S. Lavirotte
Polytech'Nice – Sophia
SI 3
2014-2015

TD séance n° 18
Introduction à la Programmation Shell Unix 4

Nous avons réalisé lors de la dernière séance des scripts qui commencent à avoir une certaine taille, même si celle-ci est encore très raisonnable. Mais dans le cas de scripts plus complexes, il devient vite nécessaire de regrouper les instructions pour pouvoir réutiliser certains traitements. Comme dans les langages de programmation classiques, nous pouvons alors avoir recours à des fonctions[1].

Vous savez qu'un script Shell n'est rien d'autre qu'une série de commandes. Mais parfois cela pose des problèmes, car lorsqu'un script devient un peu long et surtout lorsqu'il est obligé de se répéter, les risques de bogues (dysfonctionnements) augmentent.

L'usage des fonctions permet alors:

- d'éviter ces répétitions ;
- de diminuer les risques de bogues ;
- d'augmenter la lisibilité du script pour un humain.

1 Fonctions : regrouper des commandes

Pour exécuter une suite de commandes sur une seule ligne, on peut les placer les unes à la suite des autres en les séparant par des points-virgules. Dans un script, cela n'a que peu d'intérêt. Mais il est possible de les entourer par des accolades { }. Alors cela sera considéré comme un sous-script et l'expression totale vaudra le code de retour de la dernière commande. C'est ce qui est utilisé pour regrouper des commandes mais une fonction possède un nom qui permet de faire appel à ce groupe de commandes, voir à lui passer des valeurs en paramètre.

L'utilisation des fonctions se fait en deux temps :

- d'abord, il faut définir la fonction : vous décrivez quelle série de commandes il faudra exécuter lorsque l'on appellera la fonction ;
- ensuite, il faut appeler la fonction à chaque endroit que vous voulez.

1.1 Définir une fonction

Définir une fonction est extrêmement simple :

- On commence par trouver un nom à la fonction. Vous pouvez choisir ce nom à votre guise (il doit commencer par une lettre), par exemple `liste_des_fichiers`, `effacer_fichier`, etc. Il est toutefois fortement recommandé :
  - de ne pas donner à ses fonctions des noms de commandes existant déjà, par exemple `ls`, `grep`, etc. Cela pourrait en effet poser de graves problèmes, car les fonctions définies dans un programme sont prioritaires sur les commandes intégrées du shell (`cd`, `alias`, etc.) et les commandes externes (`thunderbird`, `mozilla`, etc.). Le comportement devient difficilement prévisible, et surtout, le script sera très difficile à débuguer…
  - de ne pas utiliser de signes de ponctuation, d'espaces ni de caractères accentués dans les noms de fonction ;
- une fois que vous avez donné un nom à la fonction, notez l'utilisation des parenthèses ouvrantes et fermantes : ce sont elles qui indiquent à l'interpréteur du script qu'il s'agit de la définition d'une fonction ;
- enfin, entre accolades, notez la série des instructions qu'il faudra exécuter à chaque appel de la fonction.

```
function ma_fonction ()
{
 #Liste des commandes
}
```

[1] *Ce cours/TD a été réalisé en réutilisant des données issues de plusieurs. Nous tenons donc à remercier Tian et Baptiste Mélès pour leurs supports de cours.*

TD séance n° 18
Introduction à la Programmation Shell Unix 4

Le mot clé `function` n'est pas obligatoire, mais pour la lisibilité, il est recommandé de l'utiliser

1.2 Utiliser une fonction

Pour utiliser une fonction dans un script, il suffit de l'appeler par son nom :

```
ma_fonction
```

Sachez enfin que des fonctions peuvent appeler d'autres fonctions, ce qui donne une extrême souplesse à leur utilisation.

1.3 Paramètres d'une fonction

Tout l'intérêt d'une fonction est dans le fait de pouvoir lui passer des valeurs en paramètre sur lesquelles appliquer l'algorithme qui est dans la fonction.

Les variables passées en paramètre lors de l'appel à la fonction sont accessibles par les variables positionnelles. Cela se passe de la même manière que pour accéder aux variables du script avec `$1`, `$2`, ... mais aussi `$*` et `$#`. Par conséquent, si une fonction à l'intérieur d'un script doit accéder à un des paramètres du script, il faut lui passer explicitement lors de l'appel car elle ne les voit pas directement.

Une fonction est terminée lorsqu'il y a une accolade fermante. Mais parfois on souhaite en sortir avant, à la suite d'un test par exemple. Pour cela on peut utiliser la commande `return`. On peut lui passer une valeur. Ce sera le code de retour de la fonction qui peut être utilisé directement pour un test ou dans une boucle, ou alors par la variable `$?`. Comme pour `exit`, le code de retour sera en l'absence de cette valeur celui de la dernière commande exécutée.

```
function ma_fonction ()
{
    param1=$1
    echo $param1
    return 5
}

ma_fonction 10
echo $?
```

2 Conclusions

De là, passons à un conseil de programmation : abusez des fonctions ! N'hésitez pas à créer des fonctions pour tout et n'importe quoi. Vous en tirerez :

- un gain de lisibilité ;
- un gain d'efficacité ;
- un gain de souplesse.

2.1 Gain de lisibilité

Un programme sans fonction n'est lisible que s'il est très petit : une vingtaine de lignes tout au plus. Dès qu'un programme dépasse cette taille, il cesse d'être intelligible d'un seul coup d'œil pour un humain.

Supposons qu'un programme soit amené à répéter n fois un même fragment de code comportant p lignes ; en utilisant des fonctions on économise (n - 1) * p lignes (toutes les occurrences de la répétition, moins la définition de la fonction).

Ce gain de lignes est indissociable d'un gain de lisibilité, car les répétitions fatiguent inutilement le cerveau humain.

De plus, choisissez bien vos noms de fonctions, cela facilitera aussi la lisibilité du code.

2.2 Gain d'efficacité

En recopiant « à la main » des fragments identiques de codes, vous risquez toujours la faute de frappe. Or, la moindre coquille peut avoir des conséquences, au mieux imprévisibles, au pire catastrophiques.

En isolant les séries d'instructions dans des définitions de fonctions, vous concentrez en un seul endroit la situation possible d'un bogue donné. Vous pouvez circonscrire le bogue.

2.3 Gain de souplesse

En utilisant des fonctions, vous donnez à vos programmes une grande souplesse. Si vous voulez apporter une modification d'ensemble à un programme, vous n'avez plus à corriger autant de morceaux du script qu'il y a d'occurrences du même fragment : vous apportez vos modifications de manière centralisée.

Pour décrire ce phénomène, on parle souvent de modularité, ou encore d'encapsulation. Il s'agit en effet de privilégier l'assemblage simple d'éléments simples à l'assemblage complexe d'éléments complexes. Ainsi, chaque fonction cache aux fonctions qui l'appellent la complexité de son travail, pour leur offrir une interface simple : le travail est divisé en autant de petites tâches que nécessaires, au lieu d'une seule tâche gigantesque et labyrinthesque.

Pour toutes ces raisons, n'hésitez surtout pas à créer des fonctions et à les emboîter entre elles. La programmation vous paraîtra de plus en plus facile, à mesure que vous réaliserez des tâches de plus en plus complexes.

3 Complément sur les variables d'environnement

Vous avez appris à manipuler les variables dans vos scripts Shell. Toutefois, nous avons vu dans le premier TD qu'il existe plusieurs types de variables. Tout d'abord, les variables utilisateur que vous avez créées et manipulées dans vos scripts.

Mais il y a aussi des variables prédéfinies que vous pouvez utiliser dans vos scripts ou depuis le terminal. Vous avez lors du dernier exercice du premier TD, vous avez utilisé par exemple les variables USER et HOSTNAME qui contiennent respectivement le nom de l'utilisateur connecté et le nom de la machine.

Vous avez une autre variable très utile qui est la variable `PATH`. Cette variable contient l'ensemble des chemins dans lesquels le Shell va chercher où trouver les programmes que vous tentez de lancer.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

Si le chemin où se trouve le programme n'est pas spécifié dans la variable `PATH`, celui-ci ne sera pas trouvé et donc vous obtiendrez une erreur comme quoi la commande est introuvable.

Vous remarquerez que le répertoire courant n'est pas présent dans la variable `PATH`. Pour des raisons de sécurité, cela est préférable ainsi. Mais cela explique aussi pourquoi, quand vous tentez de lancer un de vos script Shell, vous êtes obligés de spécifier l'endroit où il se trouve en ajoutant `./` devant le nom du script (ou bien le chemin absolu).

Pour éviter cela, soit vous pouvez ajouter le chemin courant à votre `PATH` (ce n'est pas recommandé) :

```
$ PATH=$PATH:.
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:.
```

soit vous pouvez ajouter vos scripts dans un des chemins du `PATH` (par exemple `/usr/local/bin`).

TD séance n° 18
Introduction à la Programmation Shell Unix 4

4 Exercices

Nous avons vu un des intérêts du scripting Shell la semaine dernière qui est de pouvoir enchainer des commandes simples avec les redirections. L'autre intérêt majeur des scripts Shell est pouvoir simplement mettre en place des traitements pas lot sur des fichiers. Nous allons utiliser cette semaine tout ce que l'on a appris lors de la programmation Shell pour pouvoir mettre en place de tels traitements sur de potentiellement très gros volumes de données.

Exercice n°1:

Ecrire un script `ma_fct.sh` qui prend *n* fichier en paramètres. Ce script contiendra une fonction qui vous demande si vous souhaitez remplacer un fichier, qui lit la réponse et vous retourne la valeur `0` si l'utilisateur a répondu oui (avec o ou O), sinon renvoie `1`. Cette fonction sera appelée pour chacun des fichiers passés en paramètre du script.

```
$ ma_fct.sh mon_fichier.txt fichier.txt file.txt
Voulez-vous vraiment remplacer mon_fichier.txt ? (o/n) o
Voulez-vous vraiment remplacer fichier.txt ? (o/n) n
Voulez-vous vraiment remplacer file.txt ? (o/n) o
```

Exercice n°2:

Ecrire un script `tri_photos.sh` qui va prendre `2` arguments : le nom d'un dossier contenant des photos `jpg` (vous utiliserez le dossier contenant une cinquantaine de photos que vous avez récupérées dans le fichier de ressource pour ce TD) et le nom d'un dossier dans lequel il va recopier ces photos en les triant et les renommant.

Voici quelques précisions pour créer votre script.

1. Installez la commande `exif` sur votre machine. Cette commande vous permettra de récupérer les méta-données qui se trouvent dans un fichier image `.jpg` comme la date de prise de vue ou la date de création (mais bien d'autre encore comme l'ouverture, la focale, …). Dites dans votre compte-rendu la commande que vous avez utilisé pour installer le paquetage `exif`.
2. Vous commencerez par récupérer la date de numérisation de l'image (tag `0x9003`) et si celui-ci n'existe pas (c'est le cas pour certaines photos de l'exemple), vous prendrez alors la date « simple » (tag `0x0132`)
3. Après avoir extrait dans des variables la date (année, mois, jour), l'heure (heure, minute, seconde) et l'année, vous utiliserez celles-ci pour :
    a. Créer un dossier correspondant à l'année (dans le dossier de destination passé en paramètre `2`) si celui-ci n'existe pas déjà.
    b. Créer une copie de l'image dans ce nouveau dossier en la renommant avec un nom du type `anne-mois-jour_heure-minute_seconde.jpg`
    c. Imprimer un message d'erreur sur la sortie standard d'erreur dans le cas ou un fichier de ce nom existe déjà dans le dossier de destination.

Exercice n°3:

Utilisez la fonction faite dans l'exercice `1` pour demander dynamiquement à l'utilisateur dans le programme fait dans l'exercice `2` s'il veut remplacer le fichier s'il existe.