Bash

1 - Qu’est-ce que [Bb]ash

Bash - Le shell

Un logiciel qui fait l'interface entre l'utilisateur et le système.

Créé par Steve Bourne (Bell Labs, 1975).

Bash = Bourne Again Shell ("Encore un shell, celui de Bourne")

bash - Le langage

Bash intègre un langage de scriptage qu'on appelle aussi du bash, par extension.

Après avoir étudié le Bash "outil shell", c'est le bash "langage" qu'on va étudier maintenant.

Pourquoi un langage dans un shell ?

Le shell est un outil très intimement lié au système à son administration.

Avoir un langage intégré permet d'automatiser très simplement de nombreuses actions répétitives ou qui nécessiteraient d'écrire en C des programmes complexes pour faire ces actions.

Un langage contient généralement :

- Des instructions et des fonctions (librairies).
- Des structures de contrôle.
- Des variables.

Le shell dispose de tout cela aussi.

On connaît déjà une partie des instructions et des fonctions. Il s'agit des commandes et des filtres, qu'on a déjà étudiés.

2 - Scriptage

Le bash est un langage de scriptage (scripting).

Un script est un programme :

- Non compilé : exécutable directement par un interpréteur (ici, la commande Bash elle-même).
- Modifiable (script et exécutable ne font qu'un).
- Stocké dans un fichier.

Créer un script ne nécessite qu'un éditeur de texte (vi, emacs, nano...)

Autres langages de scriptage

Le langage bash n'est pas le seul langage de scriptage :

- Perl
- Python
- PHP
- Ruby

Mais aussi AWK, bien sûr !

Exécutable

Un script a besoin d'un interpréteur pour être exécuté.

La commande Bash est l'interpréteur du langage bash.

Et pour les autres ? Comment Bash fait-il la différence entre tous ces langages de scriptage ? Comment sait-il de quel langage il s'agit, et quel interpréteur doit être appelé ?

Bash utilise un “magic code” pour savoir le contenu d’un fichier et déterminer :

- S’il s’agit d’un script.
- Quel interpréteur de script utiliser.

Le magic code se trouve au tout début du fichier (1ère ligne dès le 1er caractère). Pour un script, ce code est : #!

Il est suivi du chemin complet de l’interpréteur. Exemple : #!/bin/bash

Pour savoir où se trouve un interpréteur de script, comme aussi pour n'importe quelle autre commande, on peut faire appel à which. Exemples :

```
which bash
which perl
which php
```

Pour être exécutable comme avec n'importe quelle autre commande, un script doit avoir le droit de s'exécuter.

Pour rendre exécutable un fichier contenant un script, il faut lancer :

chmod u+x fichier_script

Les droits seront vu plus tard dans le cours.

Exécution

Pour exécuter un script (ou n'importe quel autre exécutable, script ou binaire), qui se trouve dans le répertoire courant, il ne suffit pas de taper :

fichier_script

Bash est incapable de trouver le script dans le répertoire courant, même s'il est devant son nez !

Les commandes (binaires ou scripts) lancées sans préciser le chemin pour les trouver, doivent obligatoirement être dans l'un des répertoires où on trouve traditionnellement les commandes, tels que :

- /bin
- /usr/bin
- /usr/local/bin
- ~/bin
- /sbin

Pour les autres commandes qui ne sont dans aucun de ces chemins classiques, il faut obligatoirement préciser le chemin pour y accéder. Exemple pour un script dans le répertoire courant :

```
./fichier_script
```

PATH

Trouver le chemin où se trouve une commande ne relève pas de la magie.

Tout repose sur une variable d'environnement qui s'appelle PATH.

echo $PATH affiche "/bin:/usr/bin:/usr/local/bin..."

Bash dispose, dans cette variable, d'une liste de chemins où il peut éventuellement trouver la commande qu'on veut exécuter. Il va faire le tour de ces endroits pour trouver la commande.

Les chemins de cette liste sont séparés par des :

Le répertoire courant (.) ne figure pas dans cette liste, c'est pour cela que Bash ne "cherche" pas, de lui même, les commandes dans le répertoire courant.

.bashrc

La variable PATH est définie dans un fichier de configuration du Bash qui s'appelle .bashrc (un fichier caché) situés dans le Home de chaque utilisateur :

~/.bashrc

On y trouve beaucoup d'autres choses (variables et paramètres divers).

.bashrc est lui-même un script Bash.

Il est exécuté à chaque fois que l'utilisateur lance un Bash, en ouvrant une fenêtre de Terminal par exemple.

En tant que script, il est modifiable et personnalisable par son propriétaire, l'utilisateur, à l'aide d'un simple éditeur de texte.

3 - Variables

Variables

Les variables bash sont comme les variables dans un programme en C par exemple. Elles sont utilisables dans les scripts.

Elles servent à stocker des informations :

- Texte.
- Valeurs numériques.
- Tableaux de valeurs.

Certaines variables sont définies pour un usage spécifique dans Bash :

- PATH
- PS1 et PS2 : Prompts du shell
- PWD : Le répertoire courant
- PPID : Le PID du processus parent du shell courant
- UID : Le User ID de l'utilisateur connecté

Un script ou l'utilisateur peuvent aussi définir leurs propres variables.

Il y a 2 syntaxes pour utiliser les variables d'environnement.

Affectation : <nom>=<valeur>
Exemple : nomfic=logo.png

Utilisation : $<nom>
Exemple : echo $nomfic

ATTENTION ! Pas d'espace autour de =, tout est collé.

Guillemet et Apostrophe

```
utilisateur=Jean
utilisateur="Jean Peuplu"
utilisateur='Gérard Mensoif'
reponse="C'est sûr"
reponse='C\'est sûr'
reponse="Il a dit \"oui\" !"
reponse='Il a dit "non" !'
```

Guillemet et Apostrophe : différences et pour quels usages ?

Les guillemets peuvent contenir des variables qui sont remplacées par leurs valeurs. Exemple :

echo "La variable PATH contient : $PATH"

Avec les apostrophes, les variables ne sont pas interprétées : echo '$PATH' affiche simplement “$PATH”.

Noms

Les noms de variables :

- Doivent commencer par une lettre ou un _.
- Peuvent contenir des minuscules et MAJUSCULES.
- Peuvent contenir des chiffres.
- Peuvent contenir des _.
- Ne peuvent pas contenir d'autres symboles ou des espaces.

Si une variable doit être concaténée à un texte :

```
fic=logo
echo $fic_old
```

Bash va croire que la variable s'appelle fic_old.

Pour éviter cette erreur, il faut placer le nom de variable entre { }. Exemple : echo ${fic}_old

Exemple d'usage des variables avec un script "compil" :

```
#!/bin/bash
projet=test
mkdir $projet
cd $projet
echo '#include <stdio.h>' > ${projet}.c
echo 'int main() {' >> ${projet}.c
echo '    printf("Hello World!");' >> ${projet}.c
echo '    return EXIT_SUCCESS;' >> ${projet}.c
echo '}' >> ${projet}.c
cc ${projet}.c -o $projet -Wall
```

4 - Entrée & Sortie

Un script n'est pas différent des autres commandes du système ou des programmes compilés.

Un script a aussi :

- Une entrée standard, le STDIN.
- Une sortie standard, le STDOUT.

STDOUT

STDOUT, la sortie standard, est évidemment l'écran.

Elle peut aussi être redirigée vers un fichier ou entrer dans le jeu d'un tube (pipe), de façon totalement transparente pour le script.

Pour écrire sur STDOUT : echo <texte...>

Exemples :

```
echo Ceci est un message
echo 'Ceci est aussi un message'
echo "Ceci est aussi un message"
echo My name is $bond
```

Par défaut, echo fait un retour à la ligne, pour empêcher ça :

```
echo -n "Pas de retour en fin de ligne"
```

STDIN

STDIN, l'entrée standard, est évidemment le clavier.

Elle peut aussi être redirigée depuis un fichier ou entrer dans le jeu d'un tube (pipe), de façon totalement transparente pour le script.

Pour lire sur STDIN : read <var> ...

Exemples :

```
read nom
read nom prenom
read -p "Qui t'es toi ? " nom prenom
```

Quand plusieurs variables sont utilisées avec read, l'espace sert de séparateur à la saisie au clavier. Pas de touche Retour entre chaque valeur saisie.
-p permet d'afficher une question puis lecture du clavier.

Amélioration du script “compil” :

```
#!/bin/bash
read -p "Nom du projet : " projet
mkdir $projet
cd $projet
echo '#include <stdio.h>' > ${projet}.c
echo 'int main() {' >> ${projet}.c
echo '    printf("Hello World!");' >> ${projet}.c
echo '    return EXIT_SUCCESS;' >> ${projet}.c
echo '}' >> ${projet}.c
cc ${projet}.c -o $projet -Wall
```

5 - Paramètres

Comme les commandes du système, un script peut aussi recevoir des paramètres et des options. Exemple :

```
script -v *.c
```

Les options et paramètres se retrouvent dans des variables automatiquement affectées par le shell.

Ces variables particulières sont nommées : $1, $2... à $9
Exemples pour script : $1 = -v, $2, $3... : chaque fichier .c

Amélioration du script “compil” :

```
#!/bin/bash
projet=$1
mkdir $projet
cd $projet
echo '#include <stdio.h>' > ${projet}.c
echo 'int main() {' >> ${projet}.c
echo '    printf("Hello World!");' >> ${projet}.c
echo '    return EXIT_SUCCESS;' >> ${projet}.c
echo '}' >> ${projet}.c
cc ${projet}.c -o $projet -Wall
```

Exemple :

./compil exo1

On peut aussi utiliser $1 directement dans le script.
Exemple :

```
#!/bin/bash
mkdir $1
cd $1
echo '#include <stdio.h>' > $1.c
... etc ...
```

shift

Les paramètres sont accessibles seulement de $1 à $9.

Ca ne veut pas dire que les paramètres sont limités à 9 et que les autres sont perdus. Ils sont simplement inaccessibles directement sans une astuce.

L'astuce s'appelle "shift", c'est une commande interne au Bash.

shift permet de décaler les paramètres d'un cran vers la gauche.

$2 devient $1, $3 devient $2, etc...
$9 voit apparaître le 10ème paramètre qui était jusqu'alors inaccessible.

Et $0 ? Il existe aussi !

$0 contient simplement le nom du script, de la commande exécutée, tel qu'il a été passé sur la ligne de commande, avec son chemin si c'est le cas. Exemple :

./script a b c
$0 vaut ./script

$0 n'est jamais affecté par shift. $0 reste $0.

Effets indésirables

Les paramètres, les variables, peuvent contenir des valeurs à effets indésirables si on ne prend pas quelques précautions.

Les valeurs à problèmes sont principalement :

- Valeurs/Variables vides.
- Valeurs/Variables avec des espaces.

Problème potentiel #1 - La valeur vide :

```
fic=liste
wc -l $fic
```

Aucun problème. Affiche le nombre de ligne dans liste.
Mais :

```
fic=
wc -l $fic
```

Ca se bloque car la commande revient à wc -l tout court.

Solution au problème des valeurs vides : les guillemets.

Exemple :

```
fic=
wc -l "$fic"
```

Ce qui revient à wc -l "". Le paramètre est effectivement vide mais il existe bien cette fois-ci.

Problème potentiel #2 - Les espaces :

```
doss="james bond 007"
mkdir $doss
```

Ca ne crée pas 1 dossier “james bond 007” mais 3 dossiers : “james”, “bond” et “007”.

La commande revient à mkdir james bond 007.

Solution au problème des espaces : les guillemets.

Exemple :

```
doss="james bond 007"
mkdir "$doss"
```

Ce qui revient à mkdir "james bond 007". Le paramètre contient des espaces mais est passé comme un unique paramètre cette fois-ci.

6 - Tests

Les guillemets sont une solution pour passer un paramètre même s'il est vide. C'est une solution du côté utilisateur.

Mais comment doit réagir une commande si un paramètre ne lui convient pas ?

- Valeur vide (parfois)
- Valeur incorrecte ou incohérente.

La solution du côté de la commande consiste à faire des tests !

On ne laisse pas la commande planter.

Un plantage n'est évidemment pas propre mais peut aussi être le début d'une catastrophe si rien n'est maîtrisé.

Règle d'Or

On ne fait jamais confiance à l'utilisateur.

Sa nature d'Etre humain rend l'utilisateur particulièrement peu fiable, et certains sont même parfois sournois !

Si une commande attend une chaîne de caractères non vide, inévitablement elle recevra une chaîne vide un jour ou l'autre.

Il faut absolument tester qu'une valeur est compatible avec ce qui est attendu par la commande.

Il faut prendre des décisions adaptées en cas de valeur incompatible.

Décisions adaptées :

- Un message d'erreur (+ exit) :
  Généralement la meilleure solution.
- Une question pour corriger l'erreur :
  Intrusif, incompatible ou compliqué à scripter en mode autonome.
- Un remplacement de la valeur en erreur par une valeur compatible :
  Une solution à bien documenter. Ne convient pas à toutes les situations.

A la différence des autres langages (C, Java etc.), bash a une manière particulière de faire des tests.
Exemple en C :

```
if <val1> != <val2> { ... }
```

Exemple en bash :

```
if test <val1> != <val2>
then
...
fi
```

La commande “test”

Le langage bash requiert, en plus, une commande test :

- C’est une commande du système (which test affiche /usr/bin/test).
- C’est aussi une commande interne à bash.

Un script shell, exécuté par l’interpréteur Bash, utilisera la version interne pour des raisons de performance.

Avec le mot clé if, le langage bash ne sait tester qu’une seule chose :

“La commande qui a été exécutée a-t-elle retourné un code d’erreur (valeur <> 0) ?”

Par contre, la commande test sait tester des choses (égalités, différences, et de nombreuses autres choses) et retourner 0 (OK) ou autre chose. Le if pourra alors fonctionner sur la base de la valeur retournée par test.

Syntaxe d’un test :

```
if test <val1> <type_test> <val2>
then
    <commande(s)>
else
    <commande(s)>
fi
```

then marque le début du if, c’est le { du C.
fi, qui est le mot clé if écrit à l’envers, marque la fin du if, c’est le } du C.

Exemple de test :

```
if test "$1" = ""
then
    echo "Paramètre vide !"
    exit 1
fi
```

Le script est arrêté (exit) après avoir affiché un message d’erreur.

Retour sur l’importance des guillemets autour de $1 :

```
if test "$1" = ""
then
    echo "Paramètre vide !"
    exit 1
fi
```

Sans guillemets, si $1 est vide, le test donne : if test = ""
Il manque quelque chose, la syntaxe est incorrecte !

test est capable de multiples types de tests (ils seront détaillés dans un prochain cours).

2 types “grands classiques” :

- Egalité :
  if test <val1> = <val2>
  Attention, seulement “=” (pas “==”)
- Différence :
  if test <val1> != <val2>

Bash est très strict sur certaines syntaxes. Un espace en trop ou manquant et le résultat peut totalement changer.

Attention à respecter les espaces dans les tests :

- if test $reponse=oui est toujours VRAI !
- if test $reponse = oui est la bonne écriture à utiliser, et c’est encore mieux ainsi : if "$reponse" = oui

TOUJOURS un espace avant et après le “type de test” (“ = ”, “ != ”, etc.).

exit

En algorithmique, les bonnes pratiques conseillent un seul retour (return) par fonction.

Un script n'est pas vraiment une fonction, c'est plutôt un programme.

Les tests sur les paramètres passés peuvent être nombreux en début de script.

Tous ces tests seraient très laborieux à écrire, et peu lisibles, avec des if… else… imbriqués. Exemple :

```
if test …
else if test …
        else if test …
                else …
                fi
        fi
fi
```

Une écriture préférable et acceptée est :

```
if test…
then
    echo 'Erreur XXX'
    exit 1
fi
if test…
then
    echo 'Erreur YYY'
    exit 2
fi
```

La valeur entière qui suit le mot clé exit est appelée le code de retour de la commande.

Un code de retour indique à l'appelant, comment s'est déroulée la commande.

C'est aussi ce que fait un main() en C avec :
return EXIT_SUCCESS ou return EXIT_FAILURE

0 signifie "pas d'erreur", autre chose indique une erreur.

7 - Redirections

Un script bash contient une succession de commandes et des structures de contrôle (tests, boucles).

Il est donc possible d'utiliser des redirections de et vers des fichiers, ainsi que les tubes. C'est même une manière de faire très fréquente.

Mais pas uniquement de et vers des fichiers ou d'autres commandes...

Redirection vers une variable

Un script bash sait aussi rediriger le résultat d'une commande vers une variable.

Syntaxe : <var>=$(<commande>)
Syntaxe (ancienne, à éviter) : <var>=`<commande>`

Exemple : nb_lignes=$(wc -l "$1")

Autres exemples :

- oldest=$(ls -tr | head -1)
  oldest contient le fichier/dossier le plus ancien.
- upper_last=$(tail -1 personnes | tr [a-z] [A-Z]
  upper_last contient le dernier nom d'une liste, en MAJUSCULES).
- today=$(date +%Y%m%d)
  today contient la date du jour au format YYYYMMDD.

Il est aussi possible d'utiliser la syntaxe d'affectation de variable avec le résultat d'une commande, dans un autre contexte que l'affectation de variable. Exemple :

```
mkdir $(date +%Y%m%d)
```

Permet de créer un dossier portant comme nom, la date d'aujourd'hui.

La commande date est appelée en 1er, et mkdir ensuite.

8 - Calculs arithmétiques

Bash sait faire des opérations avec les 5 opérateurs arithmétiques de base (+, -, /, * et %), et uniquement sur des entiers. Syntaxe :

$((<expression>))

La syntaxe est très similaire à celle de l'appel d'une commande (pour affecter une variable par exemple).

Attention à ne pas confondre les 2 !!!

Exemple :

```
nb_lignes1=$(cat *.c | wc -l)
nb_lignes2=$(cat *.h | wc -l)
total=$((nb_lignes1+nb_lignes2))
```

ATTENTION :

- Pas de $ pour affecter (écrire) une variable.
- Un $ pour utiliser (lire) une variable
  SAUF dans un calcul avec $(( )) !!

9 - Boucles

Boucle for

Syntaxe :

```
for ((<init>; <condition_continuation>; <avancement>))
do
…
done
```

Exemple :

```
for ((loop=0; loop<10; loop++))
do
   echo $loop
done
```

On note :

- L'utilisation (lecture) de la variable se fait sans $.
- Avec for(( )), bash sait faire des tests sans appel à test ! Bizarre…
- Le code a exécuter est dans un bloc do…done.

Autre syntaxe :
for <var> in <liste>

Il s'agit d'une boucle d'énumération.

Exemple pour parcourir une liste de fichiers .c :

```
for fic in *.c
do
   echo $fic
done
```

Autre exemple :

```
for i in 0 1 2 3 4 5 6 7 8 9
do
   for j in 0 1 2 3 4 5 6 7 8 9
   do
      mkdir $i$j
   done
done
```

Crée 100 dossiers nommés de 00 à 99.

Autre exemple :

```
for fic in $(head -10 liste_fics)
do
   echo Traitement du fichier $fic
   mv $fic ${fic}.old
done
```

Renomme les 10 premiers fichiers contenus dans une liste (fichier liste_fics).

Boucle while

Syntaxe :
while test <val> <type_test> <val>
do
…
done

On retrouve la syntaxe du if avec l'utilisation de la commande (interne) test et le bloc do…done.

10 - Le nombre mystère

Règles du jeu :

- Se joue à 2.
- Joueur 1 entre un nombre.
- Joueur 2 doit découvrir ce nombre.
- A chaque tour on indique au joueur 2 s'il a trouvé ou si sa réponse est inférieure ou supérieure au nombre mystère.

Etapes

1. Demander au joueur 1 un nombre mystère
2. Demander au joueur 2 de deviner le nombre
   - Si c'est le bon nombre, on félicite et on quitte
   - Si c'est un nombre < mystère, on dit "Trop petit"
   - Si c'est un nombre > mystère, on dit "Trop grand"
3. Retourner à l'étape 2.

1) Demander au joueur 1 un nombre mystère :

1) Demander au joueur 1 un nombre mystère :

```
echo -n "Proposez un nombre mystère : "
```

1) Demander au joueur 1 un nombre mystère :

```
echo -n "Proposez un nombre mystère : "
read myst
```

1) Demander au joueur 1 un nombre mystère :

```
echo -n "Proposez un nombre mystère : "
read myst
```

autre écriture :

1) Demander au joueur 1 un nombre mystère :

```
echo -n "Proposez un nombre mystère : "
read myst
```

autre écriture :

```
read -p "Proposez un nombre mystère : " myst
```

1) Demander au joueur 1 un nombre mystère :

```
read -p "Proposez un nombre mystère : " myst
```

encore mieux :

```
read -s -p "Proposez un nombre mystère : " myst
```

-s permet de saisir sans que rien ne s'affiche (silent).

2) Demander au joueur 2 de deviner le nombre :

2) Demander au joueur 2 de deviner le nombre :

```
read -p "A vous de jouer : " rep
```

2) Demander au joueur 2 de deviner le nombre :

```
read -p "A vous de jouer : " rep
```

Si c'est le bon nombre, on félicite et on quitte :

2) Demander au joueur 2 de deviner le nombre :

```
read -p "A vous de jouer : " rep
```

Si c'est le bon nombre, on félicite et on quitte :

```
if test $rep = $myst
then
   echo "Bravo !"
   exit 0
fi
```

2) Demander au joueur 2 de deviner le nombre :
Si c'est le bon nombre, on félicite et on quitte :

```
if test $rep = $myst
then
   echo "Bravo !"
   exit 0
fi
```

Est-ce une écriture satisfaisante ?

2) Demander au joueur 2 de deviner le nombre :
Si c'est le bon nombre, on félicite et on quitte :

```
if test "$rep" = "$myst"
then
   echo "Bravo !"
   exit 0
fi
```

2) Demander au joueur 2 de deviner le nombre :
Si c'est un nombre < mystère, on dit "Trop petit" :

2) Demander au joueur 2 de deviner le nombre :
Si c'est un nombre < mystère, on dit "Trop petit" :

```
if test "$rep" < "$myst"
```

Est-ce ainsi qu'on doit l'écrire ?

2) Demander au joueur 2 de deviner le nombre :
Si c'est un nombre < mystère, on dit "Trop petit" :

```
if test "$rep" < "$myst"
```

Hélas, avec test on n'écrit pas ainsi ! On doit écrire :

```
if test "$rep" -lt "$myst"
```

-lt = less than, et -gt = greater than.

2) Demander au joueur 2 de deviner le nombre :
Si c'est un nombre < mystère, on dit "Trop petit" :

```
if test "$rep" -lt "$myst"
then
   echo "Trop petit"
fi
```

2) Demander au joueur 2 de deviner le nombre :
Si c'est un nombre > mystère, on dit "Trop grand" :

```
if test "$rep" -gt "$myst"
then
   echo "Trop grand"
fi
```

```
#!/bin/bash
read -p "Prop. mystère : " myst
while test 1
do
   read -p "Jouez : " rep
   if test "$rep" = "$myst"
   then
      echo "Bravo !"
      exit 0
   fi
   if test "$rep" -lt "$myst"
   then
      echo "Trop petit"
   fi
   if test "$rep" -gt "$myst"
   then
      echo "Trop grand"
   fi
done
```

```
#!/bin/bash
cnt=0
read -p "Prop. mystère : " myst
while test 1
do
   read -p "Jouez : " rep
   cnt=$((cnt+1))
   if test "$rep" = "$myst"
   then
      echo "Bravo ! $cnt coup(s)"
      exit 0
   fi
   if test "$rep" -lt "$myst"
   then
      echo "Trop petit"
   fi
   if test "$rep" -gt "$myst"
   then
      echo "Trop grand"
   fi
done
```

https://commons.wikimedia.org/wiki/File:Thats_all_folks.svg