Cours 5 : Les flux de redirection

Rabii El Ghorfi

1. Les flux de redirection
   - `>` et `»` redirigiriger le résultat dans un fichier
   - `2>, 2» et 2>&1` : rediriger les erreurs
   - `< et «` : lire depuis un fichier ou le clavier

2. Processus
   - Définition
   - Cycle de vie d'un processus
   - Enchaînement de processus

Vous devriez maintenant avoir l'habitude d'un certain nombre de commandes que propose la console de `Linux`. Le fonctionnement est toujours le même :

1. Vous tapez la commande (par exemple `ls`)
2. Le résultat s'affiche dans la console

Toutefois, au lieu d'afficher le résultat dans la console, vous allez pouvoir l'envoyer ailleurs.
Où ? Dans un fichier, ou en entrée d'une autre commande pour "chaîner des commandes". Ainsi, le résultat d'une commande peut en déclencher une autre !
Comment ? A l'aide de petits symboles spéciaux, appelés flux de redirection

La manipulation la plus simple que nous allons voir va nous permettre d'écrire le résultat d'une commande dans un fichier, au lieu de l'afficher simplement dans la console.
Préparatifs : Prenons par exemple la commande `cut` et le fichier `telephone.txt`
La commande `cut` nous avait permis de découper une partie du fichier et d'afficher le résultat dans la console. Par exemple

```
cut -f 2 telephone.txt
```

Ce résultat s'est affiché dans la console. C'est ce que font toutes les commandes par défaut... à moins que l'on utilise un flux de redirection !

> : rediriger dans un nouveau fichier

Supposons que nous souhaitions écrire la liste des prénoms dans un fichier, afin de garder sous le coude la liste des prénoms des personnes inscrit dans l'annuaire `telephone.txt`.
C'est là qu'intervient le petit symbole magique > (appelé chevron)
Ce symbole permet de rediriger le résultat de la commande dans le fichier de votre choix

```
cut -f 2 telephone.txt > people.txt
```

Attention : si le fichier existait déjà il sera écrasé sans demande de confirmation !

» : rediriger à la fin d'un fichier

Le double chevron » sert lui aussi à rediriger le résultat dans un fichier, mais cette fois à la fin de ce fichier.
Avantage : vous ne risquez pas d'écraser le fichier s'il existe déjà. Si le fichier n'existe pas, il sera créé automatiquement.
Normalement, on devrait avoir créé un fichier `people.txt` lors des manipulations précédentes. Si vous faites :

```
cut -f 3 telephone.txt » people.txt
```

les commandes produisent 2 flux de données différents :

1. La sortie standard : pour tous les messages (sauf les erreurs)
2. La sortie d'erreurs : pour toutes les erreurs

Lorsque l'on applique la commande `cat` sur un fichier `test.txt` pour afficher son contenu. Il y a 2 possibilités :

- Si tout va bien : le résultat (le contenu du fichier) s'affiche sur la sortie standard
- S'il y a une erreur : celle-ci s'affiche dans la sortie d'erreurs.

Par défaut, tout s'affiche dans la console : la sortie standard comme la sortie d'erreurs
Alors...Essayons

```
cut -d , -f 1 fichier_inexistant.txt > standard.txt
```

Rediriger les erreurs dans un fichier à part

On pourrait souhaiter "logger" les erreurs dans un fichier d'erreurs à part pour ne pas les oublier et pour pouvoir les analyser ensuite. Pour cela, on utilise l'opérateur 2>.

```
cut -d , -f 1 fich_inexist.txt > standard.txt 2> erreurs.log
```

Il y a deux redirections ici :

1. `> standard.txt` : redirige le résultat de la commande (sauf les erreurs) dans le fichier eleves.txt. C'est la sortie standard.
2. `2> erreurs.log` : redirige les erreurs éventuelles dans le fichier erreurs.log. C'est la sortie d'erreurs.

Fusionner les sorties

Parfois, on n'a pas envie de séparer les informations dans 2 fichiers différents. Heureusement, il est possible de fusionner les sorties dans un seul et même fichier.
Comment ? Il faut utiliser le code suivant :`2>&1` Cela a pour effet de rediriger toute la sortie d'erreurs dans la sortie standard.

```
cut -d , -f 1 fichier_inexistant.txt > eleves.txt 2>&1
```

Pour le moment, nous avons redirigé uniquement la **sortie** des commandes. Nous avons décidé où envoyer les messages issus de ces commandes.
Maintenant, faisons l'inverse, *i.e.* de décider **d'où vient l'entrée d'une commande** !!!.
Jusqu'alors, l'entrée venait des paramètres de la commande... mais on peut faire en sorte qu'elle vienne d'un fichier ou d'une saisie au clavier !!!

< : lire depuis un fichier

Le chevron ouvrant < permet d'indiquer d'où vient l'entrée qu'on envoie à la commande.

```
cat < telephone.txt
```

On faisait pas pareil avant en écrivant juste `cat telephone.txt` par hasard ?

Si. Le fait d'écrire `cat < telephone.txt` est strictement identique à écrire `cat telephone.txt` ... du moins en apparence.

Le résultat produit est le même, mais ce qui se passe derrière est très différent :

- `cat telephone.txt` : la commande `cat` reçoit en entrée le nom du fichier “telephone.txt“ qu’elle doit ensuite se charger d’ouvrir pour afficher son contenu.
- Si vous écrivez `cat < telephone.txt` : la commande `cat` reçoit le contenu de ”telephone.txt” qu’elle se contente simplement d’afficher dans la console. C’est le shell (le programme qui gère la console) qui se charge d’envoyer le contenu de “telephone.txt“ à la commande `cat`

2 façons de faire la même chose mais de manière très différente.

« : lire depuis le clavier progressivement

Le double chevron ouvrant « fait quelque chose d'assez différent : il vous permet d'envoyer un contenu à une commande avec votre clavier. Cela peut s'avérer très utile.

```
sort -n « FIN
```

Cela évite d'avoir à créer un fichier dont on a pas besoin.
On peut faire la même chose avec une autre commande comme par exemple `wc` pour compter le nombre de mots ou de caractères.

```
wc -m « FIN
```

Rq : finir par un mot-clé qui sert à indiquer la fin de la saisie.

Nous pouvons tout à fait combiner ces symboles avec ceux qu'on a vus précédemment. Par exemple :

```
sort -n « FIN > nombres_tries.txt 2>&1
```

Définition

Un processus est un programme en cours d'exécution. On distingue deux types de processus :

- Le processus système (daemons) : assure des services généraux accessibles à tous les utilisateurs du système. Le propriétaire est le `root` et il n'est sous le contrôle d'aucun terminal.
- Le processus utilisateur : dédié à l'exécution d'une tâche particulière. Le propriétaire est l'utilisateur qui l'exécute et il est sous le contrôle du terminal à partir duquel il a été lancé.

Création

Toute exécution d'un programme déclenche la création d'un processus dont la durée de vie = la durée d'exécution du programme.
Le système alloue à chaque processus un numéro d'identification unique : `PID` (Process IDentifier).
Tout processus est créé par un autre processus : son processus père

Exemples :

Lorsqu'un utilisateur lance une commande, un processus est créé dont le père est le processus correspondant à l'exécution du shell.

États d'un processus :

- Prêt (R) : le processus attend que le processeur lui soit affecté.
- Actif (R) : le processeur exécute le processus.
- Endormi (S pour <20s ou I pour >20s) : le processus est en attente de l'arrivée d'un évènement (par exemple, une réponse du terminal).
- Zombi (Z) : le processus a pris fin sans libérer ses ressources.
- Suspendu (T) : le processus a été interrompu et attend l'arrivée d'un signal de reprise.

Exécution d'une commande :

Cinq modes d'exécution d'une commande sous Unix :

- mode interactif : commande lancée à partir d'un terminal. Le contrôle du terminal n'est rendu à l'utilisateur qu'à la fin de l'exécution de la commande.
  `<ctrl-c>` : interrompre la commande
  `<ctrl-z>` : suspendre la commande
- mode en arrière plan : permet de rendre immédiatement le contrôle à l'utilisateur (commande lancée suivie du caractère &). Si le terminal est fermé, la commande en arrière plan est interrompue automatiquement. Pour éviter ce problème, il faut lancer la commande sous le contrôle de la commande `nohup` (syntaxe : `nohup nom_commande &`).

- mode différé : `at` permet de déclencher l'exécution d'une commande à une date fixée.
  ex : `at 16:50 10/06/08 < commande`
  démarre le lancement du contenu de commande le 06 octobre 2008 à 16h50.
  (`at -l` pour lister et `at -r` pour supprimer)
- mode batch : permet de placer une commande dans une file d'attente.
- mode cyclique : tâche exécutée de façon cyclique.

La commande `ps`

Visualiser les processus avec la commande : `ps (options)`, les options les plus intéressantes

- `-e` : affichage de tous les processus
- `-f` : affichage détaillé

exemple : `ps -ef`

```
UID PID PPID C STIME TTY TIME COMMAND
root 1 0 0 Dec 6 ? 1:02 init
...
jean 319 300 0 10:30:30 ? 0:02 /usr/dt/bin/dtsession
olivier 321 319 0 10:30:34 ttyp1 0:02 csh
olivier 324 321 0 10:32:12 ttyp1 0:00 ps -ef
```

La signification des différentes colonnes :

`UID`	nom du user qui a lancé le process
`PID`	num. du process
`PPID`	num. du process parent
`C`	facteur de priorité : $\uparrow \Rightarrow$ prioritaire
`STIME`	heure de lancement du processus
`TTY`	nom du terminal
`TIME`	durée de traitement du processus
`COMMAND`	nom du process

exemple précédent : `ps -ef`
le 1^{er} p_i a pour `PID 321`, le 2^{me} 324. Le PPID du process `" ps -ef "` est 321 qui correspond au shell, par conséquent le shell est le process parent, de la commande qu'on vient de taper.
Pour voir les process d'un seul utilisateur : `ps -u olivier`

Les signaux

Il est possible d'agir sur le déroulement d'un p_i en lui envoyant un signal. Unix définit de façon standard un certain nombre de signaux dont :

- `SIGINT` (signal 2) : interrompre l'exécution d'un p_i,
- `SIGKILL` (signal 9) : arrêt définitif l'exécution d'un p_i (ne peut pas être ignoré),
- `SIGTSTP` (signal 24) : suspendre temporairement l'exécution d'un p_i,
- `SIGCONT` (signal 25) : reprendre l'exécution d'un p_i précédemment suspendu par l'envoi d'un signal `SIGTSTP`.

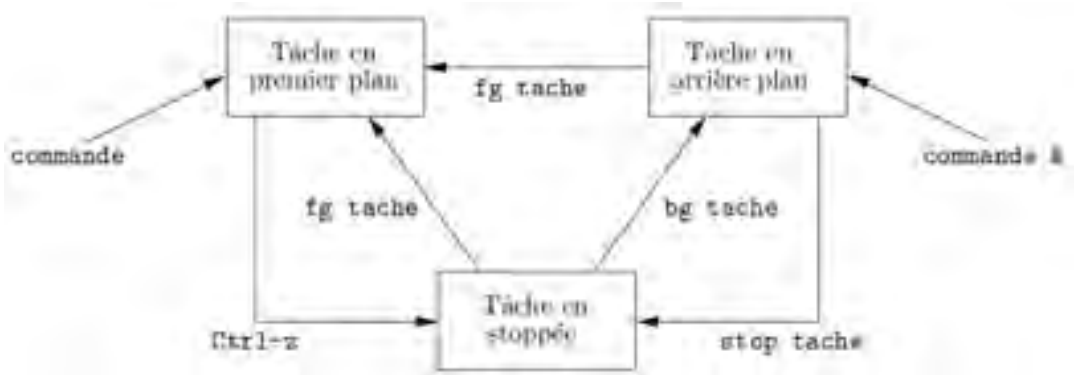
Un signal peut être envoyé par :

1. le système (ex : signaux d'erreur),
2. un autre p_i,
3. l'utiliateur :
   - soit l'utilisateur tape des caractères provoquant l'envoi d'un signal au p_i en cours d'exécution sur le terminal (ex : `<ctrl-z>` pour `SIGTSTP`, `<ctrl-c>` pour `SIGINT`),
   - soit l'utilisateur utilise la commande `kill` pour envoyer un signal à un ou plusieurs p_i lorsqu'il n'a pas accès au terminal de rattachement des p_i ou lorsque ces derniers sont exécutés en arrière plan.
     La syntaxe est la suivante :
     `kill -signal PID`
     `signal` : nom symbolique ou num.
     `PID` : donné par `ps`

Le job control

Un job est une ligne de commande shell. Chaque job est numéroté de 1 à N par le shell. Un job peut se trouver dans trois états :

- avant plan : Le job s'exécute et vous n'avez pas la main sur le shell.
- arrière plan : Le job s'exécute et vous avez la main sur le shell.
- suspendu : Le job est en attente, il ne s'exécute pas.



Quelques commandes

`jobs`	:	permet de lister les jobs en cours
`<ctrl-z>`	:	suspendre une commande en avant plan
`stop %num_job`	:	suspendre une commande en arrière plan
`bg %num_job`	:	basculer de suspendu à arrière plan
`fg %num_job`	:	passer en avant plan
`kill %num_job`	:	tuer un processus

Il est possible sur une même ligne de commande de lancer plusieurs p_i.

Lancement en séquence de plusieurs commandes

`commande1 ; commande2 ; commande3 > fich`

création du p_i qui exécute `commande1`

puis quand `commande1` est terminé,

création du p_i qui exécute `commande2`

...

Les différents p_i ne co-existent pas.

Seule la S standard de `commande3` est redirigée vers `fich`. Pour rediriger les trois commandes :

`(commande1 ; commande2 ; commande3) > fich`.

Lancement concurrent de p_i communiquant par un tube (pipe)
L'exécution simultanée de plusieurs p_i échangeant des données par l'intermédiaire de zones mémoires tubes.
Syntaxe : $\texttt{com}_1$ | $\texttt{com}_2$ | ... | $\texttt{com}_n$
Le système crée $n-1$ tubes et n p_i.

- le p_i qui exécute $\texttt{com}_1$ a sa S standard redirigée vers le 1^{er} tube, son E standard étant celle définie par défaut (clavier),
- le p_i qui exécute $\texttt{com}_2$ a son E standard redirigée vers le 1^{er} tube et sa S standard redirigée vers le 2^{me} tube, ...
- le p_i qui exécute $\texttt{com}_n$ a son E standard redirigée vers le $n-1^{me}$ tube, sa S standard étant celle par défaut (l'écran).

Les n p_i coexistent et se partagent l'accès au processeur. Le système se charge de leur synchronisation : les p_i lecteurs sont mis en attente tant que leur tube en entrée est vide, les p_i écrivains sont mis en attente si leur tube en sortie est plein.

Exemple

```
ls /bin /usr/local/bin | grep "cp" | wc -l
```

permet de compter le nombre de fichiers dont le nom contient la chaîne `cp` dans les répertoires `/bin` et `/usr/local/bin`.

La commande `tee` permet de rediriger la sortie standard d'un processus vers un fichier et vers la sortie standard.

Exemple :

```
ls /bin | grep "cp" | tee fich | wc -l
```

1. Des scripts Shell
2. premier script
3. Les variables en shell
   - Déclaration
   - Saisie
   - Opérations mathématiques
   - Les variables d'environnements
   - Les variables des paramètres
4. Conditionnelles
   - If
   - If then else
   - Sinon si
5. Les tests
6. Les Boucles
   - `while` : boucler "tant que"
   - `for` : boucler sur une liste de valeurs

la programmation shell. De quoi s'agit-il ?

Imaginez un mini-langage de programmation intégré à Linux. Ce n'est pas un langage aussi complet que peuvent l'être le `C`, le `C++` ou le `Java` par exemple, mais cela permet d'automatiser la plupart de vos tâches. Voici un aperçu de ce qu'on peut faire avec :

- Sauvegarde de vos données
- Surveillance de la charge de votre machine
- Système de gestion personnalisé de vos téléchargements
- ...etc

Pourquoi pas le `C` ?

Le gros avantage des `scripts shell`, c'est qu'ils sont totalement intégrés à Linux : il n'y a rien à installer et rien à compiler. Et surtout : vous avez trés peu de nouvelles choses à apprendre. En effet, toutes les commandes que l'on utilise dans les `scripts shells` sont des commandes du système que vous connaissez déjà : `ls, cut, grep, sort`

`shell`: Un interpréteur de commandes

Les fonctionnalités offertes par l'invite de commande peuvent varier en fonction du `shell` que l'on utilise.
Les principaux sont

- `sh` : Bourne Shell. L'ancêtre de tous les shells.
- `bash` : Bourne Again Shell. Une amélioration du Bourne Shell, disponible par défaut sous Linux et Mac OS X.
- `ksh` : Korn Shell. Un shell puissant présent sur les Unix propriétaires, mais aussi disponible en version libre, compatible avec bash.
- `csh` : C Shell. Un shell utilisant une syntaxe proche du `C`.
- `tcsh` : Tenex C Shell. Amélioration du C Shell.
- `zsh` : Z Shell. Shell assez récent reprenant les meilleures idées de `bash`, `ksh` et `tcsh`.

A quoi sert un shell

`Shell` : programme qui gère l'invite de commandes. C'est donc le programme qui attend que vous rentriez des commandes :

- Se souvenir quelles étaient les dernières commandes tapées
- Auto-complétion d'une commande ou d'un nom de fichier lorsque vous appuyez sur `Tab`
- Gérer les processus (envoi en arrière-plan, mise en pause avec Ctrl + Z ...).
- Rediriger et chaîner les commandes (les fameux symboles $>$, $<$, | ...)

Avec quel shell écrire nos scripts alors ? `bash`

- On le trouve par défaut sous Linux et Mac OS X (cela couvre assez de monde !).
- Il rend l'écriture de scripts plus simple que **`sh`**.
- Il est plus répandu que **`ksh`** et **`zsh`** sous Linux.

En clair, le **`bash`** est un bon compromis entre **`sh`** (le plus compatible) et **`ksh/zsh`** (plus puissants).

- Commençons par créer un nouveau fichier pour notre script : `gedit essai.sh` $\longrightarrow$ fichier vide
- La première chose à faire dans un script shell est d'indiquer... quel shell est utilisé : Rajouter dans `essai.sh` la ligne

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

  le #! est appelé le sha-bang
- Après le sha-bang, nous pouvons commencer à coder.
  Le principe : Ecrire les commandes que vous souhaitez exécuter. Ce sont les mêmes que celles que vous tapiez dans l'invite de commandes !

Exple :

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

- Donner les droits d'exec au script

```
chmod +x essai.sh
```

- Exécuter le script, en tapant " ./" devant le nom du script

```
./essai.sh
```

Que fait le script ? Il fait juste un `ls`, donc il affiche la liste des fichiers dans le répertoire.
On peut vouloir préciser en plus le rep courant :

```
# !/bin/bash
pwd
ls
```

Créer sa propre commande :

Actuellement, le script doit être lancé via `./essai.sh` et vous devez être dans le bon répertoire.
Comment font les autres programmes pour pouvoir être exécutés depuis n'importe quel répertoire sans "./" devant ? Ils sont placés dans un des rep. du `PATH`.
Def : Le `PATH` est une variable système qui indique où sont les programmes exec. Si vous tapez echo $PATH , vous aurez la liste de ces rep.$\longrightarrow$ déplacer ou copier le script dans un de ces rep, (`/bin, ou /usr/bin, ou /usr/local/bin`).
Rq : Il faut être `root` pour pouvoir faire ça.

Comme dans tous les langages de programmation, on trouve en `bash` ce qu'on appelle des variables.
$\longrightarrow$ stocker temporairement des informations en mémoire. C'est en fait la base de la programmation.
Les variables en `bash` sont particulières. Il faut être très rigoureux lorsqu'on les utilise $\longrightarrow$ différent du `C`

Variables :

- Un nom
- Une valeur

Exple

```
message='Bonjour tout le monde'
```

Rq :Pas d'espace autour de "="
Executons!!! `./var.sh`

echo : afficher une variable

Exple

- `echo "Salut tout le monde"`
- `echo -e "Message\n Autre ligne"`

`./varaffich.sh`

Les quotes

- Les apostrophes ' '(simples quotes)

  `./simplequote.sh`
- Les guillemets " " (doubles quotes)

  `./doublequote.sh`
- Les accents graves ` ` (back quotes)

  `./backquote.sh`

read

Demander au user de saisir du texte avec la commande `read`. La façon la plus simple de l'utiliser est d'indiquer le nom de la variable dans laquelle le message saisi sera stocké :

`./read1.sh`

La commande read propose plusieurs options intéressantes.

- `-p` : afficher un message de prompt `./readp.sh`
- `-n` : limiter le nombre de caractères `./readn.sh`
- `-s` : ne pas afficher le texte saisi `./reads.sh`

En `bash`, les var. sont toutes des chaînes de caractères
⇒Incapable de manipuler des nombres ⇒ pas opérations!!

la commande `let`

`./calcul1.sh`

Les opérations :

- L'addition : +
- La soustraction : −
- La multiplication : ∗
- La division : /
- La puissance : ∗∗
- Le modulo : %

Les var. que crées dans scripts `bash` n'existent que dans ces scripts. *ie.* une variable définie dans un `pgme A` ne sera pas utilisable dans un `pgme B`.

Les var. d'environnement :var. utilisables n'importe quel pgme. On parle aussi parfois de var. globales. Afficher toutes celles actuellement en mémoire avec la commande `env`.

Quelques variables d'environnement

- `SHELL` : type de shell est en cours d'utilisation (`sh, bash, ksh...`)
- `PATH` : une liste rep qui contiennent des exec que vous souhaitez pouvoir lancer sans indiquer leur rep.
- `EDITOR` :Editeur de txt par défaut
- `HOME` : position du dossier `home`
- `PWD` : Dossier courant

Rq : En majuscule

Les scripts `bash` acceptent des paramètres

```
./varparam.sh param1 param2 param3
```

- `$#` : contient le nombre de param.
- `$0` : contient le nom du script exécuté (ici `"./varparam.sh"`)
- `$1` : contient 1^r param.
- ...
- `$9` : contient 9^m param.

Syntaxe

```
if [ test ]
then
echo "true"
fi
```

Rq : l'espace dans `[ test ]`

`./if1.sh`

Syntaxe

```
if [ test ]
then
echo "true"
else
echo "false"
fi
```

```
./if2.sh param1
```

Syntaxe

```
if [ test ]
then
echo "premier test a été verif"
elif [ autre_test ]
echo "second test a été verif"
elif [ encore_autre_test ]
echo "troisième test a été verif"
else
echo "Aucun des tests prec. n'a été vérifié"
fi
```

```
./if3.sh param1
```

3 types de tests différents en `bash` :

1. Tests sur des chaînes de caractères
2. Tests sur des nombres
3. Tests sur des fichiers

- `$chaine1 = $chaine2`
  teste si 2 chaînes sont identiques. `B` $\neq$ `b` (sensible à la casse...)
- `$chaine1 != $chaine2`
  Teste si 2 chaînes sont $\neq$
- `-z$chaine`
  Teste si 1 chaînes est vide
- `-n$chaine`
  Teste si 1 chaînes est non vide

```
./test1.sh param1 param2
```

Exo :Ecrire un script qui teste l'existence d'un paramètre

- `$num1 -eq $num2` Teste si les nombres sont égaux(equal) "=" compare les caractères.
- `$num1 -ne $num2` Teste si les nombres sont diff(non equal) "!=" compare les caractères.
- `$num1 -lt $num2` Teste si num1 est < num2 (lower than)
- `$num1 -le $num2` Teste si num1 est <= num2 (lower or equal)
- `$num1 -gt $num2` Teste si num1 est > num2 (greater than)
- `$num1 -ge $num2` Teste si num1 est >= num2 (greater or equal)

```
./test2.sh param1
```

- `-e $nomfich` Teste si le fich. existe
- `-d $nomfich` Teste si le fich. est un rep.
- `-f $nomfich` Teste si le fich. est un... fich. Un vrai fich. pas un dossier.
- `-L $nomfich` Teste si fich est un lien symbolique
- `-r $nomfich` Teste si fich est lisible (r)
- `$fich1 -nt $fich2` Teste si fich1 est plus récent que fich2 (newer than) | `$fich1 -ot $fich2` (older than)

EXO Ecrire un script qui demande au user de rentrer le nom d'un rep et de verifier si c'est bien un rep

Effectuer plusieurs tests à la fois

Dans un `if`, il est possible de faire plusieurs tests à la fois.

- Si un test est vrai ET qu'un autre test est vrai : `&&`
- Si un test est vrai OU qu'un autre test est vrai : `||`

Rq :encadrer chaque condition par des crochets
EXO Ecrire un script qui vérifie qu'il a au moins un param et la valeur du 1^r param est `asticot`

Syntaxe

```
while [ test ]
do
echo 'Action en boucle'
done
```

```
while [ test ]; do
echo 'Action en
boucle' done
```

`./while1.sh`

La boucle `for` permet de parcourir une liste de valeurs, et de boucler autant de fois qu'il y a de valeurs.

Syntaxe

```
for variable in 'valeur1' 'valeur2' 'valeur3'
do
echo "La variable vaut $variable"
done
```

`./for1.sh`

Rq : La liste des valeurs n'a pas besoin d'être définie directement dans le code :

`./for2.sh`

EXO : Script qui renomme tous les fichiers trouvés

Un cas plus classique du `for`

```
for i in ‘seq 1 10‘;
do
echo $i
done
```

`./for3.sh`

Pour faire des sauts de 2 faire `for i in ‘seq 1 2 10‘;`