Les filtres sous *Unix*

Avertissement : Toutes les commandes données dans ce cours sont issues d'une `SuSe` **Linux 7.1** avec `bash` comme shell. Le nom et la syntaxe des commandes présentées peuvent évoluer en fonction des versions utilisées.

Notion d'entrée/sortie et de filtre

➤ On définit :

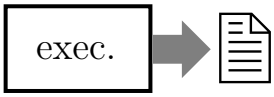
`stdin` est l'entrée standard, *cad* où les informations attendues en entrée sont lues (par défaut le clavier). Descripteur = 0.

`stdout` est la sortie standard, *cad* où les résultats d'une commande sont transmis (par défaut l'écran). Descripteur = 1.

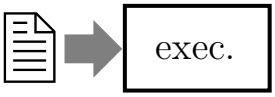
`stderr` est la sortie d'erreur, *cad* où les messages d'erreur sont transmis (par défaut l'écran). Descripteur = 2.

➤ *Unix* offre des possibilités avancées pour la gestion des entrées et des sorties d'une commande. Elles se résument en deux concepts :

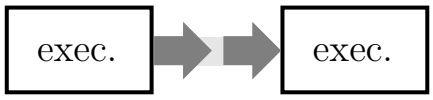
- **les redirections**
  - **redirection de sortie** :
    la sortie d'une commande est envoyée vers un fichier (par exemple).



  - **redirection d'entrée** :
    l'entrée d'une commande est lue depuis un fichier (par exemple).



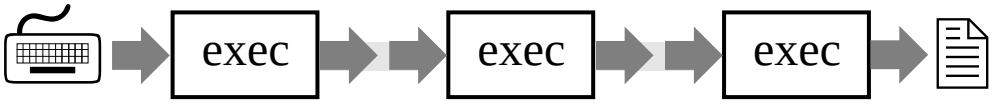
- **le pipelining**
  la sortie d'une commande est utilisée comme entrée d'une autre commande.



➤ Un **filtre** est une commande ayant les capacités :

- de lire son entrée sur le canal d'entrée standard ou depuis un fichier ou un ensemble de fichiers passés en paramètre.
- d'écrire sa sortie sur le canal de sortie standard et ses erreurs sur le canal de sortie d'erreur.

➤ **Exemple de combinaison de filtres par pipelining**



Syntaxe de redirection et de pipelining

➤ Syntaxe des redirections

`command > file` : place le résultat de la sortie de `command` (`stdout`) dans le fichier nommé `file`. Si `file` existe, il est écrasé ; sinon, il est crée.

`command 2> file` : comme ci-dessus, mais concerne la sortie `stderr` (i.e. les messages d'erreur issus de la commande).

`command >> file` : ajoute à la fin du fichier `file` le résultat de la sortie de `command` (`stdout`). Si le fichier n'existe pas, il est créé.

`command 2>> file` : comme ci-dessus, mais concerne la sortie `stderr`.

`command < file` : prend comme entrée (`stdinn`) de la commande `command` le contenu du fichier `file`. Ce fichier contient en général soit les données à traiter (cas des filtres), soit les réponses aux questions posées par un programme (cas des programmes inter-actifs).

➤ Syntaxe de pipelining

`command1 | command2` : passe le résultat de la sortie de `command1` comme entrée à `command2`.

`command1 | tee file | command2` : comme ci-dessus, mais la sortie de `command1` est en plus placée dans le fichier `file`.

➤ Le fichier virtuel `/dev/null`

`/dev/null` est un fichier virtuel toujours vide utilisée dqns les redirections.

utilisé en entrée , il n'envoie rien (on l'utilise pour s'assurer qu'une commande ne reçoit rien en entrée).

utilisé en sortie , il joue le rôle de poubelle (on redirige dans `/dev/null` les sorties que l'on ne souhaite pas).

Filtres d'affichage

❑ **cat** : affichage.

- ❑ *affichage simple* : **cat** `fichier`
- ❑ *concaténation de fichier* : **cat** `fichier_1` ... `fichier_n`
  Les fichiers `fichier_i` sont envoyés dans l'ordre et sans séparation vers `stdout`.
- ❑ *création de fichier* : **cat** > `fichier`
  mon texte.
  ^D
  Aucun fichier n'est donné en entrée, c'est donc l'entrée `stdin` standard (le clavier) qui est utilisée et redirigée vers `fichier`.

❑ **more** (**less**) : affichage avec pause à chaque page.

- ❑ *consultation d'un fichier* : **more** `fichier`
  Possède des commandes de recherche similaires à celle de **vi**.
- ❑ *consultation page à page du résultat d'une commande* :

  `command` | **more**

Cette commande étant interactive, sa sortie n'est généralement ni redirigée ni pipée.

❑ **head** : affichage du début du flux d'entrée.

options : **-n** n : affichage des n premières lignes (ou -n).
-c n : affichage des n premiers caractères.

❑ **tail** : affichage de la fin du flux d'entrée.

options : **-n** n : affichage des n dernières lignes (ou $\pm n$).
-c n : affichage des n derniers caractères.

si le paramètre n est immédiatement précédé du signe $+$, alors la référence est prise depuis le début du fichier (par exemple, la fin du fichier en commençant à la $n^{\text{ième}}$ ligne).

❑ **rev** : affichage inversé du flux d'entrée (ligne à ligne).

Analyse de flux

❑ **od** : dump au format choisi (permet l'analyse de flux texte ou binaire).

❑ **Options** :

-t c : affiche au format décrit par c où c peut prendre les valeurs :

- **c** caractères.
- **a** caractères nommés.
- **d**n entier sur n octets.
- **u**n entier non signé sur n octets.
- **x**n hexadécimal sur n octets.
- **f**n flottant sur n octets.

-wn : sortie avec n bytes par ligne.

❑ **Exemple**

```
% od -t a -w8 toto
0000000   a   a   a   b   b   c   c  sp
0000010   c   c   c   b   b   a   a   a
0000020  nl
% od -t x1 -w8 toto
0000000 61 61 61 62 62 63 63 20
0000010 63 63 63 62 62 61 61 61
0000020 0a
0000021
% od -t u1 -w8 toto
0000000  97  97  97  98  98  99  99  32
0000010  99  99  99  98  98  97  97  97
0000020  10
0000021
```

❑ **wc** : comptage des lignes, mots et caractères.

❑ **Options**

sans option équivalent à **-lwc**.

- **-c** renvoie le nombre de caractères.
- **-w** renvoie le nombre de mots.
- **-l** renvoie le nombre de lignes.

❑ **Exemple**

```
% cat toto
aaabbcc cccbbaaa
% wc toto
       1       2      17 toto
```

Filtre de tri : **sort**

Tri ligne à ligne par ordre croissant.

❑ **Principales options** :

-n	tri numérique (**-g** pour les réels).
-r	inverser l'ordre du tri.
-f	ne pas différencier majuscules et minuscules.
-t *c*	utiliser le caractère *c* comme séparateur de champs dans la ligne (par défaut, l'espace).
-k *fields*	tri en fonction du sélecteur de champ.

❑ **Principe de sélection des champs** (*fields*) :

n,n	tri selon le $n^{\text{ième}}$ champ.
n	tri selon la fin de la ligne en commençant à partir du $n^{\text{ième}}$ champ.
n.m	tri selon la fin de la ligne en commençant à partir du $m^{\text{ième}}$ caractère du $n^{\text{ième}}$ champ.
$\mathbf{n_1}, \mathbf{n_2}$	tri selon les champs à partir du champ n_1 et jusqu'au champ n_2.
$\mathbf{n_1}.\mathbf{m_1}, \mathbf{n_2}.\mathbf{m_2}$	etc ...

❑ **Exemple** : si une ligne du fichier à trier est la suivante :

```
toto titi ta :ta
```

alors :

- ❖ si l'option **-t** n'est pas précisée, il y a 3 champs : `toto`, `titi` et `ta :ta`.
- ❖ avec l'option **-t :**, il y a deux champs : `toto titi ta` et `ta`.
- ❖ l'option `-k 2,2` fait le tri sur le deuxième champ seulement (`titi`).
- ❖ l'option `-k 2.2` fait le tri à partir du deuxième caractère du deuxième champ (`ti ta :ta`).

Gestion des entrées multiples : **uniq**

Gestion des entrées multiples d'un flux trié.

❑ **Options de gestion** :

affiche les lignes en supprimant les entrées multiples
-u affiche seulement les lignes uniques
-c affiche le nombre d'occurence.
-d affiche seulement les lignes multiples.

❑ **Options sur les champs** :

-f n saute les n premiers champs.
-i ignorer la casse (minuscules/majuscules).

❑ **Exemple** :

```
% cat toto
1
1
1
2
2
3

% uniq toto
1
2
3

% uniq -d toto
1
2

% uniq -u toto
3

% uniq -c toto
      3 1
      2 2
      1 3
```

Gestion de champs

❑ **cut** : extractions et réorganisation de champs.

❑ **Options principales** :

-c *liste* coupure en mode caractère.

-f *liste* coupure en mode champ.

-d *c* utiliser le caractère *c* comme séparateur de champs (par défaut, la tabulation).

❑ **Format de la *liste*** :

n le $n^{\text{ième}}$.

n,m le $n^{\text{ième}}$ et le $m^{\text{ième}}$.

n-m du $n^{\text{ième}}$ au $m^{\text{ième}}$.

n- du $n^{\text{ième}}$ à la fin.

-n du 1^{er} au $n^{\text{ième}}$.

❑ **Exemple** :

```
% cat toto
1       2       3
12      23      34
```

```
% cut -c 2-4,5 toto
        2       3
2       23
```

```
% cut -f 3,1 toto
1       3
12      34
```

❑ **join**[1] : union de deux fichiers par un champ commun (jointure).

❑ **Syntaxe** :

`join` **-1** c_1 **-2** c_2 **-o** *field_list* $fichier_1$ $fichier_2$

La jointure des deux fichiers est réalisée en utilisant le champ c_1 dans $fichier_1$ et le champ c_2 dans $fichier_2$ (par défaut, séparés par des blancs). Les champs affichés en sortie sont ceux spécifiés par *field_list*.

❑ **Liste des champs de sortie** (*field_list*) : les indications suivantes sont séparées par des virgules.

0 le champ de jointure.

1.n le $n^{\text{ième}}$ champ du 1^{er} fichier.

2.m le $m^{\text{ième}}$ champ du 2^{nd} fichier.

❑ **Exemple** :

```
% cat auteurs
victor hugo
arthur rimbaud
% cat livres
hugo hernani
rimbaud illuminations
```

```
% join -1 2 -2 1 auteurs livres
hugo victor  hernani
rimbaud arthur  illuminations
% join -1 2 -2 1 -o 1.1,2.2 auteurs livres
victor hernani
arthur illuminations
```

[1] Ce n'est pas un filtre. **ATTENTION : les implémentations de *join* peuvent conduire à des comportements très différents d'un système à l'autre.**

Modification d'un flux

❑ **tr** : transcodage et supression des caractères. La seule entrée de ce filtre est `stdin`.

❑ **Options** :

table tablefin	transcodage de tous les caractères contenus dans *table* un à un avec les caractères contenus dans *tablefin* (de même longueur que *table*).
-d *table*	suppression de tous les caractères contenus dans *table*.
-s *table*	suppression des occurences multiples consécutives des caractères contenus dans *table*.
-c	inversion de *table*.

❑ **Table de caractères** : les tables de caractères sont à donner entre crochets, avec les facilités suivantes :

`A-F`	les caractères de `A` à `F`.
`A*4`	le caractère `A` répété 4 fois.
`A*`	(dans *tablefin* seulement) autant de `A` qu'il faut pour atteindre la longeur de *table*.

❑ **Exemples** :

```
% cat toto
aaabbcc cccbbaaa
% cat toto | tr [a-c] [A-C]
AAABBCC CCCBBAAA
% cat toto | tr -d [ac]
bb bb
```

```
% cat toto | tr -s [bc]
aaabc cbaaa
% cat toto | tr [a-c] [-*]
------- --------
% cat toto | tr -dc [ac]
aaacccccaaa
```

❑ **sed** : éditeur de texte batch (édition automatique d'un texte par une suite d'instructions programmées).

Les instructions les plus directement utilisables sont la **s**ubstitution et la suppression :

`s/`$expr_1$`/`$expr_2$`/g`	remplace partout $expr_1$ par $expr_2$.
`/`*expr*`/d`	efface toutes les lignes contenant *expr*.

❑ **Exemple :**

```
% cat toto
aaabbcc cccbbaaa
% sed s/a//g toto
bbcc cccbb
```

```
% cat titi
s/b/bc/g
s/c/de/g
% sed -f titi toto
aaabdebdedede dededebdebdeaaa
```

Expressions régulières

Les expressions régulières sont utilisées dans de nombreux programmes ou filtres *Unix* lorsque des chaînes de caractères sont à rechercher. Attention, toutes les fonctionnalités ci-dessous (norme `POSIX 1003.2`) ne sont pas nécessairement présentes.

❑ **Caractères ou de groupe de caractères**

c	le catactère c si ce n'est pas un meta-caractère.
$\backslash c$	le caractère littéral c dans le cas d'un meta-caractère.
`.`	un caractère, n'importe lequel (saut de ligne compris).
`M-R`	la plage des caractères de `M` à `R`.
`[`*abc...*`]`	un caractère parmi *abc...* (ceux spécifiés entre crochets).
`[^`*abc...*`]`	un caractère qui n'est pas parmi *abc...* (aucun de ceux spécifiés entre crochets).

❑ **Répétition** : on répète une expression en plaçant immédiatement après l'un des opérateurs suivants :

`*`	se répète un nombre quelconque de fois (0 compris).
`+`	se répète au moins une fois.
`?`	se répète au plus une fois.
$\backslash\{n\backslash\}$	se répète exactement n fois
$\backslash\{n,\backslash\}$	se répète n fois ou plus.
$\backslash\{,m\backslash\}$	se répète au plus m fois.
$\backslash\{n,m\backslash\}$	se répète n fois au moins et m fois au plus.

❑ **Groupement** : un groupement est une expression régulière *exp* contenant toute combinaison de caractères, d'opérateurs, ou de groupement.

$exp_1 exp_2$	correspond à exp_1 **puis** exp_2 (concaténation).
$exp_1 \vert exp_2$	correspond à exp_1 **ou** exp_2.
`(`*exp*`)`	groupe le contenu de l'expression *exp* et la marque.
$\backslash n$	fait référence à la $n^{\text{ième}}$ expression marquée.

❑ **Position**

`^`	début de ligne.
`$`	fin de ligne.
`\<`	début de mot.
`\>`	fin de mot.

Exemples d'expression régulière

`[0-9]+`	un nombre entier qui peut commencer par un ou plusieurs zéros.
`[^aeyuio]`	un caractère qui n'est pas une voyelle.
`0x[0-9a-fA-F]+`	un nombre hexadécimal (par exemple `0xf2a3`).
`[1-9][0-9]\{3\}`	un nombre entier à quatre chiffres (dont le premier n'est pas 0).
`[A-Z][a-z]*`	un mot commençant par une majuscule.
`([A-Z]+)|([a-z]+)`	un mot exclusivement écrit soit en majuscules, soit en minuscules.
`\<loin\.`	une phrase se terminant par le mot *loin*.
`\.$`	une ligne se terminant par un point.
`^toto$`	*toto* écrit seul sur une ligne.
`(1|2)|(3|4)`	un chiffre commençant par 1 ou 2 et se terminant par 3 ou 4.
`(1|2)\1`	soit 11, soit 22.
`(1|2)(3|4)\2\1`	1331 ou 1441 ou 2332 ou 2442.

Filtre de recherche : **grep**

recherche d'expression (régulière) dans un flux.

❑ **Définition du critère de recherche** :

-e *regexp* le critère de recherche est défini par *regexp* (éventuellement entre double quote).

-f *file* les critères de recherche sont définis dans le fichier *file* (un par ligne).

❑ **Options de gestion** :

affiche les lignes répondant au critère.

-n préfixe chaque ligne répondant au critère par son numéro de ligne dans le fichier.

-c compte les lignes répondant au critère.

-l affiche les fichiers répondant au critère.

❑ **Autres options** :

-i ignorer la casse (minuscules/majuscules).

-v inversion du critère.

-E utilisation d'expressions régulières étendues (**grep -E** ≡ **egrep**).

-F utilisation de chaînes fixes comme critère (**grep -F** ≡ **fgrep**).

❑ **Exemple** :

```
% cat toto
1       2       3
12      23      34
% grep 4$ toto
12      23      34
% grep -n 4$ toto
2:12    23      34
% grep -c 4$ toto
1
% grep -vn 4$ toto
1:1     2       3
% grep -E "([1234]).\1" toto
12      23      34
```

Recherche de fichiers : **find**
(1) les arguments

recherche et exécution de commandes récursivement sur une arborescence[2]. Toutes les expressions régulières sur les noms de fichier *regexf* sont à donner entre simple cote (')

❑ **Argument numérique** n :

n	exactement égal à n.
$+n$	plus grand que n.
$-n$	moins grand que n.

❑ **Critères de recherche** (extrait) :

-name *regexf*	nom du fichier (**-iname** pour ignorer la casse).
-path *regexf*	nom du chemin (**-ipath** pour ignorer la casse).
-amin n	accès au fichier il y a n minutes (**-atime** jours).
-mmin n	données du fichier modifiées il y a n minutes.
-empty	le fichier est vide.
-perm n	la permission du fichier est n (en octal).
-size n	la taille du fichier est n (en blocks, n`c` en byte, n`k` en Kb).
-type c	le type du fichier est c (`d`=répertoire, `f`=fichier, `l`=lien).

❑ **Tests booleens** : si $test_1$ et $test_2$ sont deux test de recherche, alors :

($test_1$)	force la précédence.
$test_1$ **-and** $test_2$	$test_1$ **et** $test_2$ (le **-and** est facultatif).
$test_1$ **-or** $test_2$	$test_1$ **ou** $test_2$.
-not $test_1$	test inverse de $test_1$.

❑ **Actions** : elles sont exécutées sur tous les fichiers ou les répertoires satisfaisant le critère.

-print	affichage simple.
-fprint *file*	comme **-print**, mais sauvegarde dans *file*.
-ls	affichage comme `ls -dils`
-fls *file*	comme **-ls**, mais sauvegarde dans *file*.
-exec *commande* \ **;**	exécute *commande*. \{\} représente le nom du fichier ou répertoire courant.

[2] ce n'est pas un filtre

Recherche de fichiers : **find**
(2) exemples

❑ **Syntaxe** :

find *chemin(s) critère(s) action(s)*

- ❖ *chemin* est le nom du répertoire initial d'où la recherche va commencer. Au moins un *chemin* est obligatoire (et en première position).
- ❖ si aucun critère n'est précisé, tous les fichiers ou répertoires sont traités.
- ❖ si aucune commande n'est précisée, l'action par défaut est **-print**.

❑ **Exemples** :

```
% find /home/pascal -name toto -print
```

recherche le fichier `toto` dans l'arborescence en dessous de `/home/pascal`.

```
% find /home/pascal -type f -exec chmod =rwxr-xr-x \{\}\ ;
```

change tous les droits d'accés aux fichiers dans `/home/pascal` à la valeur `rwxr-xr-x`.

```
% find /home/pascal -mtime +180 -atime +180 -print
```

affiche tous les fichiers depuis `/home/pascal` qui n'ont été ni modifiés, ni changés depuis plus de 6 mois (180 jours).

```
% find /home/pascal -size +100k -type f -print
```

trouve tous les fichiers plus gros que 100 ko depuis `/home/pascal`.

awk : recherche et traitement de motifs. (1)

awk est un langage de programmation à lui seul dont la syntaxe est proche de celle du `C`.

- **Passage du flux dans awk**
  - **awk** traite le flux enregistrement par enregistrement (*i.e.* une exécution par enregistrement). Par défaut, le séparateur d'enregistrement est la nouvelle ligne.
  - chaque ligne est divisée en champs en fonction d'un séparateur (par défaut l'espace).
- **Structure d'un programme awk**

BEGIN { *actions*$_i$ **}**	ces actions sont exécutées avant l'ouverture du flux (par exemple, l'initialisation des variables `FS` et `RS`).
critere$_1$ **{** *actions*$_1$ **}** *critere*$_2$ **{** *actions*$_2$ **}** ⋮	**programme principal** : pour enregistrement, pour chaque *critère* vérifié, l'*action* est exécutée.
END { *actions*$_f$ **}**	ces actions sont exécutées après la fermeture du flux.

- **Les variables prédéfinies** : lors de l'*action*, les variables suivantes sont définies à chaque enregistrement :
  - **NF** le nombre de champs dans l'enregistrement courant.
  - **NR** le nombre d'enregistrement déjà lu.
  - **$0** l'enregistrement complet.
  - **$***n* le champ numéro n (de 1 à **NF**).
- **Les critères**
  - **critère vide** : si aucun critère n'est spécifié, l'action est exécutée sur tout l'enregistrement.
  - **critère de recherche** : */regex/*
    le critère est vérifié si l'expression régulière *regex* est trouvée dans l'enregistrement. Le critère peut être limité à un champ avec :
    $$\$n \sim /regex/.$$
  - **critère de comparaison** : porte sur les variables ou des opérations sur elles, et suit la syntaxe du `C` (`>`,`<`,`>=`,`<=`,`==`,`!=`,`&&`,`||`,`!`).

awk : recherche et traitement de motifs. (2)

- **Programmation**
  - **Structures de controle** : syntaxe pareille au `C`.
    if, **while**, **do**, **for**, **break**, **continue**, **exit**.
  - **Entrée/Sortie**

getline	passe à l'enregistrement suivant.
print *var(s)*	affichage de variables.
printf(...)	affichage formaté (idem `C`)

  - **Traitement de chaîne de caractères**

sub, **gsub**, **gensub**	substitution.
index, **match**	position d'un motif dans une chaîne.
length	longueur de chaîne.
split	découpage à partir d'un séparateur.
substr	extraction d'une sous-chaîne.
tolower,**toupper**	changement de la casse.

- **Options**

-f *fichier*	lecture du programme **awk** depuis le *fichier*.
-F *i*	changement du séparateur de champs.
-v *var=val*	assignation d'une valeur à une variable utilisable dans **awk**.

- **Exemples**

```
awk 'NR % 2 == 0 { print $0 }' toto
```
affiche une ligne sur deux du fichier `toto`.

```
awk -F ":" '{ print $1 }' /etc/passwd
```
affiche la liste des comptes déclarés dans `/etc/passwd`.

```
awk -F ":" '/^a/{ print $1 }' /etc/passwd
```
affiche la liste des comptes commençant par `a`.

```
awk '{ print NR }' toto
```
affiche le nombre de ligne du fichier `toto`.

```
% wc toto | awk '{ printf("%s: %d lignes, %d mots, %d caracteres\n",\
$4,$1,$2,$3) }'
toto: 1 lignes, 2 mots, 17 caracteres
```
formatage de la sortie de **wc**.