420-436-SH Développement de scripts

P1 – Scripts Bash

Partie 1

cégep de Sherbrooke

Plan

- Concepts de base
- Scripts Bash : généralités
- Scripts Bash : programmation
  - Syntaxe
  - Variables
  - Substitution de commandes et de variables
  - Tableaux
  - Variables d'environnement
  - Opérateurs
  - Structures de base
  - Fonctions

Quelques concepts de base

- Qu'est-ce qu'un programme informatique ?
  - Une série d'ordres (instructions) indiquant à un ordinateur ce qu'il doit faire, pour atteindre un but spécifique
  - Un programme comporte une ou plusieurs séquences d'instructions
  - Les instructions sont codées (écrites) dans un langage spécifique (langage de programmation)

- Qu'est-ce qu'un langage de programmation ?
  - Ensemble de symboles et d'une syntaxe, utiles pour créer des programmes informatiques

- Langages compilés vs langages interprétés ?
  - Langage compilé :
    - Le code source est complétement traduit en code binaire exécutable avant d'être exécuté par le processeur
    - La traduction est faite par un compilateur
    - Une vérification de la syntaxe est effectuée lors de la compilation

  - Langage interprété :
    - Le code source est traduit, une ligne à la fois, en code binaire exécutable. Une ligne traduite est immédiatement exécutée par le processeur
    - La traduction est faite par un interpréteur
    - Des erreurs de syntaxe sont détectées durant l'exécution de chaque ligne

Quelques concepts de base

- Qu'est-ce qu'un script ?
  - Un programme fait avec un langage interprété (attention : certains de ces langages sont devenus *hybrides*)
  - L'exécution des commandes est gérée par un moteur de script ou par l'environnement pour lequel le script est fait
    - Exemple : un script *bash* est exécuté par l'interpréteur de lignes de commandes du noyau Linux
  - Les scripts sont très utiles pour :
  - Automatiser l'exécution de tâches à effectuer dans un système d'exploitation
  - Automatiser l'exécution de tâches répétitives
  - Automatiser l'exécution de tâches interdépendantes et complexes

- Quelques langages utilisés pour faire des scripts
  - JavaScript (web)
  - PHP (web)
  - Python
  - Ruby
  - Groovy
  - Perl
  - Lua
  - Bash
  - PowerShell

Mise en garde

- Plagiat :
  - Utilisation partielle ou complète d'une production qui vient d'une autre personne, référence ou d'un outil d' IA.
    Exemple : copier des longues portions de code de quelqu'un d'autre, d'une référence ou d'un outil d'IA.
  - **Paraphrase** : reprendre dans ses propres mots, les mots ou les idées d'une autre personne, référence ou d'un outil d'IA
  - **Collusion** : plusieurs élèves se mettent ensemble pour faire un travail scolaire, et à le faire passer pour sien, alors que l'enseignant avait indiqué qu'il s'agissait d'un travail individuel.
  - **Attention** : le fait de citer les sources ne veut pas dire nécessairement que ces pratiques sont permises tout le temps.

- Tricherie :
  - Actions prises par un élève pour compléter ou présenter un travail/examen qui n'a pas été fait par lui/elle-même.
  - Vous devez soumettre le travail qu'on a réussi à compléter par soi-même et jamais le travail produit par une autre personne, ressource ou outil d'IA

- **Attention** :
  - Le plagiat et la tricherie sont des fautes graves qui entraineront automatiquement une note de zéro. De plus, la direction du Cégep sera informée d'une telle situation.

Src: https://www.alloprof.qc.ca/fr/parents/chroniques/difficultes-ecole/mieux-comprendre-plagiat-k1504.

Scripts *Bash*

- Bash
  - C'est l'interpréteur de commandes le plus populaire en Linux
  - Les commandes possibles à utiliser sont celles disponibles en *bash* (*https://ss64.com/bash/*)

- Comment créer un script *bash* ?
  - À l'aide d'un éditeur de texte (ex : *gedit*, *nano*, *pico*, *vi*, *vim*, etc.)
  - Pas besoin d'extension pour le fichier source, mais habituellement on utilise .sh
    - Exemple : script1.sh

- Comment exécuter un script *bash* ?
  - Le fichier source doit avoir les permissions d'exécution appropriées

```
-rwxr-xr-x 1 user1 user1 1201 sep 28 12:01 script1.sh
```

  - Exécution – méthode 1 : ./script1.sh
  - Exécution – méthode 2 : bash script1.sh
  - Exécution – méthode 3 : source script1.sh

Programmation de scripts *Bash*

Exemple 1

```
1 #!/bin/bash
2
3 clear
4
5 echo "*** Dossier courant : "
6 pwd
7
8 echo "*** Contenu du dossier courant : "
9 ls -l
```

Déclaration du *shell* à utiliser (ligne obligatoire)

Cette commande fait un *clear screen*

Il est fortement suggéré de placer chaque commande dans une nouvelle ligne

Il faut faire attention aux espaces dans les différentes commandes

Résultat de l'exécution du script

```
*** Dossier courant :
/scripts
*** Contenu du dossier courant
total 12
drwxr-xr-x. 2 root root  182 Jan  2 08:21 lab1
drwxr-xr-x. 2 root root 4096 Jan  2 10:38 lab2
-rwxr-xr-x. 1 root root  101 Jan  3 11:11 script1a.sh
-rwxr-xr-x. 1 root root  349 Jan  3 11:05 script1.sh
```

Programmation de scripts *Bash*

Exemple 1

Même script de la page précédente mais on a ajouté des commentaires

```
1 #!/bin/bash
2
3 # ------------------------------------------------------------------------
4 # Script : script1.sh
5 # Auteur : Alex J
6 # Description: Afficher le dossier courant et son contenu
7 # Paramètres : Aucun
8 # Date : 2023-01-02
9 # ------------------------------------------------------------------------
10
11 # Cette commande fait un clear screen
12 clear
13
14 echo "*** Dossier courant"
15 pwd     # Cette commande affiche le dossier courant
16
17 # La commande suivante liste le contenu du dossier courant
18 echo "*** Contenu du dossier courant : "
19 ls -l # Cette commande affiche le dossier courant
20
21 : '
22   Ceci est un bloc de commentaires. Ce bloc ne sera pas exécuté
23   Très utile lorsqu''on veut mettre en commentaire plusieurs lignes
24 '
```

Entête

Commentaire

Commentaire sur plusieurs lignes

```
#!/bin/bash

# ------------------------------------------------------------------------------
# Script : script1.sh
# Auteur : Alex J
# Description: Afficher le dossier courant et son contenu
# Paramètres : Aucun
# Date : 2023-01-02
# ------------------------------------------------------------------------------

# Cette commande fait un clear screen
#clear

echo "*** Dossier courant"
pwd     # Cette commande affiche le dossier courant

# La commande suivante liste le contenu du dossier courant
echo "*** Contenu du dossier courant : "
ls -l # Cette commande affiche le dossier courant

: '
  Ceci est un bloc de commentaires. Ce bloc ne sera pas exécuté
  Très utile lorsqu''on veut mettre en commentaire plusieurs lignes
'
```

Programmation de scripts *Bash*

- Débogage

Affichage de l'exécution (ligne par ligne) : `bash -x nom_script.sh`

```
[alexj@localhost scripts]$ bash -x script1.sh
+ echo '*** Dossier courant'
*** Dossier courant
+ pwd
/scripts
+ echo '*** Contenu du dossier courant : '
*** Contenu du dossier courant :
+ ls -l
total 36
drwxr-xr-x. 2 root root  182 Jan  2 08:21 lab1
drwxr-xr-x. 2 root root 4096 Jan  3 11:33 lab2
-rwxr-xr-x. 1 root root    0 Jan  4 09:35 mon_fichier2.txt
-rwxr-xr-x. 1 root root    0 Jan  4 09:35 mon_fichier.txt
-rwxr-xr-x. 1 root root  104 Jan  3 11:15 script1a.sh
-rwxr-xr-x. 1 root root  732 Jan 17 16:13 script1.sh
-rwxr-xr-x. 1 root root  353 Jan  3 12:00 script2.sh
-rwxr-xr-x. 1 root root  243 Jan  3 12:10 script3.sh
-rwxr-xr-x. 1 root root  435 Jan  3 12:22 script4.sh
-rwxr-xr-x. 1 root root  499 Jan  5 13:21 script_code_retour.sh
-rwxr-xr-x. 1 root root  698 Jan  4 09:39 script_course1.sh
-rwxr-xr-x. 1 root root  588 Jan  5 13:15 script_course2.sh
+ : '
  Ceci est un bloc de commentaires. Ce bloc ne sera pas exécuté
  Très utile lorsquon veut mettre en commentaire plusieurs lignes
'
```

Affichage de la lecture ligne par ligne : `bash -v nom_script.sh`

```
#!/bin/bash

# ------------------------------------------------------------------------------
# Script : script1.sh
# Auteur : Alex J
# Description: Afficher le dossier courant et son contenu
# Paramètres : Aucun
# Date : 2023-01-02
# ------------------------------------------------------------------------------

# Cette commande fait un clear screen
#clear

echo "*** Dossier courant"
*** Dossier courant
pwd     # Cette commande affiche le dossier courant
/scripts

# La commande suivante liste le contenu du dossier courant
echo "*** Contenu du dossier courant : "
*** Contenu du dossier courant :
ls -l # Cette commande affiche le dossier courant
total 36
drwxr-xr-x. 2 root root  182 Jan  2 08:21 lab1
drwxr-xr-x. 2 root root 4096 Jan  3 11:33 lab2
-rwxr-xr-x. 1 root root    0 Jan  4 09:35 mon_fichier2.txt
-rwxr-xr-x. 1 root root    0 Jan  4 09:35 mon_fichier.txt
-rwxr-xr-x. 1 root root  104 Jan  3 11:15 script1a.sh
-rwxr-xr-x. 1 root root  732 Jan 17 16:13 script1.sh
-rwxr-xr-x. 1 root root  353 Jan  3 12:00 script2.sh
-rwxr-xr-x. 1 root root  243 Jan  3 12:10 script3.sh
-rwxr-xr-x. 1 root root  435 Jan  3 12:22 script4.sh
-rwxr-xr-x. 1 root root  499 Jan  5 13:21 script_code_retour.sh
-rwxr-xr-x. 1 root root  698 Jan  4 09:39 script_course1.sh
-rwxr-xr-x. 1 root root  588 Jan  5 13:15 script_course2.sh

: '
  Ceci est un bloc de commentaires. Ce bloc ne sera pas exécuté
  Très utile lorsqu''on veut mettre en commentaire plusieurs lignes
'
```

Programmation de scripts *Bash*

- Création de variables
  - Bash est un langage non typé : on n'indique pas le type de variable lors de sa déclaration (création)
  - Selon la commande ou l'opérateur utilisé, le type de variable est défini implicitement

Déclaration de variables en C++

```
int a = 3;
float b = 5.47;
string c = "Bonjour";
bool d = false;
```

Déclaration de variables en Bash

```
a=3
b=5.47
c="Bonjour"
d=false
```

- Exemple d'utilisation (*appel*) des variables

```
echo $a
echo "Voici la valeur de a : $a"
echo $b
echo $c
echo $d
```

Résultat

```
3
Voici la valeur de a : 3
5.47
Bonjour
false
```

Programmation de scripts *Bash*

Substitution de commandes et de variables :

- **$()** : technique pour exécuter une commande et se servir du résultat dans une autre commande. Exemple :

```
touch $(pwd)/mon_fichier.txt
```

La commande pwd retourne le dossier courant. Ce résultat est utilisé par la commande touch pour créer un fichier

  - $() est équivalent à ` `

- **${ } :** technique utilisée lorsqu'une variable doit être concaténée avec d'autres éléments. Exemple :

Situation pratique :
On aimerait ajouter un préfix à l'affichage de quelques textes

```
prefix="420"
echo "Le code du cours est $prefix_436_SH"
echo "Le code du cours est $(prefix)_436_SH"
echo "Le code du cours est ${prefix}_436_SH"
```

Résultat

```
Le code du cours est
```

L'interpréteur considère qu'on veut afficher la variable $prefix_436_SH, laquelle n'existe pas

```
./script_course1.sh: line 37: prefix: command not found
Le code du cours est _436_SH
```

L'interpréteur considère qu'on veut exécuter la commande prefix, laquelle n'existe pas

```
Le code du cours est 420_436_SH
```

L'interpréteur accède à la valeur de la variable prefix, qui est le texte « 420 »

Programmation de scripts *Bash*

- Tableaux : Bash peut manipuler des tableaux simples

```
tableauFruits=("Pomme" "Orange" "Banane")

echo "${tableauFruits[0]}"
echo "${tableauFruits[1]}"
echo "${tableauFruits[2]}"
echo "${tableauFruits[@]}"
echo "${tableauFruits[*]}"
```

Résultat

```
Pomme
Orange
Banane
Pomme Orange Banane
Pomme Orange Banane
```

- Variables d'environnement
  - Variables présentes dans le système qui peuvent être accédées à partir de la console ou à partir de scripts.

Obtenir les variables d'environnement :

```
[alexj@localhost scripts]$ printenv
SHELL=/bin/bash
SESSION_MANAGER=local/unix:@/tmp/.ICE-unix/2009,unix/unix:/tmp/.ICE-
COLORTERM=truecolor
HISTCONTROL=ignoredups
XDG_MENU_PREFIX=gnome-
HISTSIZE=1000
HOSTNAME=localhost
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=gnome
PWD=/scripts
LOGNAME=alexj
XDG_SESSION_DESKTOP=gnome
XDG_SESSION_TYPE=wayland
SYSTEMD_EXEC_PID=2042
XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.ZA2FY1
GDM_LANG=en_CA.UTF-8
HOME=/home/alexj
USERNAME=alexj
LANG=en_CA.UTF-8
```

Créer une variable d'environnement :

```
export NAME=VALUE
```

```
[alexj@localhost scripts]$ export MA_VARIABLE="Ceci est la valeur de la variable"
[alexj@localhost scripts]$ echo $MA_VARIABLE
Ceci est la valeur de la variable
```

Ajouter le dossier /scripts à la variable d'environnement PATH:

```
[alexj@localhost scripts]$ export PATH="/scripts:$PATH"
[alexj@localhost scripts]$ echo $PATH
/scripts:/home/alexj/.local/bin:/home/alexj/bin:/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin
```

Programmation de scripts *Bash*

Exemple 2 : variables d'environnement

```
1  #!/bin/bash
2
3  clear
4
5  # Affichage d'un message
6  echo "***** Voici mon deuxième script"
7
8  echo "Calendrier : "
9
10 # Carriage Return
11 echo -e "\n"
12
13 cal
14
15 : '
16  Ceci est un commentaire
17  sur plusieurs lignes
18 '
19
20 # Affichage des valeurs de deux variables d'environnement
21 echo "Dossier courant : " $PWD
22 echo "Nom de cette machine : " $HOSTNAME
23
24 # Fin du script
```

Type de *shell* à utiliser. Cette ligne est obligatoire

Affichage d'un message sur la sortie standard (normalement l'écran)

Exécution de la commande *cal*

Un commentaire qui s'étale sur plusieurs lignes

Affichage de variables d'environnement

Résultat de l'exécution du script

```
***** Voici mon deuxième script
Calendrier :


    January 2023
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

Dossier courant :  /scripts
Nom de cette machine :  localhost
```

Programmation de scripts *Bash*

Calculs mathématiques

Commande let :
https://phoenixnap.com/kb/bash-let

Commande expr :
https://www.geeksforgeeks.org/expr-command-in-linux-with-examples/

Commande bc :
https://www.geeksforgeeks.org/bc-command-linux-examples/

```
# Calcul mathématique simple - Commande let

echo -e "\n***** Exemple d'un calcul simple - Je suis en train de calculer 1 + 2 ..."

let "a=1"
let "b=2"
let "c=a+b"
echo "Résultat : $c"
```

Résultat

```
***** Exemple d'un calcul simple - Je suis en train de calculer 1 + 2 ...
Résultat : 3
```

```
# Calcul mathématique simple - Commande 'expr' (evaluate expression)

echo -e "\n***** Même calcul - Commande 'expr'"

a=1
b=2
c=$(expr $a + $b)
echo "$c"
```

Résultat

```
***** Même calcul - Commande 'expr'
3
```

```
# Calcul mathématique simple - Commande 'bc' (basic calculator)

echo -e "\n***** Même calcul - Commande 'bc'"

a=1
b=2
echo $a + $b | bc
```

Résultat

```
***** Même calcul - Commande 'bc'
3
```

Programmation de scripts *Bash*

Opérateurs arithmétiques

```
a=10
b=20
```

Operator	Description	Example
+	Addition - Adds values on either side of the operator	`expr $a + $b` will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
*	Multiplication - Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/	Division - Divides left hand operand by right hand operand	`expr $b / $a` will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
=	Assignment - Assign right operand in left operand	a=$b would assign value of b into a
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=	Not Equality - Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

Src : http://www.tutorialspoint.com/unix/unix-basic-operators.htm

Programmation de scripts *Bash*

Opérateurs booléens

```
a=10
b=20
```

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	[$a -le $b] is true.

Src : http://www.tutorialspoint.com/unix/unix-basic-operators.htm

Programmation de scripts *Bash*

Opérateurs booléens

```
a=10
b=20
```

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true then condition would be true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	[$a -lt 20 -a $b -gt 100] is false.

Opérateurs de comparaison de textes

```
a="texte1"
b="texte2"
```

Operator	Description	Example
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero. If it is zero length then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero. If it is non-zero length then it returns true.	[-z $a] is not false.
str	Check if str is not the empty string. If it is empty then it returns false.	[$a] is not false.

Programmation de scripts *Bash*

Opérateurs de test de fichiers

```
file=/home/user1/Desktop/fichier1.txt
```

Operator	Description	Example
-b file	Checks if file is a block special file if yes then condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file if yes then condition becomes true.	[-c $file] is false.
-d file	Check if file is a directory if yes then condition becomes true.	[-d $file] is not true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set if yes then condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe if yes then condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.	[-t $file] is false.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable if yes then condition becomes true.	[-r $file] is true.
-w file	Check if file is writable if yes then condition becomes true.	[-w $file] is true.
-x file	Check if file is execute if yes then condition becomes true.	[-x $file] is true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.	[-s $file] is true.
-e file	Check if file exists. Is true even if file is a directory but exists.	[-e $file] is true.

Programmation de scripts *Bash*

Paramètres (arguments) d'un Script

- Paramètre : une donnée que l'on fournit à un script
- Il est possible de fournir plusieurs paramètres à un script
- Les paramètres sont fournis au moment où le script est exécuté

```
# ./script1.sh param1 param2 param3
```

Appel d'un script en indiquant plusieurs paramètres

- Code du fichier script1.sh

```
# Paramètres

echo -e "\n***** Paramètres du script"
echo "Nombre de paramètres en entrée : $#"

echo "Valeur du premier paramètre (s'il y en a) : $1"
```

Pour connaître le nombre de paramètres en entrée

Pour connaître la valeur du premier paramètre

- Résultat de l'exécution du script

```
***** Paramètres du script
Nombre de paramètres en entrée : 3
Valeur du premier paramètre (s'il y en a) : param1
```

Programmation de scripts *Bash*

Variables réservées

$0 : nom du script
$# : nombre de paramètres reçus
$* : liste des paramètres reçus
$$: numéro du processus en cours d'exécution
$? : code de retour de la dernière commande [1]

[1] **Attention** : le code de retour n'est pas la valeur qui s'affiche suite à l'exécution de la commande. Généralement, $? retourne la valeur 0 pour indiquer qu'il n'y a pas eu d'erreur d'exécution de la commande. Autrement, un code d'erreur est retourné par $?

Codes d'erreurs standard en Bash : https://www.adminschoice.com/exit-error-codes-in-bash-and-linux-os

Programmation de scripts *Bash*

Variables réservées

```
alexj@debian8:~/scripts$ ./script_reserved_vars.sh param1 param2
```

Appel d'un script en indiquant des paramètres

```
echo -e "Nom de ce script : " $0
echo -e "Nombre de paramètres envoyés à ce script : " $#
echo -e "Liste de paramètres envoyés à ce script : " $*
echo -e "Premier paramètre envoyé à ce script : " $1
echo -e "Deuxième paramètre envoyé à ce script : " $2
echo -e "Numéro du processus sous lequel ce script s'exécute actuellement : " $$
echo -e $PWD
ls
echo "Valeur de retour de la dernière commande exécutée : " $?
ls fichier_non_present
echo "Valeur de retour de la dernière commande exécutée : " $?
```

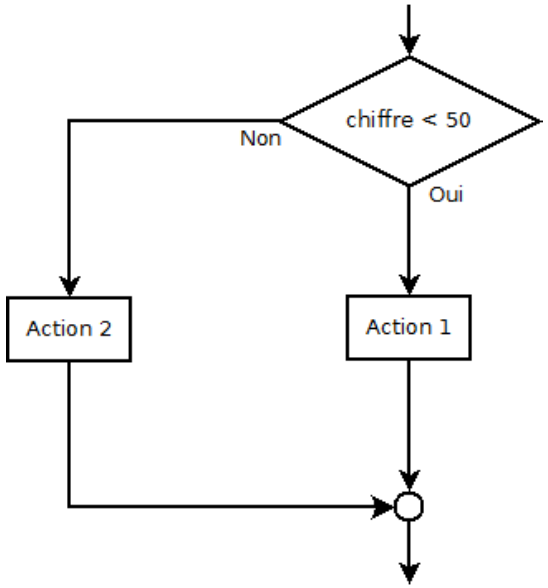
Code source du script

Résultat

```
Nom de ce script :  ./script_reserved_vars.sh
Nombre de paramètres envoyés à ce script :  2
Liste de paramètres envoyés à ce script :  param1 param2
Premier paramètre envoyé à ce script :  param1
Deuxième paramètre envoyé à ce script :  param2
Numéro du processus sous lequel ce script s'exécute actuellement :  2384
/home/alexj/scripts
script1.sh  script_reserved_vars.sh
Valeur de retour de la dernière commande exécutée :  0
ls: cannot access fichier_non_present: No such file or directory
Valeur de retour de la dernière commande exécutée :  2
```

Programmation de scripts *Bash*

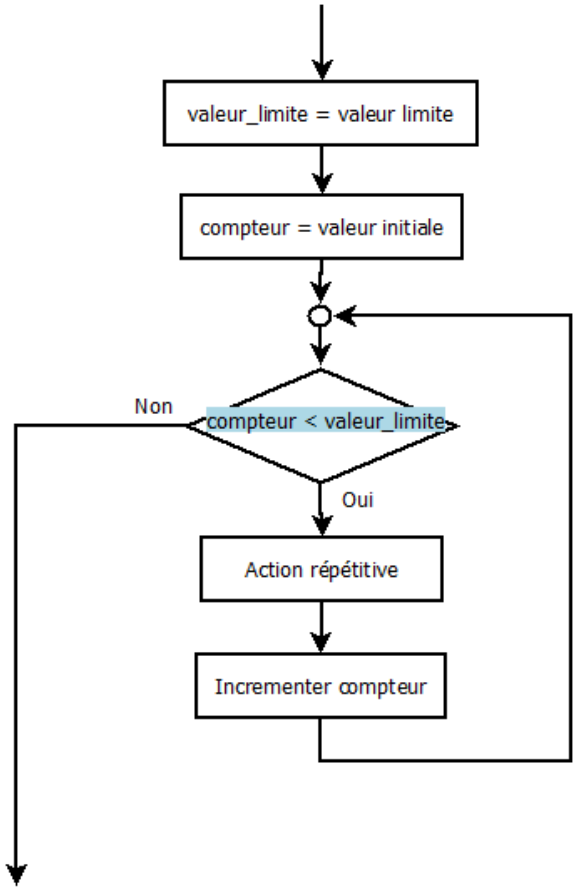
Codage des structures logiques : IF…ELSE




```
if [ $chiffre1 -lt 50 ]
then
    echo "Le chiffre est plus petit que 50"
else
    echo "Le chiffre n'est pas plus petit que 50"
fi
```

Programmation de scripts *Bash*

Codage des structures logiques : Boucle WHILE

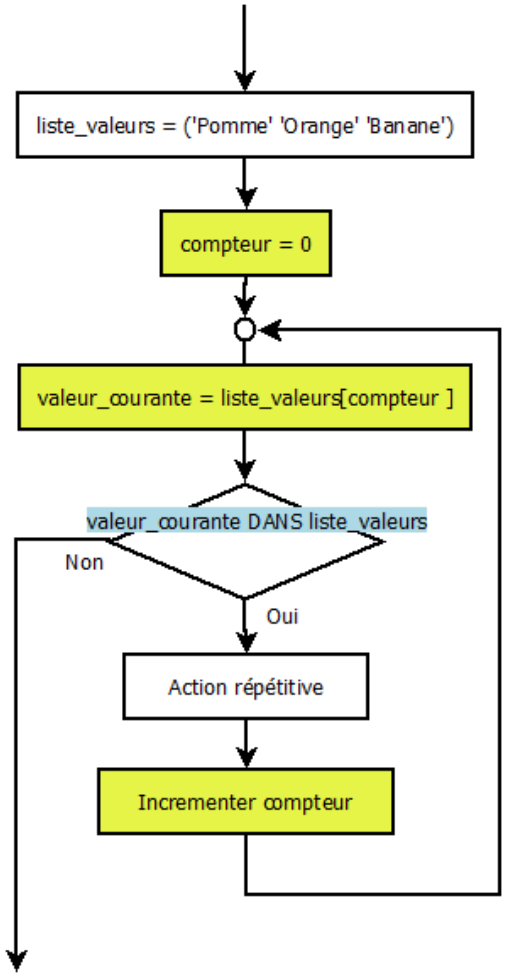



```
valeur_limite=50
compteur=0

while [ $compteur -lt $valeur_limite ]
do
    echo  "Le compteur est rendu à $compteur"
    compteur=$[$compteur+1]

done

echo "Merci !"
```

```
Le compteur est rendu à 0
Le compteur est rendu à 1
Le compteur est rendu à 2
Le compteur est rendu à 3
Le compteur est rendu à 4
Le compteur est rendu à 5
Le compteur est rendu à 6
Le compteur est rendu à 7
Le compteur est rendu à 8
Le compteur est rendu à 9
Le compteur est rendu à 10
...
Le compteur est rendu à 48
Le compteur est rendu à 49
Merci !
```

Programmation de scripts *Bash*

Codage des structures logiques : Boucle FOR




```
liste_valeurs=('Pomme' 'Orange' 'Banane')

for valeur_courante in "${liste_valeurs[@]}";
do
   echo  "Fruit = $valeur_courante"
done
```

```
Fruit = Pomme
Fruit = Orange
Fruit = Banane
```

Programmation de scripts *Bash*

Codage des structures logiques : Boucle FOR

```
for i in {1..10};
do
  echo "i = ${i}"
done
```

```
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
```

```
echo "Compte à rebours..."

for i in {30..0..3}
do
    echo "i = ${i}"
done
```

```
i = 30
i = 27
i = 24
i = 21
i = 18
i = 15
i = 12
i = 9
i = 6
i = 3
i = 0
```

```
unMot="Bonjour"

for i in $(seq 1 ${#unMot})
do
 echo "${unMot:i-1:1}"
done
```

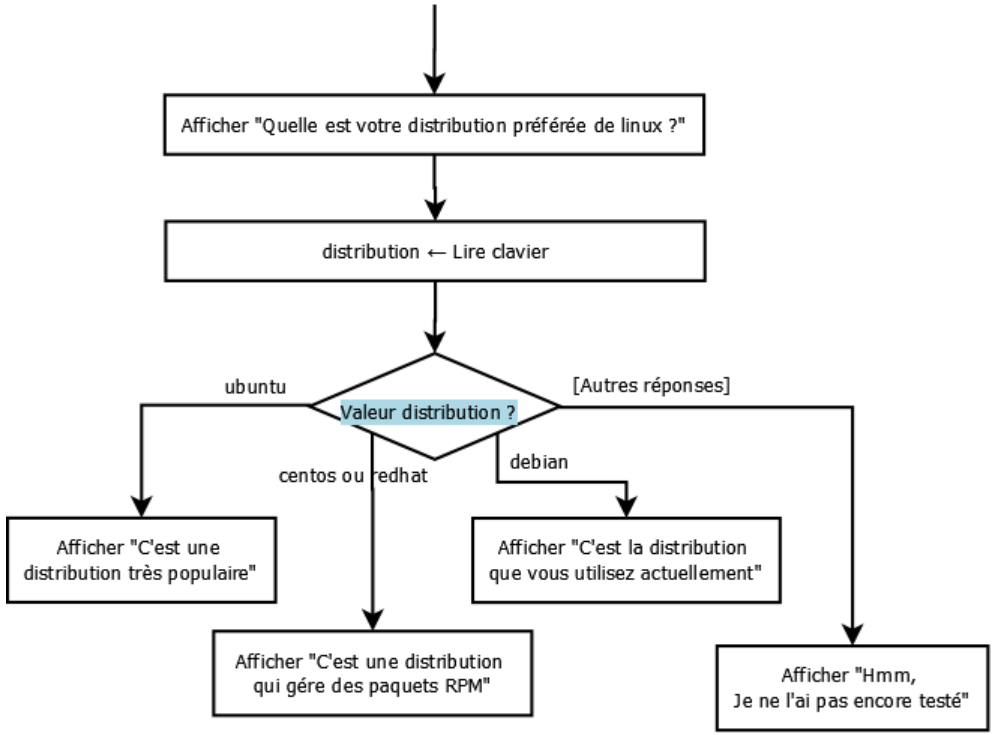
```
B
o
n
j
o
u
r
```

```
i=1

for day in Lundi Mardi Mercredi Jeudi Vendredi
do
 echo "JourSem $((i++)) : $day"
done
```

```
JourSem 1 : Lundi
JourSem 2 : Mardi
JourSem 3 : Mercredi
JourSem 4 : Jeudi
JourSem 5 : Vendredi
```

Programmation de scripts *Bash*

Codage des structures logiques : SWITCH…CASE




```
case $distribution in
     ubuntu)
          echo "C'est une distribution très populaire"
          ;;
     centos|redhat)
          echo "C'est une distribution qui gére des paquets RPM"
          ;;
     debian)
          echo "C'est la distribution que vous utilisez actuellement"
          ;;
     *)
          echo "Hmm, Je ne l'ai pas encore testé"
          ;;
esac
```

En Bash, il n'est pas recommandé d'utiliser cette structure pour vérifier d'intervalles numériques

Programmation de scripts *Bash*

```
function nom_fonction() paramètres
{
  <<commandes>>
  return valeur_retournée  #optionnel
}
```

- Fonctions
  - Une fonction est un mini-programme dont le but est d'effectuer une tâche spécifique
  - Une fonction peut recevoir 0, 1 ou plusieurs paramètres
  - Une fonction peut retourner un code (résultat) d'exécution, si désiré
  - Une fonction ne peut pas retourner une donnée
  - Les variables créées à l'intérieur d'une fonction sont globales par défaut
  - Une fonction est créée à l'intérieur d'un programme (programme principal)
  - Le programme principal appelle la fonction autant de fois que nécessaire
  - Une fonction peut appeler d'autres fonctions
  - Les fonctions sont placées généralement au début du programme

Programmation de scripts *Bash*

Création et utilisation d'une fonction

```
1  #!/bin/bash
2
3  # Cette fonction vérfie si un groupe existe déjà dans le système
4  function verifier_groupe()
5  {
6    nom_group=$1
7
8    /bin/egrep  -i "^${nom_group}" /etc/group 1>/dev/null
9    if [ $? -eq 0 ]
10   then
11     echo "Le groupe $nom_group existe déjà"
12     return 1
13   else
14     echo "Le groupe $nom_group n'existe pas"
15     return 0
16   fi
17 }
```

Création de la fonction

Un paramètre passé à la fonction

Retour du chiffre 1

Retour du chiffre 0

```
71         # On vérifie si le groupe primaire de l'utilisateur existe avant de le créer
72         verifier_groupe $p_group
73         if [ $? -eq 0 ]
74         then
75           echo "Création du groupe $p_group"
76           sudo groupadd "$p_group"
77         fi
```

Appel de la fonction, avec un paramètre

Utilisation de la valeur de retour de la fonction

Programmation de scripts *Bash*

Variables globales dans une fonction

```
1  #!/usr/bin/bash
2
3  function genererInfosPays()
4  {
5    pays="Canada"
6    langue1="Français"
7    langue2="Anglais"
8  }
9
10
11 # ---------- Corps du programme ----------------------------
12
13 # Appel de la fonction
14 genererInfosPays
15
16 # Utilisation des variables déclarées dans la fonction
17 echo "Au $pays on parle les langues $langue1 et $langue2 "
```

Création de la fonction

Déclaration des variables. Ces variables sont globales par défaut

Appel de la fonction

Utilisation des variables

Résultat

```
[alexj@localhost scripts]$ ./script_fonctions.sh
Au Canada on parle les langues Français et Anglais
```

Programmation de scripts *Bash*

Variables locales dans une fonction

```
1 #!/usr/bin/bash
2
3 function genererInfosPays()
4 {
5   local pays="Canada"
6   local langue1="Français"
7   local langue2="Anglais"
8 }
9
10
11 # ---------- Corps du programme ----------
12
13 # Appel de la fonction
14 genererInfosPays
15
16 # Utilisation des variables déclarées dans la fonction
17 echo "Au $pays on parle les langues $langue1 et $langue2 "
```

Création de la fonction

Déclaration des variables **locales**. Ces variables sont globales par défaut

Appel de la fonction

Utilisation des variables

Résultat

```
[alexj@localhost scripts]$ ./script_fonctions2.sh
Au  on parle les langues  et
```

Programmation de scripts *Bash*

Pour simuler un return d'une valeur spécifique (toute en gardant les variables 'locales')

```
1 #!/usr/bin/bash
2
3 function genererInfosPays()
4 {
5   local pays="Canada"
6   local langue1="Français"
7   local langue2="Anglais"
8
9   echo "$pays"
10 }
11
12
13 # ---------- Corps du programme -----------------------
14
15 # Appel de la fonction
16 monPays=$(genererInfosPays)
17
18 # Utilisation des variables déclarées dans la fonction
19 echo "Le pays généré est : $monPays"
```

Création de la fonction

Déclaration des variables locales. Ces variables sont globales par défaut

Les valeurs compris dans les commandes echo de la fonction sont inclus dans le « return »

Appel de la fonction et assignation à une variable

Résultat

```
[alexj@localhost scripts]$ ./script_fonctions3.sh
Le pays généré est : Canada
```

Références intéressantes

- Scripts Linux.pdf (Notes de cours)

- *Bash Beginners Guide*
  https://tldp.org/LDP/Bash-Beginners-Guide/html/chap_01.html

- *Shell Scripting Tutorial*
  https://www.shellscript.sh/first.html