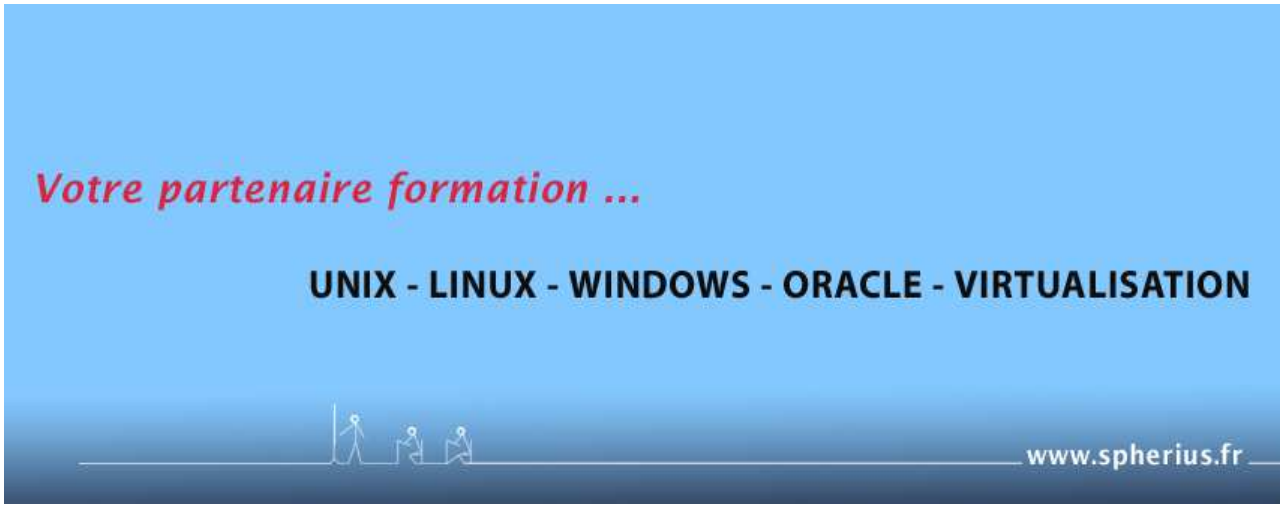
Script Shell

Linux - Unix



SOMMAIRE

Les auteurs de ce document sont :

- Monsieur Baranger Jean-Marc,
- Monsieur Schomaker Theo.

La version du support de cours est:

ShellScript_v1.1

L'introduction et le shell

Dans ce chapitre nous allons découvrir les différents shells ainsi que leurs fonctionnalités.

L'introduction et le shell

- Le rôle du shell
- Les différents shells
- La compatibilité des différents Shells
- Les alias de commandes
- L'ordre de recherche de commandes
- L'interprétation d'une commande

Introduction

Le rôle du shell

- Le shell est un programme
- Affecté à l'utilisateur lors de sa connexion
- Interpréteur de commandes
- Historique des shells : sh, ksh, bash
- Règles d'interprétation du shell

Le rôle du shell

Chaque utilisateur se connectant sur un serveur se voit affecté un shell. Celui-ci est déterminé par le fichier /etc/passwd. Plusieurs shells peuvent cohabiter sur le même serveur. Le shell est un programme, c'est un interpréteur de commandes. Cela correspond à une interface entre l'utilisateur et le système d'exploitation. Il y a au moins un shell par utilisateur connecté.

Le shell se présente sous la forme d'une interface en ligne de commande accessible depuis un terminal. L'utilisateur tape une commande au clavier. Celle-ci est interprétée selon les règles du shell et est ensuite exécutée.

Son but est d'interpréter les lignes de commandes saisies par l'utilisateur, d'envoyer le résultat interprété au noyau pour exécution. Puis le shell récupérera le résultat de l'exécution de la commande du noyau pour l'affichage.

Un shell est unique à un utilisateur, il sera lancé lorsque l'utilisateur ouvrira une invite de commande (prompt).
Le prompt d'un utilisateur est caractérisé par le caractère «**$**». A l'exception de l'administrateur « root » qui est caractérisé par le caractère «**#**».

Un shell est paramétrable, ainsi l'environnement de l'utilisateur est personnalisable.

1971 : Le premier shell est apparu pour la première version d'Unix : Thompson Shell (en) écrit par Ken Thompson.

1977 : Il a été remplacé par le Bourne Shell (sh) développé par Stephen Bourne pour la version 7 d'unix.

1978 : Le C shell (csh), développé par Bill Joy (université de Berkeley, donnant le modèle BSD), avec une syntaxe proche du langage C. Dans ce shell a été introduit la réutilisation de l'historique de commandes. Une évolution du csh est le tcsh (turbo c shell).

1983 : Le Korn Shell (ksh), de David Korn. Il est compatible avec le Bourne Shell et intègre certaines fonctionnalités du csh.

1988 : Le Bourne-Again Shell (bash), de Brian Fox pour la FSF dans le cadre du projet GNU. Ce shell est utilisé par défaut dans de nombreux Linux. Il est compatible avec le Bourne Shell.

1990 : Le zsh, de Paul Falstad. Il reprend les fonctionnalités les plus pratiques du bash, csh et tcsh.

L'introduction et le shell

Les différents shells

- Shell à la connexion
- Rappels de commandes : mode emacs et mode vi
- Complétion de la commande
- Paramétrage du shell : set -o
- Historique de commandes

Les différents shells

Tous les shells n'ont pas exactement les mêmes fonctionnalités. Le Bourne Again Shell est considéré par beaucoup d'utilisateur comme le shell le plus convivial, notamment grâce au rappel simple de commandes et à la complétion de la commande. Ces fonctionnalités existent aussi en Korn Shell. A noter que sur presque tous les systèmes récents le sh est un lien symbolique sur le bash.

Le fichier /etc/passwd détermine avec quel shell l'utilisateur est connecté sur le système.

Remarque: il faut que le shell soit listé dans le fichier /etc/shells qui contient la liste des shells valides pour se connecter graphiquement.

```
# grep user1 /etc/passwd
user1:x:1001:1001::/home/user1:/bin/ksh
# grep -w ksh /etc/shells
/bin/ksh
```

Le rappel des commandes peut se faire grâce à deux modes :

- le mode emacs activé par défaut sur la plupart des shell.
- le mode vi. Mode moins convivial car il utilise les commandes internes vi.

En mode emacs, le rappel de commandes se fait grâce au flèches du clavier (nativement avec les touches h, j, k, l en mode vi).

La complétion de la commande s'effectue en appuyant une à plusieurs fois sur la touche de tabulation. Le système complète alors automatiquement la commande ou affiche les commandes commençant par la séquence tapée.

La commande **set -o** permet d'afficher le paramétrage du shell sous la forme d'une fonctionnalité qui est activé (on) ou non. Certaines sont activées par défaut. Pour activer une fonctionnalité, il faut exécuter **set -o** suivi de la fonctionnalité à activer. Pour la désactiver, il faut exécuter **set +o**. Ce paramétrage est temporaire à la session. Pour un paramétrage permanent, il faut configurer les fichiers exécutés lors de l'appel du shell (.bashrc, .kshrc,...).

Le shell intègre un certain nombre de builtin (primitives du shell) qui sont des commandes directement intégrées au shell.

La commande **history** permet d'afficher les commandes de l'historique de commandes. La commande **fc -s no_commande** permet de ré-exécuter la commande indiquée par le numéro. Cette commande peut aussi prendre le nom d'une commande comme argument. Elle exécutera alors la dernière commande commençant par ce nom.

L'introduction et le shell

La compatibilité des différents shells

- sh

  POSIX - compatible avec tous les shells

- ksh et bash

  pas POSIX - shells ayant des fonctionnalités similaires.

La compatibilité des différents shells

Historiquement le Bourne Shell fut le shell original depuis lequel ont été développé de nombreux shells. Le Bourne Shell est POSIX et respecte ainsi cette norme.

Le Korn Shell et le Bourne Again Shell ont des fonctionnalités et des comportements similaires. Les différences entre les deux shells sont minimes (commande print qui n'existe pas en bash par exemple).

Par défaut le bash et le ksh ne sont pas POSIX ce qui permet d'avoir des fonctionnalités étendues notamment pour le traitement des variables. Il est possible de rendre ces shells compatibles POSIX mais on perdra certaines fonctionnalités.

Si vous désirez développer un script entièrement portable sur tous les systèmes, il est conseillé de l'écrire en sh. Attention cependant sur beaucoup de systèmes, c'est devenu un lien symbolique qui pointe sur le bash.

Pour rendre un script bash compatible avec ksh, il faut le tester et éventuellement corriger les erreurs minimes s'il y en a.

L'introduction et le shell

Les alias de commandes

- alias alias h
- alias h=history
- unalias h
- \ls

Les alias de commandes

Un alias permet d'exécuter une commande à la place d'un autre. La commande alias sans options liste les alias positionnés par défaut grâce à des fichiers de configuration lus au moment du démarrage du serveur.
Pour l'utilisateur root, les commandes comme cp, mv et rm sont aliassées avec l'option '-i' pour que le système demande confirmation avant de supprimer ou d'écraser un fichier.
Beaucoup de commandes sont aliassées avec des options de coloration de l'affichage.

Les alias sont exécutés avant les commandes.

```
$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot -show-tilde'
```

Création et affichage d'un alias.

```
$ alias cls=clear
$ alias h=history
$ alias cls h
alias cls='clear'
alias h='history'
```

La commande unalias permet de supprimer la définition d'un alias.

```
$ unalias cls
$ alias cls
-bash: alias: cls : non trouvé
```

Si une commande porte le même nom qu'un alias, par défaut c'est l'alias qui est exécuté. Pour exécuter la commande, il faut la faire précéder du caractère \.

```
$ ls
fic1  fic2  fic3  rep1  rep2
```

En faisant précéder du signe \ c'est la commande qui est exécuté et non plus l'alias. Les répertoires ne sont plus affichés en bleu.

```
$ \ls
fic1  fic2  fic3  rep1  rep2
```

L'introduction et le shell

L'ordre de recherche de commandes

- Alias
- Commandes internes
- Fonctions
- Commandes externes recherchées dans les répertoires de la variable PATH

L'ordre de recherche de commandes

Lors de l'appel d'une commande, le shell va rechercher cette commande selon des règles très précises.

Le shell va rechercher les commandes en respectant l'ordre ci-dessous :

- dans les alias (la commande 'alias' les liste),
- dans les commandes internes (la commande 'type' permet d'afficher le type de la commande),
- dans les fonctions (la commande 'declare -f' ou 'typeset -f' permet de les lister),
- dans les répertoires de la variable PATH.

L'introduction et le shell

L'interprétation d'une commande

- Les mots sont isolés par la variable IFS
- Traitement des caractères de protection (',",\)
- Substitution des variables ($)
- Substitution des commandes (`cmde` et $(cmde))
- Substitution des méta-caractères
- Traitement des pipes et des redirections

L'interprétation d'une commande

Le shell a ses propres règles pour interpréter la ligne de commande.

Par défaut les mots sont séparés par le contenu de la variable IFS. Cette variable contient par défaut les valeurs espace, tabulation et retour chariot dans cet ordre.

Le shell va ensuite traiter les caractères de protection qui sont ' , " et \.

Puis le traitement des variables est effectué pour substituer le contenu par leur valeur.

Ensuite le shell effectue la substitution des commandes (` cmde` ou $(cmde)).

Les méta-caractères sont ensuite traités pour identifier les noms de la ligne de commande.

Enfin le shell met en place le traitements des tubes (pipes) et des redirections (>, <, >> , <<, 2> , …).

Notes

L'exécution d'un script et débogage

Dans ce chapitre, nous allons nous familiariser avec l'exécution d'un script et de son débogage.

L'exécution d'un script et son débogage

- Les différentes méthodes d'exécution
- Le Shebang, les commentaires
- La structure d'un script
- Les bonnes règles de conception d'un script
- Le débogage d'un script

L'exécution d'un script et son débogage

Les différentes méthodes d'exécution

- ./nom_script
- . ./nom_script
- source ./nom_script
- bash script1
- exec script1

Les différentes méthodes d'exécution

Il existe différentes méthodes pour exécuter un script. La méthode usuelle étant d'ajouter le droit d'exécution avec la commande chmod, puis de le lancer.

```
$ ./nom_script
```

Cette méthode génère un processus fils dans lequel le script est exécuté.

```
$ . ./nom_script        ou        $ source ./nom_script
```

Cette méthode force le script à s'exécuter dans le shell courant et ne génère donc par de processus fils. C'est la méthode utilisée pour recharger un fichier de configuration du shell.

```
$ bash nom_script
```

Avec cette méthode, on fait appel au programme bash qui va lire le script en argument. Le script n'a pas besoin d'être exécutable, le droit de lecture suffit.

```
$ exec nom_script
```

La commande exec indique au shell de remplacer le processus courant par la commande ou le script en argument.

Pour lancer un shell fils, il suffit de l'appeler par son nom (sh, ksh, bash).

Remarque : si un shell n'est pas présent sur le système, il est possible de l'installer. Pour installer ksh, tapez en tant qu'administrateur la commande :

```
# yum install ksh              ou              # apt-get install ksh
```

L'exécution d'un script et son débogage

Le shebang, les commentaires

- Shebang

  #! sur la première ligne d'un script

  Défini le shell pour le script

- Commentaire

  Le caractère #

Le shebang, les commentaires

Le shebang (#!/bin/bash) permet de définir le shell utilisé pour l'exécution du script.

Il est positionné en première ligne du script (#!/bin/ksh ou #!/bin/bash par exemple).
Si cette ligne n'apparaît pas, le script sera lancé avec le shell en cours d'utilisation.
C'est le seul cas où le caractère # n'est pas interprété comme un commentaire.

Un commentaire est une chaîne de caractères qui n'est pas interprété par le shell. Cela permet, comme son nom l'indique, de faire des descriptifs, des saisies de texte.

Le caractère dièse est le caractère indiquant le début d'un commentaire qu'il soit positionné en début de ligne ou en milieu de ligne.

```
$ cat script1
#!/bin/bash

# La première ligne du script s'appelle le shebang
# Ce script affiche le contenu d'une variable

a=10    # initialisation de la variable a
echo $a # affichage de la valeur de a

# Fin de script
```

L'exécution d'un script et son débogage

La structure d'un script

- Positionner le shebang sur la première ligne
- Commentaires de présentation du script

- Les variables modifiables
- Les variables non modifiables
- La déclaration des fonctions

- Les tests de cohérences
- Le programme principal

La structure d'un script

En première ligne est défini le shebang.

Puis des commentaires présenteront le script. C'est à dire tout apport donnant une meilleure maîtrise du script : fonctionnalités, syntaxe, etc.

Puis la déclaration des variables, idéalement un commentaire de fin de ligne pour la présenter.
Une première section permettrait de définir les variables modifiables. C'est à dire, les variables que l'utilisateur du script pourrait un jour être amené à modifier pour une adaptation à leur environnement de travail (par exemple une adresse email ou les références à un serveur au sein de l'entreprise).
Une deuxième section permettrait de définir les variables non modifiables. Celles qui sont nécessaires au bon fonctionnement du script, mais qui ne doivent en aucun cas être modifiées par un utilisateur.

Puis la déclaration des fonctions. Éventuellement précédées par un commentaire pour les présenter. Les fonctions traiteront les opérations répétitives et permettront de simplifier le code principal du script.

Puis une partie pour les tests de cohérences. Ce sont des tests qui analysent toutes les situations qui pourraient faire « planter » votre programme. En cas d'anomalie détectée, il sera possible soit de la corriger, soit d'arrêter le programme.

Enfin, le corps principal du script.

L'exécution d'un script et son débogage

Les bonnes règles de conception d'un script

- Une mise en page rigoureuse
- Les tests de cohérences et la phase des tests
- Les commentaires
- Le respect du cahier des charges
- L'utilisation de différents fichiers
- Le suivi de la première phase de vie du script

Les bonnes règles de conception d'un script

Une mise en page rigoureuse doit être appliquée à la rédaction du script. Et on doit s'y tenir !
Il faut adopter un code d'écriture, utiliser les sauts de lignes et l'indentation. Il ne faut pas cependant tomber dans l'excès. Le but est d'améliorer la lisibilité d'un script et d'éviter de nombreuses erreurs telles que l'oubli de fermer un bloc d'instructions.

Après avoir écrit le programme, il est important de le tester, surtout pour avoir une meilleure idée des effets de bords. En effet, il faut que tous les cas de mauvaise utilisation du programme aient été prévus. Dans ce cas, un message d'aide est affiché indiquant la syntaxe d'utilisation du script.
Le script doit également gérer toutes les anomalies, une attention toute particulière sera donc portée sur les différents cas de figures d'incohérences.
Si vous soupçonnez un mauvais fonctionnement du script, vous pouvez faire afficher des messages visuels grâce à la commande 'echo'. Cela permet d'être sûr que le programme passe bien par la boucle ou l'instruction désirée.
Le script peut aussi être lancé en mode débogage pour avoir une meilleure idée des actions qu'il exécute.
Après exécution du script vérifier bien que les actions demandées ont été effectuées (création d'utilisateur, sauvegardes, etc).

Il est important de positionner des commentaires. Ils sont utiles si vous décidez de modifier un script, afin d'indiquer le rôle de certaines variables, fonctions ou codes.

Le cahier des charges est un document contractuel du maître d'ouvrage (le demandeur) définissant le travail attendu par le maître d'œuvre (le réalisateur).

Il définit les besoins et les contraintes du projet que le maître d'œuvre doit respecter (environnement réseau, plate-forme de déploiement, conventions de nommage, etc).
Il est important que le maître d'œuvre et le maître d'ouvrage soient bien d'accord avant de commencer le travail. Il faudra porter une attention toute particulière et bien différencier une contrainte (à respecter obligatoirement) et une préférence (à respecter si on peut, en général si cela n'est pas trop contraignant).

Une première approche de la structure d'un programme :
L'ensemble des fonctions et des variables sont centralisées au sein d'un seul script. L'avantage est qu'il a un fonctionnement totalement autonome et qu'il est facilement portable sur un autre système. L'inconvénient est que les fonctions de ce script sont utilisables qu'au sein de ce programme, même si leurs fonctionnalités auraient pu être exploitées par d'autres programmes.

Une deuxième approche de la structure d'un programme :
Une autre approche permet de résoudre les inconvénients spécifiés précédemment mais alourdi la procédure de déploiement sur différents serveurs.
Elle consiste à exploiter différents fichiers : les scripts, les fichiers des variables, les fichiers des fonctions, etc. L'intérêt de cette évolution du script se justifie uniquement par son intégration au sein d'une application plus complexe, avec des fonctions et des variables communes à plusieurs scripts.
Lors de la rédaction d'un script, il faut bien analyser ce qui sera spécifique au script et ce qui sera commun à plusieurs scripts.
Il est important de bien structurer le script. Pour cela on utilise souvent des fonctions. L'avantage des fonctions est que l'on peut les regrouper dans un fichier, et indiquer au script grâce à une instruction «source» d'aller consulter le fichier pour la définition des fonctions qui sont utilisés au sein du script. Cette méthode de centralisation des fonctions au sein d'un fichier est très usitée. Si vous avez besoin au sein d'un autre script d'utiliser à nouveau cette fonction, il suffit d'ajouter l'instruction source correspondante.
La même méthode peut être utilisée pour définir des variables qui interviennent dans plusieurs scripts.

Après avoir effectué tous les contrôles nécessaires sur des machines de tests, la solution est déployée sur les machines en production. Lors de cette phase, le maître d'ouvrage valide la solution mis en place par le maître d'œuvre.
Cette phase est appelée « recette » car le client réceptionne la solution que vous avez mis en place.
Après la validation du client, il faut tout de même durant une période définie s'assurer que le script ne génère pas des effets indésirables (encombrement réseau, ralentissement du serveur, etc).
Pour cela, on visualise les fichiers de journalisations (les logs) qu'il génère et qu'il effectue bien les actions demandées.
Il sera aussi utile de surveiller la charge système qu'engendre les scripts en utilisant les commandes appropriées pour visualiser : la vitesse de lecture, d'écriture, la bande passante du réseau, etc. Cette surveillance peut mettre en évidence des goulots d'étranglement que ce soit au niveau du réseau ou au niveau des accès disque, ou tout simplement identifier des périodes plus propices à l'exécution des scripts.

Voici quelques commandes de surveillance :

1- Pour la mémoire et les processus :

```
#  free
#  top
#  vmstat
#  vmstat 5  10   (10 requêtes, une toutes les 5 secondes)
```

2- Pour les statistiques d'entrée / sortie sur les disques :

```
#  iostat
#  iostat  -d  3  10  (10 requêtes, une toutes les 3 secondes)
```

3- Pour l'activité CPU :

```
#  sar
#  sar  -u  2   5    (5 requêtes, une toutes les 2 secondes)
#  sadc
#  mpstat
```

Le paquetage « sysstat » regroupe la plupart des commandes de surveillance système ci-dessus.

L'exécution d'un script et son débogage

Le débogage d'un script

- bash -x nom_script
- set -x
- set -o xtrace

Le débogage d'un script

Au niveau du shell, vous pouvez activer la fonction 'xtrace'.
Lors de la substitution de variables ou de paramètres, le caractère + apparaît.
Ce qui serait affiché à l'écran sans la fonctionnalité' xtrace' est ce qui apparaît sans signe +.

Vous pouvez aussi activer le mode débogage au niveau du shell avec la commande 'set -x' ('set +x' pour désactiver). Dans ce cas les commandes et les scripts lancés seront tracés comme avec xtrace.

Vous pouvez aussi simplement lancer votre script en mode débogage en appelant le shell.
L'option -x fait apparaître le résultat après la substitution de commandes ou paramètres.
L'option -v fait apparaître le résultat avant la substitution de commande ou paramètres. Vous pouvez utiliser les deux options conjointement.

```
$ bash –x script1
+ a=10
+ echo 10
10
```

```
$ bash –v script1
#!/bin/bash

### La première ligne du script s'appèle le shebang ###
### Ce script affiche le contenu d'une variable  ###

a=10    ## initialisation de la variable a
echo $a ## affichage de la valeur de a
10

# Fin de script
```

Notes

Rappel sur des commandes

Dans ce chapitre nous allons revoir les commandes les plus utilisées pour travailler sur des fichiers.

Rappel sur des commandes

- Les commandes «wc», «head» et «tail»
- Les commandes «file, «strings» et «od»
- Les commandes «cmp», «diff» et «paste»
- Les commandes «cut» et «awk»
- La commande «sort»
- Les commandes «uniq» et «tr»

Rappel sur des commandes

Les commandes «wc», «head» et «tail»

- wc [-cwl] fichier
- head [-n] fichier
- tail [-n] fichier

Les commandes wc, head et tail

La commande «wc»

Cette commande nous permet d'afficher des informations sur un fichier comme : le nombre d'octets, le nombre de mots et le nombre de lignes.

Plusieurs options sont disponibles pour cette commande.

-c : affiche le nombre d'octets.
-w : affiche le nombre de mots.
-l : affiche le nombre de lignes.

Exemples :

```
$ wc        /etc/passwd
40     68     1993   /etc/passwd

$ wc   -l   /etc/passwd
40     /etc/passwd
```

La commande «head»

Cette commande nous permet d'afficher les premières lignes d'un fichier, par défaut les 10 premières.
Nous pouvons personnaliser le nombre de lignes à afficher en positionnant l'option «**-n**» (n étant le nombre de lignes).

Exemple :

```
$ head     -5    /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
```

La commande «tail»

Cette commande permet d'afficher les dernières lignes d'un fichier, par défaut les 10 dernières.
Comme la commande «**head**» nous pouvons personnaliser le nombre de lignes à afficher grâce à l'option «**-n**».

Exemple :

```
$ tail     -5    /etc/passwd
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tcpdump:x:72:72::/:/sbin/nologin
user1:x:1000:1003:user1:/home/user1:/bin/bash
user2:x:1001:1004::/home/user2:/bin/bash
user3:x:1002:1005::/home/user3:/bin/bash
```

Rappel sur des commandes

Les commandes «file», «strings» et «od»

- file fichier
- strings fichier
- od [-odxc] fichier

Les commandes file, strings et od

La commande «file»

Cette commande permet de déterminer le type du contenu d'un fichier.

Syntaxe : file fic1

Exemple :

```
$ file /etc/*
/etc/abrt:          directory
/etc/adjtime:       ASCII text
/etc/aliases:       ASCII text
/etc/alsa:          directory
```

La commande «strings»

Cette commande permet d'afficher, sur la console, les caractères imprimables d'un fichier.

Syntaxe : strings fic1

La commande «od»

Cette commande permet d'afficher le contenu d'un fichier sous plusieurs formats (octal, hexadécimal, etc ...)

Exemple : Par défaut, l'affichage est en octal.

```
$ od        /etc/passwd
0000000 067562 072157 074072 030072 030072 071072 067557 035164
0000020 071057 067557 035164 061057 067151 061057 071541 005150
0000040 064542 035156 035170 035061 035061 064542 035156 061057
0000060 067151 027472 061163 067151 067057 066157 063557 067151
0000100 062012 062541 067555 035156 035170 035062 035062 060544
0000120 066545 067157 027472 061163 067151 027472 061163 067151
..etc
```

Il existe plusieurs options pour cette commande :

-o : permet d'afficher un contenu en octal.
-d : permet d'afficher un contenu en décimal.
-x : permet d'afficher un contenu en hexadécimal.
-c : permet d'afficher un contenu en ascii.

Exemple :

```
$ od        -xc          /etc/passwd
0000000    6f72    746f    783a    303a    303a    723a    6f6f    3a74
          r   o   o   t   :   x   :   0   :   0   :   r   o   o   t   :
0000020    722f    6f6f    3a74    622f    6e69    622f    7361    0a68
          /   r   o   o   t   :   /   b   i   n   /   b   a   s   h  \n
0000040    6962    3a6e    3a78    3a31    3a31    6962    3a6e    622f
          b   i   n   :   x   :   1   :   1   :   b   i   n   :   /   b
0000060    6e69    2f3a    6273    6e69    6e2f    6c6f    676f    6e69
          i   n   :   /   s   b   i   n   /   n   o   l   o   g   i   n
0000100    640a    6561    6f6d    3a6e    3a78    3a32    3a32    6164
         \n   d   a   e   m   o   n   :   x   :   2   :   2   :   d   a
0000120    6d65    6e6f    2f3a    6273    6e69    2f3a    6273    6e69
          e   m   o   n   :   /   s   b   i   n   :   /   s   b   i   n
0000140    6e2f    6c6f    676f    6e69    610a    6d64    783a    333a
          /   n   o   l   o   g   i   n  \n   a   d   m   :   x   :   3
0000160    343a    613a    6d64    2f3a    6176    2f72    6461    3a6d
          :   4   :   a   d   m   :   /   v   a   r   /   a   d   m   :
0000200    732f    6962    2f6e    6f6e    6f6c    6967    0a6e    706c
          /   s   b   i   n   /   n   o   l   o   g   i   n  \n   l   p
0000220    783a    343a    373a    6c3a    3a70    762f    7261    732f
          :   x   :   4   :   7   :   l   p   :   /   v   a   r   /   s
0000240    6f70    6c6f    6c2f    6470    2f3a    6273    6e69    6e2f
          p   o   o   l   /   l   p   d   :   /   s   b   i   n   /   n
0000260    6c6f    676f    6e69    730a    6e79    3a63    3a78    3a35
          o   l   o   g   i   n  \n   s   y   n   c   :   x   :   5   :
0000300    3a30    7973    636e    2f3a    6273    6e69    2f3a    6962
          0   :   s   y   n   c   :   /   s   b   i   n   :   /   b   i
0000320    2f6e    7973    636e    730a    7568    6474    776f    3a6e
          n   /   s   y   n   c  \n   s   h   u   t   d   o   w   n   :
...etc
```

Rappel sur des commandes

Les commandes «cmp», «diff» et «paste»

- cmp fic1 fic2

```
$ cmp fic1 fic2
fic1 fic2 sont différents: octet27, ligne2
```

- diff fic1 fic2

```
1c1,5 < texte
6a5 ------
3d3 > texte
```

- paste fic1 fic2

```
$ paste fic1 fic2
1 Jean-Marc 5 Responsable
5 Robin 8 Stagiaire
```

Les commandes cmp, diff et paste

La commande «cmp»

Cette commande permet de comparer deux fichiers, elle indique si les deux fichiers son identiques ou pas.

Syntaxe : cmp fichier1 fichier2

Exemple :

```
$ cmp fic1 fic2
```

Si les deux fichiers sont identiques, le **prompt** sera **vide** en retour.
Par contre si les deux fichiers sont différents, la commande «**cmp**» affichera le numéro de la ligne et le numéro du caractère à partir duquel les deux fichiers diffèrent.

Exemple :

```
$ cmp fic1 fic2
fic1 fic2 sont différents: octet27, ligne2
```

La commande «diff»

Cette commande nous informe sur les modifications à apporter au premier fichier afin qu'il ait le même contenu que le second.

Syntaxe du code : ligne(s)_du_fichier1 Action ligne(s)_du_fichier1
Action 'c' : pour changer des lignes.
Action 'd' : pour supprimer des lignes.
Action 'a' : pour ajouter des lignes.

Exemple :

```
$ cat      fic1
ceci est un exemple.
Il fait beau aujourd'hui.
La pluie c'est pour demain.
```

```
$ cat      fic2
Il ne fait pas beau aujourd'hui.
La pluie c'est pour demain.
Ceci est la fin de l'exemple.
```

```
$ diff     fic1       fic2
1,2c1                                  les lignes 1 à 2 de fic1 doivent être
< ceci est un exemple.                 changées par la ligne 1 de fic2
< Il fait beau aujourd'hui.
---
> Il ne fait pas beau aujourd'hui.
3a3                                    Après la ligne 3 de fic1 il faut
> Ceci est la fin de l'exemple.        Ajouter la ligne 3 de fic2
```

```
$ diff     fic2       fic1
1c1,2                                  la ligne 1 de fic1 doit être
< Il ne fait pas beau aujourd'hui.     Changée par les lignes 1 à 2 de fic1
---
> ceci est un exemple.
> Il fait beau aujourd'hui.
3d3                                    il faut supprimer la ligne 3 de fic2
< Ceci est la fin de l'exemple.
$
```

La commande «paste»

Cette commande permet de fusionner des fichiers ligne par ligne.

Exemple :

```
$ cat      fic1
1     Jean-Marc
5     Robin

$ cat      fic2
5     Responsable
8     Stagiaire

$ paste    fic1       fic2
1     Jean-Marc    5     Responsable
5     Robin        8     Stagiaire
```

Rappel sur des commandes

Les commandes «cut» et «awk»

- cut

```
$ cut   -c  2-5   fic1
$ cut   -c  -3    fic1
$ cut   -c  3-    fic1
$ cut   -d:   -f  1     /etc/passwd
$ cut   -d:   -f  1,4   /etc/passwd
$ cut   -d:   -f  1-4   /etc/passwd
```

- awk

```
$ awk   '{print $1}'   /etc/host
$ awk   -F:   '{print $3, $1}'   /etc/passwd
$ awk   -F:   '{print "Uid = "$3" Login = " $1}'   /etc/passwd
```

Les commandes cut et awk

Commande «cut»

Cette commande sélectionne une partie de chaque ligne d'un fichier texte.
L'option «**-c**» permet de travailler avec la notion de caractère, alors que l'option «**-f**» permet de travailler avec la notion de champs.

Exemples :

Pour conserver les caractères 2 à 5 de chaque ligne :

```
$ cut      -c  2-5    fic1
```

Pour conserver les 3 premiers caractères :

```
$ cut      -c  -3     fic1
```

Pour conserver à partir du 3ème caractère jusqu'au dernier caractère de la ligne :

```
$ cut      -c  3-     fic1
```

Pour délimiter un champs à conserver, nous utiliserons l'option «**-d**» et l'option «**-f**».

-d : indique le caractère séparateur de champs (par défaut : la tabulation).
-f : indique le numéro du champs à conserver.

Exemples :

Pour récupérer le champs 1, avec le caractère séparateur de champs «:» :

```
$ cut -d: -f 1 /etc/passwd
```

Pour récupérer les champs 1 et 4, avec le caractère séparateur de champs «:» :

```
$ cut -d: -f 1,4 /etc/passwd
```

Pour récupérer les champs 1 à 4, avec le caractère séparateur de champs «:» :

```
$ cut -d: -f 1-4 /etc/passwd
```

Commande «awk»

La commande awk (nawk, gawk) permet de manipuler le contenu d'un fichier en exploitant les champs. Cette instruction peut avoir une syntaxe très complexe car elle intègre toutes les fonctionnalités des scripts.

Syntaxe : awk [-options] 'actions' fichier

```
$ awk '{print $1}' /etc/hosts
127.0.0.1
::1
```

La section entre **{ }** définit la liste des actions à réaliser sur chaque ligne du fichier.
La référence à un champ est indiquée par le caractère **$** suivi du numéro de champ, les caractères 'espace' et 'tabulation' sont les caractères séparateurs de champs par défaut.
Ainsi l'exemple ci-dessus affiche le premier champ du fichier /etc/hosts.

```
$ awk -F: '{print $3, $1}' /etc/passwd
0 root
1 bin
2 daemon
3 adm
4 lp
5 sync
Etc..
```

L'option **-F** redéfinie le caractère séparateur de champs (ici le «**:**»).
L'exemple affiche le champ 3 puis le champ 1 séparés par un caractère «**espace**». Ce dernier est présent à cause du caractère spécial «**,**».

```
$ awk -F: '{print "Uid = "$3" Login = " $1}' /etc/passwd
Uid = 0 Login = root
Uid = 1 Login = bin
Uid = 2 Login = daemon
Uid = 3 Login = adm
Uid = 4 Login = lp
Uid = 5 Login = sync
Etc..
```

Il est possible d'agrémenter l'affichage avec du texte, il suffit de l'écrire entre guillemets.

Rappel sur des commandes

La commande «sort»

- sort = tri du contenu du fichier

```
$ sort    /etc/passwd
$ sort    -t:  -k3  /etc/passwd
$ sort    -t:  -k3n /etc/passwd
$ sort    -n   fichier
$ sort    -r   -n  fichier
```

La commande sort

Cette commande permet de trier un fichier par ligne dans l'ordre croissant alphabétique.
Plusieurs options sont disponibles pour cette commande comme :

-n	:	pour un tri numérique.
-k	:	pour trier à partir d'une colonne spécifique.
-t	:	défini le caractère séparateurs de champs. Par défaut : espace, tabulation et retour chariot.
-r	:	pour trier dans l'ordre décroissant.

Syntaxes :	sort	fic1		Tri alphabétique des lignes du fichier.
	sort	-n	fic1	Le fichier correspond à une colonne numérique. Tri numérique du fichier.
	sort	-k3	fic1	Trier par rapport à la colonne 3 avec les caractères séparateurs de champs par défaut.
	sort	-t:	-k3 fic1	Tri alphabétique dans l'ordre croissant par rapport au champs numéro 3. Le caractère séparateur de champs étant le '**:**'.
	sort	-t:	-k3n fic1	Idem que la commande précédente mais le tri est un tri numérique sur le champs 3.

Exemples :

```
$ sort     /etc/passwd
abrt:x:173:173::/etc/abrt:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
avahi-autoipd:x:170:170:Avahi IPv4LL Stack:/var/lib/avahi-autoipd:/sbin/nologin
avahi:x:70:70:Avahi mDNS/DNS-SD Stack:/var/run/avahi-daemon:/sbin/nologin
bin:x:1:1:bin:/bin:/sbin/nologin
chrony:x:994:993::/var/lib/chrony:/sbin/nologin
colord:x:997:995:User for colord:/var/lib/colord:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
dbus:x:81:81:System message bus:/:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
… etc
```

```
$ sort     -t:   -k3   /etc/passwd
root:x:0:0:root:/root:/bin/bash
user1:x:1000:1003:user1:/home/user1:/bin/bash
user2:x:1001:1004::/home/user2:/bin/bash
user3:x:1002:1005::/home/user3:/bin/bash
qemu:x:107:107:qemu user:/:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
usbmuxd:x:113:113:usbmuxd user:/:/sbin/nologin
bin:x:1:1:bin:/bin:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
… etc
```

```
$ sort     -t:   -k3n  /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
… etc
```

```
$ cat  fichier
1002
25
1
34
20
2
999
56
```

```
$ sort  -r -n  fichier
1002
999
56
34
25
20
2
1
$
```

Rappel sur des commandes

Les commandes «uniq» et «tr»

- uniq fichier — Lignes uniques

```
$ uniq    fic1
$ uniq    -c   fic1
$ uniq    -d   fic1
$ uniq    -u   fic1
```

- tr — Traduction

```
$ tr  'otu'  'say'    <  fic1
$ tr  "a-z"  "A-Z"    <  fic1
$ cat  fic2  |  tr  -s 'o'
$ who  |  tr  -s  ' '
```

Les commandes uniq et tr

La commande «uniq»

Cette commande détecte et supprime les lignes identiques successives dans un fichier.

Exemple :

```
$ cat      fic1
Jean
Jean
Marcel
Patrice
Patrice
Jean
```

```
$ uniq     fic1
Jean
Marcel
Patrice
Jean
```

Des options sont disponibles pour cette commande comme :

-c : affiche le nombre d'occurrences.
-d : affiche uniquement les lignes dupliquées dans le fichier.
-u : affiche que les lignes uniques du fichier.

Exemples :

```
$ uniq     -c    fic1
2 Jean
1 Marcel
2 Patrice
1 Jean

$ uniq     -d    fic1
Jean
Patrice

$ uniq     -u    fic1
Marcel
Jean
```

La commande «tr»

Cette commande permet de remplacer un caractère par un autre.
Dans l'exemple ci-dessous, nous remplaçons les caractères (o,t et u) respectivement par (s,a et y) :

```
$ cat      fic1
Bonjour tout le monde.

$ tr  'otu'  'say'    <  fic1
Bsnjsyr asya le msnde.
```

On peut utiliser des intervalles de caractères en utilisant le caractère «-». Ainsi il est possible de remplacer les caractères minuscules d'un fichier par des majuscules :

```
$ cat      fic1
Bonjour tout le monde.

$ tr  "a-z"  "A-Z"    <  fic1
BONJOUR TOUT LE MONDE.
```

Option « -s » : cette option remplace un succession d'un même caractère par un seul.

```
$ cat  fic2
Booooonjour nous avoooons
souvent oorganise des fooooormatiooons
$
$ cat  fic2  |  tr  -s 'o'
Bonjour nous avons
souvent organise des formations

$ who
user1    :0           2016-04-29 09:25 (:0)
user1    pts/0        2016-04-29 09:25 (:0)
user1    pts/1        2016-04-29 10:43 (spheriusform-pc.home)
$
$ who  |  tr  -s  ' '
user1 :0 2016-04-29 09:25 (:0)
user1 pts/0 2016-04-29 09:25 (:0)
user1 pts/1 2016-04-29 10:43 (spheriusform-pc.home)
```

Notes

Les caractères spéciaux du shell, les redirections et le pipe

Dans ce chapitre nous allons nous familiariser avec les caractères spéciaux et les redirections.

Les caractères spéciaux du shell, les redirections et le pipe

- Les métacaractères
- Les redirections
- Le caractère pipe

Les caractères spéciaux du shell, les redirections et le pipe

Les métacaractères

- **Caractère spécial du shell**
- "texte"
- 'texte'
- `commande` ou $(commande)
- \x

Les métacaractères

Les métacaractères sont des caractères qui ont une signification spéciale au sein du shell.

Métacaractère	Signification
" "	Cela permet de prendre l'ensemble de la chaîne de caractères qu'il y a entre guillemets comme un seul argument. Les caractères spéciaux au sein des guillemets sont interprétés.
' '	Cela permet de prendre l'ensemble de la chaîne de caractères qu'il y a entre simple côte comme un seul argument mais les caractères spéciaux sont neutralisés.
`cmd` **ou $(cmd)**	La chaîne de caractères entre côte inverse est exécutée comme une commande. On récupère donc le résultat de cette commande.
\x	Le backslash change la signification du caractère suivant (x). Cela signifie que si ce caractère est un caractère spécial, il est neutralisé. Par contre si c'est un caractère lambda, il peut devenir un caractère spécial. Exemple : \$ Le caractère dollar n'a plus sa signification spéciale. \t Représente une tabulation. \n Représente un saut de ligne.

Exemples :

```
$ echo " Répertoire $HOME=$HOME et date "
 Répertoire /home/user1=/home/user1 et date

$ echo ' Répertoire $HOME=$HOME et date '
 Répertoire $HOME=$HOME et date

$ echo " Répertoire $HOME=$HOME et `date` "
 Répertoire /home/user1=/home/user1 et mar. avril 26 16:52:38 CEST 2016

$ echo " Répertoire $HOME=$HOME et $(date) "
 Répertoire /home/user1=/home/user1 et mar. avril 26 16:53:12 CEST 2016

$ echo " Répertoire \$HOME=$HOME et `date` "
 Répertoire $HOME=/home/user1 et mar. avril 26 16:54:00 CEST 2016

$ echo -e " Répertoire \$HOME=$HOME \n et\t\t\t`date` "
 Répertoire $HOME=/home/user1
 et mar. avril 26 16:54:00 CEST 2016
```

Les caractères spéciaux du shell, les redirections et le pipe

Les métacaractères - suite

- **Caractère spécial du shell**

- * 0 à n caractères quelconques
- ? 1 caractère quelconque
- [abc] Liste de caractères
- [a-z] Intervalle de caractères
- [!abc] Exclusion d'une liste de caractères
- [!a-z] Exclusion d'un intervalle de caractères

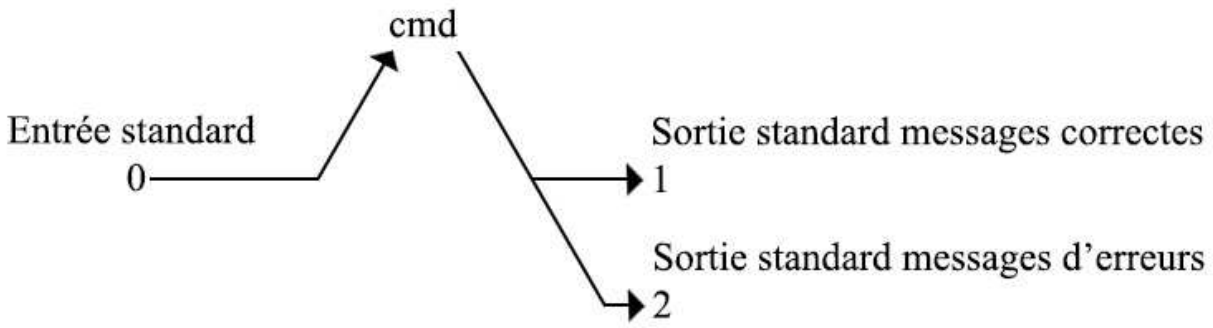
Métacaractère	Signification
*****	Représente 0 à n caractères quelconques.
?	Représente 1 caractère quelconque.
[abc]	Représente un caractère parmi la liste entre [ ].
[a-z]	Représente un caractère parmi l'intervalle de caractères (utilisation du tiret). Remarques : [a-z] pour un caractère minuscule. [A-Z] pour un caractère majuscule. [a-zA-Z] pour un caractère minuscule ou majuscule. [0-9] pour un chiffre.
[!abc]	Représente un caractère quelconque sauf ceux de la liste.
[!a-z]	Représente un caractère quelconque sauf ceux de l'intervalle.

Exemples :

```
$  ls
fic1   fic2   fic3   fic4   fic5
fic6   fic7   fic8   fic9   fic10
fic11  fic12  fic20  ficAA  fic0z
fichier   file  foret  autre  nouveau  liste


$  ls  fi*
fic1   fic2   fic3   fic4   fic5
fic6   fic7   fic8   fic9   fic10
fic11  fic12  fic20  ficAA  fic0z
fichier   file


$  ls  fic?
fic1   fic2   fic3   fic4   fic5
fic6   fic7   fic8   fic9


$  ls  ?i*
fic1   fic2   fic3   fic4   fic5
fic6   fic7   fic8   fic9   fic10
fic11  fic12  fic20  ficAA  fic0z
fichier   file  liste


$  ls  fic??
fic10  fic11  fic12  fic20  ficAA  fic0z


$  ls  fic[0-9][0-9]
fic10  fic11  fic12  fic20


$  ls  fic[!0-9][!0-9]
ficAA


$  ls  ?[!i]*
foret  autre  nouveau
```

Les caractères spéciaux du shell, les redirections et le pipe

Les Entrée / Sorties standards



Les Entrée / Sorties standards

Lorsque l'on exécute une commande, il existe 3 flux de données par défaut :
L'entrée standard, la sortie standard, et la sortie standard des messages d'erreur.

Entrée standard

C'est le flux qui permet de lire les données d'entrée. Appelé «**stdin**», il est identifié par le flux numéro **0**.
Par défaut ce flux correspond au clavier.

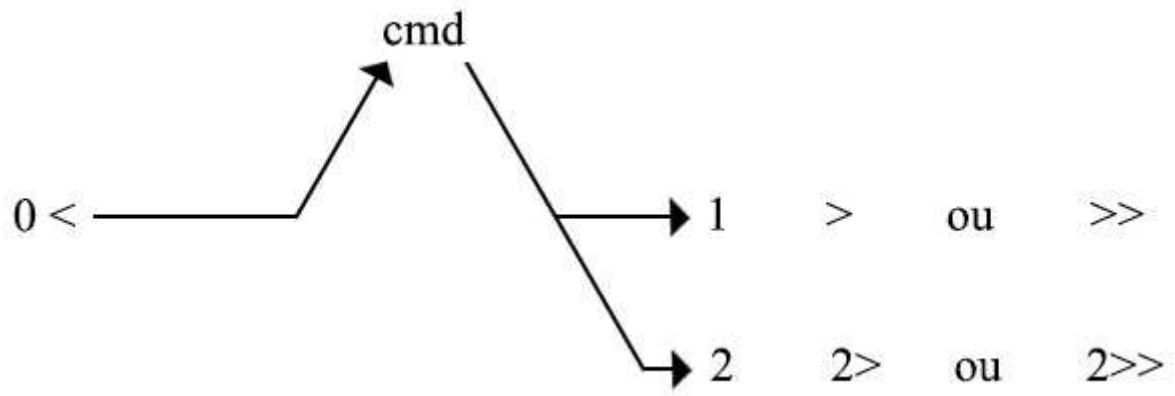
Sortie standard / Sortie d'erreur

Lors de l'exécution d'une commande ou d'un programme, les résultats sont envoyés dans deux flux distincts :

La sortie standard	:	pour les messages corrects. Appelé «**stdout**» et identifié par le flux numéro **1**. Par défaut cela correspond au pseudo-terminal (tty) auquel est rattaché la commande ou le programme.
La sortie d'erreur	:	pour les messages d'erreurs. Appelé «**stderr**» et identifié par le flux numéro **2**. Par défaut cela correspond à la même sortie que la sortie standard.

Les caractères spéciaux du shell, les redirections et le pipe

La redirection de l'entrée standard




```
$  commande    < fichier1    > fichier2    2> fichier3
```

La redirection de l'entrée standard

Entrée standard - flux 0

Il est possible de changer la source de l'entrée standard par le contenu d'un fichier.
Ainsi lorsqu'une commande attend par défaut une saisie au clavier (flux 0 de l'entrée standard), on peut faire en sorte qu'à la place cette même commande lise le contenu d'un fichier. Il est donc nécessaire pour cela de changer l'entrée standard en la connectant non pas au clavier mais au fichier concerné. Ceci est possible avec la syntaxe suivante :

commande 0< fichier_d_entrée

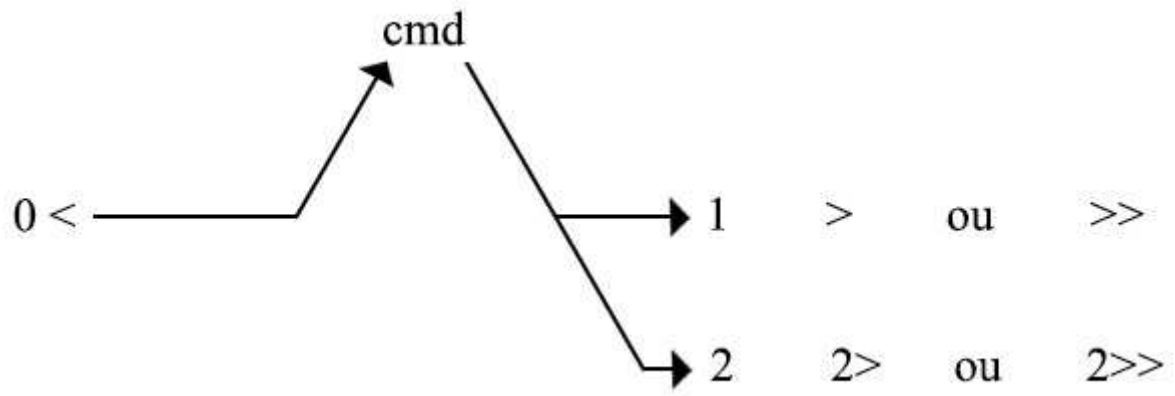
Plus communément commande < fichier_d_entrée

Exemple :

```
$  mail   user1@spherius.fr
_                                attente de saisie au clavier
                                 du texte du corps du mail


$  mail   user1@spherius.fr   <   fichier_du_corps_du_mail
$
```

Les caractères spéciaux du shell, les redirections et le pipe

Les redirections des sorties standards




```
$  commande    < fichier1    > fichier2    2> fichier3
```

Les redirections des sorties standards

Sortie standard des messages corrects - flux 1

Il est possible de changer la destination de la sortie standard des messages corrects pour alimenter le contenu d'un fichier.
Ainsi lorsqu'une commande affiche son résultat à l'écran, en fait le pseudo-terminal (flux 1 de la sortie standard), on peut faire en sorte qu'à la place le résultat de cette commande ne s'affiche pas à l'écran mais soit redirigé vers un fichier.

Première syntaxe – le 1> ou le >

commande 1> fichier_de_sortie

Plus communément commande > fichier_de_sortie

Si le fichier n'existe pas, il est créé. Si le fichier existe, le contenu est écrasé avec le résultat de la commande.

Deuxième syntaxe – le 1>> ou le >>

commande 1>> fichier_de_sortie

Plus communément commande >> fichier_de_sortie

Si le fichier n'existe pas, il est créé. Si le fichier existe, le résultat de la commande est ajouté à la fin du fichier.

Exemple :

```
$  cat fichier1
bonjour, ceci est le contenu de fichier1
$
$  date  >>  fichier1
$
$  cat fichier1
bonjour, ceci est le contenu de fichier1
mar. avril 26 16:54:00 CEST 2016
$
$  pwd  >>  fichier1
$  cat fichier1
bonjour, ceci est le contenu de fichier1
mar. avril 26 16:54:00 CEST 2016
/home/user1
$
$  date  >  fichier1
$
$  cat fichier1
mar. avril 26 16:56:10 CEST 2016
$
```

Sortie standard des messages d'erreurs - flux 2

Comme pour la sortie des messages corrects, il est possible de changer la destination de la sortie standard des messages d'erreurs pour alimenter le contenu d'un fichier.
Ainsi lorsqu'une commande affiche un message d'erreur à l'écran, en fait le pseudo-terminal (flux 2 de la sortie standard des messages d'erreurs), on peut faire en sorte qu'à la place le message d'erreurs ne s'affiche pas à l'écran mais soit redirigé vers un fichier.

Première syntaxe – le 2>

commande 2> fichier_de_sortie

Si le fichier n'existe pas, il est créé. Si le fichier existe, le contenu est écrasé avec le résultat de la commande.

Deuxième syntaxe – le 2>>

commande 2>> fichier_de_sortie

Si le fichier n'existe pas, il est créé. Si le fichier existe, le résultat de la commande est ajouté à la fin du fichier.

Exemple :

```
$  cat fichier1
bonjour, ceci est le contenu de fichier1
$
$  datex
Error : command 'datex' not found.
$
$
$  datex  2>>  fichier1
$
$  cat fichier1
bonjour, ceci est le contenu de fichier1
Error : command 'datex' not found.
$
$  datexyz  2>>  fichier1
$
$  cat fichier1
bonjour, ceci est le contenu de fichier1
Error : command 'datex' not found.
Error : command 'datexyz' not found.
$
$
$  cmdxxxx  2>  fichier1
$
$  cat fichier1
Error : command 'cmdxxxx' not found.
$
```

Syntaxe complète

nous pouvons cumuler sur la même ligne de commandes les redirections nécessaires :

Exemple 1 : pour rediriger les deux sorties standards :

```
$ commande > fichier1 2> fichier2
$

ou

$ commande 2> fichier2 > fichier1
$
```

Ainsi, les messages corrects résultant de la commande sont stockés au sein du fichier1 et les messages d'erreurs au sein du fichier2.

Exemple 2 : pour rediriger les trois flux standards :

```
$ commande < fichier1 > fichier2 2> fichier3
$
```

Ainsi, la commande utilise le contenu du fichier1 en entrée, les messages corrects résultant de la commande sont stockés au sein du fichier2 et les messages d'erreurs au sein du fichier3.

Remarque

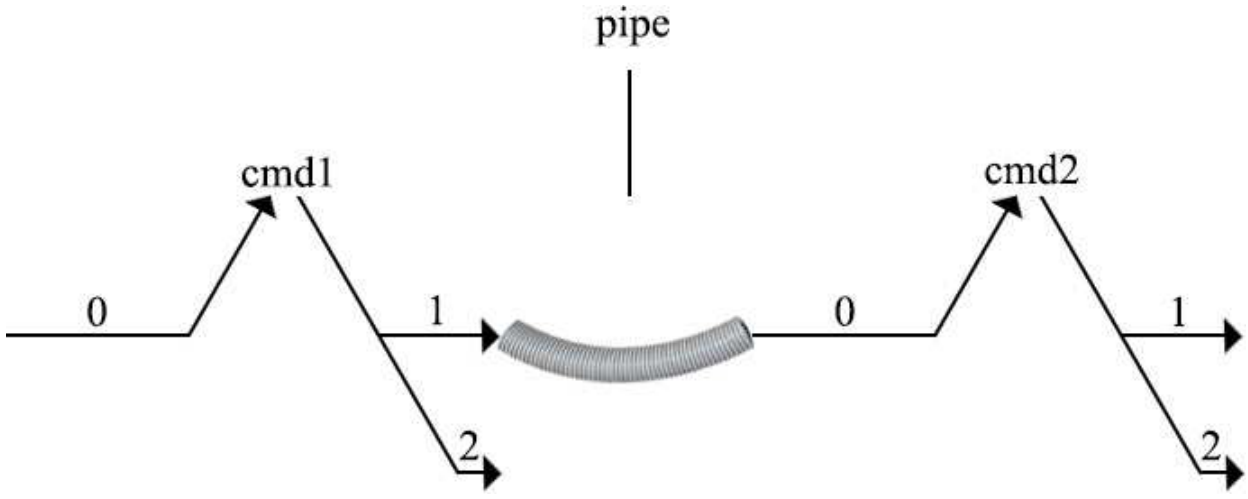
Pour la redirection de la sortie, l'utilisation du fichier spécial /dev/null permet de ne pas conserver les messages reçus .

```
$ commande 2> /dev/null
$
```

Ainsi, la commande affiche les messages corrects, les messages d'erreurs n'apparaissent pas et ne sont pas conservés.

Les caractères spéciaux du shell, les redirections et le pipe

Le pipe



$ commande1 | commande2 | commande3

Le pipe

Le «**pipe**» est une passerelle qui permet de transférer le résultat d'une commande «**cmd1**» à une autre commande «**cmd2**».
Son symbole est la barre verticale «**|**».
Plus précisément, le flux numéro **1** (messages corrects) de la commande «**cmd1**» est envoyé en entrée du flux numéro **0** (flux d'entrée) de la commande «**cmd2**».

Premier exemple - ls -Rl / | more

La première commande est 'ls -Rl /'. Elle permet d'afficher de manière récursive toute l'arborescence du système au format long. Le résultat de cette commande fournissant énormément de lignes, il en résulte un défilement à l'écran qui n'est pas exploitable pour une personne.

L'idéal serait que cette affichage se fasse page par page. Or justement c'est la fonction de la commande more.

Ainsi en redirigeant le résultat de première commande en entrée de la commande more, l'affichage sera paginée. C'est que permet le pipe pour combiner ces deux commandes.

Deuxième exemple - grep root /etc/passwd | wc -l | lp

cette exemple permet d'imprimer le nombre de lignes contenant « root » du fichier /etc/passwd.

La première commande est 'grep root /etc/passwd'. Elle affiche du fichier /etc/passwd uniquement les lignes contenant la chaîne de caractères « root ».

'grep root /etc/passwd | wc -l' affiche le nombre de ligne résultant de la commande 'grep'.

L'affichage précédent est envoyé en entrée de la commande 'lp' qui a pour fonction d'imprimer ce qu'elle reçoit.

Exemple avec la commande «tee»

Grâce à la commande «**tee**» récupère les données de l'entrée standard, pour les rediriger simultanément vers un fichier (passé en argument) et vers l'écran (la sortie standard des messages corrects).
Elle est pratique pour le pipe car cela permet de faire une sauvegarde dans un fichier des données d'un état intermédiaire du pipe.

Exemple :

```
$ echo "Bonjour tout le monde." | tee fic3 | wc
1          4          22
$ cat fic3
Bonjour tout le monde.
```

Le résultat de la commande 'echo' a été stocké dans un fichier fic3 et redirigé vers la commande suivante. 'wc' affichera à l'écran le nombre de caractères, de mots et de lignes résultant du 'echo'.

Notes

Les variables

Dans ce chapitre nous allons nous allons traiter des variables.

Les variables

- La déclaration et l'utilisation
- Les variables locales et globales
- Les manipulations avancées
- La concaténation, l'isolation et la substitution
- La personnalisation de l'environnement de travail

Les variables

La déclaration et l'utilisation

- echo $variable
- env et set
- Quelques variables
  - HOME LOGNAME SHELL IFS
  - PATH HOSTNAME PWD
  - PS1 LANG OLDPWD

La déclaration et l'utilisation

Les variables contiennent des chaînes de caractères qui permettent de stocker, d'une manière temporaire, des informations en mémoire.
Pour afficher le contenu d'une variable : echo $variable

Variables d'environnements

Les variables d'environnement permettent de configurer le système et un programme.
Les commandes «**env**» et «**set**» affichent la liste des variables situées dans l'environnement.

Quelques exemples de variables

HOME	:	affiche le nom du répertoire de connexion.
PATH	:	liste les répertoires où le système va rechercher les commandes.
PS1	:	affiche le prompt principal.
LANG	:	affiche la langue du système.
LOGNAME	:	affiche le nom de l'utilisateur.
HOSTNAME	:	affiche le nom de l'hôte du système.
SHELL	:	affiche le shell utilisé.
PWD	:	affiche le nom du répertoire courant.
OLDPWD	:	affiche le nom du répertoire précédent.
IFS	:	variable de séparation des arguments.

Les variables

Les variables locales et globales

- set / unset
- Variables locales / Variables globales
  - export variable=valeur

Les variables locales et globales

set / unset

La commande «**set**» affiche l'ensemble des variables qu'elles soient dans environnement ou non.
La commande «**unset**» annule la définition d'une variable.

Exemple :

```
$  var=bonjour
$  echo  $var
bonjour

$  unset var
$  echo  $var
--- affiche rien ---
```

Variables locales / Variables globales

Les variables globales sont transmises (dupliquées) dans le «**sous-shell**» contrairement aux variables locales.

Ainsi, l'exploitation d'une variable locale ne peut être faite qu'au niveau du shell dans lequel elle a été créée.
La valeur d'une variable globale est exploitable au sein de sous-shells. La commande «**export**» définie une variable comme globale.

Aucune variable n'est transmise au shell père.

Par convention, les variables locales sont notées en minuscules et les variables globales en majuscules.

Exemple de variable locale : village=Salon

Exemple de variable globale : export VILLE=«Aix en Provence»
ou
VILLE=«Aix en Provence»
export VILLE

Exemple :

```
$  village=Salon                          variable locale
$  export  VILLE=«Aix en Provence»        variable globale
$  echo  $village
Salon
$  echo  $VILLE
Aix en Provence

$  bash                                   un sous-shell

$  echo  $village
--- affiche rien ---
$  echo  $VILLE
Aix en Provence

$  VILLE=Toulouse
$  echo  $VILLE
Toulouse

$  sante=Vacances                         variable locale au sous-shell
$  export  PAYS=France                    variable globale au sous-shell
$  echo  $sante
Vacances
$  echo  $PAYS
France

$  exit                                   retour au shell père

$  echo  $village
Salon
$  echo  $VILLE
Aix en Provence
$  echo  $sante
--- affiche rien ---
$  echo  $PAYS
--- affiche rien ---
```

Les variables

Les manipulations avancées

- Les variables spéciales du shell

  $? code retour de la dernière commande exécuté

  $$ PID du shell courant

  $! PID du dernier programme exécuté en arrière plan

Les manipulations avancées

La variable $? contient le code retour de la dernière instruction passée au shell. Si ce code est égal à 0, cela signifie que la dernière instruction passée s'est correctement exécutée. Une valeur différente de 0 indiquant un mauvais fonctionnement.

```
$ pwd
/home/theo
$ echo $?
0
```

```
$ daaate
bash: daaate: commande inconnue...
$ echo $?
127
```

La variable $$ contient le PID du shell dans lequel s'exécute la commande.

```
$ ps
  PID TTY          TIME CMD
 3467 pts/1    00:00:00 bash
 3572 pts/1    00:00:00 ps
$ echo $$
3467
```

La variables $! contient le PID de la dernière commande exécutée en arrière plan.

```
$ sleep 2000 &
[1] 16337
$ echo $!
16337
```

Les variables

La concaténation, l'isolation et la substitution

- La concaténation des variables

  var=$var1$var2$var3

- L'isolation des variables : les accolades

  {var}

- La substitution des variables

  {var:-10} {var:=10} {var:+10} {var:?''message''}

La concaténation, l'isolation et la substitution

Les variables peuvent être concaténées. Les accolades permettent d'isoler le nom d'une variable.

```
$ cat script_var
#!/bin/bash

var1=in
var2=variable

deuxvar=$var1$var2

echo "Ce script affiche le mot: $deuxvar"
echo "Ce script affiche le mot: ${deuxvar}ment"
echo "Ce script n'affiche pas le mot: $deuxvarment"
```

```
$ ./script_var
Ce script affiche le mot: invariable
Ce script affiche le mot: invariablement
Ce script n'affiche pas le mot:
```

Les accolades permettent la prise en charge d'une valeur par défaut d'une variable ou de la tester.

${var:-default_value}

La syntaxe suivante permet d'afficher une valeur par défaut pour une variable non initialisée. La valeur par défaut est affichée si la variable n'est pas initialisée. Sinon c'est la valeur de la variable qui est affichée. Cette syntaxe n'initialise pas la variable.

Syntaxe : **echo ${var:-default_value}**

Exemple avec la variable var1 initialisée.

```
$ echo $var1
1
$ echo "La valeur de var1 est ${var1:-20}"
La valeur de var1 est 1
$ echo $var1
1
```

Exemple avec la variable var1 non initialisée.

```
$ echo $var1

$ echo "La valeur de var1 est ${var1:-20}"
La valeur de var1 est 20
$ echo $var1
$
```

${var:=default_value}

La syntaxe suivante permet d'afficher une valeur par défaut pour une variable non initialisée. La valeur par défaut est affichée si la variable n'est pas initialisée. Sinon c'est la valeur de la variable qui est affichée. Cette syntaxe initialise la variable si elle ne l'était pas.

Syntaxe : **echo ${var:=default_value}**

Exemple avec la variable var1 initialisée.

```
$ echo $var1
1
$ echo "La valeur de var1 est ${var1:=20}"
La valeur de var1 est 1
$ echo $var1
1
```

Exemple avec la variable var1 non initialisée.

```
$ echo $var1

$ echo "La valeur de var1 est ${var1:=20}"
La valeur de var1 est 20
$ echo $var1
20
```

${var:+default_value}

La syntaxe suivante permet d'afficher une valeur par défaut pour une variable initialisée. La valeur par défaut est affichée si la variable est initialisée. Si la variable n'est pas initialisée, un champ vide apparaît.

Syntaxe : **echo ${var:+default_value}**

Exemple avec la variable var1 initialisée.

```
$ echo $var1
1
$ echo "La valeur de var1 est ${var1:+20}"
La valeur de var1 est 20
$ echo $var1
1
```

Exemple avec la variable var1 non initialisée.

```
$ echo $var1

$ echo "La valeur de var1 est ${var1:+20}"
La valeur de var1 est
$ echo $var1
```

${var:?"message à afficher"}

La syntaxe suivante permet de tester si une variable est initialisée et non nulle. Dans ce cas, le message est affiché à l'écran. Si le message n'est pas spécifié, un message par défaut est affiché.

Syntaxe : **echo ${var:?"message à afficher"}**

Exemple avec un message par défaut.

```
$ echo ${var1:?}
-bash: var1 : paramètre vide ou non défini
```

Exemple avec un message personnalisé.

```
$ echo ${var1:?"La variable var1 n'est pas initialisée"}
-bash: var1: La variable var1 n'est pas initialisée
```

```
$ var1=1
$ echo ${var1:?"La variable var1 n'est pas initialisée"}
1
```

Les variables

La personnalisation de l'environnement de travail

- Ouverture d'une session utilisateur
  - /etc/profile
  - $HOME/.bash_profile
  - La variable BASH_ENV
  - $HOME/.bashrc
- Fermeture d'une session utilisateur
  - $HOME/.logout

La personnalisation de l'environnement de travail

Plusieurs fichiers peuvent être exploités afin de personnaliser l'environnement d'un utilisateur.
Au sein de ces fichiers, nous pouvons définir des variables, des alias et/ou avoir du scripting shell.

Ordre de traitement des fichiers de personnalisation lors de l'ouverture d'une session bash :

/etc/profile	fichier modifiable que par root, permet de centraliser tout ce qui sera commun aux utilisateurs.
$HOME/.bash_profile	s'il existe, modifiable par l'utilisateur.

Au sein de ce fichier peut être défini la variable BASH_ENV, en général positionnée à la valeur «$HOME/.bashrc»

$HOME/.bashrc	le fichier défini par la variable BASH_ENV, s'il existe. Fichier modifiable par l'utilisateur.

Fichier traité lors du lancement d'un sous-shell bash :

$HOME/.bashrc	le fichier défini par la variable BASH_ENV, s'il existe.

Fichier traité lors de la fermeture d'une session bash :

$HOME/.bash_logout	s'il existe, modifiable par l'utilisateur.

Le même principe est utilisé pour le korn shell mais le nom de fichiers de configuration ne sont pas les mêmes.

$HOME/.profile	s'il existe, modifiable par l'utilisateur.

Au sein de ce fichier peut être défini la variable ENV, en général positionnée à la valeur «$HOME/.kshrc»

$HOME/.kshrc	le fichier défini par la variable ENV, s'il existe.

Notes

L'interactivité avec un script

Dans ce chapitre nous allons aborder les scripts interactifs.

L'interactivité avec un script

- La commande read
- Le passage d'arguments
- Les instructions set et shift
- L'affichage (echo, print, printf)

L'interactivité avec un script

La commande read

- Saisie interactive des variables

  read variable read nom prenom autre

- read -p : prompt à afficher pour le message
- read -n : nombre de caractères maximum
- read -t : limiter le temps de saisie
- read -s : ne pas afficher la saisie

La commande read

La commande 'read' attend une saisie clavier. Elle est suivie d'une variable dans laquelle sera stockée la réponse tapée au clavier.

Syntaxe :

```
read  variable
read  nom  prenom  autre
read  -p « Votre nom : »  nom      pour afficher un prompt,

read  -n  5  nom                   pour fixer un nombre de caractères maximum,
read  -t  10  nom                  pour limiter le temps de saisie,
read  -s  nom                      pour ne pas afficher la saisie.
```

Si la commande 'read' est suivie par une seule variable, tout le texte tapé sera stocké au sein de cette variable.

Si le nombre de variables excède le nombre de réponses, les variables excédentaires ne seront pas initialisées.

Si le nombre de réponses excèdent le nombre de variables, la dernière variable contiendra les réponses excédentaires.

Exemple 1 – Syntaxe standard :

```
$ cat script_read1
#!/bin/bash
echo "Quel est votre nom? "
read nom
echo -e "Quel est votre prenom? \c"
read prenom
echo "bonjour $prenom $nom"
```

```
$ ./script_read1
Quel est votre nom?
Cruyff
Quel est votre prenom? Johan
bonjour Johan Cruyff
```

Exemple 2 – Syntaxe standard :

```
$ cat script_read2
#!/bin/bash
echo -e "Quel est votre nom et prenom? \c"
read nom prenom
echo "bonjour $prenom $nom"
```

```
$ ./script_read2
Quel est votre nom et prenom? Di Stefano Alfredo
bonjour Stefano Alfredo Di
```

Exemple 3 – Les options -p et -n :

```
$ cat script_read3
#!/bin/bash

read -p "Indiquez votre prenom: " -n 5 prenom
echo
read -p "Indiquez votre nom: " -n 6 nom
echo
echo "bonjour $prenom $nom"
```

```
$ ./script_read3
Indiquez votre prenom: Miche
Indiquez votre nom: Platin
bonjour Miche Platin
```

Exemple 4 – Les options -t et -s :

```
$ cat script_read4
#!/bin/bash

read -p "Indiquez votre prenom: " -t 10 prenom
echo
read -p "Indiquez votre mot de passe : " -t 10 -s pass
echo
echo "bonjour $prenom ton mot de passe est $pass"
```

```
$ ./script_read4
Indiquez votre prenom: theo

Indiquez votre mot de passe :
bonjour theo ton mot de passe est secret
```

L'interactivité avec un script

Le passage d'arguments

- $0 le nom du script

- $1 le 1er argument du script
- $2 le 2ème argument du script
- ${10} le 10ème argument du script

- $# le nombre d'arguments du script

- $@ la liste des arguments du script
- $* la liste des arguments du script

Le passage d'arguments ($0, $n, $#, $*, ...)

Le shell intègre des variables de postions. Ce sont des variables numériques et sont assignées de manière séquentielle par le shell.

$0 représente le nom du script ou du programme

$1 représente le premier argument passé au script
$2 représente le deuxième argument passé au script
$n représente le nième argument passé au script

$# représente le nombre d'arguments passés au script

Attention $10 est interprété par défaut comme $1 suivie d'un zéro. Il faut utiliser les accolades pour isoler les paramètres de positions supérieurs à 10. Il faudra donc écrire ${10}.

$@ et $* sont similaires et contiennent l'ensemble des arguments passés au script. $@ est interprété comme un seul argument alors que $* comme une liste d'arguments.

```
$ ./script_var2 titi tata toto tutu
Le nom de ce script est: ./script_var2                    # pour $0
Le 1er argument du script est: titi                       # pour $1
Le 2nd argument du script est: tata                       # pour $2
Le 3ème argument du script est: toto                      # pour $3
Le nombre d'arguments du script est: 4                    # pour $#
La liste des arguments du script est: titi tata toto tutu # pour $*
```

```
$ cat etoile_versus_at
#!/bin/bash

OLDIFS=$IFS
IFS="
"

echo 'Sans les "  " le caractère * et @ ont la meme signification'
echo Voici la liste des arguments avec etoile: $*
echo Voici la liste des arguments avec at: $@
echo 'Entoure de "  " le caractère * et @ n ont plus la meme signification'
echo "Voici la liste des arguments avec etoile: $*"
echo "Voici la liste des arguments avec at: $@"

IFS=$OLDIFS
```

IFS est la variable séparateur de champs des arguments (par défaut correspond à trois caractères : espace, tabulation et retour chariot).
Attention : $@ utilise le premier caractère de la variable IFS.

```
$ ./etoile_versus_at  AA  BB  CC
Sans les "  " le caractère * et @ ont la meme signification
Voici la liste des arguments avec etoile: AA BB CC
Voici la liste des arguments avec at: AA BB CC
Entoure de "  " le caractère * et @ n ont plus la meme signification
Voici la liste des arguments avec etoile: AA
BB
CC
Voici la liste des arguments avec at: AA BB CC
```

L'interactivité avec un script

Les instructions set et shift

- set valeur1 valeur2 valeur3
- set $(commande)
- shift
- shift n
- set --

Les instructions set et shift

La commande 'set' permet de redéfinir les variables de positions du shell. Elle peut prendre en argument des valeurs ou le résultat d'une commande. L' option ' - -' de la commande 'set' supprime l'initialisation des variables de position (à part $0).

La commande 'shift' permet d'effectuer un décalage des variables de positions, par défaut d'un cran vers la gauche. La variable $# est alors décrémentée de 1, et la variable $2 devient $1.

```
$ set pim pam poum
$ echo $2
pam
$ echo $@
pim pam poum
$
$ shift
$
$ echo $#
2
$ echo $2
poum
```

La commande shift peut prendre un argument pour indiquer le nombre d'arguments à décaler.

```
$ set $(ls /etc)
$ echo "Il y a $# fichiers et répertoires dans /etc"
Il y a 282 fichiers et répertoires dans /etc
$ shift 3
$ echo "Après décalage, Il y a $# fichiers et répertoires dans /etc"
Après décalage, Il y a 279 fichiers et répertoires dans /etc
```

L'interactivité avec un script

L'affichage (echo, print, printf)

- echo -e (en bash et en ksh)
- print (spécifique ksh)
- printf formatage de l'affichage des données

L'affichage (echo, print, printf)

La commande 'echo' permet d'afficher l'argument vers la sortie standard. Le texte est usuellement entre doubles quotes.

L'option '-e' permet d'interpréter des caractères qui sont échappés avec le caractère \ (exemples : \t , \n, \011, …). L'option '-n' permet d'inhiber le retour chariot de la commande 'echo'.

La commande 'print' est une spécificité du korn shell. Aussi est-il conseillé d'utiliser que la commande 'echo' pour permettre une compatibilité optimale.

```
$ echo –e "Voici un exemple \t\t d'utilisation \n des \v\v caractere \012 echapees"
Voici un exemple                d'utilisation
 des

      caractere
 echapees
```

```
$ echo –en "L'option –n supprime le retour \n chariot en fin de ligne"
L'option –n supprime le retour
 chariot en fin de ligne$
```

La commande 'printf' permet d'effectuer un formatage des données et de l'affichage de ceux-ci. Cette commande fonctionne de la même manière que la fonction 'printf' du langage C.

La syntaxe générale de la commande 'printf' est :

printf format [argument]

'printf' permet d'utiliser des masques pour spécifier le type de valeur :

%d	valeur entière
%s	valeur chaîne
%c	valeur caractère
%f	valeur point flottant
%x	valeur hexadécimal

Pour tout masque dans la chaîne de caractère, il doit y avoir une valeur à la suite. Ces valeurs sont séparées par des virgules.

Pour chaque masque, des options de formatage existent :

%<chiffre>d	La valeur est cadrée à droite dans la taille indiquée du champ.
%-<chiffre>d	La valeur est cadrée à gauche dans la taille indiquée du champ.
%.<chiffre>f	Indique le nombre de décimales pour les valeurs à points flottants.
%<chiffre>.<chiffre>f	Indique la taille du champ et le nombre de décimales.

Exemples :

```
$ var="10" ; printf "%05d\tfin" "$var"
00010        fin$
```

```
$ printf "Le resultat est %10.4f\n" "3,1234556"
Le resultat est     3,1235
```

Remarque la commande 'printf' n'a pas tronqué le résultat mais l'a arrondi selon les règles mathématiques en vigueur.

```
$ printf "Le resultat est %10.4f\n" "3,12"
Le resultat est     3,1200
```

Notes

Les tests, les opérateurs if et case

Dans ce chapitre nous allons traiter de la commande test, ainsi que les instructions conditionnelles if et case.

Les tests, les opérateurs if et case

- Le code retour $?
- Les opérateurs && et ||
- La commande test
- L'instruction conditionnelle if
- L'instruction conditionnelle case

Les tests, les opérateurs if et case

Le code retour $?

- La variable $? = code de retour d'une commande
  = 0 → commande ou instruction correcte
  ≠ 0 → commande ou instruction incorrecte
- Le code de retour indique le type d'erreur
- Le code de retour est utilisé par la commande test

Le code retour $?

Pour chaque commande ou script exécuté, un code retour est stocké dans la variable spéciale du shell $?.

Si cette variable contient la valeur 0, alors la dernière instruction s'est correctement déroulée.

Si cette variable contient une valeur différente de 0, alors la dernière instruction ne s'est pas correctement déroulée.

```
$ pwd
/home/theo
$ echo $?
0
```

```
$ pwwwd
bash: pwwwd: commande inconnue...
$ echo $?
127
```

Le code retour d'une commande peut indiquer le type d'erreur pour certaines commandes.
Consultez la page de man de la commande correspondante pour connaître les différents codes et types d'erreurs d'une commande.
Par exemple, consultez le manuel de la commande useradd pour visualiser les différents types d'erreurs possibles.

Les tests, les opérateurs if et case

Les opérateurs && et ||

- cmde1 && cmde2

  La commande 2 est traitée si la dernière commande exécutée renvoie un code de retour égal à 0

- cmde1 || cmde2

  La commande 2 est traitée si la dernière commande exécutée renvoie un code de retour différent de 0

Les opérateurs && et ||

L'opérateur && traite la seconde commande si la dernière commande exécutée renvoie un code de retour égal à 0.

```
$ pwd  &&  date
/home/theo
jeu. janv. 12 16:12:42 CET 2017
$ pwwwd  &&  date
bash: pwwwd: commande inconnue...
```

L'opérateur || traite la seconde commande si la dernière commande exécutée renvoie un code de retour différent de 0.

```
$ pwd  ||  date
/home/theo
$ pwwwd  ||  date
bash: pwwwd: commande inconnue...
jeu. janv. 12 16:13:20 CET 2017
```

Exemples

```
read –p «Un repertoire : » repertoire
cd $repertoire 2> /dev/null  ||  echo «Echec. $repertoire n existe pas.»


read –p «Un repertoire : » repertoire
cd  $repertoire   &&   touch fichier   &&   echo «Création réussie.»
```

Les tests, les opérateurs if et case

La commande test

- test -f fichier
- test -d fichier
- test -r fichier
- test -w fichier
- test -x fichier

test ch1 = ch2
test ch1 != ch2
test -z chaîne
test -n chaîne
test chaîne

test nb1 -eq nb2
test nb1 -ne nb2
test nb1 -lt nb2
test nb2 -le nb2
test nb1 -gt nb2
test nb1 -gt nb2

- [ ] [[ ]] (( )) ! expression

La commande test

La commande 'test' permet d'évaluer une expression. Le code de retour est 0 si le test est vrai.
Les expressions testées peuvent être unaires ou binaires.

```
$ test -d /home/tux
$ echo $?
0
```

L'écriture [ ] remplace la commande 'test'. Elle demande cependant une rigueur d'écriture, des espaces étant indispensable après le [et avant le].

```
$ [ -d /home/tux ]
$ echo $?
0
```

```
$ [ tux != theo ]
$ echo $?
0
```

```
$ [ 10 -lt 5 ]
$ echo $?
1
```

L 'écriture (( )) est utilisée pour effectuer des tests sur les nombres.

La commande 'test' est une primitive du shell dont l'écriture réduite est [ ].

Les doubles crochets sont plus récents et sont usuellement utilisés pour effectuer des tests sur les chaînes de caractères. Il n'est pas recommandé de les utiliser pour effectuer des tests sur les nombres.

Tests sur les fichiers :

Opérateur	Code Retour égal à 0 si
[-b fic]	le fichier est un fichier spécial en mode bloc
[-c fic]	le fichier est un fichier spécial en mode caractère
[-d fic]	le fichier est un répertoire
[-e fic]	le fichier existe
[-f fic]	le fichier est un fichier ordinaire
[-g fic]	le fichier a un setgid de positionné
[-k fic]	le fichier a le stickybit de positionné
[-L fic]	le fichier est un lien symbolique
[-p fic]	le fichier est un tube nommé
[-r fic]	le fichier a le droit de lecture
[-s fic]	le fichier a une taille supérieure à 0
[-S fic]	le fichier est une socket
[- t fd]	le descripteur de fichier est ouvert sur le terminal. Si fd est omis par défaut c'est le descripteur de fichier 1 qui est utilisé (sortie standard)
[-u fic]	le setuid est positionné
[-w fic]	le fichier a le droit d'écriture
[-x fic]	le fichier a le droit d'exécution
[-O fic]	le fichier appartient à l'UID effectif de l'exécuteur
[-G fic]	le fichier appartient au GID effectif de l'exécuteur
[fic1 -nt fic2]	fic1 est plus récent que fic2 (utilise la date de dernière modification)
[fic1 -ot fic2]	fic1 est plus ancien que fic2 (utilise la date de dernière modification)
[fic1 -ef fic2]	fic1 et fic2 ont les mêmes numéros de périphériques et d'inode

Important : Le caractère ! inverse le sens du test. Le code retour sera égal à zéro si le test est faux.

Exemple :

```
$ [ ! -d /home/tux ]
$ echo $?
1
```

Autres exemples :

```
[ ! -d $rep ]  &&  echo «$rep n est pas un repertoire.»

[ -d $rep ]    ||  ( echo «$rep n est pas un repertoire.» ; rep=/etc )

[ -d $rep ] && cd  $rep &&  touch fichier || echo «Echec.»

[ -d $rep ] && [ -f $fic ] && [ -f $fic_autre ] && echo Alors_Actions
```

Tests sur les nombres :

Opérateur	Code Retour égal à 0 si	(( ))
[nb1 -eq nb2]	nb1 est égale à nb2	((nb1 == nb2))
[nb1 -ne nb2]	nb1 n'est pas égale à nb2	((nb1 != nb2))
[nb1 -lt nb2]	nb1 est strictement inférieur à nb2	((nb1 < nb2))
[nb1 -le nb2]	nb1 est inférieur ou égale à nb2	((nb1 <= nb2))
[nb1 -gt nb2]	nb1 est strictement supérieur à nb2	((nb1 > nb2))
[nb1 -ge nb2]	nb1 est supérieur ou égale à nb2	((nb1 >= nb2))

Remarque : Avec l'écriture (( )) les espaces ne sont pas obligatoires.

Tests sur les chaînes :

Opérateur	Code Retour égal à 0 si
[ch1 = ch2]	ch1 est égale à ch2
[ch1!= nb2]	ch1 est différent de ch2
[-z chaîne]	La chaîne est nulle (taille égale à zéro)
[-n chaîne]	La chaîne est non nulle (taille différente de zéro)
[chaîne]	La chaîne n'est pas la chaîne vide
[ch1 < ch2]	ch1 est avant ch2 au niveau lexical
[ch1 > ch2]	ch1 est après ch2 au niveau lexical

Tests sur les expressions :

Opérateur	Code Retour égal à 0 si
[! expression]	L'expression est fausse
[expr1 -a expr2]	expr1 et expr2 sont toutes les deux vraies
[expr1 -o expr2]	expr1 ou expr2 est vraie
[-n chaîne]	La chaîne est non nulle (taille différente de zéro)
[chaîne]	La chaîne n'est pas la chaîne vide

Les tests, les opérateurs if et case

L'opérateur conditionnel if

if *commande*	if *test*	if *condition*
if test then commande … fi	if test then commande else commande fi	

L'opérateur conditionnel if

Syntaxe :

```
if code_de_retour
then
        cmd si code de retour = 0
fi
```

```
if test
then
        commandes si test vrai
else
        commandes si test faux
fi
```

L'instruction if permet d'exécuter une séquence de commande si le test est vrai (code de retour égal 0). Optionnellement, une section else peut être présente. Dans ce cas, la séquence de commandes correspondante est traitée si le test est faux (code de retour différent de 0).

```
$ cat script_if
#!/bin/bash
echo « Saisir un nombre : » ; read num

if (( $num == 5 ))
then
   echo « La saisie est 5 »
else
   echo « La saisie est differente de 5 »
fi
```

ou

```
if (( $num == 5 ))
then echo « La saisie est 5 »
else echo « La saisie est differente de 5 »
fi
```

Les tests, les opérateurs if et case

L'opérateur conditionnel if

```
if   test1
   then   commande
elif   test2
   then   commande
elif   test3
   then   commande
else   commande
fi
```

```
if  grep $user /etc/passwd > /dev/null
if  ! grep $user /etc/passwd > /dev/null

if  test  -f  $file
if  [  -f  $file  ]
if  ((  $num == 5  ))
if  [[  "$nom" !=  "theo"  ]]
```

```
[  "$(whoami)"  !=  'root'  ]  &&  ( echo tu n es pas root ;  exit 1 )
```

« elif » : une syntaxe pour des if-else imbriqués.

Exemple :

```
if  ((  $val  <  10  ))
   then  tarif=1
elif  ((  $val  <  20  ))
   then  tarif=2
elif  ((  $val  <  30  ))
   then  tarif=3
else   tarif=4
fi
```

Quelques utilisations du if :

```
if  grep  $user  /etc/passwd  >  /dev/null

if  !  grep  $user  /etc/passwd  >  /dev/null

if   test  -f  $file      équivalent à       if   [  -f  $file  ]

if   ((  $num  ==  5  ))

if   [[  "$nom"  !=  "theo"  ]]

if [  "$(whoami)"  !=  'root'  ]

[  "$(whoami)"  !=  'root'  ] && ( echo tu n es pas root ; exit 1 )

test  "$(whoami)"  !=  'root'  &&  (echo tu n es pas root  ;  exit 1)
```

Les tests, les opérateurs if et case

L'opérateur conditionnel case

```
case   variable   in
          valeur_1)   cmde_1 ;;
          valeur_2)   cmde_2 ;;
          valeur_n)   cmde_n ;;
                *)    cmde ;;          optionnel : tous les autres cas
esac
```

L'opérateur conditionnel case

Syntaxe :

```
case  variable  in
      valeur_1)  cmde_1              traitement si variable=valeur_1
                 ;;                  jusqu'à ;; , puis sortie du case
      valeur_2)  cmde_2
                 ;;
      valeur_3)  cmde_3
                 ;;
      valeur_n)  cmde_n
                 ;;
            *)  cmde                 * pour le traitement de tous les autres cas
esac
```

L'opérateur conditionnel case permet d'exécuter une séquence de commandes spécifique en fonction de la valeur de la variable du case.

Ainsi la section de commandes « cmde_2 » sera traitée si variable=valeur2. La fin du traitement est signifiée par le ;;. Suite à cette instruction, il y a une une sortie du case, la suite du programme se réalise après l'instruction esac.

Le caractère * est optionnel, il permet un traitement pour tous les autres cas.

Exemple 1 :

```
case  $nom  in
      «jean»)       echo  tu es jean
                    echo  tu as reussi l examen
      ;;
      «marc»)       echo  tu es marc
      ;;
      *)            echo je ne te connais pas
esac
```

Exemple 2 :

```
case  $nom  in
      «jean»)              echo  tu es jean ;;
      «marc»|«theo»)       echo  tu es $nom ;;
      «celine»)            echo tu as reussi l examen
esac
```

pour jean :	pour marc :	pour theo:	pour celine:
tu es jean	tu es marc	tu es theo	tu as reussi l examen

Exemple 3 :

```
case   «$(date  +'%H')»  in
      0[0-9]|1[01])echo  C  est  la  matinee ;;
      1[2-9])  echo  C  est  l  apres midi ;;
      2[0-3])  echo  C est la soiree ;;
esac
```

Notes

Les boucles

Dans ce chapitre nous allons traiter des boucles for, while et until, ainsi que d'instructions telles que break, continue et exit.

Les boucles

- La boucle for
- La boucle while
- La boucle until
- Les instructions break, continue et exit

Les boucles

La boucle for

for var in liste_de_paramètres
do
commande
…
done

for ((initialisation ; test_de_fin ; incrémentation))

for var in {a..d} **for var in {10..20..2}**

La boucle for

La boucle for permet d'exécuter plusieurs fois une même séquence de commandes.

Syntaxe classique :

```
for var in liste_de_paramètres
do
        commande
        …
done

for var in aa bb cc dd
for var in $(date)
```

La séquence de commandes inscrite entre le « do … done » sera exécutée autant de fois qu'il y a de paramètres au sein de la liste « liste_de_paramètres ».
chaque paramètre de liste sera exploitable via la variable 'var' à chaque itération.

```
$ cat boucle_for
#!/bin/bash

for element in aa bb cc
do
   echo « Traitement de $element »
done
echo fin
```

```
$ ./boucle_for
Traitement de aa
Traitement de bb
Traitement de cc
fin
```

```
$ for i in $(seq 10 2 30); do printf "%03d\t" "$i"; done ; echo
010     012     014     016     018     020     022     024     026     028     030
```

Autre syntaxe en bash :

for (initialisation ; test_de_fin ; incrémentation)
do
commande
…
done

La première phase, avant le traitement du « do ...done », la séquence « initialisation » est traitée.
Puis la séquence de commandes inscrite entre le « do … done » sera exécutée tant de le « test_de_fin » est vrai.
A la fin de chaque itération, la séquence « incrémentation est traitée, avant le test de fin.

```
$ cat boucle_for
#!/bin/bash

for (( i=0 ; i < 3 ; i++ ))
do
   echo « Traitement de $i »
done
echo fin

$ ./boucle_for
Traitement de 0
Traitement de 1
Traitement de 2
fin
```

Compléments :

for ((i=0 ; i < 10 ; i++))

for ((i=0 , j=20 ; i < 10 ; i++ , j- -))

Utilisation de {a..d}

Pour définir une liste, la syntaxe suivante est possible :

{d..k} une liste de caractère de d à k.

{30..50} une liste de nombres de 30 à 50.

Possibilité de définir un pas d'incrémentation :

{0..50..5} une liste de 0 à 50 avec un pas d'incrémentation de 5.

Exemples

```
$ cat script_for
#!/bin/bash

for var in {a..d}
do
        echo $var
done
```

```
$ ./script_for
a
b
c
d
```

```
$ cat script_for2
#!/bin/bash

for var in {10..20..2}  ; do
        echo $var
done
```

```
$ ./script_for2
10
12
14
16
18
20
```

Les boucles

La boucle while

while ***test_de_continuité***
do
commande
…
done

while ((i < 10)) ; do
((i++)) ; echo $i
done

La boucle while

Syntaxe :

while test_de_continuité
do
commande
…
done

La séquence de commandes inscrite entre le « do … done » sera exécutée tant que le test de continuité est vrai (c'est à dire le code de retour à 0).

```
$ cat boucle_while
#!/bin/bash

i=0
while (( i < 3 ))
do
   echo « Traitement de $i »
   (( i++ ))
done
echo fin

$ ./boucle_while
Traitement de 0
Traitement de 1
Traitement de 2
fin
```

Il est donc facile de réaliser une boucle infinie avec un test qui est toujours vrai :

```
while (( 10 == 10 ))
while true
while :
```

Autres exemples

```
$ cat script_while
#!/bin/bash

while (($# != 0)) ; do
        echo $*
        shift
done
```

```
$ ./script_while aa bb cc dd
aa bb cc dd
bb cc dd
cc dd
dd
```

```
$ cat script_while2
#!/bin/bash

while (($# != 0)) ; do
        echo $1
        shift
done
```

```
$ ./script_while2 aa bb cc dd
aa
bb
cc
dd
```

Les boucles

La boucle until

until ***test_de_fin***
do
commande
…
done

until ((i == 10)) ; do
((i++)) ; echo $i
done

La boucle until

Syntaxe :

```
until  test_de_fin
do
        commande
        …
done
```

La séquence de commandes inscrite entre le « do … done » sera exécutée jusqu'à ce que le test de fin soit vrai.

La boucle until (« jusqu'à ce que ») est l'inverse de la boucle while (« tant que ») pour la phase du test de fin de la boucle.

```
$ cat boucle_until
#!/bin/bash

i=0
until (( i == 3 ))
do
   echo « Traitement de $i »
   (( i++ ))
done
echo fin

$ ./boucle_until
Traitement de 0
Traitement de 1
Traitement de 2
fin
```

Les boucles

Les instructions break, continue et exit

break	sortie de la boucle en cours.
continue	passe directement à l'itération suivante de la boucle.
exit	sortie du script avec un code de retour.

Les instructions break, continue et exit

L'instruction break :

'break' permet de sortir de la boucle en cours de traitement.

```
$ cat script_break
#!/bin/bash

i=0
while (( i < 5 ))
do
   (( i++ ))
   echo « Phase 1 de $i »
   if  (( i == 3 )) ; then echo « Phase break »
                           break
   fi
   echo « Phase 2 de $i »
done
echo fin
```

```
$ ./script_break
Phase 1 de 1
Phase 2 de 1
Phase 1 de 2
Phase 2 de 2
Phase 1 de 3
      Phase break
fin
```

L'instruction 'break' peut faire sortir de plusieurs boucles imbriquées en précisant le nombre de boucles en argument. Par exemple : break 2.

L'instruction continue:

'continue' permet de passer directement à l'itération suivante de la boucle en cours de traitement.

```
$ cat script_break
#!/bin/bash

i=0
while (( i < 5 ))
do
   (( i++ ))
   echo « Phase 1 de $i »
   if  (( i == 3 )) ; then echo « Phase continue»
                           continue
   fi
   echo « Phase 2 de $i »
done
echo fin
```

```
$ ./script_break
Phase 1 de 1
Phase 2 de 1
Phase 1 de 2
Phase 2 de 2
Phase 1 de 3
      Phase continue
Phase 1 de 4
Phase 2 de 4
Phase 1 de 5
Phase 2 de 5
fin
```

L'instruction exit :

'exit' arrête le script. Cette instruction peut être utilisée avec un nombre en argument qui initialisera le code de retour. Le code de retour par défaut est 0.

exit	echo $? affichera 0.
exit 0	echo $? affichera 0.
exit 1	echo $? affichera 1.
exit 5	echo $? affichera 5.

Ainsi, pour un arrêt normal d'un script, on utilisera une syntaxe avec un code de retour à 0. Pour spécifier un arrêt du script suite à un événement anormal, on utilisera un code de retour différent de 0.
Ce code de retour peut être exploité afin de définir un type d'anomalie en fonction de sa valeur.

```
$ cat traitement
#/bin/bash
… sequence de commandes
[  !  -d  $1  ]  &&  exit 1
… sequence de commandes
[  !  -d  $2  ]  &&  exit 2
… sequence de commandes
[  !  -f  $3  ]  &&  exit 3
… sequence de commandes
```

```
$ cat  script_maitre
#/bin/bash
… commandes, initialisation de var1, var2 et var3

./traitement $var1  $var2  $var3
etat_retour=$?

(( $etat_retour == 1 )) && echo «$var1 n est un repertoire»  && exit $etat_retour
(( $etat_retour == 2 )) && echo «$var2 incompatible»  && exit $etat_retour
(( $etat_retour == 3 )) && (echo «Correction champs3» ; var3=file ; echo «on continu»)

… suite du code, le code de retour de traitement est 0, tout va bien !
             Ou  Le code de retour de traitement est 3, mais l'anomalie est corrigée.
```

Notes

Le traitement arithmétique

Dans ce chapitre, nous allons traiter les différentes méthodes pour effectuer des traitements arithmétiques.

Le traitement arithmétique

- Les instructions expr, let, bc
- L'utilisation de (( ))

Le traitement arithmétique

Les instructions expr, let, bc

- expr 1 + 1 expr $a + 1
- let ((expression))
- bc bc -l

Les instructions expr, let, bc

La commande expr

La commande 'expr' permet d'évaluer une expression et d'afficher le résultat sur la sortie standard.
Si l'expression est invalide, cette commande renvoie un code de retour égal à 2.
Si l'expression est nulle ou 0, le code de retour est égal à 1.
Dans les autres cas, le code de retour est égal à zéro.

'expr' accepte les opérateurs suivants pour effectuer des opérations arithmétiques :

Opérateur	Signification
+	addition
-	soustraction
*	multiplication
/	division
%	modulo

L'utilisation de la commande 'expr' demande une certaine rigueur d'écriture. De part et d'autre de l'opérateur un espace est obligatoire pour que la commande fonctionne correctement.

Exemple 1 : Attention à la syntaxe.

```
$ expr 10 + 20
30
$ expr 10 +20
expr: erreur de syntaxe
$ expr 10+ 20
expr: erreur de syntaxe
$ expr 10+20
10+20
```

Exemple 2 :

```
$ a=5
$ expr $a \* 3
15
$ echo $a
5
$ a=$(expr $a \* 3)
$ echo $a
15
```

La commande let

La commande 'let' s'utilise avec un ou plusieurs arguments. Chaque argument est une expression arithmétique à évaluer. Si l'évaluation de dernier argument donne 0, 'let' renvoie la valeur 1 sinon elle renvoie la valeur 0.

La commande 'let' est équivalente à ((expression)). Elle utilise les mêmes opérateurs que 'expr'. Néanmoins le caractère '*' n'a pas besoin d'être échappé. Les parenthèses permettent de prioriser les opérations.

```
$ a=10
$ let a=a+20
$ echo $a
30
$ let a=a*2
$ echo $a
60
$ let a=a+1
$ echo $a
61
$ let a=a%2
$ echo $a
1
$ let b=a\*2
$ echo $b
2
$ let b++
$ echo $b
3
```

```
$ a=2
$ echo $a
2

$ let a=$a+1*2+5
$echo $a
9
$ let a=($a+1)*2+5
$ echo $a
25
```

La commande bc

La commande 'bc' (beautiful calculator) permet d'effectuer des calculs comme avec une calculatrice. Le comportement par défaut, sans argument, est le mode interactif.

Les nombres sont définis en deux attributs :

- La longueur : nombre total de chiffres décimaux significatifs d'un nombre.
- L'échelle : nombre total de chiffres décimaux après le point décimal.

Exemples :

0.0000001	a une longueur de 8 et une échelle de 7
192.56	a une longueur de 5 et une échelle de 2

```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
3*5
15
./2                 Le caractère point (.) permet de rappeler le dernier résultat.
7
scale=3             La commande interne 'scale' permet de définir l'échelle.
./2
3.500
.*4
14.000
scale=5
./3
4.66666
^D
```

Par défaut, 'bc' se lance sans la gestion des nombres flottants. Pour sa prise en charge, indiquez l'option -l à la commande 'bc'.

```
$ bc -l
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
8/3
2.66666666666666666666
```

L'utilisation non interactive utilise les redirections des entrées standards.

```
$ a=$(echo "scale=2 ; 10/3" | bc -l)
$
$ echo $a
3.33
```

Le traitement arithmétique

L'utilisation de (( ))

- (( )) écriture abrégée pour let
- (( )) pour des opérations arithmétiques ((a=b+5)) ((a*b))
- (( )) pour effectuer des tests ((a != 5))

L'utilisation de (( ))

(( )) est une syntaxe simplifiée pour réaliser des opérations arithmétiques. Elles seront également très pratiques pour les tests.

Une syntaxe souple : ((5*2)) est équivalent à ((5*2)) ou ((5 * 2)).

Utilisation pour des opérations :

```
$ a=5
$ (( b = a * 2 ))
$ echo $b
10
$ var1=$((a+10))
$ echo $var1
15
$ var2=$(($a+10))
$ echo $var2
15
```

Utilisation pour des tests en exploitant le code de retour :

```
$ a=10
$ ((a>5))
$ echo $?
0                       pour indiquer que ce test est vrai
$ ((a<5))
$ echo $?
1                       pour indiquer que ce test est faux
```

Nous pourrons exploiter : <, >, <=, >=, ==, !=

Notes

Le traitement des chaînes de caractères

Dans ce chapitre nous allons traiter les différentes méthodes pour manipuler des chaînes de caractères.

Le traitement des chaînes de caractères

- La commande expr
- La commande typeset
- Manipulations avancées

Le traitement des chaînes de caractères

La commande expr

- expr length $var expr "$var" : '.*'
- expr substr $var position longueur
- expr index $var chaîne_recherchée
- expr match $var chaîne

La commande expr

La commande 'expr' permet d'évaluer une expression et d'afficher le résultat sur la sortie standard. Cette commande accepte un certain nombre de mots clefs pour effectuer des opérations sur les chaînes de caractères.

Afficher la longueur d'une variable.

```
$ var=formidable
$ expr length $var                $ expr "$var" : '.*'
10                                10
```

Extraire une sous-chaîne d'une variable.

```
$ expr substr $var 5 3
ida
```

Afficher la position de la première occurrence de la variable correspondant à la chaîne recherchée.

```
$ expr index $var ida
5
```

Afficher le nombre de caractères qui correspond à une expression, à partir du début de la chaîne.

```
$ surnom=Pauline_4625
$ expr match $surnom '[a-zA-Z]*_'
8
```

Afficher une sous-chaîne qui correspond à une expression (identifiée par '\(...\)').

```
$ expr match $surnom '\([a-zA-Z]*\)_'                début de chaîne
Pauline
$ expr match $surnom '.*_\([0-9]*\)'                 fin de chaîne
4625
```

Le traitement des chaînes de caractères

La commande typeset en ksh

typeset -u var	transforme var en majuscules
typeset -l var	transforme var en minuscules
typeset -LZ var	élimine les 0 non significatifs de var
typeset -Lnum var	cadre à gauche var dans la longueur indiquée par num
typeset -Rnum var	cadre à droite var dans la longueur indiquée par num

La commande typeset en ksh

La commande 'typeset' permet de formater une chaîne de caractères :

- typeset -u var transforme var en majuscules,
- typeset -l var transforme var en minuscules,
- typeset -LZ var élimine les 0 non significatifs de var,
- typeset -Lnum var cadre à gauche var dans la longueur indiquée par num,
- typeset -Rnum var cadre à droite var dans la longueur indiquée par num.

```
$ cat script_typeset
#!/bin/ksh

nom="Linus Thorvald"
echo "La variable nom contient: $nom"
typeset -u nom      ; echo "en majuscule: $nom" ;
typeset -l nom      ; echo "en minuscule: $nom" ;
typeset -L5 nom     ; echo "cadré à droite avec une longueur de 5: $nom"
typeset -R30 nom    ; echo "cadré à gauche avec une longueur de 30: $nom"

var=000024
echo "La variable var contient: $var"
typeset -LZ var     ; echo "Sans les zéros cela donne: $var"
```

```
$ ./script_typeset
La variable nom contient: Linus Thorvald
en majuscule: LINUS THORVALD
en minuscule: linus thorvald
cadré à droite avec une longueur de 5: linus
cadré à gauche avec une longueur de 30:                          linus
La variable var contient: 000024
Sans les zéros cela donne: 24
```

Le traitement des chaînes de caractères

Manipulation avancée

- ?(expression) 0 à 1 fois l'expression
- *(expression) 0 à n fois l'expression
- +(expression) 1 à n fois l'expression
- @(expression) 1 fois l'expression
- !(expression) 0 fois l'expression
- ?(expr1|expr2) 0 à 1 fois expr1 ou expr2

Manipulation avancée

```
$ ls
fic100.tar fic10.gz  fic10.tar  fic1.gz  fic1.tar  fic20.tar  fic2.gz  fic2.tar  fic.tar
tp10.tar  tp2.tar  tp3.tar  tp.tar
```

?(expression) 0 à 1 fois l'expression

fic?([0-9]).tar correspond à fic.tar, fic1.tar, fic2.tar, …. ,fic9.tar

```
$ ls fic?([0-9]).tar
fic1.tar  fic2.tar  fic.tar
```

***(expression) 0 à n fois l'expression**

fic*([0-9]).tar correspond à fic.tar, fic1.tar, fic2.tar, …., ficN.tar

```
$ ls fic*([0-9]).tar
fic100.tar  fic10.tar  fic1.tar  fic20.tar  fic2.tar  fic.tar
```

+(expression) 1 à n fois l'expression

fic+([0-9]).tar correspond à fic1.tar, fic2.tar, …., ficN.tar

```
$ ls fic+([0-9]).tar
fic100.tar  fic10.tar  fic1.tar  fic20.tar  fic2.tar
```

@(expression) **1 fois l'expression**
fic@([0-9]).tar correspond à fic1.tar, fic2.tar, …. ,fic9.tar

```
$ ls fic@([0-9]).tar
fic1.tar  fic2.tar
```

!(expression) **0 fois l'expression**
fic!([0-9]).tar exclusion de l'expression

```
$ ls fic!([0-9]).tar
fic100.tar  fic10.tar  fic20.tar  fic.tar
```

(expression1|expression2) **expression1 ou expression2**

```
$ ls fic?([0-9]).@(tar|gz)
fic1.gz  fic1.tar  fic2.gz  fic2.tar  fic.tar
```

Avec l'opérateur **[[ ]]**, ces expressions complexes peuvent-être utilisées au sein de scripts.
&& signifie un ET logique (équivalent du -a pour l'opérateur test).
|| signifie un OU logique (équivalent du -o pour l'opérateur test).

```
$ cat script_chaine
#!/bin/bash

read -p "Veuillez saisir un nom: " nom

while [[ $nom != +([a-zA-Z]) ]]
do
        echo "Veuillez ne saisir que des caractères alphabétiques: "
        echo -e "Nouvelle saisie: \c"
        read nom
done

echo "Le nom saisie est $nom"
```

```
$ ./script_chaine
Veuillez saisir un nom: jean-marc
Veuillez ne saisir que des caractères alphabétiques:
Nouvelle saisie: theo
Le nom saisie est theo
```

Notes

Les fonctions

Dans ce chapitre nous allons traiter des fonctions pour mieux organiser un script.

Les fonctions

- Déclaration et syntaxe
- Passage d'arguments
- Les variables, $? et le mot clé return
- L'externalisation des fonctions
- L'externalisation en ksh : FPATH et autoload

Les fonctions

Déclaration et syntaxe

Rôle d'une fonction

```
ma_fonction () {
        …
}

function ma_fonction {
        …
}
```

Déclaration et syntaxe

Une fonction permet de regrouper un ensemble de commandes. Cette fonction peut être exécutée à partir du programme principale ou au sein d'une autre fonction. Elle est utilisée comme une commande. La déclaration d'une fonction doit être faite avant qu'elle soit sollicitée.

Une fonction sert à regrouper un ensemble d'instructions qui est sollicité plusieurs fois. Cela améliore la lisibilité et l'organisation d'un script.
Elle permet de simplifier le code du programme principale, en déportant des commandes dans des fonctions.

Syntaxe :

```
ma_fonction () {          ou          function ma_fonction {
        …                                     …
}                                     }
```

Exemple :

```
$ cat script_fonction
#!/bin/bash

affichage ()  {
      echo   '***************'
}

affichage
echo  bonjour, ceci est le code principal.
affichage
```

```
$   ./script_fonction
***************
bonjour, ceci est le code principal.
***************
```

Les fonctions

Passage d'arguments

```
$ cat script_fonction
#!/bin/bash
function essai () {
      var2=20
      echo « fonction : parametre 1 = $1  parametre 2 = $2  parametre 3 = $3 »
      echo « fonction : variable 1 = $var1  variable 2 = $var2 »
      var1=50
}

echo « Programme principal : parametre 1 = $1  parametre 2 = $2  parametre 3 = $3 »
echo « Programme principal : variable 1 = $var1  variable 2 = $var2 »
essai
echo  « Programme principal : variable 1 = $var1  variable 2 = $var2 »

var1=10 ; essai
echo « Programme principal : parametre 1 = $1  parametre 2 = $2  parametre 3 = $3 »
echo  « Programme principal : variable 1 = $var1  variable 2 = $var2 »

essai   Jean   Marc
echo « Programme principal : parametre 1 = $1  parametre 2 = $2  parametre 3 = $3 »

$ ./script_fonction   Celine   Eve   Valerie
```

Passage d'arguments

Une fonction s'exécute au sein du shell du script. Ainsi toutes variables modifiées ou crées dans une fonction est modifiée et exploitable au sein du reste du script.

Les arguments passés à la fonction initialisent les variables $1 à $n, uniquement au sein de la fonction.

Le mot clé 'local' :
'local' déclare une variable en tant que locale à la fonction. La durée de vie de cette variable est donc limitée à la fonction dans laquelle elle a été déclarée.

Exemple :

```
$ cat script_fonction
#!/bin/bash
function essai () {
       var2=20
       echo « fonction : parametre 1 = $1  parametre 2 = $2  parametre 3 = $3 »
       echo « fonction : variable 1 = $var1  variable 2 = $var2 »
       var1=50
}

echo « Programme principal : parametre 1 = $1  parametre 2 = $2  parametre 3 = $3 »
echo « Programme principal : variable 1 = $var1  variable 2 = $var2 »
echo

essai

echo  « Programme principal : variable 1 = $var1  variable 2 = $var2 »
echo

var1=10
essai

echo « Programme principal : parametre 1 = $1  parametre 2 = $2  parametre 3 = $3 »
echo  « Programme principal : variable 1 = $var1  variable 2 = $var2 »
echo

essai   Jean   Marc
echo « Programme principal : parametre 1 = $1  parametre 2 = $2  parametre 3 = $3 »


$ ./script_fonction  Celine  Eve  Valerie
Programme principal : parametre 1 = Celine  parametre 2 = Eve  parametre 3 = Valerie
Programme principal : variable 1 =   variable 2 =

fonction : parametre 1 =   parametre 2 =   parametre 3 =
fonction : variable 1 =   variable 2 = 20
Programme principal : variable 1 = 50  variable 2 = 20

fonction : parametre 1 =   parametre 2 =   parametre 3 =
fonction : variable 1 = 50  variable 2 = 20
Programme principal : parametre 1 = Celine  parametre 2 = Eve  parametre 3 = Valerie
Programme principal : variable 1 = 10  variable 2 = 20

fonction : parametre 1 = Jean  parametre 2 = Marc  parametre 3 =
fonction : variable 1 = 50  variable 2 = 20
Programme principal : parametre 1 = Celine  parametre 2 = Eve  parametre 3 = Valerie
```

```
$ cat fonction_variable_locale
#!/bin/bash
function hello {
       local var=10
       nom=20
       echo «fonction :var=$var et nom=$nom»
}

var=jean ; nom=marc
echo «Programme principal : var=$var et nom=$nom»
hello
echo «Programme principal : var=$var et nom=$nom»



$ ./fonction_variable_locale
Programme principal :  var=jean et nom=marc
fonction :var=10 et nom=20
Programme principal :  var=jean et nom=20
```

Les fonctions

Les variables, $? et le mot clé return

- Une fonction peut initialiser des variables
- return
  - Sortie d'une fonction
  - Initialise le code de retour $? du programme principal
  - Peut prendre un nombre en argument

Les variables, $? et le mot clé return

Une fonction peut initialiser des variables :

```
$ cat script_fonction
#!/bin/bash
function essai () {
       echo «bonjour»
}

echo « Phase 1 »
essai

echo
echo « Phase 2 »
var=$(essai)

echo « Resultat : $var »

$ ./script_fonction
Phase 1
bonjour

Phase 2
Resultat : bonjour
```

Le mot clé 'return' permet de sortir d'une fonction et d'initialiser le code de retour $? au sein du programme principal. Optionnellement, 'return' peut prendre un nombre en argument (valeur de 0 à 255).

```
$ cat script_fonction
#!/bin/bash
function essai () {
       echo «bonjour»
       return 2
}

echo « Phase 1 »
essai
echo « Code de retour : $? »

echo ; echo « Phase 2 »
var=$(essai)
echo « Code de retour : $? » ; echo « Resultat : $var » ;

echo ; echo « Phase 3 »
echo « Resultat : $(essai) » ; echo « Code de retour : $? »
```

```
$ ./script_fonction
Phase 1
bonjour
Code de retour : 2

Phase 2
Code de retour : 2
Resultat : bonjour

Phase 3
Resultat : bonjour
Code de retour : 0              0 car correspond au code de retour du echo
```

```
$ cat calmax
#!/bin/bash
function maxi () {
       echo «Operation entre $1 et $2»
       if (( $1 > $2 ))
       then return $1
       else return $2
       fi
}

maxi 50 10
resultat=$?
echo « Max entre 50 et 10 : $resultat » ; echo

maxi 10 20
resultat=$?
echo « Max entre 10 et 20 : $resultat » ; echo

resultat=$(maxi 10 20)
echo « Attention : $resultat »
```

```
$ ./script_fonction
Operation entre 50 et 10
Max entre 50 et 10 : 50

Operation entre 10 et 20
Max entre 10 et 20 : 20

Attention : Operation entre 10 et 20
```

Les fonctions

L'externalisation des fonctions

```
$ cat $HOME/rep/mes_fonctions
function affichage () {
       echo '**********'
}
function somme () {
       echo $(( $1 + $2 ))
}

$ cat mon_script
#!/bin/bash

. $HOME/rep/mes_fonctions     ou     source $HOME/rep/mes_fonctions

affichage
echo « Somme de 10 et 30 : $(somme 10 30) »
affichage

$ ./mon_script
**********
Somme de 10 et 30 : 40
**********
```

L'externalisation des fonctions

Lorsque qu'une ou plusieurs fonctions peuvent être utilisées par différents programmes, il est possible de les externaliser. Il suffit de les regrouper au sein d'un fichier qui sera exécuté au sein du script principal.

Pour que les fonctions soient déclarées dans le programme principal, et donc reconnues, le fichier des fonctions doit être exécuté au sein du shell père avec les syntaxes '. fichier_fonctions' ou 'source fichier_fonctions'.

```
$ cat $HOME/rep/mes_fonctions
function affichage () {
       echo '**********'
}
function somme () {
       echo $(( $1 + $2 ))
}

$ cat mon_script
#!/bin/bash

. $HOME/rep/mes_fonctions                    # ou   source $HOME/rep/mes_fonctions

affichage
echo « Somme de 10 et 30 : $(somme 10 30) »
affichage

$ ./mon_script
**********
Somme de 10 et 30 : 40
**********
```

Les fonctions

L'externalisation en ksh : FPATH et autoload

```
$ cat $HOME/rep_fonctions/affichage
function affichage () {
      echo '**********'
}
```

```
$ cat $HOME/rep_fonctions/somme
function somme () {
     echo $(( $1 + $2 ))
}
```

```
$ cat mon_script
#!/bin/ksh

FPATH=${FPATH}:$HOME/rep_fonctions

autoload affichage                ou    autoload affichage somme
autoload somme

affichage
echo « Somme de 10 et 30 : $(somme 10 30) »
affichage

$ ./mon_script
**********
Somme de 10 et 30 : 40
**********
```

L'externalisation en ksh : FPATH et autoload

L'externalisation des fonctions en ksh, peuvent se faire par '. fichier_fonctions'.

```
$ cat mon_script
#!/bin/ksh

… commandes …

. $HOME/rep/mes_fonctions              ou     . ./mes_fonctions

affichage
echo « Somme de 10 et 30 : $(somme 10 30) »
```

Il est possible de créer au sein d'un répertoire un fichier par fonction. Le fichier doit avoir le même nom que la fonction. La variable FPATH doit contenir le chemin pointant sur ce répertoire.
Au sein d'un script, on peut utiliser une de ces fonctions, à condition d'avoir été déclarée (ou chargée) via l'instruction 'autoload'.

```
$ cat $HOME/rep_fonctions/affichage
function affichage () {
      echo '**********'
}

$ cat $HOME/rep_fonctions/somme
function somme () {
      echo $(( $1 + $2 ))
}
```

```
$ cat mon_script
#!/bin/ksh

FPATH=${FPATH}:$HOME/rep_fonctions

autoload  affichage                          ou     autoload  affichage  somme
autoload  somme

affichage
echo « Somme de 10 et 30 : $(somme 10 30) »
affichage
```

```
$ ./mon_script
**********
Somme de 10 et 30 : 40
**********
```

Notes

Les expressions régulières et les commandes grep

caractères ainsi que les commandes «grep», «fgrep» et «egrep».

Les expressions régulières et les commandes grep

- La commande grep
- Les expressions régulières
- Les expressions régulières : compléments
- Les commandes fgrep et egrep
- Les expressions régulières étendues

Les expressions régulières et les commandes grep

La commande grep

- Introduction
- grep 'chaîne_de_caractères' nom_du_fichier
- Options

  -i -v -l -c -q

La commande grep

Introduction

Cette commande permet de filtrer le contenu d'un fichier.
Elle permet donc d'afficher toute les lignes qui contiennent la chaîne de caractère au sein d'un fichier.

Syntaxe : grep 'chaîne_de_caractères' nom_du_fichier

Exemples :

```
$ grep 'root' /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

```
$ grep 'root' /etc/passwd /etc/group
/etc/passwd:root:x:0:0:root:/root:/bin/bash
/etc/passwd:operator:x:11:0:operator:/root:/sbin/nologin
/etc/group:root:x:0:
```

```
$ ps -ef | grep bash
root       706     1  0 09:38 ?        00:00:00 /bin/bash /usr/sbin/ksmtuned
user1     2761  2609  0 09:39 ?        00:00:00 /usr/bin/ssh-agent /bin/sh -cexec
-l /bin/bash -c "env GNOME_SHELL_SESSION_MODE=classic gnome-session --session
gnome-classic"
user1     3298  3285  0 09:39 pts/0    00:00:00 -bash
user1     3395  3298  0 09:42 pts/0    00:00:00 grep --color=auto bash
```

Les options de «grep»

-i : pour ignorer la casse.

Exemple :

```
$ grep -i 're' /etc/passwd
unbound:x:998:996:Unbound DNS resolver:/etc/unbound:/sbin/nologin
rtkit:x:172:172:RealtimeKit:/proc:/sbin/nologin
```

-v : pour récupérer les lignes qui ne contiennent pas la chaîne de caractères.

Exemple :

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
```

```
$ grep -v 'nologin' /etc/passwd
root:x:0:0:root:/root:/bin/bash
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
user1:x:1000:1003:user1:/home/user1:/bin/bash
user2:x:1001:1004::/home/user2:/bin/bash
```

-l : pour afficher que la liste des fichiers qui contiennent la chaîne de caractères.

Exemple :

```
$ grep -l 'mount' /etc/init.d/*
/etc/init.d/functions
/etc/init.d/network
```

-c : indique le nombre de lignes trouvées.

Exemple :

```
$ grep -c 'root' /etc/passwd
2
```

-q : mode silencieux, n'écrit rien sur la sortie standard.
Pratique pour la programmation (scripts shells).

Exemple :

```
$ grep -q 'root' /etc/passwd
$
```

Les expressions régulières et les commandes grep

Les expressions régulières

- Liste de caractères [abc]
- Intervalle de caractères [a-z] [A-Z] [a-zA-Z] [0-9]
- Exclusion de liste [^abc] [^a-z]
- Un caractère quelconque .
- 0 à n fois le caractère précédent *
- marqueur de début de ligne ^
- marqueur de fin de ligne $
- marqueur de début de mot \<
- marqueur de fin de mot \>

Les expressions régulières

Introduction

Les expressions régulières sont les caractères spéciaux pour les chaînes de caractères.

Liste de caractères [abc]

Représente un caractère parmi la liste de ceux spécifiés entre les crochets «**[**» et «**]**» .

Exemple :

```
$ grep      'r[aio]'      /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
libstoragemgmt:x:995:994:daemon account for
libstoragemgmt:/var/run/lsm:/sbin/nologin
rtkit:x:172:172:RealtimeKit:/proc:/sbin/nologin
radvd:x:75:75:radvd user:/:/sbin/nologin
chrony:x:994:993::/var/lib/chrony:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
```

Dans l'exemple ci-dessus, nous recherchons les lignes contenant un caractère «**r**» suivi d'un caractère a ou i ou o.

Intervalle de caractères [a-z]

L'utilisation du «-» permet de définir une liste de caractères.

[a-z] : un caractère en minuscule, de «**a**» à «**z**».

Exemple :

```
$ grep '[a-z]' /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
```

[A-Z] : un caractère en majuscule, de «**A**» à «**Z**».

Exemple :

```
$ grep '[A-Z]' /etc/passwd
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
dbus:x:81:81:System message bus:/:/sbin/nologin
polkitd:x:999:998:User for polkitd:/:/sbin/nologin
```

[a-zA-Z]: un caractère en minuscule ou majuscule.

Exemple :

```
$ grep '[a-zA-Z]' /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
```

[0-9] : un chiffre de **0** à **9**.

Exemple :

```
$ grep '[0-9]' /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
```

[^abc] : un caractère exclu de la liste.

Exemple :

```
$ grep '[^abc]' /etc/passwd
```

[^a-z] : un caractère exclu de l'intervalle de caractères.

Exemple :

```
$ grep '[^a-z]' /etc/passwd
```

Un caractère quelconque .

La caractère «**.**» représente un caractère quelconque.

Exemple :

```
$ grep 'r..t' /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
```

L'exemple ci-dessus filtre les lignes contenant la chaîne de caractères «**r**», suivi de 2 caractères quelconques, suivi de «**t**».

**La closure *

Le caractère «*****» est la closure, il représente 0 à n fois le caractère précédent.

Exemple :

```
$ grep 'ro*t' /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
abrt:x:173:173::/etc/abrt:/sbin/nologin
rtkit:x:172:172:RealtimeKit:/proc:/sbin/nologin
```

L'exemple ci-dessus filtre les lignes contenant la chaîne de caractères «**r**», suivi de 0 à n fois le caractère «**o**», suivi de «**t**». C'est à dire rt, rot, root, roooooot, ...

Le marqueur de début de ligne ^

Le caractère «**^**» est le marqueur de début de ligne, défini «les lignes qui commencent par».

Exemple :

```
$ grep '^u' /etc/passwd
unbound:x:998:996:Unbound DNS resolver:/etc/unbound:/sbin/nologin
usbmuxd:x:113:113:usbmuxd user:/:/sbin/nologin
user1:x:1000:1003:user1:/home/user1:/bin/bash
user2:x:1001:1004::/home/user2:/bin/bash
user3:x:1002:1005::/home/user3:/bin/bash
```

Le marqueur de fin de ligne $

Le caractère «**$**» est le marqueur de fin de ligne, défini «les lignes qui finissent par».

Exemple :

```
$ grep 'nologin$' /etc/passwd
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
```

Le marqueur de début de mot \<

L'ensemble de caractères «**\<**» représentent les lignes dans lesquelles des mots commencent par la chaîne de caractères.

Exemple :

```
$ grep '\<bin' /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
user1:x:1000:1003:user1:/home/user1:/bin/bash
user2:x:1001:1004::/home/user2:/bin/bash
```

Le marqueur de fin de mot \>

L'ensemble de caractères «**\>**» représentent les lignes dans lesquelles des mots se finissent par la chaîne de caractères.

Exemple :

```
$ grep 'mu\>' /etc/passwd
qemu:x:107:107:qemu user:/:/sbin/nologin
```

Les expressions régulières et les commandes grep

Les expressions régulières : compléments

- n fois le caractère précédent \{n\}
- au moins n fois le caractère précédent \{n,\}
- entre n et p fois le caractère précédent \{n,p\}
- mémorisation d'une expression \(...\)
- réutilisation d'une expression mémorisée \n

```
$ tr -s '\t' < fichier_source \
  | sed 's/\([a-zA-Z]*\)\t\([a-zA-Z]*\)\t\([0-9]*\)/\3:\2_\1/'
```

Les expressions régulières : compléments

Exemples

```
$ cat fic_source
aaaa bbbbb ccccc
aaa bbb ccc
aa bb cc
aaabbbcccddd
```

\{n\} **n fois le caractère précédent**

ainsi 'a\{3\}' ne concerne que : aaa

```
$ grep 'a\{3\}.\{3\}c' fic_source
aaabbbcccddd
```

\{n,\} **au moins n fois le caractère précédent**

ainsi 'a\{3,\}' concerne : aaa, aaaa, aaaaa, aaaaaa, etc

```
$ grep 'a\{3,\}.\{3,\}c' fic_source
aaaa bbbbb ccccc
aaa bbb ccc
aaabbbcccddd
```

\{n,p\} **entre n et p fois le caractère précédent**

ainsi 'a\{3,5\}' ne concerne que : aaa, aaaa, aaaaa

```
$ grep 'a\{3,5\}.\{3,6\}c' fic_source
aaa bbb ccc
aaabbbcccddd
```

\(...\) mémorisation et réutilisation d'une expression

Les parenthèses anti-slashés permettent de mémoriser dans l'ordre d'apparition le contenu, ré-exploitable par \1, \2, etc.

```
$ cat fichier_source
Linus   Thorvald        123456789
Richard Stallman        01564
Denis   Ritchie         98
```

```
$ cat -etv fichier_source
Linus^IThorvald^I123456789$
Richard^IStallman^I01564$
Denis^IRitchie^I^I98$
```

```
$ tr -s '\t' < fichier_source \
  | sed 's/\([a-zA-Z]*\)\t\([a-zA-Z]*\)\t\([0-9]*\)/\3:\2_\1/'
123456789:Thorvald_Linus
01564:Stallman_Richard
98:Ritchie_Denis
```

Les expressions régulières et les commandes grep

Les commandes fgrep et egrep

- Commande «fgrep»

```
$ fgrep    '.gif'       fic
```

- Commande «egrep»

```
$ egrep    'var|user'   /etc/passwd
```

```
$ grep     'var'    /etc/passwd   |   grep   'user'
```

Les commandes fgrep et egrep

La commande «fgrep»

Cette commande n'interprète pas les caractères spéciaux.

Exemple :

```
$ grep      '.gif'      fic
```

Ci-dessus avec grep, le «**.**» représente le caractère spécial définissant un caractère quelconque.

```
$ fgrep     '.gif'      fic
```

Ci-dessus avec fgrep, le «**.**» est inhibé et représente donc le simple caractère «**.**». La commande affichera toutes les lignes du fichier «**fic**» qui contiennent la chaîne de caractères «**.gif**».

La commande «egrep»

Cette commande est plus puissante que la commande «**grep**» mais elle est moins utilisée. La commande «**egrep**» supporte des fonctionnalités et caractères spéciaux supplémentaires.

Par exemple, le caractère «**|**» représente un «**OU logique**» pour deux chaînes de caractères.

Exemple :

```
$ egrep      'var|user'      /etc/passwd
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
usbmuxd:x:113:113:usbmuxd user:/:/sbin/nologin
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/sbin/nologin
```

Dans l'exemple ci-dessus, la caractère «**|**» (pipe) signifie un «**OU logique**». Cette commande affichera toute les lignes qui contiennent «**var**» OU «**user**».

Remarque complémentaire

Pour réaliser un «**ET logique**», il faut simplement utiliser le pipe.

Exemple :

```
$ grep    'var'    /etc/passwd    |    grep    'user'
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/sbin/nologin
```

Notes

La commande sed

Dans ce chapitre nous allons nous traiter les syntaxes de la commande sed.

La commande sed

- Les bases
- Les options d, p et w
- L'insertion
- Les compléments
- Quelques cas

La commande sed

Les bases

```
sed   'action'   fichier
sed   's/ancien/nouveau/g'   fichier

sed   'zone_de_traitement   action'   fichier
      '5                    action'
      '5,10                 action'          1,$
      '/filtre/             action'
      '/debut/,/fin/        action'
```

Les bases

Présentation

La commande «**sed**» réalise les opérations d'édition de texte de manière non interactive :
Substitution de chaînes de caractères, suppression de lignes, insertion de texte…

Par défaut, «**sed**» n'est pas destructrice du fichier d'origine car le résultat de cette commande est uniquement envoyé vers la sortie standard.

Syntaxe : $ sed 'action' fichier

La substitution

action = s/ancienne_chaîne/nouvelle_chaîne/

Substitue l'ancienne chaîne de caractères par la nouvelle chaîne de caractères. Cette modification est réalisée uniquement pour la première occurrence sur la ligne.

action = s/ancienne_chaîne/nouvelle_chaîne/g

Comme précédemment mais pour toutes les occurrences sur la ligne.

Exemple :

```
$ cat       fichier
Votre premiere commande :
Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
Oranges 40 dollars Mures 50 dollars Framboises 60 dollars
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
Prunes 40 dollars Oranges 50 dollars Poires 60 dollars
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
Mures 40 dollars Fraises 50 dollars Oranges 60 dollars
Groseilles 40 dollars Pommes 50 dollars Raisins 60 dollars
Oranges 40 dollars Mandarines 50 dollars Clementines 60 dollars
Framboises 40 dollars Raisins 50 dollars Mures 60 dollars
Fin de la commande


$ sed       's/dollar/euro/'       fichier
Votre premiere commande :
Pommes 10 euros Cerises 20 dollars Fraises 30 dollars
Oranges 40 euros Mures 50 dollars Framboises 60 dollars
Mangues 40 euros Ananas 50 dollars Bananes 60 dollars
Prunes 40 euros Oranges 50 dollars Poires 60 dollars
Votre deuxième commande :
Fraises 40 euros Mirabelles 50 dollars Peches 60 dollars
Mures 40 euros Fraises 50 dollars Oranges 60 dollars
Groseilles 40 euros Pommes 50 dollars Raisins 60 dollars
Oranges 40 euros Mandarines 50 dollars Clementines 60 dollars
Framboises 40 euros Raisins 50 dollars Mures 60 dollars
Fin de la commande


$ sed       's/dollars/euros/g'       fichier
Votre premiere commande :
Pommes 10 euros Cerises 20 euros Fraises 30 euros
Oranges 40 euros Mures 50 euros Framboises 60 euros
Mangues 40 euros Ananas 50 euros Bananes 60 euros
Prunes 40 euros Oranges 50 euros Poires 60 euros
Votre deuxième commande :
Fraises 40 euros Mirabelles 50 euros Peches 60 euros
Mures 40 euros Fraises 50 euros Oranges 60 euros
Groseilles 40 euros Pommes 50 euros Raisins 60 euros
Oranges 40 euros Mandarines 50 euros Clementines 60 euros
Framboises 40 euros Raisins 50 euros Mures 60 euros
Fin de la commande
```

Zone de traitement

Il est possible d'insérer une zone de traitement avant d'effectuer l'action. Cela permet de limiter les lignes du fichier qui subiront l'action.

Syntaxe : $ sed 'zone_de_traitement action' fichier

numéro_de_ligne

L'action est appliquée que sur une seule ligne du fichier.

Exemple :

```
$ sed       '3s/Oranges/KIWIS/'       fichier
1     Votre premiere commande :
2     Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
3     KIWIS 40 dollars Mures 50 dollars Framboises 60 dollars
4     Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
5     Prunes 40 dollars Oranges 50 dollars Poires 60 dollars
6     Votre deuxième commande :
7     Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
8     Mures 40 dollars Fraises 50 dollars Oranges 60 dollars
9     Groseilles 40 dollars Pommes 50 dollars Raisins 60 dollars
10    Oranges 40 dollars Mandarines 50 dollars Clementines 60 dollars
11    Framboises 40 dollars Raisins 50 dollars Mures 60 dollars
12    Fin de la commande
```

Dans l'exemple ci-dessus, nous avons attribué le numéro «**3**» à la zone de traitement puis une action de substitution.
Ce qui entraîne la substitution du terme «**Oranges**» par «**KIWIS**» uniquement sur la 3ème ligne.

numéro_de_ligne_de_début,numéro_de_ligne_de_fin

L'action est appliquée sur un ensemble de ligne, depuis le numéro de ligne de début jusqu'au numéro de ligne de fin.

Exemple :

```
$ sed       '4,8 s/Oranges/KIWIS/'       fichier
1     Votre premiere commande :
2     Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
3     Oranges 40 dollars Mures 50 dollars Framboises 60 dollars
4     Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
5     Prunes 40 dollars KIWIS 50 dollars Poires 60 dollars
6     Votre deuxième commande :
7     Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
8     Mures 40 dollars Fraises 50 dollars KIWIS 60 dollars
9     Groseilles 40 dollars Pommes 50 dollars Raisins 60 dollars
10    Oranges 40 dollars Mandarines 50 dollars Clementines 60 dollars
11    Framboises 40 dollars Raisins 50 dollars Mures 60 dollars
12    Fin de la commande
```

Dans l'exemple ci-dessus, nous avons attribué les lignes de «**4**» à «**8**» à la zone de traitement puis une action de substitution.
Ce qui entraîne la substitution du terme «**Oranges**» par «**KIWIS**» à partir de la ligne «**4**» jusqu'à la ligne «**8**».

/chaîne_de_caractères/

L'action est appliquée sur la ligne contenant la chaîne de caractères précisée (un filtre).

Exemple :

```
$ sed       '/Oranges/s/Mures/KIWIS/'       fichier
Votre premiere commande :
Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
Oranges 40 dollars KIWIS 50 dollars Framboises 60 dollars
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
Prunes 40 dollars Oranges 50 dollars Poires 60 dollars
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
KIWIS 40 dollars Fraises 50 dollars Oranges 60 dollars
Groseilles 40 dollars Pommes 50 dollars Raisins 60 dollars
Oranges 40 dollars Mandarines 50 dollars Clementines 60 dollars
Framboises 40 dollars Raisins 50 dollars Mures 60 dollars
Fin de la commande
```

Dans l'exemple ci-dessus, nous avons attribué le filtre «**Oranges**» à la zone de traitement puis une action de substitution du terme «**Mures**» par «**KIWIS**».
Ce qui entraîne la substitution du terme «**Mures**» par «**KIWIS**» uniquement sur les lignes qui contiennent le mot «**Oranges**».

/chaîne_de_début/,/chaîne_de_fin/

L'action est appliquée sur une zone délimitée par deux chaînes de caractères, de la ligne contenant la chaîne de début jusqu'à la ligne contenant la chaîne de fin.

Exemple :

```
$ sed '/Pommes/,/Poires/s/0/5/g' fichier
Votre premiere commande :
Pommes 15 dollars Cerises 25 dollars Fraises 35 dollars
Oranges 45 dollars Mures 55 dollars Framboises 65 dollars
Mangues 45 dollars Ananas 55 dollars Bananes 65 dollars
Prunes 45 dollars Oranges 55 dollars Poires 65 dollars
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
Mures 40 dollars Fraises 50 dollars Oranges 60 dollars
Groseilles 45 dollars Pommes 55 dollars Raisins 65 dollars
Oranges 45 dollars Mandarines 55 dollars Clementines 65 dollars
Framboises 45 dollars Raisins 55 dollars Mures 65 dollars
Fin de la commande
```

Dans l'exemple ci-dessus, nous avons attribué un filtre de début «**Pommes**» et un filtre de fin «**Poires**» à la zone de traitement, puis une action de substitution du caractère «**0**» par «**5**».
Ce qui entraîne la substitution des «**0**» par des «**5**» dans l'intervalle des lignes qui contiennent le terme «**Pommes**» à «**Poires**».

La commande sed

Les options d, p et w

d suppression de lignes

'5d'

'5,10d'

'/filtre/d'

'/début/,/fin/d'

sed -n '/Pommes/p' fichier

sed -n '/Pommes/w newfic' fichier

Les options d, p et w

La suppression «d»

Le caractère «**d**» (delete) permet de supprimer une ligne.

Syntaxe : $ sed 'filtre d' fichier

Exemples :

```
$ sed '5d' fichier
Votre premiere commande :
Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
Oranges 40 dollars Mures 50 dollars Framboises 60 dollars
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
Votre deuxième commande :
Etc…
```

Entraîne la suppression de la ligne «**5**» du fichier.

```
$ sed '2,11d' fichier
Votre premiere commande :
Fin de la commande
```

Entraîne la suppression des lignes «**2**» à «**11**» du fichier.

```
$ sed       '/Pommes/d'       fichier
Votre premiere commande :
Oranges 40 dollars Mures 50 dollars Framboises 60 dollars
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
Prunes 40 dollars Oranges 50 dollars Poires 60 dollars
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
Mures 40 dollars Fraises 50 dollars Oranges 60 dollars
Oranges 40 dollars Mandarines 50 dollars Clementines 60 dollars
Framboises 40 dollars Raisins 50 dollars Mures 60 dollars
Fin de la commande
```

Entraîne la suppression des lignes contenant le terme «**Pommes**» du fichier.

```
$ sed       '/Pommes/,/Poires/d'       fichier
Votre premiere commande :
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
Mures 40 dollars Fraises 50 dollars Oranges 60 dollars
```

Entraîne la suppression des lignes contenant le terme «**Pommes**» jusqu'à la ligne contenant le terme «**Poires**».

Le print «p»

Le caractère «**p**» permet d'afficher les lignes qui contiennent le terme précisé dans le filtre.
L'option «**-n**» permet de ne pas afficher le reste du fichier.

Syntaxe : $ sed -n '/filtre/p' fichier

Exemple :

```
$ sed       -n       '/Pommes/p'       fichier
Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
Groseilles 40 dollars Pommes 50 dollars Raisins 60 dollars
```

Affiche uniquement les deux lignes du fichier contenant le terme «**Pommes**».

Le write «w»

Le caractère «**w**» permet de créer un nouveau fichier contenant le résultat de la commande.

Syntaxe : $ sed -n '/filtre/w nouveau_fichier' fichier

Exemple :

```
$ sed       -n       '/Pommes/w    newfic'       fichier
```

Un fichier nommé «**newfic**» à été créé avec toutes les lignes contenant «**Pommes**».

La commande sed

L'insertion

```
sed          '/filtre/i\
> nouveau_texte'        fichier

sed          '/filtre/a\
> nouveau_texte'        fichier

sed          '/filtre/c\
> nouveau_texte'        fichier

sed          '/filtre/r  autre_fichier'        fichier
```

L'insertion

Cette commande nous permet d'insérer du texte avant ou après une ligne, ou insérer le contenu d'un autre fichier.
Il peut aussi remplacer le texte initial par un nouveau.

Utilisation du code «i»

La caractère d'insertion «**i**» (insert) permet d'insérer du texte avant la ligne définie par le filtre.
Dans l'exemple ci-dessous, nous avons inséré du texte avant chaque ligne contenant «**Oranges**».

Exemples :

```
$ sed       '/Oranges/i\
> un nouveau texte'       fichier
Votre premiere commande :
Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
un nouveau texte
Oranges 40 dollars Mures 50 dollars Framboises 60 dollars
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
un nouveau texte
Prunes 40 dollars Oranges 50 dollars Poires 60 dollars
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
un nouveau texte
Mures 40 dollars Fraises 50 dollars Oranges 60 dollars
Groseilles 40 dollars Pommes 50 dollars Raisins 60 dollars
un nouveau texte
Oranges 40 dollars Mandarines 50 dollars Clementines 60 dollars
Framboises 40 dollars Raisins 50 dollars Mures 60 dollars
Fin de la commande
```

Utilisation du code «a»

La caractère d'insertion «**a**» (append) permet d'ajouter du texte après la ligne définie par le filtre.
Dans l'exemple ci-dessous, nous avons inséré du texte après chaque ligne contenant «**Oranges**».

Exemples :

```
$ sed       '/Oranges/a\
un nouveau texte'       fichier
Votre premiere commande :
Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
Oranges 40 dollars Mures 50 dollars Framboises 60 dollars
un nouveau texte
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
Prunes 40 dollars Oranges 50 dollars Poires 60 dollars
un nouveau texte
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
Mures 40 dollars Fraises 50 dollars Oranges 60 dollars
un nouveau texte
Groseilles 40 dollars Pommes 50 dollars Raisins 60 dollars
Oranges 40 dollars Mandarines 50 dollars Clementines 60 dollars
un nouveau texte
Framboises 40 dollars Raisins 50 dollars Mures 60 dollars
Fin de la commande
```

Utilisation du code «c»

La caractère d'insertion «**c**» (change) permet de remplacer la ligne définie par le filtre.
Dans l'exemple ci-dessous, les lignes contenant le terme «**Oranges**» ont été remplacé par «**un nouveau texte**».

Exemples :

```
$ sed       '/Oranges/c\
un nouveau texte'       fichier
Votre premiere commande :
Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
un nouveau texte
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
un nouveau texte
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
un nouveau texte
Groseilles 40 dollars Pommes 50 dollars Raisins 60 dollars
un nouveau texte
Framboises 40 dollars Raisins 50 dollars Mures 60 dollars
Fin de la commande
```

Utilisation du code «r»

La caractère d'insertion «**r**» permet de rajouter le contenu d'un fichier après la ligne définie par le filtre.
Dans l'exemple ci-dessous, nous avons rajouté le contenu du fichier «**autrefichier**» à la suite des lignes contenant le terme «**Pommes**».

Exemples :

```
$ cat       autrefichier
et voici un nouveau fichier
un texte quelconque
sur 3 lignes

$ sed       '/Pommes/r      autrefichier'       fichier
Votre premiere commande :
Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
et voici un nouveau fichier
un texte quelconque
sur 3 lignes
Oranges 40 dollars Mures 50 dollars Framboises 60 dollars
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
Prunes 40 dollars Oranges 50 dollars Poires 60 dollars
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
Mures 40 dollars Fraises 50 dollars Oranges 60 dollars
Groseilles 40 dollars Pommes 50 dollars Raisins 60 dollars
et voici un nouveau fichier
un texte quelconque
sur 3 lignes
Oranges 40 dollars Mandarines 50 dollars Clementines 60 dollars
Framboises 40 dollars Raisins 50 dollars Mures 60 dollars
Fin de la commande
```

La commande sed

Les compléments

-e

```
sed  –e 's/Pommes/Bananes/g' –e 's/Poires/Bananes/' –e '/Mures/d'  fic
```

-f

```
cat      fic_actions
s/Pommes/Bananes/g
s/Poires/Bananes/
/Mures/d

sed      –f  fic_actions      fichier
```

Remplacement du caractère spécial /

Les compléments

Option «-e»

L'option «**-e**» permet de cumuler plusieurs actions sur une même ligne de commande.

Exemple :

```
$ sed –e 's/Pommes/Bananes/g' –e 's/Poires/Bananes/' –e '/Mures/d'
fichier
Votre premiere commande :
Bananes 10 dollars Cerises 20 dollars Fraises 30 dollars
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
Prunes 40 dollars Oranges 50 dollars Bananes 60 dollars
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
Groseilles 40 dollars Bananes 50 dollars Raisins 60 dollars
Oranges 40 dollars Mandarines 50 dollars Clementines 60 dollars
Fin de la commande
```

Dans l'exemple ci-dessus, nous avons remplacé les «**Pommes**» et les «**Poires**» par des «**Bananes**», et nous avons supprimé les lignes contenant «**Mures**» du fichier.

Option «-f»

L'argument suivant l'option «**-f**» indique un fichier où se trouvent les actions qui doivent être appliquées à la commande sed.

Syntaxe : $ sed -f fichier_des_actions fichier

Exemple :

```
$ cat       fic_actions
s/Pommes/Bananes/g
s/Poires/Bananes/
/Mures/d

$ sed       -f  fic_actions       fichier
Votre premiere commande :
Bananes 10 dollars Cerises 20 dollars Fraises 30 dollars
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
Prunes 40 dollars Oranges 50 dollars Bananes 60 dollars
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
Groseilles 40 dollars Bananes 50 dollars Raisins 60 dollars
Oranges 40 dollars Mandarines 50 dollars Clementines 60 dollars
Fin de la commande
```

Remplacement du caractère spécial «/»

Pour remplacer le caractère spécial «**/**» que nous avons communément l'habitude d'utiliser, nous pouvons en réalité utiliser n'importe qu'elle autre caractère. Ceci est tout particulièrement utile pour éviter un conflit dans la ligne de commande.

Exemple :

```
$ cat      fichier2
/home/user1/rep1
/home/user1/repertoire
/home/user222/fic
/home/user1/fichier

$ sed       's?/?|?g'       fichier2
|home|user1|rep1
|home|user1|repertoire
|home|user222|fic
|home|user1|fichier
```

Dans l'exemple ci-dessus, nous avons remplacer le caractère spécial «**/**» par le «**?**», ce qui nous a permis d'obtenir une syntaxe simple pour la substitution de «**/**» par «**|**».

La commande sed

Quelques cas

```
sed    's/Oranges//'    fichier

sed    's/^Oranges/Mures/'    fichier

sed    's/\//|/g'    fichier

find   /   -name vi   2>   /dev/null    |    sed    's/\//|/g'

sed    '/^[   \t]*$/d'    fichier
```

Quelques cas

sed 's/Oranges//' fichier

```
$ sed       's/Oranges//'       fichier
Votre premiere commande :
Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
 40 dollars Mures 50 dollars Framboises 60 dollars
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
Prunes 40 dollars  50 dollars Poires 60 dollars
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
Mures 40 dollars Fraises 50 dollars  60 dollars
Groseilles 40 dollars Pommes 50 dollars Raisins 60 dollars
 40 dollars Mandarines 50 dollars Clementines 60 dollars
Framboises 40 dollars Raisins 50 dollars Mures 60 dollars
Fin de la commande
```

On constate que la section de «**la nouvelle chaîne**» est vide ou tout simplement inexistante.
L'instruction «**sed**» permet dans cet exemple de supprimer la chaîne de caractères «**Oranges**».

sed 's/^Oranges/Mures/' fichier

```
$ sed       's/^Oranges/Mures/'       fichier
Votre premiere commande :
Pommes 10 dollars Cerises 20 dollars Fraises 30 dollars
Mures 40 dollars Mures 50 dollars Framboises 60 dollars
Mangues 40 dollars Ananas 50 dollars Bananes 60 dollars
Prunes 40 dollars Oranges 50 dollars Poires 60 dollars
Votre deuxième commande :
Fraises 40 dollars Mirabelles 50 dollars Peches 60 dollars
Mures 40 dollars Fraises 50 dollars Oranges 60 dollars
Groseilles 40 dollars Pommes 50 dollars Raisins 60 dollars
Mures 40 dollars Mandarines 50 dollars Clementines 60 dollars
Framboises 40 dollars Raisins 50 dollars Mures 60 dollars
Fin de la commande
```

Nous pouvons utiliser les expressions régulières. L'exemple ci-dessus substitue le mot «**Oranges**» en début de ligne par «**Mures**».

sed 's/\//|/g' fichier

```
$ cat       fichier2
/home/user1/rep1
/home/user1/repertoire
/home/usre222/fic
/home/user1/fichier

$ sed       's/\//|/g'       fichier2
|home|user1|rep1
|home|user1|repertoire
|home|usre222|fic
|home|user1|fichier
```

comme précédemment, le «****» est le caractère spécial inhibant le caractère suivant. Ainsi, les caractères «**/**» du fichier sont remplacé par «**|**».

commande | sed 'action'

```
$ find  /  -name vi  2>  /dev/null   |   sed  's/\//|/g'
|usr|bin|vi
|usr|share|locale|vi
|usr|share|help|vi
|usr|share|vim|vim74|lang|vi
|usr|share|espeak-data|voices|asia|vi
```

L'utilisation de la commande «**sed**» est pratique avec le pipe pour modifier l'affichage du résultat.

sed '/^[\t]*$/d' fichier

```
$ cat       fichier3
Bonjour,

ceci est nouveau

puis

suite

fin

$ cat       -evt       fichier3
Bonjour,$
$
ceci est nouveau$
  $
puis$
  ^I^I^I^I$
suite$
^I^I$
fin$

$ sed       '/^[ \t]*$/d'       fichier3
Bonjour,
ceci est nouveau
puis
suite
fin
```

Cette commande supprime les lignes blanches. C'est à dire les lignes vides ou contenant une combinaison du caractères espaces et/ou de tabulations.

[\t] : liste de caractères, ici le caractère espace ou le caractère de tabulation (\t).

[\t]* : * est la «**closure**» pour définir 0 à n fois le caractère précédent. Ici l'espace ou la tabulation. 0 fois représente une ligne vide, n fois représente 0 à n espaces éventuellement combinés avec 0 à n tabulations.

^xxx$ sont respectivement les caractères spéciaux de marqueurs de début de ligne et de fin de ligne. Ainsi la ligne est exclusivement constitué de descriptif 'xxx'.
Dans notre exemple cela correspond aux lignes dites blanches.

Notes

La commande awk

Dans ce chapitre, nous allons nous traiter les syntaxes de la commande awk.

La commande awk

- Les bases, les options
- Les filtres
- L'option -f
- Les variables internes
- Les blocs BEGIN et END
- Les opérations arithmétiques

La commande awk

Les bases, les options

awk '{print $1}' /etc/hosts

awk -F: '{print $3, $1}' /etc/passwd

awk -F: '{print "Uid = "$3" Login = " $1}' /etc/passwd

Les bases, les options

La commande awk (nawk, gawk) permet de manipuler le contenu d'un fichier en exploitant les champs. Cette instruction peut avoir une syntaxe très complexe car elle intègre toutes les fonctionnalités des scripts.

Syntaxe : awk [-options] 'actions' fichier

Les bases

```
$ awk '{print $1}' /etc/hosts
127.0.0.1
::1
```

La section entre **{ }** définit la liste des actions à réaliser sur chaque ligne du fichier.
La référence à un champ est indiquée par le caractère **$** suivi du numéro de champ, les caractères 'espace' et 'tabulation' sont les caractères séparateurs de champs par défaut.
Ainsi l'exemple ci-dessus affiche le premier champ du fichier /etc/hosts.

```
$ awk       -F:       '{print $3, $1}'       /etc/passwd
0 root
1 bin
2 daemon
3 adm
4 lp
5 sync
Etc..
```

L'option **-F** redéfinie le caractère séparateur de champs (ici le «**:**»).
L'exemple affiche le champ 3 puis le champ 1 séparés par un caractère «**espace**». Ce dernier est présent à cause du caractère spécial «**,**».

```
$ awk    -F:     '{print "Uid = "$3" Login = " $1}'     /etc/passwd
Uid = 0 Login = root
Uid = 1 Login = bin
Uid = 2 Login = daemon
Uid = 3 Login = adm
Uid = 4 Login = lp
Uid = 5 Login = sync
Etc..
```

Il est possible d'agrémenter l'affichage avec du texte, il suffit de l'écrire entre guillemets.

La commande awk

Les filtres

awk -F: '/root/{print "Login "$1" Ligne : "$0}' /etc/passwd

awk -F: '/^root/{print "Login "$1" Ligne : "$0}' /etc/passwd

awk -F: '($3 > 99){print "Utilisateur Ligne : "$0}' /etc/passwd

ou

awk -F: '{if ($3 > 99) print "Utilisateur Ligne : "$0}' /etc/passwd

Les filtres

Filtres

Syntaxe : awk '**/filtre/**{action}' fichier

```
$ awk -F: '/root/{print "Login "$1" Ligne : "$0}' /etc/passwd
Login root Ligne : root:x:0:0:root:/root:/bin/bash
Login operator Ligne : operator:x:11:0:operator:/root:/sbin/nologin
```

Le filtre permet de sélectionner les lignes qui vont subir l'action. Il est utilisé le «**/**» pour définir une chaîne de caractères.
L'exemple ci-dessus applique l'action «**print**» pour toutes les lignes qui contiennent la chaîne de caractères «**root**».
L'expression «**$0**» représente la ligne complète.

```
$ awk -F: '/^root/{print "Login "$1" Ligne : "$0}' /etc/passwd
Login root Ligne : root:x:0:0:root:/root:/bin/bash
```

L'utilisation des expressions régulières est autorisée.

Syntaxe : awk '**(test)** {action}' fichier

ou

awk '{ **if(test) action**}' fichier

```
$ awk  -F: '($3 > 99){print "Utilisateur Ligne : "$0}'     /etc/passwd

ou

$ awk  -F: '{if ($3 > 99) print "Utilisateur Ligne : "$0}' /etc/passwd

      Utilisateur Ligne : polkitd:x:999:998:User for polkitd:/:/sbin/nologin
      Utilisateur Ligne : abrt:x:173:173::/etc/abrt:/sbin/nologin
      Utilisateur Ligne : colord:x:997:995:User for colord:/var/lib/colord:/sbin/nologin
      Utilisateur Ligne : usbmuxd:x:113:113:usbmuxd user:/:/sbin/nologin
      Utilisateur Ligne : qemu:x:107:107:qemu user:/:/sbin/nologin
      Utilisateur Ligne : rtkit:x:172:172:RealtimeKit:/proc:/sbin/nologin
      Utilisateur Ligne : chrony:x:994:993::/var/lib/chrony:/sbin/nologin
      Utilisateur Ligne : user1:x:1000:1003:user1:/home/user1:/bin/bash
      Utilisateur Ligne : user2:x:1001:1004::/home/user2:/bin/bash
      Utilisateur Ligne : user3:x:1002:1005::/home/user3:/bin/bash
```

Les deux syntaxes sont équivalentes, elles permettent de sélectionner les lignes sur lesquelles vont s'appliquer l'action en fonction d'un test.
L'exemple ci-dessus sélectionne les lignes dont le contenu du champ 3 est strictement supérieur à 99.

Les opérateurs possibles sont :

<, <=, >, >=,
== (équivalent à), != (différent), ~ (qui contient), !~ (qui ne contient pas)

La commande awk

L'option -f

L'option -f fichier_des_actions

```
$ cat    actions_awk
/root/{print "Login "$3" Ligne : "$0}

/^root/{print "La ligne commence par root : "$0}

($3 > 99){print "Utilisateur Uid="$3" et Login="$1}

{print "Uid="$3" et Login="$1}


$ awk    -F:   -f   actions_awk   /etc/passwd
```

L'option -f

Cette option permet de simplifier la syntaxe de la commande awk, ainsi que de mémoriser au sein d'un fichier une liste d'actions. C'est tout particulièrement utile pour une utilisation répétée, ou lorsqu'il y a de nombreuses actions à réaliser.

```
$ cat    actions_awk
/root/{print "Login "$3" Ligne : "$0}

/^root/{print "La ligne commence par root : "$0}

($3 > 99){print "Utilisateur Uid="$3" et Login="$1}

{print "Uid="$3" et Login="$1}

$ awk    -F:   -f   actions_awk   /etc/passwd
Login 0 Ligne : root:x:0:0:root:/root:/bin/bash
La ligne commence par root : root:x:0:0:root:/root:/bin/bash
Uid=0 et Login=root
Uid=1 et Login=bin
Uid=2 et Login=daemon
Login 11 Ligne : operator:x:11:0:operator:/root:/sbin/nologin
Uid=11 et Login=operator
Uid=12 et Login=games
Utilisateur Uid=999 et Login=polkitd
Uid=999 et Login=polkitd
Utilisateur Uid=173 et Login=abrt
Uid=173 et Login=abrt
Utilisateur Uid=998 et Login=unbound
Uid=998 et Login=unbound
Etc...
```

La commande awk

Les variables internes

FS : caractère séparateur de champs en entrée.
OFS : caractère séparateur de champs en sortie.

RS : caractère séparateur d'enregistrement en entrée.
ORS : caractère séparateur d'enregistrement en sortie.

NF : nombre de champs sur l'enregistrement courant.

NR : nombre d'enregistrements lus.
FNR : nombre d'enregistrements du fichier.

FILENAME : nom du fichier.
...

Les variables internes

Il existe des variables spécifiques à la commande awk, entre autres :

FS : caractère séparateur de champs en entrée.
OFS : caractère séparateur de champs en sortie.
RS : caractère séparateur d'enregistrement en entrée.
ORS : caractère séparateur d'enregistrement en sortie.
NF : nombre de champs sur l'enregistrement courant.
NR : nombre d'enregistrements lus.
FNR : nombre d'enregistrements du fichier.
FILENAME : nom du fichier.

```
$ cat    actions_awk_3
BEGIN {
  FS=":"
  OFS="?"
  ORS="|"
}
{print $3,$1}

$ awk    -f    actions_awk_3    /etc/passwd
0?root|1?bin|2?daemon|3?adm|4?lp|5?sync|6?shutdown|7?halt|8?mail|11?operator|12?
games|14?ftp|99?nobody|81?dbus|999?polkitd|173?abrt|998?unbound|997?colord|113?
usbmuxd|38?ntp|70?avahi|170?avahi-autoipd|996?saslauth|107?qemu|995?
libstoragemgmt|32?rpc|29?rpcuser|65534?nfsnobody|172?rtkit|75?radvd|994?chrony|
171?pulse|42?gdm|993?gnome-initial-setup|89?postfix|74?sshd|72?tcpdump|1000?user1|
1001?user2|1002?user3|$
```

```
$ cat    actions_awk_3_bis
BEGIN {
  FS=":"
}
{print "La ligne " NR " login="$1" avec " NF " champs"}


$ awk    -f    actions_awk_3_bis    /etc/passwd
La ligne 1 login=root avec 7 champs
La ligne 2 login=bin avec 7 champs
La ligne 3 login=daemon avec 7 champs
La ligne 4 login=adm avec 7 champs
La ligne 5 login=lp avec 7 champs
La ligne 6 login=sync avec 7 champs
La ligne 7 login=shutdown avec 7 champs
La ligne 8 login=halt avec 7 champs
La ligne 9 login=mail avec 7 champs
```

La commande awk

Les blocs BEGIN et END

Les blocs BEGIN { … } et END { … }

```
$ cat    actions_awk_2
      BEGIN {
        FS=":"
        print "Debut du traitement"
      }

      {print "Uid="$3" et Login="$1" et la ligne : "$0}

      END {
        print "Fin du traitement"
      }
```

Les blocs BEGIN et END

Syntaxe :

```
BEGIN {
   action-1
   action-2
   …
}

END {
   action-1
   …
}
```

Les actions définies au sein du bloc BEGIN sont exécutées avant le traitement des lignes du fichier.

Les actions définies au sein du bloc END sont exécutées après le traitement des lignes du fichier.

```
$ cat    actions_awk_2
BEGIN {
  FS=":"
  print "Debut du traitement"
}

{print "Uid="$3" et Login="$1" et la ligne : "$0}

END {
  print "Fin du traitement"
}
```

```
$ awk    -f    actions_awk_2    /etc/passwd
Debut du traitement
Uid=0 et Login=root et la ligne : root:x:0:0:root:/root:/bin/bash
Uid=1 et Login=bin et la ligne : bin:x:1:1:bin:/bin:/sbin/nologin
Uid=2 et Login=daemon et la ligne : daemon:x:2:2:daemon:/sbin:/sbin/nologin
Uid=3 et Login=adm et la ligne : adm:x:3:4:adm:/var/adm:/sbin/nologin
Uid=4 et Login=lp et la ligne : lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
Uid=5 et Login=sync et la ligne : sync:x:5:0:sync:/sbin:/bin/sync
Uid=7 et Login=halt et la ligne : halt:x:7:0:halt:/sbin:/sbin/halt
Uid=8 et Login=mail et la ligne : mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
Uid=12 et Login=games et la ligne : games:x:12:100:games:/usr/games:/sbin/nologin
Uid=14 et Login=ftp et la ligne : ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
Uid=99 et Login=nobody et la ligne : nobody:x:99:99:Nobody:/:/sbin/nologin
… etc …
Uid=72 et Login=tcpdump et la ligne : tcpdump:x:72:72::/:/sbin/nologin
Uid=1001 et Login=user2 et la ligne : user2:x:1001:1004::/home/user2:/bin/bash
Uid=1002 et Login=user3 et la ligne : user3:x:1002:1005::/home/user3:/bin/bash
Fin du traitement
```

La commande awk

Les opérations arithmétiques

```
$ cat     actions_awk_4
BEGIN {
  FS=":"
  num=0
}
{ num = num + $3}
END {
  print "Somme = "num
}

$ awk     -f     actions_awk_4     /etc/passwd
Somme = 77189
```

Les opérations arithmétiques

il est possible de réaliser des opérations arithmétiques (addition, multiplication, …) de manière très simple sur tous types de variables ($num, $1, …).

```
$ cat     actions_awk_4
BEGIN {
  FS=":"
  num=0
}
{ num = num + $3}
END {
  print "Somme = "num
}

$ awk     -f     actions_awk_4     /etc/passwd
Somme = 77189
```

Autres syntaxes :

num = num + $3 équivalent à : num = num+$3 ou : num=num+$3

num+=5 num++

num=5*10

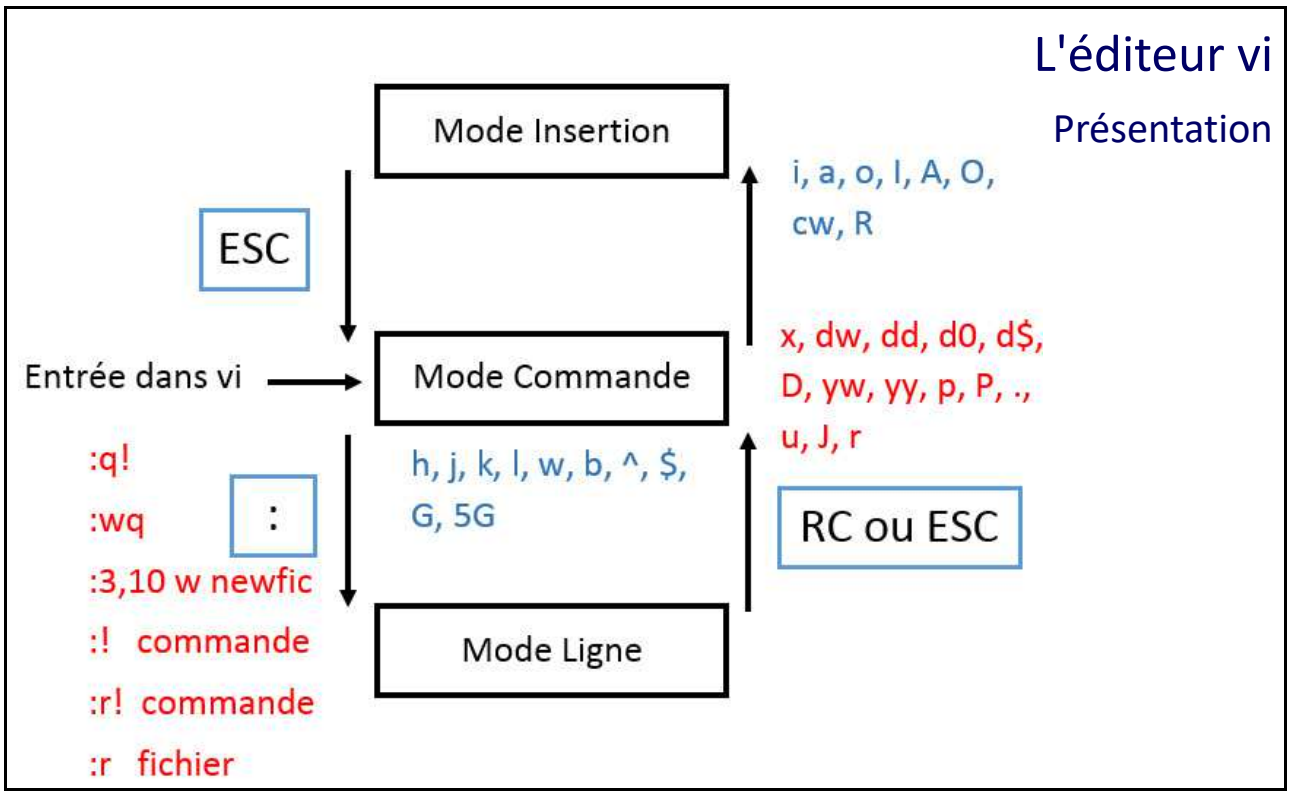
Notes

L'éditeur vi

Dans ce chapitre nous allons traiter l'utilisation de l'éditeur de texte vi.

L'éditeur vi

- Présentation
- Les déplacements du curseur
- Mode insertion
- Suppression – Mode commande
- Compléments – Mode commande
- Mode ligne
- Mode ligne – suite
- Fichier «.exrc»



Présentation

«**vi**» et «**vim**» sont des éditeurs de texte très puissant mais non conviviaux.
Toutes les fonctionnalités sont disponible via le clavier minimum (c'est à dire sans le pavé numérique, ni les flèches, ni les touches fonction).

Indispensable lorsqu'il n'y a pas d'interface graphique installé sur le serveur Linux.
Toute les fonctionnalités présentent sur ce slide sont détaillées par la suite dans ce module.

Les trois modes de «vi»

Mode commande	:	permet de ce déplacer et de modifier le texte dans l'éditeur.
Mode insertion	:	permet d'insérer des caractères dans le document.
Mode ligne	:	permet de réaliser des actions globales.

Syntaxe : vi fic

Édite le fichier fic, si le fichier n'existe pas il est créé.

vi +5 fic

Édite le fichier fic, et positionne le curseur sur la ligne 5.

L'éditeur vi

Les déplacements du curseur

- h j k l + -
- w b 3w 3b
- ^ $
- G 1G 5G

Les déplacements du curseur

La plupart des actions ce font par rapport à la position du curseur. Les déplacements du curseur se font en mode commande.

Les touches de déplacement

Caractère	«**l**»	:	déplacement vers la droite.
Caractère	«**h**»	:	déplacement vers la gauche.
Caractère	«**k**»	:	déplacement sur la ligne du haut.
Caractère	«**j**»	:	déplacement sur la ligne du bas.
Caractère	«**+**»	:	déplacement au début de la ligne suivante.
Caractère	«**-**»	:	déplacement au début de la ligne précédente.

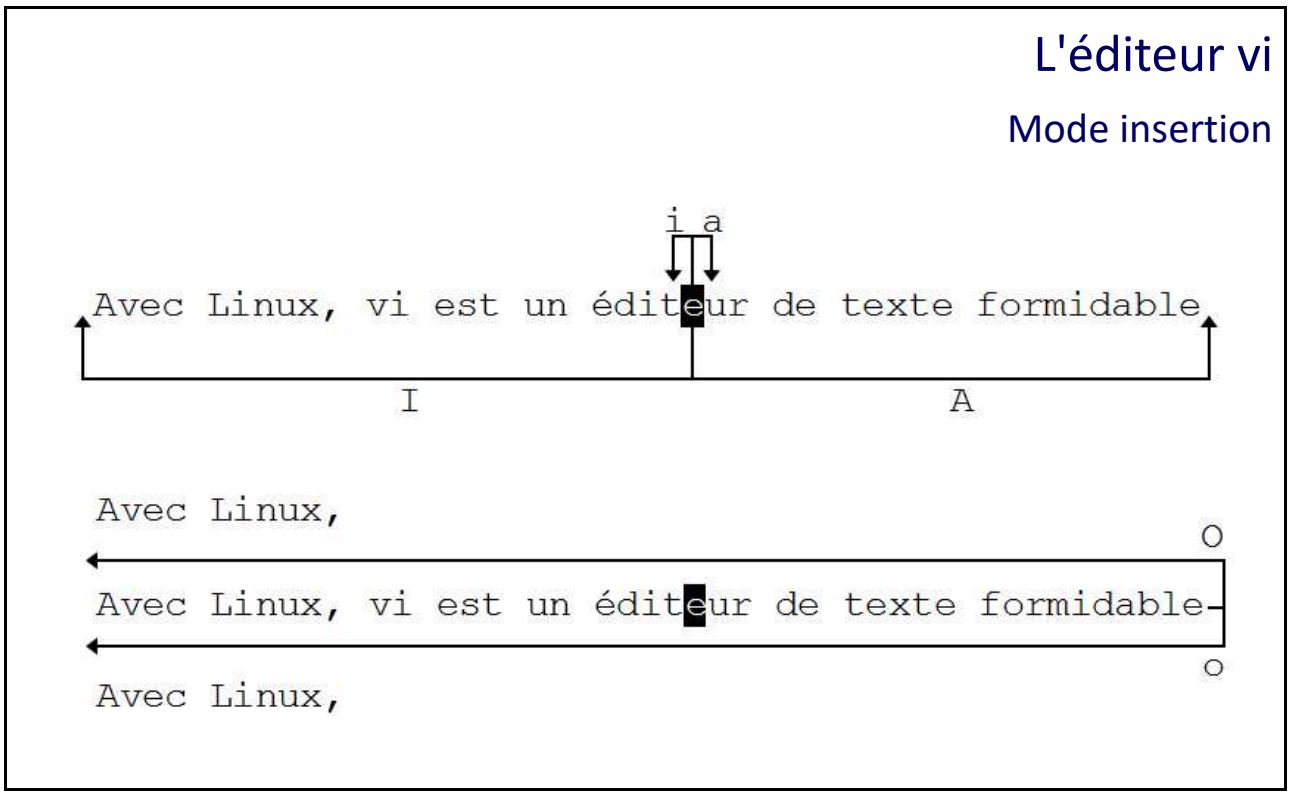
Nous pouvons aussi utiliser les flèches directionnelles.

Caractère	«**w**»	:	déplacement au début du mot suivant.
Caractère	«**b**»	:	déplacement au début du mot précédent.

En ajoutant une valeur devant le caractère, on spécifie le nombre de déplacements.

Caractère	«**^**»	:	déplacement sur le premier caractère de la ligne.
Caractère	«**$**»	:	déplacement sur le dernier caractère de la ligne.

Caractère	«**G**»	:	déplacement sur la dernière ligne du fichier.
	«**1G**»	:	déplacement sur la première ligne du fichier.
	«**5G**»	:	déplacement sur la cinquième ligne du fichier.



Mode insertion

Lorsque nous sommes dans le **mode insertion**, tout ce que nous tapons au clavier est inséré dans le document.

Pour basculer dans le mode insertion

Caractère	«**i**»	:	pour insérer du texte avant le curseur.
	«**I**»	:	pour insérer du texte au début de la ligne.
Caractère	«**a**»	:	pour insérer du texte après le curseur.
	«**A**»	:	pour insérer du texte à la fin de la ligne.
Caractère	«**o**»	:	pour insérer une ligne après la ligne où est localisé le curseur.
	«**O**»	:	pour insérer une ligne avant la ligne où est localisé le curseur.

Pour sortir du mode insertion

Pour quitter le **mode insertion** et retourner dans le **mode commande**, nous utilisons la touche «**ESC**».

L'éditeur vi

Suppression - Mode commande

- x
- dd 5dd D
- dw 5dw
- db 5db
- d^ d$
- dG d1G d5G

Suppression – Mode commande

Ce mode nous permet de supprimer des paragraphes, des lignes, des mots ainsi que des caractères. Tout se fait par rapport à la position du curseur.

«**x**» : suppression du caractère sous le curseur.

«**dd**» : suppression de la ligne courante.
«**5dd**» : suppression de 5 lignes.
«**D**» : suppression du curseur à la fin de la ligne.

«**dw**» : supprime du curseur à la fin du mot.
«**5dw**» : supprime 5 mots après du curseur .
Donc si le curseur est positionné sur le premier caractère, cela revient à supprimer un mot.

«db» : supprime du curseur au début du mot.
«**5db**» : supprime 5 mots avant le curseur.

«**d^**» : suppression du curseur jusqu'au début de la ligne.
«**d$**» : suppression du curseur jusqu'à la fin de la ligne.

«**dG**» : supprime du curseur jusqu'à la fin du fichier.
«**d1G**» : supprime du curseur jusqu'au début du fichier.
«**d5G**» : supprime du curseur jusqu'à la 5ème ligne du fichier.

L'éditeur vi

Compléments - Mode commande

• u	.			
• cw	5cw	r	R	
• J				
• yy	5yy	p	P	
• yw	5yw			
• /chaine	n	N		

Compléments - Mode commande

«**u**» : undo - annule le changement précédent.
«**.**» : répète la dernière action.

«**cw**» : change le texte du curseur à la fin du mot. On finalise la saisie par **ESC**.
«**5cw**» : change le texte du curseur jusqu'à la fin du 5ème mot.

«**r**» : remplacement du caractère sous le curseur.
Exemple : «**rA**» remplace le caractère sous le curseur par «**A**».
«**R**» : remplacement d'une suite de caractères.

«**J**» : joint la ligne suivante à la ligne courante.

Copier/coller
«**yy**» : sélection d'une ligne à copier.
«**5yy**» : sélection de 5 lignes à copier.
«**p**» : colle la sélection après le curseur.
«**P**» : colle la sélection avant le curseur.

«**yw**» : sélection d'un mot à copier. «**5yw**», pour 5 mots à copier.

Recherche
«**/xxx**» : pour rechercher un texte «**xxx**» dans le document.
«**n**» : pour passer à l'occurrence suivante.
«**N**» : pour passer à l'occurrence précédente.

L'éditeur vi

Mode ligne

- :q!
- :w :wq
- :w newfic
- :3,10w newfic
- :! commande
- :r fichier :r! commande
- :6
- :4m6 :2,5m10

Mode ligne

Pour accéder au mode ligne, il faut saisir le caractère «**:**». Pour le quitter et revenir au **mode commande**, nous utiliserons la touche «**Entrée**» ou la touche «**Echap**».

Sauvegarder et quitter

:q!		:	quitte l'éditeur sans sauvegarder.
:wq		:	sauvegarde le fichier et quitte l'éditeur.
:w	newfic	:	sauvegarde le fichier dans un nouveau fichier nommé «**newfic**» sans quitter l'éditeur.
:3,10w	newfic	:	créer un nouveau fichier «**newfic**» avec les lignes 3 à 10 du fichier courant.

Interagir avec le Shell

:! commande	:	lance une commande puis revient à l'édition.
:r! commande	:	insère le résultat d'une commande après la ligne du curseur.
:r fichier	:	insère le contenu du fichier après la ligne du curseur.
:6	:	positionne le curseur sur la ligne 6.
:4m8	:	déplacement. La ligne 4 est déplacée après la ligne 8.
:2,5m10	:	déplacement. Les lignes 2 à 5 sont déplacées après la ligne 10.

L'éditeur vi

Mode ligne - suite

- :set nu :set nonu
- :set list :set nolist

- map X 10dd unmap X
- ab rep repertoire unab rep

- 1,$s/ancien/nouveau/g
- %g/filtre/s/ancien/nouveau/g
- %g/filtre/d

Mode ligne - suite

Complément des commandes du mode ligne

:set	**nu**	:	numérote les lignes.
:set	**nonu**	:	supprime la numérotation.
:set	**list**	:	affiche les caractères invisibles.
:set	**nolist**	:	supprime l'affichage des caractères invisibles.
:**map**		:	permet de créer une macro remplaçant l'action d'une touche dans l'éditeur.
:**unmap**		:	pour supprimer une macro créée par map.

Exemple :

```
:map        X          10dd

:map
 X          10dd

:unmap      X
```

Dans l'exemple ci-dessus, nous avons attribué à la touche «**X**» l'action «**10dd**» qui correspond à la suppression de 10 lignes.
Puis nous avons supprimé cette macro grâce à la commande «**unmap**».

:ab : permet d'attribuer une abréviation à une chaîne de caractère.
:unab : permet d'annuler une abréviation.

Exemple :

```
:ab         rep         repertoire

:ab
! rep       repertoire

:unab       rep
```

Dans l'exemple ci-dessus, nous avons créé une abréviation pour la chaîne de caractères «**repertoire**».
Désormais, il suffira de rentrer «**rep**» suivi d'une touche de fin de mot («**,**», «**.**», «**Entrée**», etc...) pour que «**vi**» affiche «**repertoire**».
Puis nous avons supprimé l'abréviation grâce à la commande «**unab**».

:1,$s/ancien/nouveau/g :

Permet de modifier toutes les haines de caractères «**ancien**» par «**nouveau**» sur tout le document («**1,$**»).

Au lieu de «**1,$**», nous aurions pu préciser une autre zone de traitement telle que «**5,20**» pour un remplacement uniquement entre les lignes 5 à 20.

Le «**g**» final indique «**globale**» pour toutes les occurrences sur la ligne. En omettant le «**g**», seule la première occurrence sur chaque ligne sera substituée.

:%g/filtre/s/ancien/nouveau/g :

La substitution présentée précédemment sera réalisée sur l'ensemble du fichier uniquement que pour les lignes qui contiennent la chaîne de caractères du filtre.

:%g/filtre/d :

Toutes les lignes qui contiennent la chaîne de caractères du filtre seront supprimées.

L'éditeur vi

Fichier «.exrc»

- Personnalisation du comportement de vi

- vi $HOME/.exrc

  set nu

  set list

  map X 10dd

Fichier « .exrc »

Le fichier «**.exrc**» est un fichier de configuration qui est chargé automatiquement au démarrage de l'éditeur «**vi**».

Ce fichier permet de définir un comportement spécifique à «**vi**» pour un utilisateur. Les actions possibles sont principalement les opérations «**set xxx**», «**map xxx**» et «**ab xxx**».

Exemple :

```
$  vi   $HOME/.exrc
set  nu
set  list
map  X  10dd
```

Notes

Fin de session de Formation

Je vous recommande de relire ce support de cours d'ici les deux semaines à venir, et de refaire des exercices.

Il ne vous reste plus qu'à mettre en œuvre ces nouvelles connaissances au sein de votre entreprise.

Merci, et à bientôt.

Jean-Marc Baranger

Theo Schomaker