Initiation à l'environnement Unix : TP n° 3

septembre 2024 — Pierre Rousselin

1 Cuisine du shell : guillemets anglais et variables d'environnement

Exercice 1 : L'inhibition des caractères spéciaux : les guillemets anglais

1. Pour chacun des items suivants, entrer la ou les commandes données puis prendre des notes sur ce que révèle (ou non) cette expérience.

a)
```
$ echo "? * et [ sont utilisés pour le développement de chemins"!
```

b)
```
$ echo "~ provoque un développement du tilde"
```

c)
```
$ echo "Entre \", on peut aussi
> écrire     sur plusieurs
> lignes"
```

d)
```
$ nom=Alice
$ echo '$nom scripte en shell'
$ echo "$nom scripte en shell"
$ echo "\$nom scripte en shell"
```

e)
```
$ echo "le chemin absolu du répertoire courant est `pwd`"
$ echo "le chemin absolu du répertoire courant est \`pwd\`"
$ echo "le chemin absolu du répertoire courant est $(pwd)"
$ echo "le chemin absolu du répertoire courant est \$(pwd)"
```

f)
```
$ echo "Aussi sûr que 2 et 2 font $((2 + 2))"
$ echo "Aussi sûr que 2 et 2 font \$((2 + 2))"
```

g)
```
$ echo "\\\\\"\$\`\*\'"
```

h)
```
$ mavar="Alice
> et
> Bob"
$ echo $mavar font plein de choses
$ echo "$mavar font plein de choses"
```

2. Au vu des expériences précédentes (et d'autres à inventer si nécessaire), répondre aux questions suivantes :

a) Quels sont les caractères qui sont spéciaux entre guillemets anglais ?

b) Quel est le rôle du caractère `\` entre guillemets anglais ? Dans quel contexte est-il spécial, littéral ?

c) Quels sont les développements qui n'ont jamais lieu entre guillemets anglais ?

d) Selon vous, pourquoi avoir créé plusieurs mécanismes d'inhibition ?

--- * ---

Exercice 2 : Variables d'environnement

Les variables d'environnement sont des variables qui modifient le comportement de certaines commandes et se transmettent de processus parent à processus enfant (nous en reparlerons). En général, elles sont écrites en lettres capitales.

Pour le moment, nous allons nous contenter d'écrire un script qui affiche la valeur de certaines de ces variables.

1. Avec un éditeur de texte, écrire le script suivant que vous appellerez `infos`.

```
#!/bin/sh
# infos : donne des informations sur l'utilisateur
```

```
echo "Bonjour $USER !
Le répertoire courant est $PWD.
Amusez-vous bien !"
```

2. Rendre ce script exécutable et l'exécuter.

3. Complétez ce script pour qu'il affiche également les informations suivantes :

a) le `PATH` de l'utilisateur (les répertoire où trouver les commandes externes) ;
b) la `LANG` de l'utilisateur ;
c) le répertoire courant (`PWD`) ;
d) le répertoire courant précédent (`OLDPWD`) ;
e) le `SHELL` par défaut de l'utilisateur ;
f) l'éditeur de texte (`EDITOR`) par défaut de l'utilisateur.

Remarque : certaines de ces variables peuvent ne pas être définies (ce n'est pas grave), cela dépend du système et des fichiers de configuration de l'utilisateur.

4. Enfin, afficher la date et l'heure, puis le calendrier du mois en cours.

5. Afficher la liste des variables d'environnement avec la commande `env`. Si vous en avez envie, complétez votre script pour afficher le contenu d'autres variables d'environnement.

--- * ---

2 La commande `printf`

Mauvaise nouvelle : notre chère commande `echo` est en fait presque inutilisable à cause de nombreuses versions différentes et non compatibles… et on évitera maintenant de l'utiliser dans les scripts, sauf peut-être dans des cas très simples (affichage d'une chaîne littérale).

Heureusement, il y a `printf` qui est beaucoup mieux mais il faut apprendre à s'en servir. Les 3 exercices suivants présentent cette commande de façon assez exhaustive, mais ce n'est pas la peine de tout apprendre par cœur.

Exercice 3 : La commande `printf` et les séquences d'échappement

La commande `printf` (pour *print formatted*) est assez semblable à la fonction `printf` du langage C (pas de panique, on va de toute façon expliquer entièrement son fonctionnement).

La commande `printf` interprète certaines séquences dites « d'échappement » (les seules à retenir sont `\n` `\\` et `\t`) :

— `\r` désigne un retour chariot *carriage return* (retourner au début de la ligne courante) ;
— `\a` désigne une alerte (qui peut être sonore) ;
— `\t` désigne une tabulation ;
— `\n` désigne une nouvelle ligne (`<newline>`) ;
— `\\` désigne une contre-oblique (`\`) ;
— `\b` (*backspace*) est un retour en arrière d'un caractère ;
— `\v` (*vertical tabulation*) descend le curseur d'une ou plusieurs lignes ;
— `\f` (*form feed*) est censé éjecter la page (souvenez-vous des téléscripteurs…), les résultats sont divers sur les émulateurs de terminal ;
— `\ooo` (où chaque `o` est un chiffre octal) désigne le caractère d'un octet codé par ce nombre octal.

Tester les commandes ci-dessous (exceptionnellement, vous pouvez activer les haut-parleurs de l'ordinateur) et expliquer les résultats obtenus.

```
$ printf '\aAlerte!'
$ printf 'Étudiant\tMatière\t\tNote\nAlice\t\tshell\t\t20\n'
$ printf "une contre-oblique ? \\n"
$ printf 'une contre-oblique ? \\n'
$ printf 'abcd\b\bef\n'
$ printf 'abcd\vefgh\n'
$ printf 'Alice est très forte en shell\rZineb\n'
$ printf 'Au secours, un film de \112ames \102ond \007\012'
```

On pourra se poser les questions suivantes :

1. Est-ce qu'un affichage avec `printf` fait toujours un retour à la ligne ? Et avec `echo` ?
2. Comment fonctionnent les tabulations ? Quelle est leur largeur ?
3. Vaut-il mieux mettre le format (le premier argument de `printf`) entre apostrophes ou entre guillemets anglais ?
4. Comment expliquer le dernier affichage (et le son qui va avec) ? Indice : `man ascii`.

--- * ---

La commande `printf` permet de mettre en forme les arguments qui suivent le format (son premier argument).

Voici un premier exemple :

```
$ printf 'Le continent %s a %u habitants\n' Asie 4504428000
Le continent Asie a 4504428000 habitants
```

Les *spécifications de format* qui apparaissent dans le format sont de petites chaînes de caractères qui commencent par le symbole `%` et finissent par un *caractère spécificateur de format* qui peut être `d i u o x X s b c %`
Chaque spécification de format met en forme l'argument correspondant (dans l'ordre où ils apparaissent) en l'interprétant comme :

- un nombre entier signé (`d i`) ;
- un nombre entier non signé (`u o x X`) ;
- une chaîne de caractères (`s b`) ;
- un seul caractère (`c`).

La largeur *minimale* du champ peut être précisée entre le symbole `%` et le caractère spécificateur de format.

```
$ printf 'Continent:%12s|%10d|\n' Europe 742074000
Continent:      Europe| 742074000|
```

Le champ en sortie, s'il est plus petit que la largeur minimale du champ, est complété à gauche (à droite si le drapeau `-` est présent) par des espaces.

```
$ printf 'Continent:%-12s|%-10d|\n' Europe 742074000
Continent:Europe      |742074000 |
```

Le spécificateur de format `%%` sert juste à imprimer un caractère `%`. Une spécificité très agréable de la commande `printf` (contrairement à la fonction de la bibliothèque standard du langage C !) est que s'il y a plus d'arguments que de spécificateurs de format, le format est réutilisé :

```
$ printf '%7s %2d,%d%%\n' Asie 59 7 Afrique 16 6 Europe 9 8
   Asie 59,7%
Afrique 16,6%
 Europe  9,8%
```

Si la liste des arguments est épuisée avant celle des spécifications de format, les spécifications de type `b s c` sont évaluées comme si une chaîne vide était fournie et les autres, comme si un nombre nul était fourni :

```
$ printf '%5s:%6s:%-4d:\n' aaaaaaaa
aaaaaaaa:      :0   :
$ printf '%5s:%6s:%-4d:\n' aaaaaaaa b 1 c
aaaaaaaa:     b:1   :
    c:      :0   :
```

Exercice 4 : La commande `printf` et les spécifications de format

1. Tester les commandes suivantes et expliquer les résultats obtenus :

a) `printf '%3d %s\n' 3 'petits cochons'`

b) `printf '%s\n' sh ksh bash dash`

c) `printf 'pour %d,\n' 1 2 3 4 5 6 7 8 9; printf 'Boeuf !\n'`

d) `printf '%s\n' /usr/include/std*.h`

2. On s'intéresse maintenant aux entiers non signés.
Tester les commandes suivantes puis donner la signification des caractères spécificateurs `u x X o`, du drapeau `#` et dire comment donner à printf un argument numérique écrit en octal ou en hexadécimal.

```
$ printf '%3u %3o %3x %3X\n' 4 4 4 4 8 8 8 8 10 10 10 10 11 11 11 11
$ printf '%3u %3o %3x %3X\n' 15 15 15 15 16 16 16 16 67 67 67 67
$ printf '%3u %3o %3x %3X\n' 010 010 010 010 0XA3 0XA3 0XA3 0XA3
$ printf '%3u %3o %3x %3X\n' 0xfb 0xfb 0xfb 0xfb
$ printf '%3u %#3o %#3x %#3X\n' 253 253 253 253
```

3. Le caractère spécificateur `b` ne fait pas partie du `printf` de la bibliothèque standard C. On va voir son utilité.
Tester les commandes suivantes et dire à quoi sert le caractère spécificateur `b`.

```
$ printf '**début**\n%s\n**fin**\n' '3\n2\n1\n0!\a'
$ printf '**début**\n%b\n**fin**\n' '3\n2\n1\n0!\a'
$ printf 'Alice a dit :"\n%s\n"\n' '\tJe\n\tscripte\n\ten\n\tshell!'
$ printf 'Alice a dit :"\n%b\n"\n' '\tJe\n\tscripte\n\ten\n\tshell!'
```

4. Dans chaque commande suivante, remplacer la chaîne `format` de façon à obtenir l'affichage donné :

```
$ printf format 1964 Multics 1969 Unix 1991 Linux
1964 Multics
1969 Unix
1991 Linux
$ printf format 1964 Multics 1969 Unix 1991 Linux
1964    Multics
1969       Unix
1991      Linux
$ printf 'format' \
> 1990 'Le langage C' 304 'Kernighan et Ritchie' \
> -51 'La guerre des Gaules' 247 'Jules César' \
> 2008 'Algorithmique' 1296 'Cormen, Leiserson, Rivest et Stein'
 1990|        Le langage C|304 |Kernighan et Ritchie
  -51|La guerre des Gaules|247 |Jules César
 2008|       Algorithmique|1296|Cormen, Leiserson, Rivest et Stein
$ printf format 'abcd\vefgh\n' 'abcd\vefgh\n'
La chaîne 'abcd\vefgh\n' se traduit par :
abcd
    efgh
```

--- * ---

3 Permissions

Exercice 5 : Permissions associées aux fichiers

1. Créer un répertoire et un fichier vide. Utiliser la commande `ls`, et les options `-l` et `-d` (pour ne pas descendre dans les répertoires donnés en argument) sur ces deux nouveaux fichiers pour déterminer les permissions que vous (respectivement votre groupe et les autres) avez sur ces fichiers. Comment reconnaît-on un répertoire ?
2. Ci-dessous sont affichés deux champs (le premier et le dernier) d'une sortie de `ls -ld *`

   ```
   drwxr-xr-x a
   dr-xr--r-- b
   -rw-r--r-- c.txt
   --w--w-r-- d.c
   -rwxr-xr-x op
   ```

   Dire quels fichiers sont des répertoires, donner pour chacun d'eux les permissions associées.
3. Quelles sont les permissions associées au programme `ls`, à votre répertoire personnel, au fichier `/usr/include/stdio.h` et au répertoire racine ? Quel utilisateur et quel groupe est associé à chaque fichier ?

--- * ---

Exercice 6 : La commande `chmod`

1. Tester chacune des commandes suivantes et décrire le fonctionnement de la commande `chmod` en notation symbolique.

   ```
   $ touch f; ls -l f
   $ chmod a= f; ls -l f
   $ chmod o+rw f; ls -l f
   $ chmod u=o f; ls -l f
   $ chmod o-wx f; ls -l f
   $ chmod g+u f; ls -l f
   $ chmod a+x,g-w f; ls -l f
   ```
2. Donner les permissions suivantes au fichier `f` :
   a) Exécution pour tous, seul le propriétaire a le droit de modifier et de lire.
   b) Lecture et exécution pour tous, écriture pour personne.
   c) Toutes les permissions pour tous, sauf écriture pour les autres.
   d) La permission de lire et d'écrire pour le propriétaire et d'exécuter pour le groupe ; aucune permission pour les autres.

--- * ---

Exercice 7 : Permissions associées aux fichiers normaux

1. Créer deux fichiers `f` et `g`, et entrer quelques mots dans chacun d'eux, par exemple avec les commandes

   ```
   $ echo 'fichier f' >f; echo 'fichier g' >g
   ```
2. Pour vous, enlever la permission de lire dans le fichier `f` et la permission d'écrire dans `g`.
3. Tester les commande `cat f` et `cat g`.
4. Essayer de modifier `g` avec un éditeur de texte.
5. Tester les commandes `cp f h` puis `cp g h`. Voir le contenu de `h` et les permissions associées au fichier `h`.
6. La commande `echo 'salut' >>f` permet d'écrire la chaîne `salut` à la fin du fichier `f` (plus de détails dans un autre TP). La tester, puis vous redonner les droits en lecture sur `f` et regarder le contenu de `f` avec `cat`.

7. Tester la commande `rm g`, (taper `n` pour refuser). Enfin, tester la commande `rm -f g`.

--- * ---

Exercice 8 : Permissions associées aux répertoires

Schématiquement, un répertoire est une liste de noms de fichiers associés à un indice appelé *numéro d'inode* permettant de connaître les informations (contenues dans l'*inode*) concernant ce fichier (taille, permissions, horodatage, où trouver le contenu du fichier, ...). Sur un répertoire,

- la permission `r` est nécessaire pour consulter cette table, donc pour lister le contenu du répertoire ;
- la permission `w` est nécessaire pour modifier cette table donc de créer ou de supprimer des fichiers ;
- la permission `x` (traverser) est de loin la plus importante : elle permet de consulter l'inode associé au numéro d'inode et donc d'*ouvrir* le fichier associé à ce numéro d'inode (appel système `open`) en lecture et/ou en écriture ou de modifier cet inode (supprimer le fichier par exemple). On ne peut presque rien faire sans.

1. Créer un répertoire nommé `rep` et deux fichiers normaux `a` et `b` dans ce répertoire.

2. Retirez toutes les permissions associées à `rep`, puis essayer de

a) faire de `rep` votre répertoire courant ;
b) lister le contenu du répertoire `rep` ;
c) afficher le contenu du fichier `a` ;
d) créer un nouveau fichier dans le répertoire `rep` ;
e) supprimer un fichier du répertoire `rep`.

3. Même question avec seulement la permission `r` sur `rep`. Essayer aussi la commande `ls -l rep` et la commande `ls -li rep` (l'option `-i` affiche le numéro d'inode).

4. Même question avec seulement la permission `w`.

5. Avec seulement la permission `x`, essayer de :

a) faire de `rep` votre répertoire de travail ;
b) lister le contenu du répertoire `rep` ;
c) modifier le fichier `a` (par exemple avec un éditeur de texte) ;
d) afficher le contenu du fichier `a` ;
e) afficher des informations sur `a` avec `ls -l rep/a` ;
f) créer ou supprimer un fichier dans le répertoire `rep`.

6. Avec les permissions `-wx` sur `rep`, essayer de :

a) créer un nouveau fichier `c` dans `rep` ;
b) renommer le fichier `b` ;
c) enlever toutes les permissions sur le fichier `c` ;
d) supprimer le fichier `c`.

--- * ---

4 `PATH` et shebang

Exercice 9 : La variable `PATH`

Cet exercice de type « expérimentation » est délicat et important. Il faut le traiter avec un soin particulier et en prenant son temps.

1. Afficher le contenu de la variable `PATH`.

2. Créer un répertoire `bin` dans votre répertoire personnel et entrer les commandes suivantes

```
$ PATH=~/bin:$PATH
$ echo $PATH
```

3. À l'aide de la commande `type`, chercher les chemin absolus des programmes `cat` et `rm` et les noter.

4. Faire une copie de `cat` dans `~/bin` en le renommant `rm`.

5. Créer un fichier `fic`, y mettre quelques caractères et créer deux copies `fic2` et `fic3` de `fic`.

6. Essayer de détruire `fic` avec la commande `rm`. Que s'est-il passé ?

7. Entrer la commande `type rm`.

8. Lancer la commande

```
$ <chemin vers rm> fic
```

en remplaçant `<chemin vers rm>` par le chemin absolu vers la commande `rm` noté à la question 3. Que s'est-il passé ?

9. Enlever la permission `x` sur le fichier `~/bin/rm` et essayer de supprimer `fic2`.

10. Demander au shell d'oublier les emplacements enregistrés (« hachés ») avec la commande `hash -r`, puis entrer les commandes

```
$ type rm
$ rm fic2
```

11. Remettre la permission `x` sur `~/bin/rm` puis entrer les commandes suivantes (où `<chemin vers rm>` désigne le chemin absolu noté à la question 3) :

```
$ ~/bin/rm fic3
$ cd ~/bin
$ ./rm fic3
$ <chemin vers rm> rm
$ rm fic3
```

12. Faire le bilan de cet exercice en répondant aux questions suivantes :

a) Que contient la variable `PATH` ?

b) Dans quel cas est-ce qu'un nom de commande est cherché dans les répertoires du `PATH` ?

c) S'il y a plusieurs programmes correspondants dans les répertoires du `PATH`, lequel est choisi ?

--- * ---

Exercice 10 : Le shebang

On appelle *shebang*, contraction de *sharp* (dièse, `#`) et *bang* (`!`), la séquence de deux caractères `#!` en *début de fichier*.

Pour un fichier exécutable, elle indique au noyau que le programme qui suit le shebang doit être exécuté avec comme argument suivant le présent fichier.

1. Premier exemple avec `bc`.

Le shell ne sait pas calculer avec les nombres à virgule et même avec les nombres entiers, il n'est pas très rapide et c'est normal, il n'est pas fait pour ça. Pour les calculs, Unix a un programme spécialisé appelé `bc` (pour *basic calculator*). De plus, `bc` permet des calculs de précision arbitraire (c'est-à-dire aussi grande qu'on veut).

a) Premier contact : entrer la commande suivante

```
$ bc -lq
```

L'option `-l` demande de charger les fonctions mathématiques et de faire les calculs avec des nombres non entiers, l'option `-q` (pour *quiet*) de GNU `bc` enlève un message inutile au démarrage.

Ensuite, entrer les expressions suivantes :

```
2^32
2^64
123456/7
e(1)
```

Pour quitter `bc`, entrer Ctrl+D.

b) Éditer un nouveau fichier appelé `pi_e.bc` en y écrivant le contenu suivant :

```
scale = 50
"Voici une valeur approchée de pi :
"
4 * a(1)
"Et en voici une de e :
"
e(1)
quit
```

Ce fichier est un *script* en langage `bc` (oui, en fait, `bc` est un langage de programmation complet).

c) On va demander à `bc` *d'interpréter* ce fichier, c'est-à-dire de le lire et d'afficher les résultats dans le terminal. Pour ce faire, on entre la commande :

```
$ bc -lq pi_e.bc
```

ou bien (sans recherche de commande externe) :

```
$ /bin/bc -lq pi_e.bc
```

Maintenant, on aimerait bien faire de `pi_e.bc` un programme autonome et ne pas avoir à se souvenir qu'il faut le faire interpréter par `bc -lq`.

d) Rendre le fichier `pi_e.bc` exécutable pour vous (l'utilisateur propriétaire).

e) Éditer le fichier `pi_e.bc` en y insérant comme première ligne :

```
#!/bin/bc -lq
```

f) Enfin lancer la commande

```
$ ./pi_e.bc
```

g) Il n'est pas nécessaire que l'utilisateur sache que ce programme est un script `bc`. Le renommer `pi_e` (sans extension) et tester la commande

```
$ ./pi_e
```

2. Deuxième exemple. Dans un fichier nommé `hello`, écrire les deux lignes suivantes :

```
#!/bin/cat
Hello, world!
```

puis entrer les commandes suivantes et commenter.

```
$ cat hello
$ /bin/cat hello
$ ./hello
$ chmod u+x hello
$ ./hello
```

3. Troisième exemple (pour rigoler). Dans un fichier nommé `autodestructeur`, écrire les deux lignes suivantes :

```
#!/bin/rm
Ce message s'autodétruira.
```

Rendre le fichier exécutable et l'exécuter. Que s'est-il passé ?

4. Dans un fichier nommé `mon_super_script` taper les lignes suivantes :

```
cal
who
logname
```

puis entrer la commande

```
$ sh mon_super_script
```

5. Faire en sorte de pouvoir lancer `mon_super_script` de la façon suivante

```
$ ./mon_super_script
```

Pourquoi commençons-nous nos scripts par `#!/bin/sh` ?

--- * ---

Exercice 11 : Un répertoire pour ses programmes avec configuration du PATH

1. Créez un répertoire `bin` dans votre répertoire personnel. Vous y mettrez les exécutables que vous trouvez utile.
2. À l'aide d'un éditeur, ajouter dans votre fichier `~/.bashrc`, à la fin, la ligne
   ```
   PATH=~/bin:$PATH
   ```
3. À quoi sert cette ligne ?
4. Copier votre script `infos` (exercice 2) et votre script `pi_e` (exercice 10) dans votre répertoire `~/bin`.
5. Ouvrez un nouveau terminal (ou sourcez votre `.bashrc`), et tapez simplement les commande
   ```
   $ infos
   $ pi_e
   ```
   Que se passe-t-il et pourquoi ?

--- * ---