

ECRITURE DE SCRIPTS SHELL SOUS UNIX/LINUX

Plan

2

- Scripts bash
 - ▣ Définition
 - ▣ Structure du script
 - ▣ Exécution du script
 - ▣ Entrées-sorties
- Les variables
 - ▣ Utilisateur
 - ▣ Système/d'Environnement
 - ▣ Spéciales
- Manipulation des chaînes de caractères
- La commande test
- Calcul arithmétique
- Tests et débogages

Script Shell (bash)

3

- ❑ Le shell (ou interface système) est un programme qui reçoit des commandes par un utilisateur à partir du clavier pour les envoyer au système d'exploitation qui se chargera de les exécuter.
- ❑ Tous les Shell peuvent exécuter des commandes situées dans un fichier. Chaque fichier contenant des commandes destinées au shell se nomme un **script**
- ❑ Un script Shell est un ensemble de commandes dans un fichier
- ❑ Un script est un programme écrit dans le langage propre au shell.

Shell: script

4

- ❑ Shell Scripting aide à l'automatisation de certaines tâches (comme l'édition de tâches répétitives d'administration système)
- ❑ Les scripts Shell sont interprétés par le Shell pendant qu'ils sont lus
- ❑ Un script Shell utilise des structures de contrôle et des filtres
- ❑ Dans un script Shell, un filtre lit des entrées, leur applique une transformation puis génère une sortie

Scripts bash

5

- Nom du fichier script0.sh:
- Extension .sh : par convention pour indiquer que c'est un script shell, mais ce n'est obligatoire
- Par convention les scripts shell se terminent généralement par « .sh » pour le Bourne Shell et le Bourne Again Shell, par « .ksh » pour le Korn Shell et par « .csh » pour le C Shell

Scripts bash: Structure du script

6

- Toutes les instructions et les commandes sont regroupées au sein d'un script (fichier texte).
- Les commentaires : Commencent par un `#` ; Un commentaire peut être placé en fin d'une ligne comportant déjà des commandes
- Le code shell: suite de commandes bash organisées pour accomplir une ou plusieurs tâches
- Le shebang, représenté par `#!/`, est un en-tête d'un fichier texte qui indique au système d'exploitation (de type Unix) que ce fichier est un script (ensemble de commandes)
 - ▣ Toujours positionnés au tout début du fichier
 - ▣ Bien que non indispensable, la ligne `#!/bin/bash` permet de s'assurer que le script est bien exécuté avec le bon shell bash
 - ▣ Exemple:
 - `#!/bin/sh`
 - `#!/bin/bash`
 - `#!/usr/bin/perl`

Scripts bash: Exécution

7

- Exécution avec la commande bash (ou le shell pour lequel le script a été créé)
 - ▣ `$ bash script0.sh`
 - ▣ Si nous sommes sur le répertoire contenant le script, sinon indiquer le chemin relatif ou absolu du script
 - ▣ Pas besoin qu'il soit exécutable
- Rendre le fichier exécutable et le lancer directement:
 - ▣ `$ chmod +x script0.sh ; $ ./ script0.sh`
 - ▣ `./` : indique comme chemin le répertoire courant si nous sommes sur le répertoire contenant le script
 - ▣ On peut aussi bien indiquer le chemin absolu à partir de la racine
- On peut aussi placer le script exécutable dans l'un des répertoires listés dans la variable PATH,
 - ▣ taper ensuite simplement : `$ script0.sh` pour l'exécuter

Scripts bash: Exécution

8

- ❑ Lors de son exécution, les lignes seront lues une à une et exécutées
- ❑ Une ligne peut se composer de commandes internes ou externes, de commentaires ou être vide
- ❑ Plusieurs instructions par lignes sont possibles, séparées par le « ; » ou liées logiquement par « && » ou « || »
- ❑ Le « ; » est l'équivalent d'un saut de ligne
- ❑ La première ligne possède une importance particulière (Shebang) car elle permet de préciser quel shell va exécuter le script :
 - ❑ `#!/bin/sh`
 - ❑ `#!/bin/bash`

9 Démarrer un premier Script

Tapez dans un Editeur de texte le code suivant dans un fichier

Démarrons un premier script bash

10

- Via un éditeur de votre choix créer le fichier script0.sh suivant

```
$cat script0.sh
#!/bin/bash
# script0.sh
echo "Hello Evryone "
echo -n "La date du jour est: " ;
date
echo -n le répertoire courant est ; pwd
echo -n je suis l'utilisateur; whoami
exit
```

Entrées-sorties

11

- les voies de communication entre le programme Bash et la console
 - echo : affiche son argument texte entre guillemets sur la sortie standard, c-à-d l'écran
- La validation d'une commande echo provoque un saut de ligne
 - \$ echo "Bonjour à tous !"
- On peut insérer les caractères spéciaux habituels (language C)
 - ▣ ils seront interprétés seulement si l'option -e est utilisé echo
 - ▣ \n (saut ligne), \b (retour arrière), \t (tabulation), \a (alarme), \c (fin sans saut de ligne)
 - \$ echo "Bonjour \nà tous ! "
 - \$ echo -e "Bonjour \nà tous ! "
 - \$ echo -e "Bonjour \nà toutes \net à tous ! \c"

Entrées-sorties

12

□ Séquences d'échappement

<code>\a</code>	alerte (avertisseur sonore)
<code>\b</code>	retour arrière (backspace)
<code>\c</code>	suppression du retour-chariot final
<code>\f</code>	saut de page
<code>\n</code>	nouvelle ligne
<code>\r</code>	retour-chariot
<code>\t</code>	tabulation horizontale
<code>\v</code>	tabulation verticale
<code>\\</code>	backslash
<code>\nnn</code>	le caractère dont le code ASCII octal vaut nnn (un à trois chiffres)
<code>\xnnn</code>	le caractère dont le code ASCII hexadécimal vaut nnn (un à trois chiffres)

Tester echo

13

```
$cat script1.sh
#!/bin/bash
# script0.sh
echo  "Bonjour \nà tous ! "
echo  -e      "Bonjour \nà tous ! "
echo  -e      "Bonjour \nà toutes \net à tous ! \c"
exit
```

Entrées-sorties

14

- **read** : permet l'affectation directe par lecture de la valeur saisie sur l'entrée standard au clavier

```
$ read      var1 var2 ...
```

- Attend la saisie au clavier d'une liste de valeurs pour les affecter, après la validation globale, respectivement aux variables var1, var2 ..

- **Exemple :**

- ▣ \$ echo "Donnez votre prénom et votre nom:"

- ▣ \$ read prenom nom

- ▣ ...

- ▣ \$ echo "Bonjour \$prenom \$nom "

- ▣ \$read -p "donner le nom " nom

Entrées-sorties

15

```
#!/bin/bash  
read -p 'Entrez votre nom : ' nom  
echo "Bonjour $nom !"
```

Remarque : on note que le message 'Entrez votre nom' a été entouré de quotes. Si on ne l'avait pas fait, le bash aurait considéré que chaque mot était un paramètre différent !

Résultat :

```
Entrez votre nom : Mathieu  
Bonjour Mathieu
```

Entrées-sorties

16

- `-n` : limiter le nombre de caractères. Avec `-n`, on peut au besoin couper au bout de `X` caractères si vous ne voulez pas que l'utilisateur insère un message trop long

Exemple :

```
#!/bin/bash
read -p 'Entrez votre login (5 caractères max) : ' -n 5 nom
echo "Bonjour $nom !"
```

Entrez votre login (5 caractères max) : mathiBonjour mathi !

On note que le bash coupe automatiquement au bout de 5 caractères sans avoir besoin d'appuyer sur la touche Entrée. Ce n'est pas très esthétique du coup, parce que le message s'affiche sur la même ligne.

Entrées-sorties

17

- **t** : limiter le temps autorisé pour saisir un message.
On peut définir un timeout avec **-t**, c'est-à-dire un nombre de secondes au bout duquel le **read** s'arrêtera

- **Exemples**

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

```
read -p 'Entrez le code de désamorçage de la bombe (vous  
avez 5 secondes) : ' -t 5 code
```

```
echo -e "\nBoum !"
```

Entrées-sorties

18

- **-s** : ne pas afficher le texte saisi. Le paramètre **-s** masque les caractères lors de la saisie. Cela servira notamment si l'utilisateur entre un mot de passe :

- **Exemple**

```
#!/bin/bash  
read -p 'Entrez votre mot de passe : ' -s pass  
echo "\nMerci ! Je vais dire que votre mot de passe est  
$pass! "
```

Entrez votre mot de passe :

*Merci ! Je vais dire à tout le monde que votre
mot de passe est supertopsecret38 ! :)*

Les variables: définition

19

- Permet d'associer une clé (une nom) à une valeur Mémoriser une donnée (le nom d'une variable est sensible à la casse)
- Aucun typage: aucun besoin de déclaration spécifique
- Toute valeur de variable est considérée comme une chaine de caractères
- Trois types de variables:
 - ▣ Utilisateur
 - ▣ Système (variables d'environnement)
 - ▣ Spéciales

Les variables : Nomenclature

20

Un nom de variable obéit à certaines règles:

- Il peut être composé de lettres minuscules, majuscules, de chiffres, de caractères de soulignement
- Le premier caractère ne peut pas être un chiffre
- Le taille d'un nom est en principe illimité (mais!!)
- Par conventions les variables utilisateur sont souvent en minuscules pour les différencier des variables système.

Les variables :Déclaration et affectation :

21

- Une variable est déclarée dès qu'une valeur lui est affectée.
- La valeur est généralement une chaîne de caractère, ou des valeurs numériques
- L'affectation est effectuée avec le signe « = », sans espace avant et après le signe
 - ▣ Syntaxe : `variable=valeur`
 - ▣ Exemple : `variable1=Bonjour`
- On peut aussi créer des variables vides. Celle-ci existera mais sans contenu.
 - ▣ Exemple : `variable2=`

Substitution de variable

22

- Pour accéder à une variable on utilise le caractère \$ devant le nom
- Exemple :
 - ▣ `$ echo $variable1`
Bonjour
 - ▣ `$ echo ${variable1}`
Bonjour !
- Quand le Shell rencontre le \$, il tente d'interpréter le mot suivant comme étant une variable.
- Si elle existe, alors le `$nom_variable` (`$variable1`) est remplacé (substitution) par son contenu, ou par un texte vide dans le cas contraire.
- Afficher le contenu d'une variable: la commande echo

Les variables

23

- **Exemple :**

```
$ echo $variable1  
Bonjour
```
- Une variable peut contenir des caractères spéciaux notamment des espaces. Il faut dans ce cas soit verrouiller les caractères spéciaux un par un, soit de les mettre entre guillemets ou apostrophes
- La principale différence entre les guillemets et les apostrophes est l'interprétation des variables et des substitutions
- **Exemple :**
 - `echo " $variable1 "` Affiche Bonjour
 - `echo '$variable1'` Affiche \$var1

Les variables: Portée

24

On distingue:

- Par défaut : **locales**

- ▣ N'est définie et disponible que pour l'instance de bash
- ▣ N'est pas accessible par les autres programmes

- Variables d'environnement

- ▣ Accessibles par tous les programmes au sein d'un même environnement utilisateur
- ▣ Tous les programmes lancés par l'utilisateur-même ou par un des programmes qu'il ait lancé
- ▣ Par convention, les variables d'environnement sont en majuscules

Les variables : Exportation

25

- Par défaut une variable est **locale** n'est accessible que depuis le shell où elle a été définie
- Pour qu'elle devienne **globale** (d'environnement) elle doit être **exportée** avec la commande :

\$ export variable

- La commande **export** permet d'exporter une variable de manière à ce que son contenu soit visible par les scripts et autres sous-shells
- Les variables exportées peuvent être modifiées dans le script, mais ces modifications ne s'appliquent qu'au script ou au sous-shells
- Attention ! l'argument de la commande export est bien la référence (le nom) de la variable Pas sa valeur (Ne pas utiliser le caractère \$)

Les variables :Suppression et protection

26

- On supprime une variable avec la commande **unset**

unset nom_variable

- Exemple: **Unset variable1**
- La valeur de la variable est alors nulle
- à ne pas confondre avec une variable contenant une chaîne vide

- On peut protéger une variable en écriture et contre sa suppression avec la commande **readonly**

- Une variable en lecture seule même vide est figée
- Il n'existe aucun moyen de la remplacer en écriture et de la supprimer, sauf quitter le shell

Variables d'environnement

27

- Les systèmes Unix/Linux permettent de définir un environnement d'exécution pour chaque programme en cours d'exécution
- L'environnement comporte un ensemble de **paramètres**, ce sont les **variables d'environnement**
- Les variables d'environnement sont des variables dynamiques utilisées par les différents processus d'un système d'exploitation (Windows, Unix, Linux, etc.)
- Elles affectent le comportement des programmes et servent à communiquer des informations entre programmes qui ne se trouvent pas sur la même ligne hiérarchique

Variables d'environnement

28

- Chaque variable d'environnement contient une valeur de type chaîne de caractères, dont la signification lui est propre
- Il est d'usage que les noms des variables d'environnement soient écrits complètement en majuscules, mais ce n'est pas une obligation
- Chaque programme est susceptible de reconnaître un certain nombre de variables d'environnement qui lui sont propres, mais il existe également des variables standards que tous les programmes utilisent.

Variables d'environnement

29

- Exemple: la variable d'environnement **PATH**
 - ▣ Contient la liste des répertoires dans lesquels le système doit rechercher les programmes à exécuter
 - ▣ Exemple de contenu de la variable PATH :
`/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin`
- Cette variable PATH permet donc de lancer les programmes en tapant simplement leur nom, et de laisser le système rechercher le fichier de ce programme dans chacun des répertoires indiqués dans cette variable

Variables d'environnement

30

Variables d'environnement prédéfinies

- ❑ **HOME** chemin d'accès au répertoire initial (personnel) de l'utilisateur
- ❑ **PATH** suite de chemins d'accès aux répertoires des exécutables
- ❑ **PS1** invite principale du Shell en mode interpréteur, Variable contenant une chaîne de texte réalisant la mise en forme du message d'attente (prompt) en ligne de commandes. Par exemple `[\u@\h \W]\$`
- ❑ **PS2** invite secondaire du Shell en mode programmation
- ❑ **IFS** séparateurs de champ des arguments
- ❑ **MAIL** chemin d'accès à la boîte aux lettres utilisateur
- ❑ **MAILCHECK** intervalle en sec au bout duquel le mail est contrôlé

Variables d'environnement

31

Variables d'environnement prédéfinies

- **CDPATH** liste de chemins d'accès pour la commande `cd`
- **ENV** nom du fichier des variables d'environnement
- **TERM** nom du type de terminal
- **USER** Nom de login de l'utilisateur. Cette information est également disponible au travers de la variable d'environnement **LOGNAME**
- **SHELL** Chemin de l'exécutable du Shell actuellement utilisé. Sous Linux, il s'agit souvent du Shell **BASH** (le contenu sera donc `/bin/bash`)
- **PWD** Cette variable contient le répertoire de travail actuel

Variables d'environnement

32

- Par défaut, les programmes sont lancés avec l'environnement du programme qui les lance, c'est-à-dire dans la plupart des cas l'environnement d'exécution du Shell
- Les programmes peuvent également définir de nouvelles variables d'environnement, qui seront ainsi accessibles par les programmes qu'ils lanceront eux-mêmes
- Comme tout programme, le Shell dispose d'un environnement qu'il utilise pour stocker ses propres variables
- Le Shell utilise donc deux environnements différents :
 - ▣ Son propre environnement, qui contient les variables locales à la session du Shell en cours
 - ▣ L'environnement d'exécution, dont les variables d'environnement sont transmises aux programmes que le Shell lance

Variables d'environnement

33

Création d'une variable :

- Pour définir une variable du Shell, il suffit de lui affecter une valeur, à l'aide de la syntaxe suivante : `$Nom_VARIABLE=valeur`
- Il n'est pas nécessaire de fournir une valeur.
- Les variables ainsi définies ne font partie que de l'environnement du Shell, et ne sont donc pas accessibles aux programmes que le Shell lance !
- Exemple: `$ VAR="Hello World"`
 - ▣ Rq1:La valeur est encadrée entre guillemets car elle contient des espaces
 - ▣ Rq2 :Pas d'espaces entre le nom de la variable, le caractère = et la valeur affectée
- La valeur d'une variable peut être récupérée simplement en préfixant le nom de la variable du symbole dollar (\$) Exemple : `$ echo $VAR`
- Cette commande permet d'afficher le contenu de la variable VAR qui est : `Hello World`

Variables d'environnement

34

Création d'une variable d'environnement :

- Pour rendre une variable du Shell une variable d'environnement, et donc accessible aux programmes que celui-ci peut lancer, il faut l'exporter dans l'environnement d'exécution

- Cela peut être réalisé avec la commande **export** :

\$export Nom_VARIABLE

- Il est possible de déclarer et exporter une variable en une seule commande, la syntaxe est la suivante :

\$export Nom_VARIABLE=valeur

- Une variable peut être supprimée à l'aide de la commande **unset**
- Cette commande prend en paramètre le nom de la variable à supprimer,

\$unset Nom_VARIABLE

Variables d'environnement

35

- La commande **set** permet d'afficher toutes les variables du Shell (les variables d'environnement et les autres variables) ainsi que leurs contenus respectifs
- La commande **env**, sans aucun paramètre, permet de lister uniquement les variables d'environnement (exportées) ainsi que leurs contenus respectifs
- La commande **printenv** permet d'afficher les noms et les valeurs de toutes les variables d'environnement
- Les variables définies au cours d'une session seront perdues lorsque la session est terminée
- Si l'on définit une variable et on l'exporte, puis on redémarre la session, cette variable sera perdue.

Variables d'environnement

36

- Pour créer ou modifier une variable d'environnement de manière permanente pour le Shell Bash (comme la variable PATH), on peut définir cette variable :
 - ▣ Dans le fichier d'initialisation global du Shell `/etc/profile` : dans ce cas, la variable sera définie dans toutes les sessions de tous les utilisateurs
 - ▣ Dans le fichier `~/.bash_profile` qui sera chargé au lancement de la session de l'utilisateur intéressé : dans ce cas, la variable aura une portée restreinte, elle sera définie uniquement dans l'environnement de l'utilisateur en question

Les variables d'environnement

37

- En plus des variables que l'utilisateur peut définir lui-même, le shell est lancé avec un certain nombre de variables prédéfinies, utiles pour un certain nombre de commandes et accessibles par l'utilisateur
- Le contenu de ces variables Système/Environnement peut être modifié mais il faut alors faire attention car certaines ont une incidence directe sur le comportement du système
- La liste de ces variables est consultable par la commande `env`
- Les plus utilisées sont:
`$HOME`, `$PATH`, `$USER`, `$PS1`, `$SHELL`, `$ENV`, `$PWD` ...

Les variables d'environnement

38

- Variables d'environnement prédéfinies
- **HOME** chemin d'accès au répertoire initial de l'utilisateur
- **PATH** suite de chemins d'accès aux répertoires des exécutables
- **PS1** invite principale du shell en mode interpréteur
- **PS2** invite secondaire du shell en mode programmation
- **IFS** séparateurs de champ des arguments
- **MAIL** chemin d'accès à la boîte aux lettres utilisateur
- **MAILCHECK** intervalle en sec au bout duquel le mail est contrôlé
- **CDPATH** liste de chemins d'accès pour la commande cd
- **ENV** nom du fichier des variables d'environnement
- **TERM** nom du type de terminal

Variables spéciales:

39

- Elles sont gérées par le système et très utiles dans les scripts
- Elles ne sont accessibles qu'en lecture
- Ces variables, **paramètres positionnels** sont automatiquement affectées lors d'un appel de script suivi d'une liste de paramètres.
 - ▣ Leurs valeurs sont récupérables dans \$#, \$0, \$1, \$2 ...\$9, ..
 - ▣ Doivent être indiquées avec les accolades au delà de 9

Variables spéciales

40

- **\$?** : C'est la valeur de sortie (code retour) de la dernière commande. Elle vaut 0 si la commande s'est déroulée sans problème. Et elle est différente de 0 dans le cas contraire
- **\$0** Cette variable contient le nom du script
- **\$1 à \$9** : Les (éventuels) premiers arguments passés à l'appel du script. Appelées arguments positionnels
- **\$#** : Le nombre d'arguments positionnels passés au script
- **\$*** : Liste de tous les paramètres à partir de \$1 au format "\$1 \$2 \$3 . . ."
- **\$@** : Liste des paramètres sous forme d'éléments distincts "\$1" "\$2" "\$3" ...
- **\$\$** : le n° PID (Process ID) du processus courant
- **\$_** : le n° PID (Process ID) du processus fils

Variables spéciales: Exemple

41

```
$ ./affiche -e "Bonjour !"
```

- \$0 est égale à "affiche"
- \$1 à "-e"
- \$2 à "Bonjour !"
- \$* à "-e 'Bonjour !'"
- \$# à 2

Variables spéciales

42

Réorganisation des paramètres :

- La commande `shift` : permet de modifier la structure des paramètres de position
- Un appel simple décale tous les paramètres d'une position en supprimant le premier : `$2` devient `$1`, `$3` devient `$2` et ainsi de suite Le `$1` originel disparaît
- `$#`, `$*` et `$@` sont redéfinis en conséquence
- La commande `shift` suivie d'une valeur `n` effectue un décalage de `n` éléments

Variables spéciales

43

□ Exemples :

```
$ ls -l
```

```
$ echo $? #0
```

```
$ grep ro3533ot /etc/passwd
```

```
$ echo $? #1
```

□ Un exemple d'utilisation dans le script `exemple.sh` :

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

```
echo "Vous m'avez donné $# paramètres: "$@"
```

Variables spéciales (set)

44

- Donner des valeurs aux paramètres (set)
- Par la command `set`, donner les valeurs aux paramètres
- `$set val1 val2`
- Donne les valeurs `val1` et `val2` respectivement aux paramètres `$1` et `$2`

Les expressions arithmétiques

45

- Différentes méthodes permettent d'effectuer des calculs :
Calcul simple sur les entiers:
 - ▣ La commande **let**
 - ▣ Les parenthèses : La syntaxe spéciale **\$ ((operation))**
 - ▣ La commande **expr**

- Calcul avancé - Nombres flottants:
- L'outil **bc** : supporte des nombres de précision arbitraires

Les expressions arithmétiques

46

La commande **let** :

□ **Evaluation des expressions arithmétiques sur des variables let**

□ **Exemple**

```
x=5+2; echo "x =$x" #x= 7
```

```
let y=$x+3; echo "y = $y" #y= 10
```

Les expressions arithmétiques

47

Faire un calcul arithmétique avec **Les parenthèses** :

`(( expression ))`

Exemple

```
a=$((b=3+ 2)) ; echo "a= $a" ;
```

```
echo "b= $b"
```

```
a= 5    b= 5
```

- Dans sa forme la plus simple, `a=$(( 5 + 3 ))`, exécutera le calcul `5 + 3`, soit 8, et attribuera sa valeur à la variable `a`
- La construction en double parenthèse est aussi un mécanisme permettant la manipulation de variables à la manière du C dans Bash, par exemple `(( var++ ))`

Les expressions arithmétiques

48

- **La commande `expr` :**

- `expr $nbr1 opérateur $nbr2`

- `expr $chaîne` : expression régulière

- **Attention à l'interprétation de certains caractères spéciaux !!**

- **Exemple : Les expressions arithmétiques**

```
d=`expr 6 + 7`; echo "d = $d"
```

```
d=`expr 6+7`; echo "d = $d"
```

NB: sans les espaces ça sera interprétée comme chaîne de caractères.

Les arguments de la commande `expr` doivent toujours être séparés par au moins un espace ou une tabulation

Le calcul arithmétique

49

□ Opérateurs arithmétiques

Opérateur	Description
+	addition
-	soustraction
/	division
*	multiplication
**	puissance (Bash 2.02 et supérieurs)
%	modulo (reste de division entière)

- Bash ne comprend pas l'arithmétique à virgule flottante: interprète donc les nombres décimaux comme des chaînes de caractères
- Toute opération sera donc toujours entière
- Pour des calculs complexes, on utilise l'outil **bc**

Le calcul arithmétique

50

- Quelques exemples :
 - ▣ `let "a = 5 * 3" # $a = 15`
 - ▣ `let "a = 4 ** 2" # $a = 16` (4 au carré)
 - ▣ `let "a = 8 / 2" # $a = 4`
 - ▣ `let "a = 10 / 3" # $a = 3`
 - ▣ `let "a = 10 % 3" # $a = 1`

Le calcul arithmétique

51

- **Affectation arithmétique** : Associer affectation et opérateur arithmétique pour modifier la valeur d'une variable par le résultat d'une opération entre sa valeur précédente et une constante .

Opérateur	Description	
+=	incrémentatation	$((a+=2)) \rightarrow ((a=\$a+2)) \rightarrow a=\$((\$a+2))$
-=	décrémentatation	$((a-=2)) \rightarrow ((a=\$a-2)) \rightarrow a=\$((\$a-2))$
/=	affectation par division	$((a/=2)) \rightarrow a=((\$a/2))$
=	affectation par multiplication	$((a=2)) \rightarrow a=((\$a*2))$
%=	affectation du reste de la division entière(affectation du modulo)	$((a\%=2)) \rightarrow ((a=\$a\%2))$

Le calcul arithmétique

52

□ Opérateurs binaires :

Opérateur	Description
<<	décalage d'un bit à gauche (multiplication par 2)
>>	décalage d'un bit à droite (division par 2)
&	ET binaire
	OU (inclusif) binaire
~	NON binaire
^	XOR (ou exclusif) binaire

Le calcul arithmétique : exemples

53

□ Calcul simple:

- ▣ `a=1 ; a=$(( $a + 1 )) ; echo $a`
- ▣ `a=1 ; let "a=$a + 1" ; echo $a`
- ▣ `a=1 ; let a++ ; echo $a`
- ▣ `a=`expr 6 + 1`; echo a = " $a"`
- ▣ `expr 20 + 5`
- ▣ `expr 20 + 5 / 2`
- ▣ `expr 20 + 5`
- ▣ `expr 20 + 5 / 2`

□ Calcul avancé:

- ▣ `echo 20+5/2 | bc`
- ▣ `echo 'scale=4;20+5/2' | bc`
- ▣ `echo "1/3" | bc -l`
- ▣ Mode interactif bc:
 - `$ bc`
 - `5.3+5`
 - `10.3`

Manipulation des chaînes de caractères

54

- ❑ **Longueur de chaînes de caractères** : le premier caractère d'une chaîne a pour indice 0
 - ❑ `${#var}` : longueur de la variable (en caractères)
 - ❑ `expr length $var`
 - ❑ **Exemple** : `var="abcde" ; echo ${#var} // affiche 5`

- ❑ **Extraction de sous-chaînes** :
 - ❑ `${chaîne:position}` : Extrait une sous-chaîne de \$chaîne à partir de \$position
`Phrase = "Pourtant elle tourne"`
`echo ${phrase:7} // affiche « elle tourne »`

 - ❑ `${chaîne:position:longueur}` : Extrait \$longueur caractères d'une sous- chaîne de \$chaîne à la position \$position
`echo ${phrase:7:5} : sous-chaîne de 5 caractère à partir du 8ème position // affiche elle`

Manipulation des chaînes de caractères

55

Suppression de sous-chaînes:

- `${chaine#souschaine}` : Supprime la correspondance la plus petite de `$souschaine` à partir du début de `$chaine`
Exemple : `echo ${ var# ab}`
- `${chaine##souschaine}` :
Supprime la correspondance la plus grande de `$souschaine` à partir du début de `$chaine`
- `${chaine%souschaine}` :
Supprime la plus petite correspondance de `$souschaine` à partir de la fin de `$chaine`
- `$`
- `{chaine%%souschaine}` :
Supprime la plus grande correspondance de `$souschaine` à partir de la fin de `$chaine`

Manipulation des chaînes de caractères

56

Remplacement de sous-chaîne

- `${chaine/souschaine/remplacement}` Remplace la première correspondance de `$souschaine` par `$remplacement`.
- `${chaine//souschaine/remplacement}` Remplace toutes les correspondances de `$souschaine` avec `$remplacement`.
- `${chaine/#souschaine/remplacement}` Si `$souschaine` correspond au début de `$chaine`, substitue `$remplacement` à `$souschaine`.
- `${chaine/%souchaine/remplacement}` Si `$souschaine` correspond à la fin de `$chaine`, substitue `$remplacement` à `$souschaine`.

La commande test : Généralités

57

- Sert à vérifier des conditions
- Ces conditions portent sur des fichiers (le plus souvent), ou des chaînes ou une expression numérique
- Cette commande sert donc à prendre des (bonnes) décisions, d'où son utilisation comme condition dans les structures conditionnelles `if.. then ..else`, en quelque sorte à la place de variables booléennes ... qui n'existent pas

- **Syntaxe :**
- Deux syntaxes possibles pour tous les shells :
 - ▣ `test expression`
 - ▣ `[ expression ]` (Attention aux espaces autour de expression)
- Une autre syntaxe avec bash :
 - ▣ `[[ expression ]]` # Permet de mieux gérer les problèmes d'espaces dans les noms de fichiers, , les erreurs, ,,,

La commande test: Valeur de retour

58

Rappels :

- Tout programme ou fonction retourne une valeur de compte-rendu d'exécution au shell qui l'a appelée : une valeur numérique entre 0 et 255
- Cette valeur de retour est stockée dans la variable spéciale \$?
 - ▣ 0 indique que la commande s'est déroulée normalement (sauf cas particuliers)
 - ▣ Tout autre nombre (1-255) signale qu'une erreur s'est produite
- La commande test, de même, retourne 0 si la condition est considérée comme vraie, une valeur différente de 0 sinon pour signifier qu'elle est fausse

La commande test

59

- **Tester un fichier:** Elle admet 2 syntaxes (la deuxième étant la plus utilisée) :
 - `Test option fichier`
 - `[option fichier]`
- **Si le test est vrai Renvoie 0 \$? Contient 0 Sinon \$? ≠ 0**

Opérateurs de comparaison

60

Tests sur les fichiers (et sur les répertoires) :

- e fichier Vrai si le fichier/répertoire existe
- s fichier Vrai si le fichier à une taille supérieure à 0
- z fichier Vrai si le fichier fait 0 octet (donc si il est vide)
- r fichier Vrai si le fichier/répertoire est lisible
- w fichier Vrai si le fichier/répertoire est modifiable
- x fichier Vrai si le fichier est exécutable ou si le répertoire est accessible
- O fichier Vrai si le fichier/répertoire appartient à l'utilisateur
- G fichier Vrai si le fichier/répertoire appartient au groupe de l'utilisateur

Opérateurs de comparaison

61

□ Tests sur les fichiers (et sur les répertoires, suite) :

`-d nom` Vrai si nom représente un répertoire

`-f nom` Vrai si nom représente un fichier

`-L nom` Vrai si nom représente un lien symbolique

`f1 -nt f2` Vrai si les deux fichiers existent et si f1 est plus récent que f2

`f1 -ot f2` Vrai si les deux fichiers existent et si f1 est plus ancien que f2

`f1 -ef f2` Vrai si les deux fichiers représentent un seul et même fichier

La commande test

62

□ Exemples de tests sur les fichiers:

- ▣ `test -f /home/test.txt` : **tester si test.tx est un fichier ordinaire, équivalente à** `[ -f /home/test.txt ]`
- ▣ `[[ -f /home/test.txt ]]`
- ▣ `[[ -f /etc/passwd ]] ; echo $? #0`
- ▣ `[[ -e /etc/passwd ]] ; echo $? #0`
- ▣ `mkdir rep1 ; [[ -f rep1 ]] ; echo $? #1`
- ▣ `test -w /etc/passwd ; echo $?`

Opérateurs de comparaison

63

□ Tests sur les entiers :

`entier1 -eq entier2` Vrai si entier1 est égal à entier2

`entier1 -ge entier2` Vrai si entier1 est supérieur ou égal à entier2

`entier1 -gt entier2` Vrai si entier1 est strictement supérieur à entier2

`entier1 -le entier2` Vrai si entier1 est inférieur ou égal à entier2

`entier1 -lt entier2` Vrai si entier1 est strictement inférieur à entier

`entier1 -ne entier2` Vrai si entier1 est différent de entier2

La commande test

64

□ Comparaison numérique sur les entiers:

Opérateur	Résultat
<code>\$var -lt 6</code>	renvoie 0 si <code>\$var</code> est strictement inférieur à 6
<code>\$var -le 6</code>	renvoie 0 si <code>\$var</code> est inférieur ou égal à 6
<code>\$var -gt 6</code>	renvoie 0 si <code>\$var</code> est strictement supérieur à 6
<code>\$var -ge 6</code>	renvoie 0 si <code>\$var</code> est supérieur ou égal à 6
<code>\$var -eq 6</code>	renvoie 0 si <code>\$var</code> est égal à 6
<code>\$var -ne 6</code>	renvoie 0 si <code>\$var</code> est différent de 6

La commande test

65

Exemples de comparaisons d'entiers:

- `[ 2 -lt 3 ] ; echo $?`
- `[ 5 -lt 5 ] ; echo $?`
- `varx=2 ; vary=3 ;`
- `[ "$varx" -lt "$vary" ]; echo $? #Equivalente à (( "$varx" < "$vary" ));echo $?`
- `[ "$varx" -le "$vary" ]; echo $? #Equivalente à(( "$varx" <= "$vary")); echo $?`
- `[ "$varx" -ge "$vary" ]; echo $? #Equivalente à(( "$varx" >= "$vary")); echo $?`
- `[ "$varx" -le "$vary" ]; echo $? #Equivalente à(( "$varx" <= "$vary")); echo $?`

- `[ "$varx" -eq "$vary" ]; echo $?`
- `[ "$varx" -ne "$vary" ]; echo $?`

Opérateurs de comparaison

66

□ Tests sur les chaînes de caractères :

`-n "chaîne"` Vrai si la chaîne n'est pas vide

`-z "chaîne"` Vrai si la chaîne est vide

`"chaine1" = "chaine2"` Vrai si les deux chaînes sont identiques

`"chaine1" != "chaine2"` Vrai si les deux chaînes sont différentes

La commande test

67

□ Comparaison de chaînes de caractères

Opérateur	Résultat
=	est égal à
==	est égal à (synonyme de =)
!=	n'est pas égal à
<	est plus petit que, d'après l'ordre alphabétique ASCII
>	est plus grand que, d'après l'ordre alphabétique ASCII
Les options de la commande test -z / -n	-z la chaine est elle vide ? -n la chaine est elle non vide ?

- Les opérateurs > et < ont besoin d'être dans une séquence d'échappement s'ils se trouvent à l'intérieur de []

La commande test

68

□ Exemples de comparaisons de chaînes de caractères:

- ▣ `a="Bonjour" ; echo $?`
- ▣ `[ "$a" = "$b" ] ; echo $?`
- ▣ `[ "$a" == "$b" ] ; echo $?`
- ▣ `[ "$a" \< "$b" ] ; echo $?`
- ▣ `if [ "$a" \> "$b" ] ; echo $?`
- ▣ `test -n $a ; echo $?`
- ▣ `test -z $b ; echo $?`
- ▣ `[ -z "" ] ; echo $?`
- ▣ `[ -z "abc" ] ; echo $?`

La commande test

69

Les opérateurs logiques:

Opérateur	Résultat
-a	et logique
-o	ou logique
!	non logique

Trois opérateurs

- `exp1 -a exp2` renvoie vrai si à la fois `exp1` **et** `exp2` sont vrais
- `exp1 -o exp2` renvoie vrai si soit `exp1` soit `exp2` sont vrais
- Les opérateurs **-o** et **-a** fonctionnent avec la commande `test` ou à l'intérieur de simples crochets de test :

```
[ "$exp1" -a "$exp2" ]
```

La commande test

70

- Les Opérateurs `||` et `&&` peuvent également être utilisées comme des OU et ET logiques
- Opérateurs `||` : appelé aussi la composition sur erreur
- Opérateurs `&&` : appelé aussi la composition de réussite

- Avec la commande `test` :

```
[ $expr x || $expr y ] -- [ $exprx && $expr y ]
```

```
while [ "$varx" != "ubuntu" ] && [ "$vary" != 4 ] do ...  
done
```

- L'opérateur `!` permet d'inverser le retour d'une commande et donc d'agir comme un NON logique

```
$Which /usr/bin/ls; echo$?
```

```
$!which /usr/bin/ls; echo $?
```

Les commandes `basename`/`dirname`

71

- La commande `basename` élimine les répertoires en tête du chemin d'accès NOM. Si un SUFFIXE est indiqué, et s'il est identique à la partie finale du NOM, il est éliminé de celui-ci.

EXEMPLES

- ▣ `basename /usr/bin/sort` : Produit « `sort` » en sortie.
- ▣ `basename include/stdio.h .h` : Produit « `stdio` » en sortie.

- La commande `dirname` donne le chemin du répertoire dans lequel se trouve le fichier

EXEMPLES

- ▣ `dirname /usr/bin/sort` : Produit `/usr/bin/`.
- ▣ `dirname include/stdio.h .h` : Produit « `include/` » en sortie.