

Administration Système sous Linux (Ubuntu Server)

Grégory Morel

2022-2023

CPE Lyon - 3IRC / 4ETI / 3ICS

Cours 1

Bash Partie 1 : prise en main

Bash ?

Dans un environnement *serveur*, l'interface avec la machine est (souvent) un *shell* / *interpréteur de commandes*.

Plusieurs variantes; les principaux :

- **sh** : Bourne Shell, l'ancêtre de tous les shells
- **ksh** : Korn Shell, a introduit le contrôle des jobs, les alias, l'historique des commandes...
- **Bash** : Bourne Again Shell, un sh amélioré, et la version par défaut sous Linux
- **zsh** : Un autre shell très populaire

Dans la suite du cours, on utilisera **Bash**.

Bash est un programme de type **REPL**, càd une boucle *Read - Eval - Print*¹ :

- *Read* : lit la commande saisie par l'utilisateur,
- *Eval* : exécute cette commande,
- *Print* : éventuellement, affiche un résultat.

Bash est aussi un **langage de script**, permettant d'automatiser simplement et efficacement des tâches complexes.

1. De la même manière qu'un interpréteur Python

Commandes

Une *commande* est le nom d'un *programme compilé* (ou *binaire*) ou d'un *script* (Bash, Python...).

💡 Il existe aussi quelques commandes *internes* à Bash : ce ne sont pas des « programmes », mais plutôt des commandes d'interaction, avec Bash ¹.

La plupart des commandes peuvent prendre des *paramètres* ou *arguments* ² : par exemple `cat fichier` appelle la commande `cat` avec l'argument `fichier`.

La plupart des commandes peuvent également recevoir des *options*, sous forme *courte* (ex. : `cat -n fichier`) ou *longue* (`cat --number fichier`).

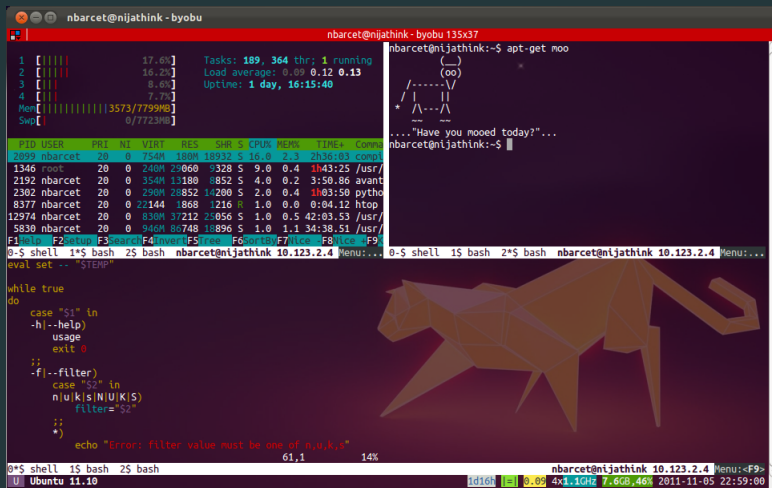
1. On retrouve par exemple `cd`, `alias`, `bg` / `fg`...

2. De la même manière que les *fonctions* dans les langages de programmation.

Travailler avec plusieurs fenêtres

6 consoles virtuelles disponibles, accessibles par **Alt+F1**, **Alt+F2**...

On peut aussi utiliser un **multiplexeur de terminal** comme *GNU Screen*, *tmux* ou *Byobu* :



```
nbarcet@nijathink - byobu
nbarcet@nijathink - byobu 135x37
1 [|||||] 17.6% Tasks: 189, 364 thr: 1 running
2 [|||||] 16.2% Load average: 0.09 0.12 0.13
3 [|||||] 8.6% Uptime: 1 day, 16:15:40
4 [|||||] 7.7%
Mem [|||||] 3573/7799MB
Swp [|||||] 0/7723MB

PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
2899 nbarcet 20 0 754M 180M 18932 S 16.0 2.3 2h36:03 compl
1346 root 20 0 240M 29060 9328 S 9.0 0.4 1h43:25 /usr/
2192 nbarcet 20 0 354M 13180 8852 S 4.0 0.2 3:50.86 avant
2302 nbarcet 20 0 290M 28852 14200 S 2.0 0.4 1h03:50 pytho
8377 nbarcet 20 0 22144 1868 1216 R 1.0 0.0 0:04.12 htop
12974 nbarcet 20 0 830M 37212 25056 S 1.0 0.5 42:03.53 /usr/
5830 nbarcet 20 0 946M 86748 18896 S 1.0 1.1 34:38.51 /usr/

F1 Help F2 Setup F3 Exit F4 Inver F5 Rec F6 Sort By F7 Lics F8 Lics F9
0-$ shell 1*$ bash 25 bash nbarcet@nijathink 10.123.2.4 Menu:...
eval set -- "$@"

while true
do
  case "$1" in
    -h|--help)
      usage
      exit 0
    ;;
    -f|--filter)
      case "$2" in
        n|u|k|s|N|U|K|S)
          filter="$2"
        ;;
        *)
          echo "Error: filter value must be one of n,u,k,s"
        ;;
      esac
    ;;
  esac
  61.1 14%
0-$ shell 1$ bash 25 bash nbarcet@nijathink 10.123.2.4 Menu:<F9>
U Ubuntu 11.10 1d1ch | 9.65 4x1.1GHz 7.6GB,46% 2011-11-05 22:59:00
```

Où trouver de l'aide sur une
commande?

Documentation électronique qui décrit le format et le fonctionnement des commandes, des fichiers, d'outils...

Section	Descriptions
1	Commandes utilisateur
2	Appels système (API du noyau)
3	Appels des bibliothèques (fonctions C)
4	Fichiers spéciaux (situés généralement dans <code>/dev</code>)
5	Formats des fichiers (ex. : <code>/etc/passwd</code>)
6	Jeux, économiseurs d'écran, gadgets...
7	Divers
8	Commandes d'administration

"RTFM !!!"

Consulter le manuel : `man page_souhaitée`

Informations sur une section : `man 3 intro`

`man -f smail` ou `whatis -r smail` : recherche les pages de manuel nommées *smail* et en affiche les descriptions courtes

`man -k printf` ou `apropos printf` : recherche les pages de manuel comportant le mot-clé *printf* dans leur résumé

`info cat` : doc de la commande *cat* au format *GNU info*

Beaucoup de programmes admettent aussi une aide "concise" en les appelant avec `--help`, ou `-h` ou encore `-?`

`adduser --help`

L'interpréteur de commandes (*Shell*) a son propre système d'aide pour les commandes internes : `help commande`

+ livres, web...

Mémento des commandes essentielles

Cette section est un condensé des principales commandes que vous découvrirez dans les premières séances de ce module.

Pas d'inquiétude si certaines vous paraissent "obscur" : vous apprendrez à les utiliser au fur et à mesure des TP.

Utilisation de la console

<code>help <cmd></code>	affiche l'aide de la commande Bash <i>cmd</i>
<code><TAB></code>	autocomplète la commande autant que possible
<code><TAB> + <TAB></code>	affiche toutes les possibilités de complétion
<code>CTRL + L</code>	efface la console
<code>CTRL + S</code>	interrompt le défilement d'un résultat trop verbeux
<code>CTRL + Q</code>	reprend le défilement
<code>CTRL + D</code>	quitte une session
<code>CTRL + U</code>	efface la ligne de commande et place le contenu dans le presse-papier
<code>CTRL + K</code>	efface ce qui se trouve après le curseur et place le contenu dans le presse-papier
<code>CTRL + Y</code>	colle le contenu du presse-papier
<code>;</code>	sépare plusieurs commandes sur la même ligne
<code>ALT + Fk</code>	affiche la k ^{ème} console virtuelle

Historique des commandes

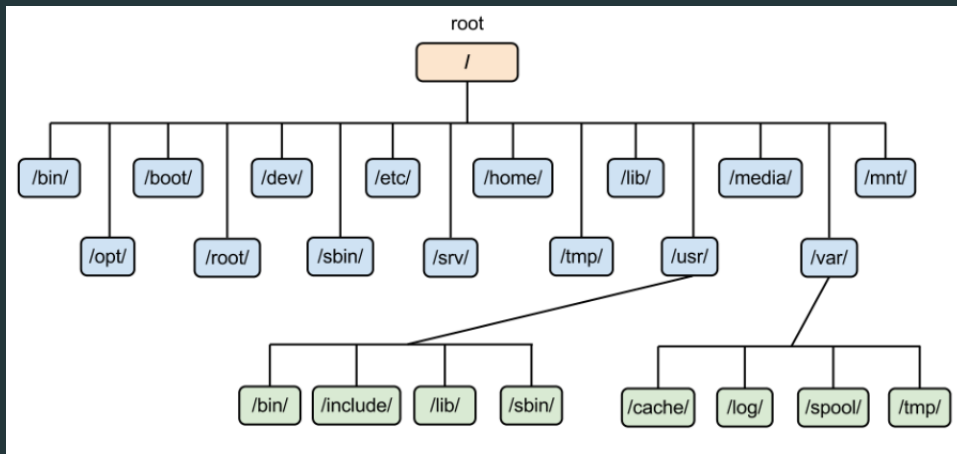
<code>!!</code>	rappelle la dernière commande
<code>history</code>	affiche l'historique des commandes tapées (avec un numéro)
<code>!n</code>	rappelle la commande <code>n</code>
<code>CTRL + R</code>	recherche dans l'historique
<code>ECHAP + .</code>	rappelle le dernier argument de la commande précédente

Navigation dans le système de fichiers

<code>cd</code>	revient au dossier <code>\$HOME</code> (dossier de l'utilisateur)
<code>cd chemin</code>	va dans le dossier spécifié par <i>chemin</i>
<code>cd ..</code>	remonte dans le dossier parent
<code>cd -</code>	revient au dossier dans lequel on était précédemment
<code>~</code>	raccourci pour <code>\$HOME</code> <code>cd ~/musique</code> \Leftrightarrow <code>cd \$HOME/musique</code>
<code>pwd</code>	affiche le dossier courant

Aparté : hiérarchie standard du système de fichiers

Une seule arborescence pour tout le système : FHS¹



1. Filesystem Hierarchy Standard

Aparté : hiérarchie standard du système de fichiers

Répertoire	Description
/	Racine de la hiérarchie primaire et du système de fichiers
/bin	Commandes (<i>binaires</i>) de base nécessaires à l'utilisation d'un système minimal
/boot	Chargeur d'amorçage (<i>bootloader</i>)
/dev	Périphériques (<i>devices</i>)
/etc	Fichiers de configuration (<i>Editable Text Configuration</i>)
/home	Répertoires personnels des utilisateurs
/lib	Bibliothèques logicielles (<i>libraries</i>)
/lost+found	Fichiers récupérés après un crash
/media	Point de montage des médias amovibles (clés USB, CD-ROM...)

Aparté : hiérarchie standard du système de fichiers

/mnt	Point de montage temporaire
/opt	Logiciels optionnels (i.e. non inclus de base)
/root	Répertoire du super-administrateur
/run	Informations sur la session en cours
/sbin	Binaires système et des tâches d'administration
/sys	Informations sur les périphériques, les drivers, le noyau...
/tmp	Fichiers temporaires
/usr	Racine de la hiérarchie secondaire : contient essentiellement les applications utilisateurs
/var	Fichiers divers ou dont le contenu est susceptible de changer en permanence (<i>variable files</i>) : logs, mails, sites web...

Pensez à consulter le manuel : **man hier**

Opérations de base sur les fichiers

<code>ls</code>	liste les fichiers d'un dossier <code>-a</code> : liste aussi les fichiers cachés <code>-l</code> : affiche les détails
<code>ll</code>	alias pour <code>ls -aLF</code>
<code>cat fichier</code>	affiche le contenu d'un fichier
<code>more fichier</code>	affiche page par page
<code>less fichier</code>	semblable à <code>more</code> mais plus élaboré
<code>head -n</code>	affiche les <i>n</i> premières lignes
<code>tail -n</code>	affiche les <i>n</i> dernières lignes
<code>touch fichier</code>	modifie l'horodatage de <i>fichier</i> , ou <i>crée fichier s'il n'existe pas</i>
<code>tar</code>	crée une archive de plusieurs fichiers
<code>gzip / gunzip</code>	compresse / décompresse
<code>zcat, zless</code>	id. <code>cat</code> et <code>less</code> , mais sur des fichiers compressés

Opérations de base sur les fichiers

cp	copie un fichier ou un dossier -r : copie récursive
mv	renomme un fichier ou le déplace dans un autre dossier
rm	supprime un fichier ¹ -r : suppression récursive
mkdir	crée un dossier
rmdir	supprime un dossier vide
ln	crée un lien sur un fichier (≈ copie synchronisée) -s : crée un lien symbolique (symlink) (≈ raccourci)
file	détermine le type d'un fichier (indépendamment de son extension)

1. Attention! Un fichier supprimé est définitivement perdu! (Pas de "corbeille")

Commandes de manipulations

wc	compte le nombre de lignes, de mots et de caractères d'un flux de données (fichiers, résultat d'une commande, etc.)
sort	trie la sortie (par ordre alphabétique, inverse, aléatoire...)
uniq	supprime les doublons de la sortie
cut	coupe chaque ligne de la sortie (selon un nombre de caractères, un séparateur...)
iconv	change l'encodage d'un fichier
grep	recherche par expressions rationnelles (regex)
awk	commande de manipulation très puissante

Chercher des fichiers

<code>locate</code>	recherche dans la base de données des fichiers indexés
<code>sudo updatedb</code>	force la mise à jour de l'index
<code>find</code>	recherche par nom, date, taille...

Redirection de flux

<code>c1 c2</code>	relie la sortie de la première commande à l'entrée de la deuxième
<code>></code>	redirige la sortie (mais pas les erreurs) dans un fichier (qui est écrasé s'il existe déjà)
<code>>></code>	redirige la sortie (mais pas les erreurs) à la fin d'un fichier
<code><</code>	prend un fichier en entrée
<code><<</code>	prend en entrée le clavier au fur et à mesure
<code>2>, 2>></code>	redirige uniquement les erreurs dans un fichier
<code>2>&1</code>	envoie la sortie d'erreurs sur la sortie standard

Multitâches

<code>commande &</code>	passe la commande en arrière-plan
<code>ps aux</code>	affiche tous les processus en cours d'exécution
<code>CTRL + Z</code>	met en pause le processus courant
<code>bg</code>	met le processus en pause en arrière-plan
<code>fg %k</code>	met le processus n°k au premier plan
<code>htop</code>	utilitaire interactif de visualisation des processus
<code>free</code>	état de la mémoire
<code>kill -9 k</code>	tue (violemment) le processus n°k

Autres commandes utiles

<code>echo</code>	affiche ce qui lui est passé en argument
<code>xargs</code>	convertit l'entrée standard en arguments pour une commande
<code>diff</code>	compare le contenu de deux fichiers
<code>which</code>	localise une commande, un binaire
<code>whereis</code>	localise une commande et sa page de manuel

Redirection des flux d'entrées / sorties

Flux d'entrées / sorties standard

Définition

Un **flux d'entrées** (resp. **de sorties**) est un canal de communication par lequel transitent des informations vers (resp. depuis) un programme.

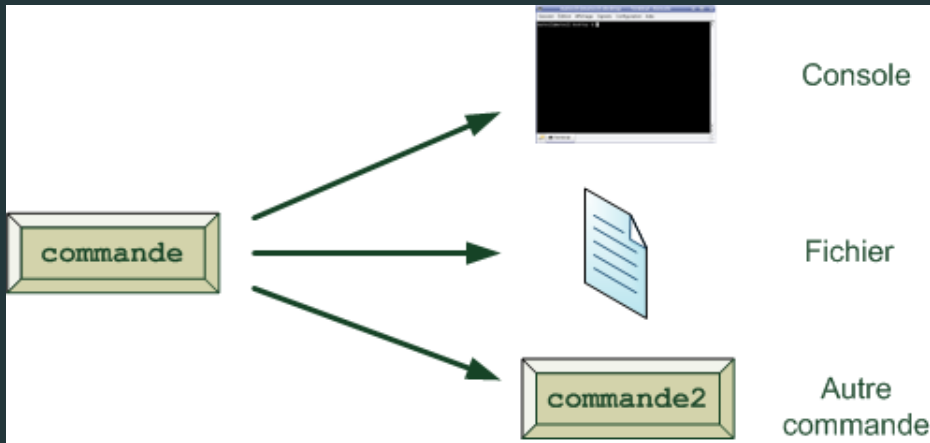
UNIX / Linux fournit trois flux d'entrées / sorties standard :

- l'**entrée standard** (stdin), connectée au clavier,
- la **sortie standard** (stdout), connectée à l'écran,
- la **sortie d'erreur** (stderr), connectée à l'écran.

💡 Sous UNIX / Linux, *tout* est vu comme un fichier, y compris les flux. Chaque fichier ouvert porte un numéro, ou **descripteur de fichier**; les flux ci-dessus portent les numéros respectifs **0**, **1** et **2**.

Redirection des sorties

Un flux peut être vu comme un **tuyau** qu'il est possible de déconnecter pour le connecter ailleurs. Par exemple, on peut **rediriger la sortie standard** vers un fichier ou vers une autre commande :



Redirection dans un fichier

L'opérateur `>` redirige la sortie d'une commande dans un fichier au lieu de l'afficher à l'écran :

```
greg@cpe:~$ ls *.xls
eleves.xls
notes.xls
planning.xls
greg@cpe:~$ ls *.xls > liste_fichiers_Excel.txt
greg@cpe:~$ cat liste_fichiers_Excel.txt
eleves.xls
notes.xls
planning.xls
```

💡 Si le fichier destination n'existait pas déjà, il est créé automatiquement

⚠ **Attention!** Si le fichier destination existait déjà, son contenu est effacé et remplacé, sans avertissement (à moins d'activer l'option Bash *noclobber*)!

Redirection dans un fichier

L'opérateur `>>` redirige la sortie à la fin d'un fichier :

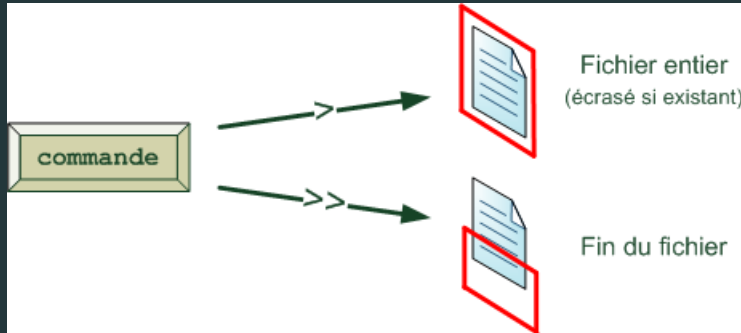
```
greg@cpe:~$ cat liste_fichiers_Excel.txt
eleves.xls
notes.xls
planning.xls
greg@cpe:~$ ls 2019/*.xls >> liste_fichiers_Excel.txt
greg@cpe:~$ cat liste_fichiers_Excel.txt
eleves.xls
notes.xls
planning.xls
notes2019.xls
```

💡 Si le fichier destination n'existait pas déjà, il est créé automatiquement

Redirection dans un fichier

En résumé :

- **>** : redirige dans un fichier et l'écrase s'il existe déjà ;
- **>>** : redirige à la fin d'un fichier et le crée s'il n'existe pas.



Source : OpenClassRooms

Redirection dans un fichier

Rem. : il n'est pas possible de rediriger un traitement sur un fichier dans le **même** fichier :

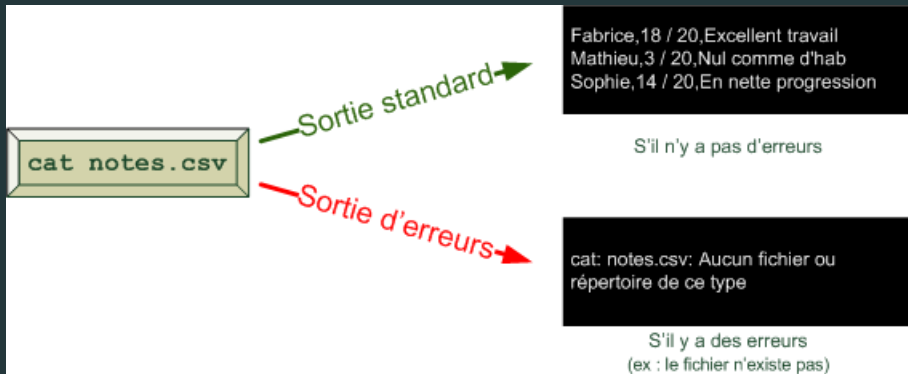
```
greg@cpe:~$ cat f.txt > f.txt
==> Efface le contenu de f.txt !
greg@cpe:~$ cat f.txt >> f.txt
cat: f.txt : le fichier d'entrée est aussi celui de sortie
```

💡 Bash ouvre le fichier de destination **avant** d'exécuter la commande. Avec le symbole **>**, le fichier est donc supprimé avant d'être traité; avec le symbole **>>**, Bash proteste car il devrait écrire à la fin d'un fichier en cours de modification.

Redirection des erreurs

Les messages affichés à l'écran peuvent l'être *via* deux flux différents :

- **sortie standard** : messages affichés en situation normale
- **sortie d'erreur** : messages affichés en cas d'erreur



Source : OpenClassRooms

Redirection des erreurs

Pour rediriger les erreurs, on utilise les opérateurs `2>` et `2>>`

A retenir

On retrouve ici le descripteur de fichier associé à la sortie d'erreur.
L'opérateur `>` (resp. `>>`) est d'ailleurs un synonyme de `1>` (resp. `1>>`).

```
greg@cpe:~$ convert *.* -o pdf 2> erreurs.txt
OK : conversion de monsite.html
greg@cpe:~$ cat erreurs.txt
Erreur : impossible de convertir linux.iso
Erreur : impossible de convertir script.sh
```

💡 Souvent, on redirige les erreurs vers le fichier spécial `/dev/null` pour les éliminer

Redirection des erreurs

Si on veut rediriger la sortie standard **et** les erreurs dans le **même** fichier, on utilise la syntaxe **2>&1** :

```
greg@cpe:~$ convert *.* -o pdf > sortie.txt 2>&1
greg@cpe:~$ cat sortie.txt
Erreur : impossible de convertir linux.iso
OK : conversion de monsite.html
Erreur : impossible de convertir script.sh
```

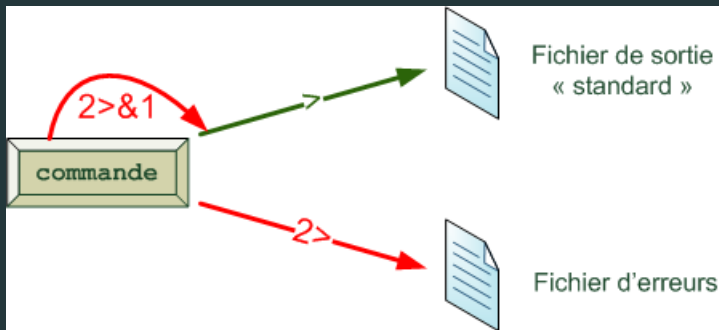
💡 Il n'est pas possible d'écrire **2>>&1**. Si on veut ajouter les messages à la fin d'un fichier, on écrira :

```
greg@cpe:~$ convert *.* -o pdf >> sortie.txt 2>&1
```

Redirection des erreurs

En résumé :

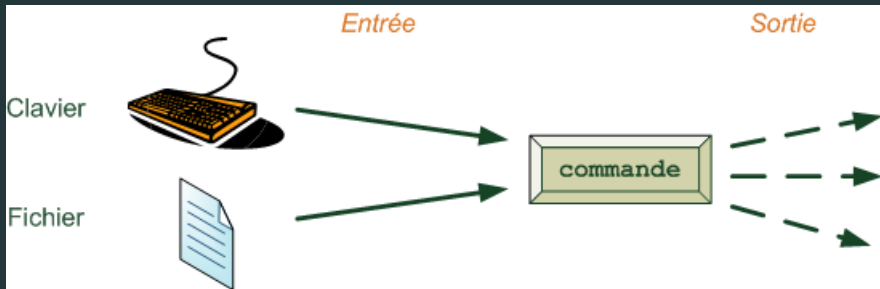
- **2>** : redirige les erreurs dans un fichier (écrasé s'il existe déjà)
- **2>>** : redirige les erreurs à la fin d'un fichier (créé s'il n'existe pas)
- **2>&1** : redirige les erreurs au même endroit et de la même façon que la sortie standard



Source : OpenClassRooms

Redirection des entrées

L' **entrée standard** d'une commande est le clavier; mais il est possible de la rediriger depuis un fichier avec l'opérateur < (ou **0<**) :



Source : OpenClassRooms

Redirection des entrées

Par exemple, la commande `tr` lit des chaînes de caractères saisie au clavier et remplace / supprime des caractères :

```
greg@cpe:~$ tr a-z A-Z      # convertit en majuscules
test
TEST
```

Mais on peut rediriger l'entrée vers un fichier, qui alimentera la commande :

```
greg@cpe:~$ cat fichier
hello world!
greg@cpe:~$ tr a-z A-Z < fichier
HELLO WORLD!
```

Bien sûr, on peut rediriger les entrées **et** les sorties dans la même commande :

```
greg@cpe:~$ cat original
hello world!
greg@cpe:~$ tr a-z A-Z < original > resultat
greg@cpe:~$ cat resultat
HELLO WORLD!
```

Ici, le fichier *original* alimente la commande **tr**, et le résultat est écrit dans le fichier *resultat*.

Redirection des entrées

L'opérateur `<<` permet de créer un *here document* (ou *document en ligne*), c'est-à-dire un document créé directement dans la console (sans passer par un fichier) :

```
greg@cpe:~$ tr a-z A-Z << FIN
> hello
> ceci est un heredoc
> FIN
HELLO
CECI EST UN HEREDOC
```

💡 La saisie du heredoc s'arrête lorsqu'on tape le délimiteur spécifié après l'opérateur `<<`.

💡 Bash dispose aussi de l'opérateur `<<<` qui lit une seule chaîne (*here string*).

Les redirections permettent d'écrire des commandes complexes comme :

```
ls > fichier && tr [:lower:] [:upper:] < fichier
```

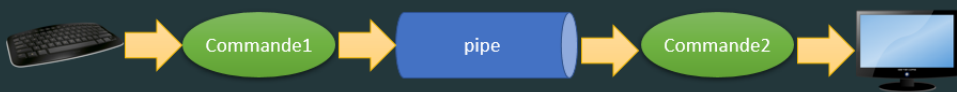
Toutefois, les commandes de ce genre ont plusieurs inconvénients :

- elle crée un fichier intermédiaire qui n'est pas forcément utile;
- la commande `tr` attend que la commande `ls` se soit terminée avant de commencer son traitement;
- dans des cas extrêmes, on pourrait manquer d'espace disque pour stocker `fichier`.

Les pipelines sont la réponse à ces inconvénients!

Pipelines

Un *pipe* (ou *tube*, ou *tuyau*) est un canal de communication qui relie directement la sortie d'une commande à l'entrée d'une autre commande. Un *pipeline* est un ensemble de commandes reliées par des pipes.



On crée un pipeline en utilisant la syntaxe suivante¹ :

`commande1 | commande2`

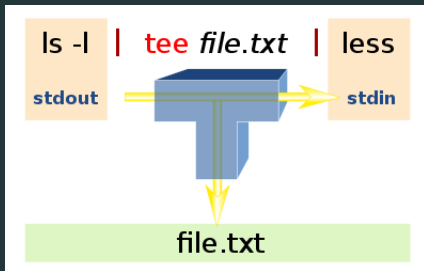
💡 L'opérateur `|` redirige uniquement la sortie standard ; pour rediriger aussi la sortie d'erreur, Bash propose l'opérateur `|&`, qui est un raccourci pour `2>&1 |`.

1. On peut bien sûr enchaîner les pipes pour créer des commandes complexes :

`ls | sort -r | tr [:lower:] [:upper:]`

Double redirection

La commande **tee** lit depuis l'entrée standard et écrit simultanément sur la sortie standard **et** un fichier :



💡 Utiliser **tee -a** pour ne pas effacer le contenu du fichier de destination.

xargs

De nombreuses commandes (**head**, **tail**, **sort**, **uniq**, **tr**...) utilisent l'entrée standard. Mais quelques autres (**ls**, **rm**...) n'utilisent que des *arguments*.

La commande **xargs** permet de *convertir l'entrée standard en arguments pour une deuxième commande*¹.

Exemple : suppression des fichiers de plus de deux semaines dans le dossier **/tmp** :

```
greg@cpe:~$ find /tmp -mtime +14 | xargs rm
```

1. Par défaut, c'est la commande **echo** qui est utilisée.

Entrée standard vs. arguments

Contrairement à la commande **tr**, la plupart des autres commandes acceptent aussi leurs entrées *via* des *arguments* :

```
greg@cpe:~$ cat -n < fichier
1 hello world!
greg@cpe:~$ cat -n fichier
1 hello world!
```

Ici, le résultat est le même, mais le principe sous-jacent est différent :

- dans le 1er cas, le fichier est ouvert par Bash et le *contenu* du fichier est transmis au programme
- dans le 2nd cas, le *nom* du fichier est transmis au programme **cat**, qui se charge de l'ouvrir et lire son contenu, *parce que le développeur du programme lui a donné cette fonctionnalité.*