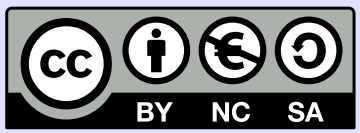
M21
Introduction à la programmation

Sami Evangelista
IUT de Villetaneuse
Département Réseaux et Télécommunications
2022–2023

http://www.lipn.univ-paris13.fr/~evangelista/cours/M21



1. Introduction

Notions d'algorithme et de programme

- **algorithme** = méthode de résolution systématique d'un problème
  - autrement dit : une suite d'instructions à réaliser afin de résoudre un problème donné dans tous les cas
- **programme** = expression d'un algorithme dans un langage de programmation
- Le rôle de la programmeuse/du programmeur :
  1. concevoir un algorithme
  2. l'exprimer dans un langage de programmation (C, bash, python)
  3. tester le programme (déboggage ou débuggage)
- Le produit de la deuxième phase est un **code source**.
- L'exécution du code source peut se faire par deux grandes approches :
  - par compilation puis exécution
  - par interprétation

Compilation

- Le code source est lu et analysé par un **compilateur**.
- Il est ensuite traduit par le compilateur dans du code binaire.
- C'est ce code binaire qui est exécuté.
- Phases de la compilation :
  1. analyse syntaxique (le code source est-il correctement formé ?)
  2. analyse sémantique (a-t-il un sens ?)
  3. génération de code

Compilation

- Le code source est lu et analysé par un **compilateur**.
- Il est ensuite traduit par le compilateur dans du code binaire.
- C'est ce code binaire qui est exécuté.
- Phases de la compilation :
  1. analyse syntaxique (le code source est-il correctement formé ?)
  2. analyse sémantique (a-t-il un sens ?)
  3. génération de code

Interprétation

- Le code source est lu, analysé et exécuté par un **interpréteur**.
- Un programme interprété est aussi appelé **script**.
- Phases de l'interprétation
  1. lire une instruction
  2. analyse syntaxique
  3. exécution de l'instruction
  4. revenir en 1

- Un programme compilé est plus rapide.
  - Le code est analysé une seule fois.
  - Le compilateur peut effectuer de nombreuses optimisations.
- Un programme compilé est (a priori) plus sûr.
  - L'étape de compilation permet de trouver des erreurs que l'interpréteur ne voit pas forcèment.
- La compilation peut prendre du temps.
- On doit compiler pour chaque OS/architecture cible (p.ex., windows 32 bit, linux 64 bit).

- Dans ce module nous programmerons en bash.
- Bash est
  - un shell (interpréteur de commandes) Unix
  - un logiciel libre
  - le shell par défaut sur les systèmes Linux
  - un langage de programmation
- Pourquoi bash (et pas un autre langage) ?
  - indispensable pour l'administration Unix/Linux
  - rapidité de développement
- Utilisation de bash
  - depuis le terminal (pour les tests) en saisissant des commandes après le message d'invite ($ dans nos exemples)
  - par interprétation d'un script : on écrit les commandes dans un fichier puis on l'exécute

2. Un premier script

Un premier script

Un script `bienvenue.sh` :

```
1 #!/bin/bash
2
3 # un script qui souhaite la bienvenue et qui donne la date
4 echo "Bonjour et bienvenue!"
5 date
```

I. 1 shebang : chemin absolu de l'interpréteur du script précédé des caractères `#!`. On doit systématiquement l'écrire en première ligne.

I. 3 commentaire. Tout ce qui suit un `#` est ignoré par l'interpréteur. On utilise ce procédé pour documenter le code source.

I. 4 la commande interne (i.e., integrée au shell) `echo` affiche un message sur la sortie standard ($\Rightarrow$ le terminal s'il n'y a pas de redirection)

I. 5 exécution de la commande externe (i.e., non integrée au shell) `date`

Un premier script — Exécution

Un script `bienvenue.sh` :

```
1 #!/bin/bash
2
3 # un script qui souhaite la bienvenue et qui donne la date
4 echo "Bonjour et bienvenue!"
5 date
```

Pour utiliser ce script :

1. (1 seule fois) rendre le script exécutable (`chmod +x`)
2. exécuter le script (`./bienvenue.sh`)

Exemple :

```
$ chmod +x bienvenue.sh
$ ./bienvenue.sh
Bonjour et bienvenue!
mer. 5 oct. 2022 16:13:25 CEST
```

Les commandes présentes dans le script sont exécutées en séquence.

3. Les variables

Les variables

- Pour produire un résultat final, un programme va procéder par étape.
- À chaque étape, le programme a en mémoire des données produites par les étapes précédentes et qui vont lui permettre d'arriver au résultat final.
- Ces données sont mémorisées dans des **variables**.
- Une variable
  - est une case de la mémoire de l'ordinateur ;
  - identifiée dans le programme par un nom ;
  - et qui contient une **valeur**.
- Règles de nommage des variables :
  - Le premier caractère doit être une lettre ou un underscore (caractère _).
  - Tout autre caractère doit être une lettre, un chiffre ou un underscore.
  - Nommage sensible à la casse : `x` et `X` n'identifient pas la même variable.
- Exemples de nommages
  - corrects : `abc`, `_x4`, `e_0`
  - et incorrects : `a-c`, `8e`, `u a`

Affecter une valeur à une variable

- **affectation** = opération de mémorisation d'une valeur dans une variable
- Pour réaliser une affectation ($\Leftrightarrow$ écrire une valeur dans une variable) :

`var=valeur`

- Quand bash exécute `var=valeur` :
  - il crée la variable `var` en mémoire si elle n'existe pas ;
  - et mémorise `valeur` dans `var`.
  - ($\Rightarrow$ Si la variable existait la valeur qui était contenue dedans est effacée.)
- Par exemple, pour créer deux variables :

```
$ x=17        # crée une variable nommée x et range 17 dedans
$ m=Bonjour   # crée une variable nommée m et range Bonjour dedans
$ m=Salut     # remplace la valeur contenue dans m (Bonjour) par Salut
```

- Attention à ne pas mettre d'espace autour du = :

```
$ toto = blahblah
bash: toto : commande introuvable
```

Tout est chaîne de caractères

- Les variables en bash contiennent toujours des **chaînes de caractères** : des suites de caractères quelconques.
- Même en faisant `x=17` le contenu de `x` ne sera pas reconnu comme l'entier 17 mais comme une chaîne constituée du caractère `1` suivi du caractère `7`.
- Quand une chaîne est composée de plusieurs mots séparés par des blancs on doit la délimiter avec les caractères `"` ou `'` :

```
$ bienvenue=bonjour mon ami
bash: mon : commande introuvable
$ bienvenue="bonjour mon ami"
$ bienvenue='bonjour mon ami'
```

(Il y a une différence entre `"..."` et `'...'`. Nous la verrons plus tard.)

Lecture du contenu d'une variable (1/2)

- Pour affecter une valeur à une variable on fait donc `var=valeur`.
- Maintenant pour récupérer (lire) la valeur contenue dans `var` :

  `${var}`

- Avant d'exécuter une commande dans laquelle `${var}` apparaît, bash va substituer le mot `${var}` par la valeur contenue dans la variable `var`.
- Si la variable n'existe pas, `${var}` sera substitué par une chaîne vide (`""`).
- Exemple 1
  afficher le contenu d'une variable

```
$ bienvenue="Bonjour mon ami"
$ echo ${bienvenue}
Bonjour mon ami
```

- Exemple 2
  afficher une phrase qui dépend du contenu d'une variable

```
$ nom="Gaston Lagaffe"
$ echo "Bonjour à toi ${nom}"
Bonjour à toi Gaston Lagaffe
```

Lecture du contenu d'une variable (2/2)

- Exemple 3

  modifier le contenu d'une variable en fonction du contenu d'une autre

```
$ nom="Gaston Lagaffe"
$ message="Bonjour à toi ${nom}"
$ echo ${message}
Bonjour à toi Gaston Lagaffe
```

- Exemple 4

  modifier le contenu d'une variable en fonction de son contenu

```
$ message="Bonjour à toi"
$ message="${message} Gaston Lagaffe"
$ echo ${message}
Bonjour à toi Gaston Lagaffe
```

- Exemple 5

  appeler une commande avec des arguments contenus dans une variable

```
$ arguments="-l -a"
$ ls ${arguments}  # équivalent à "ls -l -a"
.
..
fichier.txt
cours.pdf
```

Raccourci

- Dans la plupart des cas, on peut écrire `$var` plutôt que `${var}`.
- Par exemple :

```
$ message="Bonjour à toi"
$ message="$message Gaston Lagaffe"
$ echo $message
Bonjour à toi Gaston Lagaffe
```

- Par contre dans l'exemple ci-dessous les accolades sont nécessaires :

```
$ lettres="bc"
$ echo "a$lettresd"
a
```

sinon bash tente de lire la valeur contenue dans la variable `$lettresd`.

- On doit donc écrire :

```
$ lettres="bc"
$ echo "a${lettres}d"
abcd
```

- Règle générale : si le caractère suivant `${var}` ne peut pas faire partie d'un identifiant de variable (virgule, blanc, . . .) on peut enlever les accolades.

Substitution de commandes

- Dans un script, on fait souvent appel à des commandes qui écrivent des données sur la sortie standard (par défaut, le terminal).
- Pour pouvoir manipuler ces données dans le script (p.ex., les stocker dans des variables, faire des calculs dessus), on effectue des **substitutions de commandes**.
- Syntaxe :

  `$(cmd args)`

- Avant d'exécuter une commande dans laquelle `$(cmd args)` apparaît, bash va exécuter la commande `cmd args` puis remplacer la chaîne `$(cmd args)` par la sortie standard de la commande.
- Exemple :

```
$ pwd
/home/sami
$ repertoire=$(pwd)
$ echo "le répertoire de travail est $repertoire"
le répertoire de travail est /home/sami
$ echo "il y a $(ls | wc -l) fichiers dans ce répertoire"
il y a 17 fichiers dans ce répertoire
```

Guillemets simple et double

- Bash n'analyse pas les chaînes délimitées par des guillemets simples.
- Il n'y aura en particulier pas de substitution de variable :

```
$ var="truc"
$ echo 'le contenu de var est un $var'
le contenu de var est un $var
$ echo "le contenu de var est un $var"
le contenu de var est un truc
```

ni de substitution de commande

```
$ echo 'il y a $(ls | wc -l) fichiers dans ce répertoire'
il y a $(ls | wc -l) fichiers dans ce répertoire
$ echo "il y a $(ls | wc -l) fichiers dans ce répertoire"
il y a 17 fichiers dans ce répertoire
```

Opérations arithmétiques

- On a vu que les valeurs manipulées par bash sont des chaînes de caractères.
- Pour effectuer des opérations arithmétiques on les met entre `$((` et `))`.
- Si une variable apparaît il est inutile d'ajouter le `$`.

```
$ echo 4 + 5
4 + 5
$ echo $((4 + 5))
9
$ i=10
$ echo $(((i + 7) * 2))
34
$ i=$((i * 4))
$ echo $i
40
```

- Mais bash ne sait pas faire de calcul sur les nombres réels :

```
$ echo $((7 / 2))
3
$ echo $((2.5 * 4))
bash: 2.5 * 4 : erreur de syntaxe: opérateur arithmétique non valable
```

En résumé

- une variable = une case mémoire dans laquelle on range des valeurs
- `var="une valeur"` range `une valeur` dans la variable `var`
- `${var}` est substitué par le contenu de `var`
- `$(cmd)` est substitué par la sortie standard de la commande `cmd`
- `$((op))` est substitué par le résultat de l'évaluation de l'opération
- valeur contenue dans une variable = chaîne de caractères

4. Variables spéciales

Les variables prédéfinies

- Dans un script bash, de nombreuses variables sont prédéfinies.
- Elles sont automatiquement initialisées par l'interpréteur ou par le système avant l'exécution du script.
- Ce sont principalement :
  - les variables d'environnement ;
  - les variables de processus ;
  - et les arguments du script.

Les variables d'environnement

- À sa création, un processus hérite des variables **exportées** par son père.
- Ces variables lui fournissent des informations sur son **environnement**.
- Quelques exemples :
  - `UID` = identifiant de l'utilisateur
  - `USER` = nom de l'utilisateur
  - `HOME` = chemin absolu du répertoire personnel de l'utilisateur
  - `PATH` = répertoires contenant les commandes (voir plus loin)
- pour exporter une variable : `export la_variable`
- pour afficher la liste des variables d'environnement : `printenv`
- Le processus fils travaille sur une copie des variables exportées par le père.
  $\Rightarrow$ les modifications faites par l'un ne sont pas visibles par l'autre

La variable d'environnement PATH

- Comment est interprétée une commande `toto` dans un script ?
- Le shell va d'abord regarder si `toto` est une primitive du shell, un alias, un mot-clé ou une fonction.
- Si ce n'est pas le cas il va rechercher dans tous les répertoires contenus dans la variable `PATH` un fichier exécutable (droit `x`) nommé `toto`.

```
$ echo $PATH
/home/sami/bin:/usr/local/bin:/usr/bin:/bin
```

liste des répertoires de recherche des exécutables séparés par un `:`

- Puis exécuter le premier fichier trouvé.
- Si aucun fichier trouvé $\Rightarrow$ erreur.
- pour connaître le chemin d'un exécutable du `PATH` : `which executable`

La variable d'environnement PATH — Exemples


```
$ echo $PATH
/home/sami/bin:/usr/local/bin:/usr/bin
$ which ls
/usr/bin/ls
$ ls
lang.txt  cours.pdf  bonjour.sh
$ PATH=/home/sami/bin/:/usr/local/bin
$ ls
bash: ls : commande introuvable
```

Généralement, le répertoire courant (`.`) n'est pas dans le `PATH` ⇒ utilité de rajouter `./` devant pour préciser le chemin de l'exécutable.

```
$ cat bonjour.sh
#!/bin/bash
echo "Bonjour"
$ bonjour.sh
bash: bonjour.sh : commande introuvable
$ ./bonjour.sh
Bonjour
```

Les variables de processus

- On peut connaître le PID (Process IDentifier)
  - du processus : `$$`
  - du processus parent : `$PPID`
  - du dernier processus fils lancé en tâche de fond (avec un `&`) : `$!`
- Exemples d'utilisation :
  - envoyer un signal de terminaison au fils (`kill`)
  - attendre qu'un processus fils termine (`wait`)

Les arguments du script

- Un script peut être lancé avec des arguments (comme tout exécutable).
- Dans le script on peut les récupérer grâce aux variables suivantes :
  - `$0` = chemin de l'exécutable
  - `$#` = nombre d'arguments
  - `$1`, `$2`, ... = premier argument, deuxième argument, ... (vide si non fourni)
  - `$*` = tous les arguments séparés par un espace

Exercices

1. Soit le fichier `test_args.sh` ci-dessous :

```
#!/bin/bash

echo "$0 a été appelé avec $# arguments"
echo "argument 1 = $1"
echo "argument 2 = $2"
```

1.1 Qu'affichera la commande suivante ?

```
$ ./test_args.sh un
```

1.2 Qu'affichera la commande suivante ?

```
$ ./test_args.sh un deux trois quatre
```

2. Soit le script `text_export.sh` ci-dessous :

```
#!/bin/bash

echo "Votre UID: $UID"
echo "X vaut '$X' et Y vaut '$Y'"
X=untruc
Z=blabla
export Z
```

On exécute les commandes suivantes :

```
1 $ X=machin
2 $ Y=floup
3 $ ./test_export.sh
4 $ export X
5 $ ./test_export.sh
6 $ echo $X
7 $ echo $Z
```

Quels seront les affichages produits aux lignes 3, 5, 6 et 7 ? (On supposera que l'utilisateur à l'identifiant 1000.)

5. Entrées-Sorties

La commande echo

- `echo` est une commande interne du shell.
- Elle écrit sur la sortie standard.
- Pour préserver les espaces, il est nécessaire d'utiliser des guillemets :

```
$ echo "a   b   c"
a   b   c
$ echo  a   b   c
a b c
$ var="a   b   c"
$ echo $var
a b c
$ echo "$var"
a   b   c
```

- Par défaut `echo` termine par un saut de ligne. L'option `-n` le supprime :

```
$ echo -n salut
salut$ echo salut
salut
$
```

La commande read

- `read` est une commande interne du shell.
- Elle lit une ligne sur l'entrée standard.
- La ligne lue est affectée à une ou plusieurs variables :

```
$ read x
test
$ echo $x
test
```

- Si on fournit plusieurs variables, `read` découpe la chaîne en mots puis affecte chaque mot à une variable :

```
$ read x y
ca va?
$ echo "x contient $x et y contient $y"
x contient ca et y contient va?
```

- S'il y a plus de mots que de variables, la dernière variable stocke le reste :

```
$ read a b c d
ca va bien mon ami?
$ echo $d
mon ami?
```

3. Soient un script `inverse.sh` :

```
1 #!/bin/bash
2 read X
3 read Y
4 echo $Y
5 echo $X
```

et `ecrit.sh` :

```
1 #!/bin/bash
2 echo un
3 echo deux
```

Pour chacune des commandes ci-dessous, indiquez ses effets (lignes lues au clavier ou affichées à l'écran, et fichiers obtenus).

3.1 `./inverse.sh`
3.2 `./ecrit.sh | ./inverse.sh > sortie`
3.3 `./ecrit.sh | ./inverse.sh | ./inverse.sh`
3.4 `echo truc | ./inverse.sh`
3.5 `./inverse.sh < ./inverse.sh`
3.6 `./inverse.sh < ./inverse.sh > sortie`

6. Manipulation des chaînes de caractères

Bash fournit de nombreuses opérations pour manipuler les chaînes :

- Extraction de sous-chaînes
- Suppression de sous-chaînes
- Remplacement de sous-chaînes

Extraction

`${str:i}` = sous-chaîne de `$str` qui commence au $i^{\text{ème}}$ caractère (on compte à partir de 0)

```
str=abcdef
echo ${str:3}  # def
```

Extraction

`${str:i}` = sous-chaîne de `$str` qui commence au $i^{\text{ème}}$ caractère (on compte à partir de 0)

```
str=abcdef
echo ${str:3}  # def
```

`${str:i:n}` = sous-chaîne de `$str` qui commence au $i^{\text{ème}}$ caractère et qui contient les `n` caractères suivants

```
str=abcdef
echo ${str:2:3}  # cde
```

Extraction

`${str:i}` = sous-chaîne de `$str` qui commence au $i^{\text{ème}}$ caractère (on compte à partir de 0)

```
str=abcdef
echo ${str:3}  # def
```

`${str:i:n}` = sous-chaîne de `$str` qui commence au $i^{\text{ème}}$ caractère et qui contient les `n` caractères suivants

```
str=abcdef
echo ${str:2:3}  # cde
```

Si on sort de la chaîne on a pas d'erreur mais un résultat vide ou tronqué :

```
str=abcdef
echo ${str:10}  # chaîne vide
echo ${str:2:8000}  # cdef
```

Suppression

`${x#motif}` supprime au **début** de `$x` la **plus petite** occurrence du `motif`

```
x=10.0.0.254
echo ${x#*.}      # 0.0.254
```

Suppression

`${x#motif}` supprime au **début** de `$x` la **plus petite** occurrence du `motif`

```
x=10.0.0.254
echo ${x#*.}      # 0.0.254
```

`${x##motif}` supprime au **début** de `$x` la **plus longue** occurrence du `motif`

```
echo ${x##*.}    # 254
```

Suppression

`${x#motif}` supprime au **début** de `$x` la **plus petite** occurrence du `motif`

```
x=10.0.0.254
echo ${x#*.}      # 0.0.254
```

`${x##motif}` supprime au **début** de `$x` la **plus longue** occurrence du `motif`

```
echo ${x##*.}    # 254
```

`${x%motif}` supprime à la **fin** de `$x` la **plus petite** occurrence du `motif`

```
echo ${x%.*}    # 10.0.0
```

Suppression

`${x#motif}` supprime au **début** de `$x` la **plus petite** occurrence du `motif`

```
x=10.0.0.254
echo ${x#*.}      # 0.0.254
```

`${x##motif}` supprime au **début** de `$x` la **plus longue** occurrence du `motif`

```
echo ${x##*.}    # 254
```

`${x%motif}` supprime à la **fin** de `$x` la **plus petite** occurrence du `motif`

```
echo ${x%.*}    # 10.0.0
```

`${x%%motif}` supprime à la **fin** de `$x` la **plus longue** occurrence `motif`

```
echo ${x%%.*}    # 10
```

Suppression

`${x#motif}` supprime au **début** de `$x` la **plus petite** occurrence du `motif`

```
x=10.0.0.254
echo ${x#*.}      # 0.0.254
```

`${x##motif}` supprime au **début** de `$x` la **plus longue** occurrence du `motif`

```
echo ${x##*.}    # 254
```

`${x%motif}` supprime à la **fin** de `$x` la **plus petite** occurrence du `motif`

```
echo ${x%.*}    # 10.0.0
```

`${x%%motif}` supprime à la **fin** de `$x` la **plus longue** occurrence `motif`

```
echo ${x%%.*}    # 10
```

Ces opérations n'ont aucun effet si le motif n'est pas trouvé :

```
echo ${x%%*.}    # 10.0.0.254
```

Remplacement

`${x/motif/y}` remplace dans `$x` la **première** occurrence du `motif` par `y`

```
x=10.0.0.0
echo ${x/.0/.255}          # 10.255.0.0
echo ${x/.*/.<host-id>}    # 10.<host-id>
```

Remplacement

`${x/motif/y}` remplace dans `$x` la **première** occurrence du `motif` par `y`

```
x=10.0.0.0
echo ${x/.0/.255}          # 10.255.0.0
echo ${x/.*/.<host-id>}    # 10.<host-id>
```

`${x//motif/y}` remplace dans `$x` **toutes** les occurrences du `motif` par `y`

```
echo ${x//.0/.255}          # 10.255.255.255
```

4. Soit une variable `d` contenant une date et une heure :

```
d="20221003 09:15"
```

Donner les expressions permettant de :

4.1 trouver l'année (⇒ `2022`)
4.2 trouver le mois (⇒ `10`)
4.3 trouver l'heure (⇒ `09:15`)

5. Soit une variable `adr` contenant une adresse électronique :

```
adr="gaston.lagaffe@univ-paris13.fr"
```

Donner les expressions permettant de :

5.1 trouver le nom d'utilisateur (⇒ `gaston.lagaffe`)
5.2 trouver le nom de domaine (⇒ `univ-paris13.fr`)
5.3 trouver le TLD du nom de domaine (⇒ `fr`)
5.4 remplacer le nom de domaine par `XXX` (⇒ `gaston.lagaffe@XXX`)
5.5 supprimer chaque lettre se trouvant immédiatement après un `a` (⇒ `gaton.laafe@univ-pais13.fr`).

7. Instructions conditionnelles

- Jusqu'à maintenant nos scripts contenaient uniquement des séquences d'instructions (commandes ou affectations de variables).
- Ces instructions étaient exécutées inconditionnellement (quels que soient les arguments du script, les données saisies par l'utilisateur, ...).
- Une instruction conditionnelle est exécutée seulement dans le cas où une condition qui lui est associée est vérifiée.
- Exemples de conditions :
  - la valeur de la variable `note` est $>$ à 10
  - le fichier `donnees.txt` existe
  - le code de retour d'un processus est 0
- Exemples d'instructions conditionnelles :
  - si la valeur de la variable `note` est $>$ à 10 alors afficher "tu as la moyenne"
  - si le fichier `donnees.txt` existe alors afficher son contenu

Code de retour d'un processus

- Un processus qui se termine renvoie toujours un code de retour.
- Ce code est dans l'intervalle [0..255].
  - 0 signifie que le processus a terminé avec succès.
  - Un code dans l'intervalle [1..255] indique une erreur.
- C'est un moyen pour un processus de savoir si un processus fils qui devait lui rendre un service a correctement rendu ce service.
- Le processus père peut récupérer le code du dernier processus exécuté dans la variable `$?`.
- Exemple :

```
$ touch nouveau_fichier
$ echo $?
0
$ touch /root
touch: initialisation des dates de '/root': Permission non accordée
$ echo $?
1
```

L'instruction `if-then-else`

Syntaxe :

```
if commande_test
then
   commandes_succes
else
   commandes_echec
fi
```

Signification :

- La commande `commande_test` est exécutée.
- Si son code de retour est 0 (succès), les commandes `commandes_succes` sont exécutées (et les commandes `commandes_echec` sont ignorées).
- Si son code de retour est > 0 (échec), les commandes `commandes_echec` sont exécutées (et les commandes `commandes_succes` sont ignorées).

La partie `else` est optionnelle. On alors l'instruction suivante :

```
if commande_test
then
   commandes_succes
fi
```

En cas d'échec de la `commande_test`, l'instruction n'a aucun effet.

L'indentation

- **indentation** = ajout d'espaces au début des lignes imbriquées
- L'indentation augmente la lisibilité et la compréhension du code.

$\Rightarrow$ principe à respecter **absolument**

Code non indenté

```
if cmd_test1
then
cmd1
cmd2
else
cmd3
if cmd_test2
then
cmd4
if cmd_test3
then
cmd5
fi
fi
fi
```

$\Rightarrow$

Code indenté

```
if cmd_test1
then
    cmd1
    cmd2
else
    cmd3
    if cmd_test2
    then
        cmd4
        if cmd_test3
        then
            cmd5
        fi
    fi
fi
```

La commande interne `test`

- Elle permet d'évaluer des conditions sur des chaînes de caractères, des entiers, des fichiers, ...
- Syntaxe : `test condition`
- Syntaxe abrégée : `[ condition ]`
  (avec des espaces autour des crochets !)
- Si la `condition` est remplie, la commande renvoie 0. Sinon elle renvoie 1.

Conditions sur les entiers	
`x -eq y`	égal
`x -ge y`	supérieur ou égal (greater or equal)
`x -gt y`	strictement supérieur (greater than)
`x -le y`	inférieur ou égal (less or equal)
`x -lt y`	strictement inférieur (less than)
Conditions sur les chaînes de caractères	
`-n s`	la longueur de `s` est $>$ à 0
`-z s`	la longueur de `s` est 0
`s = t`	les chaînes `s` et `t` sont identiques
`s != t`	les chaînes `s` et `t` sont différentes
Conditions sur les fichiers :	
`-e f`	`f` existe
`-d f`	`f` existe et est un répertoire
`-f f`	`f` existe et est un fichier ordinaire
`-r f`	`f` existe et est lisible
`-w f`	`f` existe et est modifiable
`-x f`	`f` existe et est exécutable

Les opérateurs `!`, `&&` et `||`

Ils permettent de combiner des conditions.

- `! cmd`
  Exécute `cmd` puis inverse son code de retour :
  - si son code est égal à 0, il devient 1 ;
  - sinon il devient 0.
- `cmd1 && cmd2`
  Exécute `cmd1` puis, si son code de retour est **égal à** 0, exécute `cmd2`.
- `cmd1 || cmd2`
  Exécute `cmd1` puis, si son code de retour est **différent de** 0, exécute `cmd2`.

La commande interne `exit`

- La commande `exit [N]` a pour effet :
  - de provoquer la terminaison immédiate du script ;
  - et d'affecter la valeur `N` au code de retour du script.
- Si `N` n'est pas fourni, le code de retour est celui de la dernière commande exécutée (soit `$?`).
- De même, si un script se termine sans rencontrer d'instruction `exit`, le code de retour sera celui de la dernière commande exécutée.

Instructions `if-then-else` imbriquées

```
if commande_test1
then
   commandes_succes1
else
   if commandes_test2
   then
      commandes_succes2
   else
      commandes_echec
   fi
fi
```

Peut être simplifié en :

```
if commande_test1
then
   commandes_succes1
elif commandes_test2
then
   commandes_succes2
else
   commandes_echec
fi
```

6. Écrire un script `compare.sh` qui prend en arguments deux entiers puis affiche sur la sortie standard un des trois mots suivants :
   - `sup` si le premier argument est strictement supérieur au second ;
   - `inf` si le premier argument est strictement inférieur au second ;
   - ou `eq` si les deux arguments sont égaux.
7. On imagine un script `dangereux.sh` qui, par sécurité, ne doit pas être exécuté par l'utilisateur root. Ce script prend un unique argument qui est le chemin d'un répertoire existant qui doit être modifiable par l'utilisateur. Donner le début du script qui effectuera les opérations de vérification nécessaires et affichera un des trois messages suivants (sur la sortie erreurs) s'il n'est pas appelé correctement :
   - *ce script ne doit pas être lancé par root*
   - *ce script doit être appelé avec un unique argument*
   - *l'argument doit être un répertoire modifiable*

   En cas d'erreur, le script terminera immédiatement avec un code de 1, 2, ou 3 selon le cas.

8. Les boucles

Les boucles

- Elles permettent de répéter l'exécution d'une (ou de) commande(s).
- Une boucle a :
  - un schéma d'itération (comment/sous quelle condition répéter) ;
  - et un contenu (les commandes à répéter).
- Nous verrons 2 types de boucles :
  - la boucle `for` — pour itérer sur une liste
  - la boucle `while` — pour itérer tant qu'une condition est vérifiée

La boucle `for` — Exemple introductif

- Soit le script suivant qui copie des fichiers dans un nouveau répertoire :

```
#!/bin/bash
mkdir repertoire_sauvegarde
cp cours.pdf repertoire_sauvegarde
echo "fichier cours.pdf copié"
cp tps.pdf repertoire_sauvegarde
echo "fichier tps.pdf copié"
cp tds.pdf repertoire_sauvegarde
echo "fichier tds.pdf copié"
cp correction.txt repertoire_sauvegarde
echo "fichier correction.txt copié"
```

- Inconvénients de ce script :
  - peu lisible
  - sujet aux erreurs (p.ex., faute de frappe dans le nom du répertoire)
  - pénible à modifier (p.ex., si on veut modifier le répertoire de sauvegarde)
- On voit qu'un même motif se répète : une commande `cp` suivie d'une commande `echo` dans lesquelles seul un nom de fichier change.

$\Rightarrow$ utiliser une boucle `for` évite d'écrire ce motif plusieurs fois

La boucle `for` — Utilisation

Syntaxe :

```
for var in liste_de_mots
do
   commandes
done
```

Signification :

- `var` prendra successivement pour valeur chaque mot de la `liste_de_mots`.
- Les `commandes` seront exécutées pour chacune de ces valeurs.
- La `liste_de_mots` peut être vide.
  - ⇒ Dans ce cas les `commandes` ne sont pas exécutées.

La boucle `for` — Exemple

Avec une boucle `for`, notre script

```
#!/bin/bash
mkdir repertoire_sauvegarde
cp cours.pdf repertoire_sauvegarde
echo "fichier cours.pdf copié"
cp tps.pdf repertoire_sauvegarde
echo "fichier tps.pdf copié"
cp tds.pdf repertoire_sauvegarde
echo "fichier tds.pdf copié"
cp correction.txt repertoire_sauvegarde
echo "fichier correction.txt copié"
```

peut donc être réécrit en :

```
#!/bin/bash
mkdir repertoire_sauvegarde
for fichier in cours.pdf tps.pdf tds.pdf correction.txt
do
   cp $fichier repertoire_sauvegarde
   echo "fichier $fichier copié"
done
```

La boucle `for` — Itérer sur des fichiers

- Dans une boucle `for`, la liste des mots à parcourir peut être un motif avec des meta-caractères (`*`, `?`, ...).
- bash va alors tenter de lui substituer les noms de fichiers correspondant.
- Exemples :

```
# parcourir tous les fichiers du répertoire courant
for fichier in *
do
    ...
done

# parcourir tous les fichiers pdf du répertoire courant
for fichier in *.pdf
do
    ...
done

# parcourir tous les fichiers du répertoire dir
for fichier in dir/*
do
    ...
done
```

La boucle `while` — Utilisation

Syntaxe :

```
while commande_test
do
   commandes_succes
done
```

Signification :

1. exécuter la `commande_test`
2. si son code de retour est 0 (succès) alors :
   - 2.1 exécuter les `commandes_succes`
   - 2.2 revenir en 1 (on boucle)
3. si son code de retour est $>$ à 0 alors sortir du `while` (fin de la boucle)

Attention aux boucles infinies : si la `commande_test` est toujours un succès, bash ne sortira jamais de la boucle.

La boucle `while` — Lecture de fichiers

Une boucle `while` permettant de lire et d'afficher un fichier ligne par ligne :

```
1 while read ligne
2 do
3     echo "$ligne"
4 done < entree
```

- ligne 4 : la redirection permet que l'entrée standard devienne le fichier `entree` pour toutes les commandes du `while` (y compris le `read`)
- ligne 1 : la commande interne `read` retourne un code de 0 quand une ligne a pu être lue et de 1 sinon (si on a atteint la fin du fichier)

  $\Rightarrow$ On sort du `while` après avoir lu toutes les lignes.

La variable IFS

- Par défaut, l'instruction `read` découpe une ligne lue sur l'entrée standard en séparant les mots avec le caractère espace ou tabulation.
- En réalité, c'est la variable du shell `IFS` (input fields separator) qui contient la liste des caractères servant de délimiteur.
- Par exemple :

```
$ read a b c
1;2;3
$ echo $a
1;2;3
$ IFS=";"
$ read a b c
1;2;3
$ echo $a
1
$ echo $b
2
```

La boucle `while` — Lecture de fichiers structurés

- La variable `IFS` est particulièrement utile lorsque l'on lit des fichiers dont les lignes contiennent des valeurs séparées par un caractère spécial.
- Soit par exemple un fichier `modules.txt` au contenu suivant :

```
M11;Anglais;30
M12;Expression;24
M13;Droit;15
```

- On peut alors utiliser la variable `IFS` pour obtenir les code, intitulé et nombre d'heures dans des variables différentes :

```
while IFS=';' read code intitule heures
do
   echo "Le module $code - $intitule fait $heures heures."
done < modules.txt
```

(En faisant `IFS=';' read ...`, la variable `IFS` est modifiée uniquement pour l'instruction `read` et pas dans le reste du script.)

- On obtient alors :

```
Le module M11 - Anglais fait 30 heures.
Le module M12 - Expression fait 24 heures.
Le module M13 - Droit fait 15 heures.
```

8. Donner une instruction `for` équivalente à la séquence suivante :

```
systemctl start sshd.service
systemctl start dhcpd.service
systemctl start httpd.service
```

   (La commande `systemctl start` permet, dans ce cas, de démarrer un service.)

9. Écrire un script `liste_utilisateurs.sh` permettant d'afficher les logins des utilisateurs ayant un UID supérieur ou égal à 1000.
   La liste des utilisateurs est définie dans le fichier `/etc/passwd`. Chaque ligne de ce fichier contient des données sur un utilisateur sous la forme de champs séparés par le caractère `:`, par exemple :

   `root:x:0:0:root:/root:/bin/bash`

   C'est le 3$^{\text{ème}}$ champ qui contient l'UID et le 1$^{\text{er}}$ qui contient le login.

9. Les fonctions

Principe et utilité des fonctions

- Une fonction peut être vue comme un sous-script à l'intérieur d'un script.
- Pourquoi utiliser des fonctions ?
  - Pour éviter de répéter des bouts de code similaires ($\Rightarrow$ permet de factoriser du code.).
  - Pour décomposer un script long ou compliqué en plusieurs "briques" de base.
- Avantages :
  - lisibilité
  - maintenance (facilité à pouvoir modifier le code pour en corriger les bugs, ajouter de nouvelles fonctionnalités, ...)

Définir et exécuter une fonction

Pour définir une fonction :

```
nom_fonction () {
   commandes
}
```

- La définition d'une fonction ne va pas exécuter la fonction.

  Le shell se contente de lire et de mémoriser la definition pour que la fonction puisse être exécutée ensuite dans le script.

- Une fonction s'exécute comme une commande "normale" :

```
nom_fonction argument1 ... argumentN
```

- Ceci exécute les `commandes` du corps de la fonction.
- L'exécution (ou **appel**) d'une fonction ne crée pas un nouveau processus.

  La fonction est exécutée dans le shell (i.e., c'est le même processus bash qui exécute le script et la fonction).

Arguments d'une fonction

- On peut appeler une fonction avec des arguments (comme une commande).
- On utilise les mêmes variables pour les récupérer : `$#`, `$*`, `$1`, `$2`, ...
  - ⇒ Les arguments du script ne sont plus accessibles pendant l'exécution d'une fonction car il sont "écrasés" par les arguments de la fonction.

```
test_args () {
    echo "nombre d'arguments : $#"
    echo "arg 1 : $1"
    echo "arg 2 : $2"
}
test_args un deux trois quatre
# affiche :
#   nombre d'arguments : 4
#   arg 1 : un
#   arg 2 : deux
```

Retourner un code avec l'instruction `return`

- Comme une commande "normale", une fonction renvoie un code de retour.
- On a vu que l'instruction `exit [N]` termine le shell avec le code `N`.
- Dans une fonction, l'instruction équivalente est `return [N]` qui :
  - provoque la terminaison de la fonction ;
  - et affecte la valeur `N` au code de retour de la fonction.

    (Si `N` est omis, le code retourné est celui de la dernière commande exécutée.)
- Comme avec une commande "normale" :
  - si l'interpréteur exécute une fonction sans rencontrer de `return`, la fonction renvoie le code de la dernière commande exécutée ;
  - le code de retour d'une fonction exécutée se trouve dans `$?`.
- *Remarque.* À l'intérieur d'une fonction la commande `exit` garde sa signification : elle termine le script appelant la fonction.

Variables locales

- On a vu qu'une fonction s'exécute dans le shell.
  - ⇒ Une fonction a accès à toutes les variables du script.
  - ⇒ Une variable créée dans une fonction est, par défaut, utilisable hors de celle-ci.
- Une variable **locale** est une variable qui n'est utilisable que dans la fonction dans laquelle elle est définie.

  (Elle se transmet aussi aux fonctions appelées par la fonction.)
  - ⇒ Une fois la fonction terminée, toutes ses variables locales sont détruites.
- Pour déclarer qu'une variable est locale :

```
local var
```

- Bonnes pratiques :
  - Déclarer les variables locales en tout début de fonction.
  - Si une variable n'est utilisée que dans une fonction, on la déclare locale.
- *Remarque.* Si une variable locale porte le même nom qu'une variable du script, cette dernière est "cachée" pendant l'exécution de la fonction.

10. Soit le script `test_vars.sh` ci-dessous :

```
1  #!/bin/bash
2  f () {
3      local x
4      echo "dans f: x = $x"
5      echo "dans f: y = $y"
6      y="salut $y"
7      x="machin"
8  }
9  x="truc"
10 y="à toi"
11 f
12 echo "après f: x = $x"
13 echo "après f: y = $y"
```

Qu'affichera-t-il une fois exécuté ?