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 ?

--- * ---

Correction de l'exercice 1 : Correction globale.
Les seuls caractères spéciaux dans une chaîne entre guillemets sont

`$` `` ` `` `"` `\`

Les développements de variable, arithmétique (`$((...))`) et la substitution de commande (`$(...)` ou `` `...` ``) ont encore lieu. Bien sûr le guillemet est spécial car il met fin à l'inhibition. On peut rendre à ces caractères leur sens littéral en les précédant d'une contre-oblique. Laquelle est donc aussi spéciale, mais seulement lorsqu'elle précède l'un de ces quatre caractères spéciaux.

Les développements du tilde et de noms de chemins n'ont jamais lieu dans une chaîne entre guillemets.

Il y a plusieurs mécanismes d'inhibition pour pouvoir contrôler à quel point le shell met son nez dans nos affaires. De plus (sens du dernier exemple), il fallait trouver une façon d'inhiber le mécanisme (subtil) de la *séparation en champs* (*field splitting*) des résultats des développements de variable, arithmétique et substitution de commande.

--- * ---

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.

--- * ---

Correction de l'exercice 2 : Pour rendre exécutable et exécuter le script (si l'on se trouve dans le répertoire où il est présent) :

```
$ chmod u+x infos
$ ./infos
```

Mon script complété ressemble à :

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

echo "Bonjour $USER !
Votre répertoire courant est $HOME.
Le répertoire courant est $PWD.
Avant, vous étiez dans $OLDPWD.
Votre shell par défaut est $SHELL.
Votre éditeur de texte par défaut est $EDITOR.
Votre langue est $LANG.
Votre path est $PATH.
Date et heure : $(date)
Mois en cours : $(cal)
"
```

--- * ---

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`.

--- * ---

Correction de l'exercice 3 : Les seuls retours à la ligne que l'on obtient avec `printf` sont ceux demandés explicitement, ce qui n'est pas le cas avec `echo` qui termine toujours son affichage par un retour à la ligne (petite remarque au passage : c'est à cause de ça qu'ont circulé plusieurs versions de `echo` et qu'il est maintenant presque inutilisable).

Les tabulations sont interprétées comme l'insertion de blancs jusqu'au prochain multiple de leur largeur. Visiblement, au départ la largeur des tabulations est 8 (ceci peut être modifié par la commande standard `tabs`). Petite remarque : `bash` interprète le caractère tabulation comme une demande d'auto-complétion. Si l'on veut en insérer un manuellement, il faut taper la séquence `C-v` suivie d'une tabulation.

Il vaut vraiment mieux que la contre-oblique garde son sens littéral dans le format, donc sauf besoins particuliers (développement de variable par exemple), mettre le format entre apostrophes.

Les caractères affichés (ou entendus) sont ceux qui correspondent aux codes ascii (en octal) écrits dans le format. Remarque : ce TP est l'occasion d'insister un peu sur les bases de numération.

--- * ---

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
```

--- * ---

Correction de l'exercice 4 :

1. a) Il y a deux arguments qui suivent le format, la chaîne `"3"` est interprétée comme un entier (signé) et la chaîne `'petits cochons'` comme une unique chaîne (avec un espace littéral). Le premier champ en sortie est `'  3'` car la largeur minimale du champ est 3.

b) Le format est réutilisé (3 fois) pour pouvoir épuiser la liste des arguments, qui sont donc écrits chacun sur une ligne.

c) Il y a deux commandes, séparées par un `;`. Dans le premier `printf`, le format est utilisé 9 fois.

d) Le shell commence par développer les noms de chemins (donne quelque chose comme 4 ou 5 noms de chemins) qui sont autant d'arguments pour la commande. Chacun de ces chemins est affiché seul sur sa ligne par `printf`.

2. Les caractères `u o x X` correspondent à un affichage en bases respectivent 10, 8, 16 (avec chiffre supplémentaires `a, b, ..., f`) et 16 (avec chiffres supplémentaires `A, B, ..., F`). Les arguments numériques peuvent être écrits sous la forme

- `0d...` sous forme octale (chiffres `0, 1, ..., 7`) ;
- `0xd...` sous forme hexadécimale (chiffres `0, 1, ..., 9, a, ..., f`) ;
- `0Xd...` sous forme hexadécimale (chiffres `0, 1, ..., 9, A, ..., F`).

Le drapeau `#` fait que `printf` préfixe l'affichage sous forme octale par un `0` et sous forme hexadécimale par `0x` ou `0X`.

3. Le caractère spécificateur `b` fait que les séquences d'échappement *dans l'argument correspondant* sont également interprétées par `printf`. C'est évidemment inutile en C car les séquences d'échappement font partie du langage.

4. Dans l'ordre :

```
'%d %s\n' # ou '%4d %s\n'
'%d %10s\n'
'%5d|%20s|%-4d|%s\n'
'la chaîne '\''%s'\'' se traduit par :\n%b'
```

--- * ---

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 ?

--- * ---

Correction de l'exercice 5 :

1. Le résultat dépend du masque utilisateur (`umask` qu'on n'abordera pas cette année), mais fréquemment on trouve `-rw-r--r--` pour un fichier normal et `drwxr-xr-x` pour un répertoire.

Pour le fichier normal, le propriétaire a donc la permission de lire (`r` pour *read*), d'écrire (`w` pour *write*) ; le groupe et les autres ont le droit de le lire.

Pour le répertoire (qu'on repère au `d`, pour *directory*, avant les permissions), l'utilisateur a le droit de le *traverser* (`x` pour *cross*, on dit aussi *search*), de le lire (par exemple avec `ls`) et de modifier son

contenu (supprimer ou créer des fichiers dedans). Le groupe et les autres ont le droit de le traverser et de le lire.

2. Les fichiers `a` et `b` sont des répertoires, les autres des fichiers normaux. Tout le monde peut lire et traverser le répertoire `a` mais seul son propriétaire peut le modifier. Le propriétaire du répertoire `b` peut le lire et le traverser mais pas le modifier. Les autres utilisateurs peuvent seulement le lire.

Le fichier `c.txt` est lisible par tous et modifiable par son propriétaire. Le fichier `d.c` est seulement modifiable par l'utilisateur et le groupe et seulement consultable par les autres.

Enfin, le fichier `op` peut être lu et exécuté par tous et peut être modifié par son propriétaire.

3. Sur cette machine, j'ai (en enlevant les colonnes qui ne m'intéressent pas pour le moment) :

```
$ ls -ld /bin/ls ~ /usr/include/stdio.h /
drwxr-xr-x root      root  /
-rwxr-xr-x root      root  /bin/ls
drwx------ rousselin users /users/rousselin
-rw-r--r-- root      root  /usr/include/stdio.h
```

Tous les utilisateurs peuvent donc lire et traverser `/` (heureusement !) mais seul `root`, le super-utilisateur, peut le modifier. Tous les utilisateurs peuvent lire et exécuter le programme `/bin/ls` (vous pouvez essayer `more /bin/ls`), seul `root` peut le modifier. Seul moi peut faire quoi que ce soit avec mon répertoire personnel.

Enfin, tout le monde peut lire `stdio.h` et seul `root` peut le modifier.

Dernière remarque : ne jamais oublier que `root` peut de toute façon *tout faire*.

--- * ---

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.

--- * ---

Correction de l'exercice 6 :

1. La syntaxe est : l'une des lettre `augo` (all, user, group, others) suivi d'un des caractères `=+-` (affecter les permissions, ajouters des permissions, retirer des permissions) suivi d'une partie de `rwx` (qui peut être vide) ou bien, d'un caractère parmi `ugo`, pour égaler, ajouter ou supprimer les permissions associées à l'une de ces catégories d'utilisateurs.

Enfin, on peut juxtaposer ces modifications en les séparant par une virgule (sans espace, car cela doit être un seul argument).

2. Par exemple :

a) `chmod a=x,u+rw f`
b) `chmod a=rx f`
c) `chmod a=rwx,o-w f`
d) `chmod u=rw,g=x,o= f`

--- * ---

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`.

--- * ---

Correction de l'exercice 7 : Correction globale.

```
$ echo 'fichier f' >>f; echo 'fichier g' >>g
$ chmod u-r f; chmod u-w g; ls -l f g
--w-r--r-- 1 rousselin users 10 mars   6 16:57 f
-r--r--r-- 1 rousselin users 10 mars   6 16:57 g
$ cat f
cat: f: Permission non accordée
$ cat g
fichier g
$ echo 'truc' >>g
bash: g: Permission non accordée
$ cp f h
cp: impossible d'ouvrir f en lecture: Permission non accordée
$ cp g h
$ cat h
fichier g
$ ls -l h
-r--r--r-- 1 rousselin users 10 mars   6 16:58 h
$ echo 'salut' >>f
$ chmod u+r f
$ cat f
fichier f
salut
$ rm g
rm : supprimer fichier (protégé en écriture) g ? n
$ rm -f g
$ ls -l g
ls: impossible d'accéder à g: Aucun fichier ou dossier de ce type
```

On insiste sur le fait qu'un fichier protégé en écriture *peut être supprimé*.

--- * ---

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`.

--- * ---

Correction de l'exercice 8 :

1. `mkdir rep; touch rep/a rep/b`

2. `chmod a= rep`

On ne peut rien faire de tout ça.

3. Avec la permission `r`, on peut lister le contenu de `rep` et c'est tout. Très peu d'informations sont données par `ls -l` et `ls -li` (mais tout de même le numéro d'inode, ce qui signifie bien qu'on peut consulter le tableau `nom <-> numéro d'inode` mais pas les inodes eux-mêmes).

4. Avec `w` tout seul, on ne peut rien faire du tout.

5. Avec `x` tout seul, on peut `cd` dans le répertoire, lire et modifier les fichiers qu'il contient (à condition d'avoir les permissions adéquates sur ces fichiers) et lire les inodes associés à ces fichiers (appel système `stat`), *bien sûr, à condition de connaître à l'avance le nom de ces fichiers*. On ne peut pas créer ou supprimer de fichier.
6. Avec `-wx` on peut tout faire, sauf lister le contenu du répertoire. On insiste encore sur le fait que pour supprimer un fichier, il suffit d'avoir les permissions `-wx` sur le répertoire qui le contient, peu importent les permissions associées au fichier !

--- * ---

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 ?

--- * ---

Correction de l'exercice 9 :

1. `echo $PATH` ou `printf '%s\n' $PATH`
2. C'est assez standard d'avoir un répertoire `bin` dans son répertoire personnel et de l'ajouter dans le `PATH`, pour y mettre ses propres scripts et programmes.
3. Quelque chose comme `/bin/rm` ou `/usr/bin/rm` (dépend du système).
4. `cp /bin/cat ~/bin/rm`
5. Utiliser un éditeur de texte, ou bien
   ```
   $ echo 'salut' >fic; cp fic fic2; cp fic fic3
   ```
6. Le contenu de `fic` est affiché à l'écran et il n'est pas détruit.
7. `type rm` affiche maintenant le chemin `~/bin/rm`.
8. `fic` a bien été supprimé.
9. Le shell affiche un message d'erreur (permissions insuffisantes).
10. En demandant au shell d'oublier les emplacements enregistrés, il va rechercher de nouveau dans les répertoires du `PATH` un fichier *exécutable* qui s'appelle `rm`. Comme `~/bin/rm` ne l'est plus, il va trouver `/bin/rm`, qui va donc supprimer `fic2`.
11. Si un nom de commande contient une oblique, le `PATH` n'est pas utilisé, et le nom de commande est interprété comme un chemin vers un fichier qui contient le programme qui sera lancé. Ainsi, la commande `~/bin/rm fic3` lance le programmme `~/bin/rm` (qui est une copie de `cat`). De même pour `./rm` avec un chemin relatif si l'on est dans le répertoire `~/bin`.

    En revanche, `rm` ne contient pas d'oblique donc le shell cherche dans le `PATH`. Comme `~/bin/rm` n'existe plus, c'est `/bin/rm` (ou `/usr/bin/rm`, dépend du système) qui est trouvé et `fic3` est supprimé.
12. La variable `PATH` contient des chemins de répertoires séparés par le caractère `':'`. Lorsqu'un nom de commande ne contient pas de caractère oblique, n'est pas « haché » (c'est-à-dire n'a pas été mémorisé par le shell), et n'est ni une primitive du shell ni une fonction, le shell recherche, *dans l'ordre*, parmi les répertoires du `PATH`, un fichier *exécutable* de même nom que la commande, puis l'exécute.

--- * ---

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` ?

--- * ---

Correction de l'exercice 10 :

1.

2. La première commande affiche comme prévu le contenu du fichier `hello`. La seconde échoue car le fichier n'est pas encore exécutable. La quatrième réussit et donne un résultat identique à la première. La cinquième échoue probablement car il n'y a pas d'exécutable `hello` dans les répertoires du `PATH`.

3. On commence par entrer les commandes

```
$ chmod u+x autodestructeur
$ ./autodestructeur
```

On constate que le fichier a disparu... En effet c'est comme si on avait entré la commande

```
$ /bin/rm autodestructeur
```

4. La commande `sh mon_super_script` lance à la suite les commandes présentes dans le fichier. En effet, le shell `sh` lit le fichier ligne par ligne comme si nous les écrivions nous-même.

5. Pour éviter que les utilisateurs soient obligés de préciser eux-mêmes que c'est `sh` qui doit interpréter notre script. L'utilisateur ne devrait pas se poser ce genre de questions. De plus, ces shebangs permettent de pouvoir traiter les scripts (peu importe le langage dans lequel ils sont écrits) comme n'importe quel exécutable.

--- * ---

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 ?

--- * ---