

Problème et solution

Problème : Si vous essayez d'utiliser du texte avec des espaces sans guillemets :

```
$ ./test3.sh Dear Students
Hello Dear, glad to meet you.
```

Le shell interprète l'espace comme séparant deux valeurs distinctes !

Solution : Encadrer la valeur avec des guillemets :

```
$ ./test3.sh 'Dear Students'
Hello Dear Students, glad to meet you.
$ ./test3.sh "Dear Students"
Hello Dear Students, glad to meet you.
$
```

Les guillemets ne font pas partie des données.

Plus de 9 Paramètres

Utilisation des accolades

Après le neuvième paramètre, vous devez utiliser des **accolades** autour du numéro : `${10}`, `${11}`, etc.

Exemple :

```
$ cat test4.sh
#!/bin/bash
# handling lots of parameters
total=$[ ${10} * ${11} ]
echo The tenth parameter is ${10}
echo The eleventh parameter is ${11}
echo The total is $total

$ ./test4.sh 1 2 3 4 5 6 7 8 9 10 11 12
The tenth parameter is 10
The eleventh parameter is 11
The total is 110
```

Cette technique vous permet d'ajouter autant de paramètres que vous le souhaitez.

Le Paramètre $0

Accès au nom du script

Le paramètre $0 contient le nom du script exécuté sur la ligne de commande.

Utilité : Utile pour créer un utilitaire avec plusieurs fonctions.

```
$ cat test5.sh
#!/bin/bash
# Testing the $0 parameter

echo The zero parameter is set to: $0
$ bash test5.sh
The zero parameter is set to: test5.sh
```

Attention : Le chemin peut être inclus dans $0 :

```
$ ./test5.sh
The zero parameter is set to: ./test5.sh
$ /home/ilias/test5.sh
The zero parameter is set to: /home/ilias/test5.sh
```

Extraction du Nom - basename

Suppression du chemin

Solution : La commande `basename` retourne uniquement le nom du script sans le chemin.

```
$ cat test5b.sh
#!/bin/bash
# Using basename with the $0 parameter

name=$(basename $0)
echo
echo The script name is: $name

$ /home/ilias/test5b.sh
The script name is: test5b.sh
$ ./test5b.sh
The script name is: test5b.sh
```

Avantage : Vous pouvez créer des scripts multi-fonctions basés sur le nom utilisé.

Script Multi-fonctions

Utilisation de basename

Script s'adaptant selon son nom :

```
$ cat test6.sh
#!/bin/bash
# Testing a Multi-function script

name=$(basename $0)

if [ $name = "addem" ]
then
   total=$[ $1 + $2 ]
elif [ $name = "multem" ]
then
   total=$[ $1 * $2 ]
fi

echo
echo The calculated value is $total
$
```

Résultat - Script Multi-fonctions

Test avec deux noms différents

```
$ cp test6.sh addem
$ chmod u+x addem
$ ln -s test6.sh multem
$ ./addem 2 5
The calculated value is 7
$ ./multem 2 5
The calculated value is 10
$
```

Le script détermine son nom de base et exécute la fonction appropriée basée sur cette valeur.

Tester les Paramètres

Importance de la vérification

Danger : Si le script s'exécute sans les paramètres attendus :

```
$ ./addem 2
./addem: line 8: 2 +  : syntax error: operand expected
The calculated value is
$
```

Bonne pratique : Toujours vérifier que les paramètres existent avant de les utiliser.

```
$ cat test7.sh
#!/bin/bash
# testing parameters before use

if [ -n "$1" ]
then
   echo Hello $1, glad to meet you.
else
   echo "Sorry, you did not identify yourself."
fi
```

Compter les Paramètres

Variable $#

Variable spéciale $# : Contient le nombre de paramètres de ligne de commande fournis.

Utilité : Vérifier le nombre exact de paramètres avant de les utiliser.

```
$ cat test8.sh
#!/bin/bash
# getting the number of parameters

echo There were $# parameters supplied.
$ ./test8.sh
There were 0 parameters supplied.
$ ./test8.sh 1 2 3 4 5
There were 5 parameters supplied.
$
```

Vérification du Nombre

Utilisation de $# dans une condition

Exemple - tester le nombre correct de paramètres :

```
$ cat test9.sh
#!/bin/bash
# Testing parameters

if [ $# -ne 2 ]
then
   echo
   echo Usage: test9.sh a b
   echo
else
   total=$[ $1 + $2 ]
   echo
   echo The total is $total
   echo
fi
```

Résultat - Vérification

Sortie de test9.sh

```
$ bash test9.sh
Usage: test9.sh a b
$ bash test9.sh 10
Usage: test9.sh a b
$ bash test9.sh 10 15
The total is 25
$
```

Si le nombre correct de paramètres n'est pas présent, un message d'erreur s'affiche.

Dernier Paramètre

Accès sans connaître le nombre

Astuce : Accéder au dernier paramètre sans connaître le nombre total.

Syntaxe : `$!#` (note : exclamation, pas $$)

```
$ cat test10.sh
#!/bin/bash
# Grabbing the last parameter

params=$#
echo
echo The last parameter is $params
echo The last parameter is $!#
echo
$ bash test10.sh 1 2 3 4 5
The last parameter is 5
The last parameter is 5
$
```

Tous les Paramètres - $* et $@

Accès à tous les paramètres

Les variables `$*` et `$@` fournissent un accès facile à tous les paramètres.

Différences importantes :

- `$*` : tous les paramètres comme une **seule** chaîne
- `$@` : tous les paramètres comme des **mots séparés**

À première vue, elles semblent identiques :

```
$ ./test11.sh ilias amine amal
Using the $* method: ilias amine amal
Using the $@ method: ilias amine amal
$
```

Différence $* vs $@

Utilisation avec une boucle

La différence devient évidente avec une boucle :

```
$ cat test12.sh
#!/bin/bash
# testing $* and $@

for param in "$*"
do
   echo "$* Parameter #$count = $param"
   count=$[ $count + 1 ]
done

echo
count=1
for param in "$@"
do
   echo "$@ Parameter #$count = $param"
   count=$[ $count + 1 ]
done
$
```

Résultat - $ vs $@

Différence dans le traitement

```
$ ./test12.sh ilias amine amal
$* Parameter #1 = ilias amine amal
$@ Parameter #1 = ilias
$@ Parameter #2 = amine
$@ Parameter #3 = amal
$
```

`$*` traite tous les paramètres comme un seul, tandis que `$@` traite chacun séparément !