# Les scripts shell

## Protection des expressions

- Certains caractères, comme *****, **?** et **$** ont une signification particulière pour le shell, pourtant il peut arriver qu'on souhaite les utiliser comme caractères normaux.
  - Exemples de mauvaises surprises

```
$ echo ********************
ch1 play report work
$ echo Total price: $5
Total price:
$
```

- Il y a trois méthodes pour protéger les caractères de l'interprétation du shell :
  - Protection par le caractère backslash
  - Protection par apostrophes
  - Protection par guillemets






# Les scripts shell

## Protection des expressions — Backslash

- Le caractère backslash **** sert à désactiver l'interprétation du caractère qui le précède.

```
$ a=1
$ echo $a
1
$ echo \$a
$a
$
```

- Le caractère backslash peut servir à préfixer n'importe quel caractère, y compris lui-même

```
$ echo \\
\
$
```

- Utilisé à la fin d'une ligne, le caractère backslash permet de continuer une commande sur la ligne suivante
  - `echo Beginning \`
    `and end`

    dans un script produit

```
Beginning and end
```

  - Dans un shell interactif, l'invite de commande **>** indique que la commande n'est pas terminée

```
$ echo Beginning \
> and end
Beginning and end
$
```

# Les scripts shell

## Protection des expressions — Apostrophes

- Les apostrophes `' '` permettent de protéger toute une chaîne en une seule fois.
- **Tous les caractères** rencontrés perdent leur signification spéciale.
  - Y inclus le backslash

```
$ echo '#\$"&>|'
#\$"&>|
$
```

  - Donc impossible d'inclure un apostrophe dans la chaîne protégée...
- Le caractère fin de ligne perd aussi sa signification






# Les scripts shell

## Protection des expressions — Guillemets

- Il arrive que la protéction par des apostrophes est trop forte — on aimerait par exemple protéger les espaces mais continuer d'utiliser des variables
- Les guillemets `" "` **gardent la signification spéciale des trois caractères suivants**, alors que tous les autres perdent leur signification
  - **$**
  - Backquote `` ` ` `` (différent de l'apostrophe `' '` !)
  - Backslash ****

```
$ hour=14
$ minute=50
$ b="*** The time is $hour:$minute. ***"
$ echo "$b"
*** The time is 14:50. ***
$
```

- On peut utiliser backslash à l'intérieur des guillemets pour protéger **$** et guillemets **"**

```
$ c="The value of \$hour is \"$hour\"."
$ echo $c
The value of $hour is "14".
$
```

- Attention, protéger non seulement l'affectation, mais aussi l'utilisation, sinon :

```
$ b="*"
$ echo $b
ch1 play report work
$
```

- **Conseil :** Bash donne une signification spéciale à beaucoup de caractères. En cas de doute, utiliser la protection par les guillemets.

# Les scripts shell
## Extraction de motifs

- Dans la manipulation des noms de fichiers ou adresses réseau on a souvent besoin d'extraire des motifs, par exemple enlever d'un nom de fichier `program.cpp` son extension `.cpp`.
- Le shell offre des expressions de variable avec extraction de motifs
  - Syntaxe :
    - `${variable#motif}`
    - `${variable##motif}`
    - `${variable%motif}`
    - `${variable%%motif}`
  - L'extraction peut se faire en début de variable (**#**) ou fin de variable (**%**)
- Les motifs peuvent contenir les meta-caractères connus
  - Le caractère ***** correspond à n'importe quelle chaîne de caractères (éventuellement vide)
  - Le caractère **?** correspond à n'importe quel caractère
  - Les crochets **[ ]** encadrant une liste de caractères représentent n'importe quel caractère contenu dans cette liste.
  - Le caractère \ désactive l'interprétation particulière du caractère suivant.






# Les scripts shell
## Extraction de motifs

- L'expression `${variable#motif}` est remplacée par la valeur de la variable, de laquelle on ôte la chaîne initiale **la plus courte** qui corresponde au motif.
  - Exemple simple

```
$ package_name="pkg-0385"
$ package_number="${package_name#pkg-}"
$ echo "package_number"
0385
$
```

  - Exemple avec meta-caractère

```
$ package_name="pkg-beta-0385"
$ package_number="${package_name#*-}"
$ echo "package_number"
beta-0385
$
```

- L'expression `${variable##motif}` est remplacée par la valeur de la variable, de laquelle on ôte la chaîne initiale **la plus longue** qui corresponde au motif.
  - Exemple simple

```
$ package_name="pkg-0385"
$ package_number="${package_name##pkg-}"
$ echo "package_number"
0385
$
```

  - Exemple avec meta-caractère

```
$ package_name="pkg-beta-0385"
$ package_number="${package_name##*-}"
$ echo "package_number"
0385
$
```

# Les scripts shell

## Extraction de motifs

- Symétriquement, les expressions `${variable%motif}` et `${variable%%motif}` débarrassent la variable du plus court et du plus long préfixe correspondant au motif.
  - Exemples : enlever des extensions de fichier

```
$ file_name="img0459.jpg"
$ base_name="${filename%.jpg}"
$ echo "base_name"
img0459
$
```

```
$ file_name="img0459.jpg"
$ base_name="${filename%%.jpg}"
$ echo "base_name"
img0459
$
```

```
$ file_name="source.tar.gz"
$ base_name="${filename%.*}"
$ echo "base_name"
source.tar
$
```

```
$ file_name="source.tar.gz"
$ base_name="${filename%%.*}"
$ echo "base_name"
source
$
```






# Les scripts shell

## Structures de contrôle — Boucle **for-in-do-done**

- La boucle `for` sert à répéter un traitement un nombre de fois déterminé
- Syntaxe :

```
for variable in liste_mots
do
    commandes
done
```

- La variable va prendre successivement comme valeur tous les mots de la liste et le corps de la boucle sera répété pour chacune de ces valeurs.
- Exemple :

```
for ext in c cpp h o
do
    echo "Files ending in .$ext:"
    ls *.$ext
done
```

```
Files ending in .c:
robot.c
Files ending in .cpp:
main.cpp        trajectory.cpp
Files ending in .h:
robot.h       trajectory.h
Files ending in .o:
main.o        robot.o       trajectory.o
```

# Les scripts shell

## Structures de contrôle — Boucle **for-in-do-done**

- La commande `seq` est spécialement conçue pour les boucles `for-in-do-done` pour générer des séquences numériques.
- On peut spécifier la première valeur, l'incrément et la dernière valeur de la séquence (seulement la dernière valeur est obligatoire, les autres prennent des valeurs par défaut).
  - Syntaxe :
    - `seq dernier`
    - `seq premier dernier`
    - `seq premier incrément dernier`
  - Exemple :

```
for i in $(seq 5 -1 1)
do
    echo "Remaining $i seconds to liftoff."
    sleep 1
done
```

```
Remaining 5 seconds to liftoff.
Remaining 4 seconds to liftoff.
Remaining 3 seconds to liftoff.
Remaining 2 seconds to liftoff.
Remaining 1 seconds to liftoff.
```






# Les scripts shell

## Structures de contrôle — Boucle **for-in-do-done**

- On utilise assez souvent la boucle `for` dans un shell interactif. Dans ce cas on la met sur une seule ligne. Il faut alors mettre des point-virgules `;`
  - avant le mot-clé `do`
  - entre les commandes du corps
  - avant le mot-clé `done`

```
$ for ext in c cpp h o ; do echo "Files ending in $ext:" ; ls *.$ext ; done
Files ending in c:
robot.c
Files ending in cpp:
main.cpp        trajectory.cpp
Files ending in h:
robot.h       trajectory.h
Files ending in o:
main.o        robot.o       trajectory.o
$
```

# Exercice 03.04

- Écrivez un script qui affiche le nom de chaque fichier du répertoire dans lequel il se trouve, en le faisant précéder d'un numéro d'ordre, comme dans l'exemple suivant :

```
$ number_files
1) ch1
2) play
3) report
4) work
$
```

  - Utilisez pour cela une structure for-do-done.

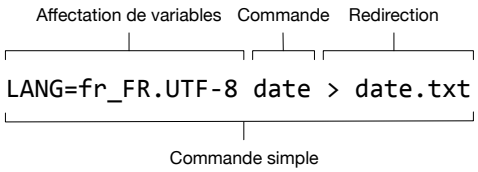





# Les scripts shell

## Commandes et code de retour

- L'opération la plus élémentaire que l'on puisse effectuer avec le shell est appelée commande simple.
  - Elle est composée de
    - affectations de variables,
    - une séquence de mots (le premier étant la commande à exécuter, les autres les paramètres),
    - des opérateurs de rédirection.
  - Chacune de ces parties est facultative.



# Les scripts shell

## Commandes et code de retour

- Lorsqu'une affectation de variable précède la commande (sur la même ligne), le contenu de la variable ne prend la valeur indiquée que pendant l'exécution de la commande.
  - Ceci permet de modifier une variable d'environnement pendant l'exécution de la commande sans changer sa valeur de manière permanente.
  - Exemple : Afficher la date en français (fr_FR.UTF-8) dans un environnement configuré pour utiliser l'anglais (en_US.UTF-8)

```
$ echo $LANG
en_US.UTF-8
$ date
Fri Apr  4 11:38:52 CEST 2014
$ LANG=fr_FR.UTF-8 date
Ven  4 avr 2014 11:39:23 CEST
$ echo $LANG
en_US.UTF-8
$
```






# Les scripts shell

## Commandes et code de retour

- Lorsqu'il se termine, un processus sous Unix renvoie toujours un *code de retour (exit status)*.
  - Un code 0 signifie une terminaison normale.
  - Un code différent de 0 indique généralement une erreur.
  - Les codes en cas d'erreur ne sont pas standardisés, mais il y a certaines conventions plus ou moins suivies comme les codes d'erreur définis dans `sysexits.h` (cf. `man 3 sysexits`).

```
int main() {
    // début du programme
    ... // instructions
    if (...) {
        // erreur
        ...
        // fin prématurée
        // du programme
        return EXIT_FAILURE;
    }
    ...
    // fin normale du programme
    return EXIT_SUCCESS;
}
```

Sous Unix valeur numérique 1

Sous Unix valeur numérique 0

# Les scripts shell

## Commandes et code de retour

- Le shell permet de consulter le code de retour de la dernière commande exécutée avec la variable spéciale `$?`
  - Si la commande est une affectation de variable, le code de retour est 0.
  - Dans un pipeline, la valeur rapportée par `$?` est le code de retour de la dernière commande.

```
$ ls
ch1 play report work
$ echo $?
0
$ ls aaaaaaaaa
ls: aaaaaaaaa: No such file or directory
$ echo $?
1
$
```

- Exemple : La commande `grep` retourne trois codes différents :
  - 0 si le motif a été trouvé dans le fichier
  - 1 si le motif n'a pas été trouvé
  - 2 si une erreur est survenue






# Les scripts shell

## Commandes et code de retour

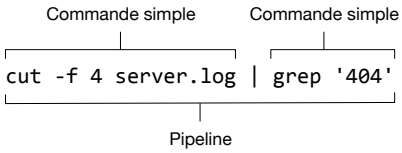
- Quelques commandes spéciales ne font rien mais retournent des codes de retour spécifiques (par exemple utile pour le débogage ou les boucles infinies) :
  - La commande `:` ne fait rien et retourne toujours 0
  - La commande `true` ne fait rien et retourne toujours 0
  - La commande `false` ne fait rien et retourne toujours 1

```
$ while true ; do echo 'Hi!' ; done
Hi
Hi
Hi
Hi
Hi
Hi
Hi
...
```

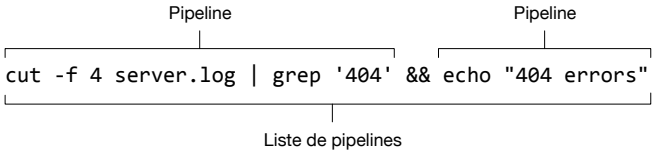
# Les scripts shell

## Commandes et code de retour

- Un *pipeline* est l'enchaînement de plusieurs commandes simples, couplées par l'opérateur |
  - La sortie standard d'une commande est connectée à l'entrée standard de la suivante
  - Chaque commande simple est exécutée par un processus indépendant et tous les processus se déroulent simultanément.
    - Parfois un processus doit attendre que son prédécesseur lui envoie des données




- Une *liste de pipelines* est la combinaison de plusieurs pipelines utilisant un de quatre symboles
  - `;` — Séquencement
  - `&` — Parallélisme
  - `&&` — Dépendance
  - `||` — Alternative









# Les scripts shell

## Commandes et code de retour

| Symbole | Connexion | Détail |
|---|---|---|
| `;` | Séquencement | La seconde opération ne commence qu'après la fin de la première. |
| `&` | Parallélisme | La première opération est lancée à l'arrière-plan, et la seconde démarre simultanément ; elles n'ont pas d'interactions entre elles. |
| `&&` | Dépendance | La seconde opération n'est exécutée que si la première a renvoyé un code de retour nul, ce qui, par convention, signifie « succès ». |
| `\|\|` | Alternative | La seconde opération n'est exécutée que si la première a renvoyé un code de retour non nul, ce qui, par convention, signifie « échec ». |

# Les scripts shell

## Commandes et code de retour

- Exemple séquencement : Attendre un certain temps entre les commandes
  - `echo Begin ; sleep 5 ; echo middle ; sleep 5 ; echo end`
- Exemple parallélisme : Mettre une tâche longue en tâche de fond
  - `ls -lR / | grep README &`
- Exemple dépendance : Créer un répertoire et en cas de succès copier des fichiers
  - `mkdir archive && cp -a package/* archive`
- Exemple alternative : Tester un host avec un ping et afficher un message en cas d'échec
  - `ping -c 1 $host || echo "$host is not accessible"`

- Les accolades { } permettent de regrouper les commandes différemment
  - `{ cat $dirlist ; echo "/home" ; echo "/var" } | sort`

- Le code de retour d'une liste de pipelines est le code de la dernière commande exécutée.






# Exercice 03.05

- Essayez de prévoir le résultat des commandes suivantes, puis exécutez-les. Que se passe-t-il et pourquoi ?

```
$ var=1234
$ echo $var
$ var=5678 echo $var
$ echo $var
```