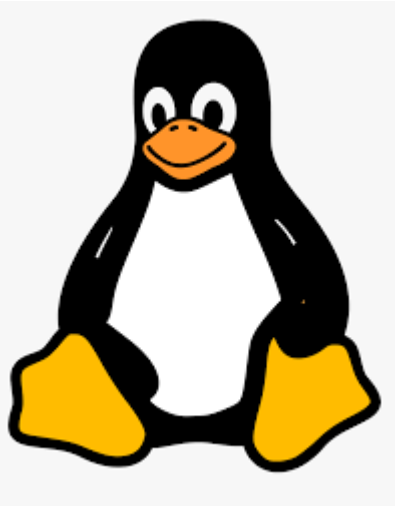
LINUX
CONCEPTS FONDAMENTAUX (2)

LES SCRIPTS



M. DRIDI

DÉPARTEMENT DISC, ESIEE PARIS

MOURAD.DRIDI@ESIEE.FR; BUREAU 4205

Plan du cours

Rappel (Commandes Linux Essentielles)

Flux & Redirection

Variables

Filtres sur les fichiers

Scripts

Rappel (Commandes Linux Essentielles)

pwd
- (Print Working Directory) Affiche le répertoire de travail actuel.

Ls
- (List) Liste les fichiers et répertoires du répertoire actuel.

cd
- (Change Directory) : Change le répertoire de travail.

Chemin Absolu / Chemin Relatif :
- Absolu : Chemin complet depuis la racine.
- Relatif : Chemin par rapport au répertoire actuel.

mkdir
- (Make Directory) :Crée un nouveau répertoire.

touch
- Crée un nouveau fichier vide ou met à jour la date de modification d'un fichier existant.

gedit
- Ouvre l'éditeur de texte Gedit.

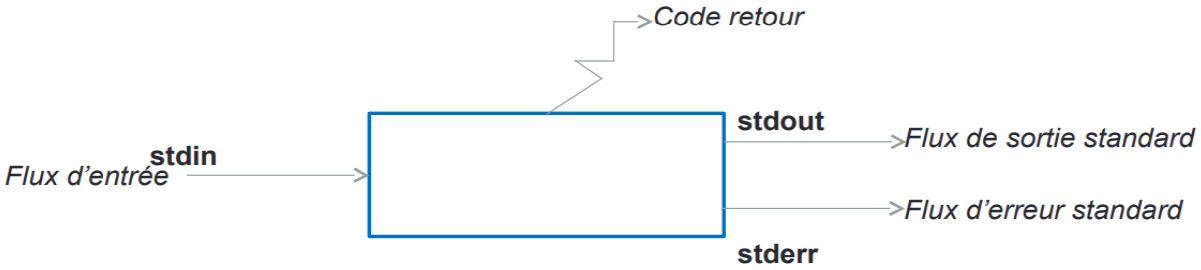
rm
- (Remove) : Supprime des fichiers.

rmdir
- (Remove Directory) : Supprime des répertoires vides.

chmod
Change les permissions d'accès aux fichiers.

- Symbolique : chmod u+r fichier (ajoute les droits de lecture pour le propriétaire).
- Numérique : chmod 755 fichier (définit les permissions 7-5-5 pour propriétaire, groupe et autres).

Processus et flux (1)



Flux usuels d'un processus interactif (shell) :

- Flux d'entrée standard (stdin) : le clavier par défaut (0)
- Flux de sortie standard (stdout) : l'écran par défaut (1)
- Flux d'erreur standard (stderr) : l'écran par défaut (2)

Un processus interactif lit les données en entrée sur **stdin** et affiche le résultat de son exécution sur **stdout** (trace de l'exécution du programme, en cas d'erreurs, sur **stderr**)

- Code retour, un octet, par convention **0 = pas d'erreur**, sinon erreurs

Code de retour d'un processus / d'une commande

- Code 0 : tout s'est bien passé
- Code non nul (!0) : une erreur s'est produite !

$? permet d'obtenir le code du retour de la dernière commande

```
$ find listing.txt
listing.txt
$ echo $?
0
```

```
$ find coucou.txt
find : coucou.txt : no such file or directory
$ echo $?
1
```

Processus et flux (2)

Le flux 0 : stdin ou entrée standard, le clavier

Le flux 1 : stdout ou sortie standard, l'écran

Le flux 2 : stderr ou sortie d'erreur, lui aussi dirigé vers l'écran

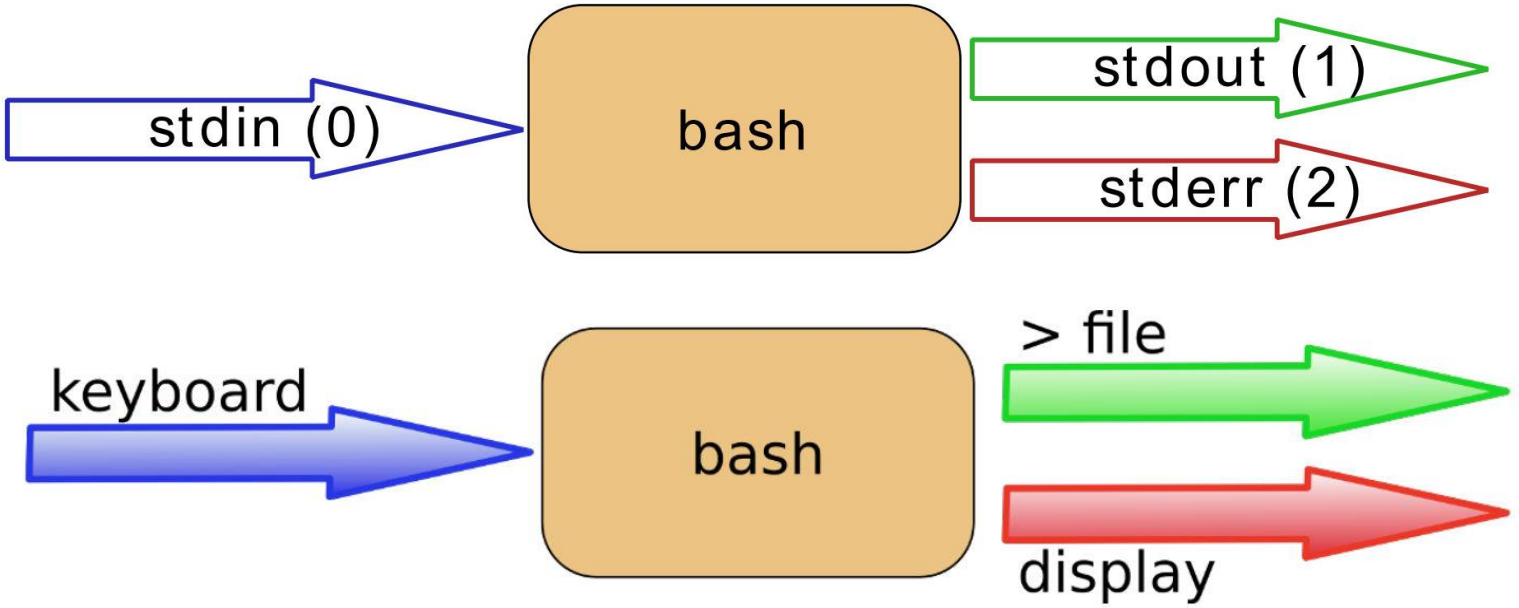
- Question (Redirection) : Les sorties standards peuvent être redirigées vers d'autres fichiers ?

Redirection (1) – Sorties Standards

- Les sorties standards peuvent être redirigées vers d'autres fichiers grâce aux caractères **>** ou **>>**.

commande **>** *file*

Exemple : ls > listeFichier.txt

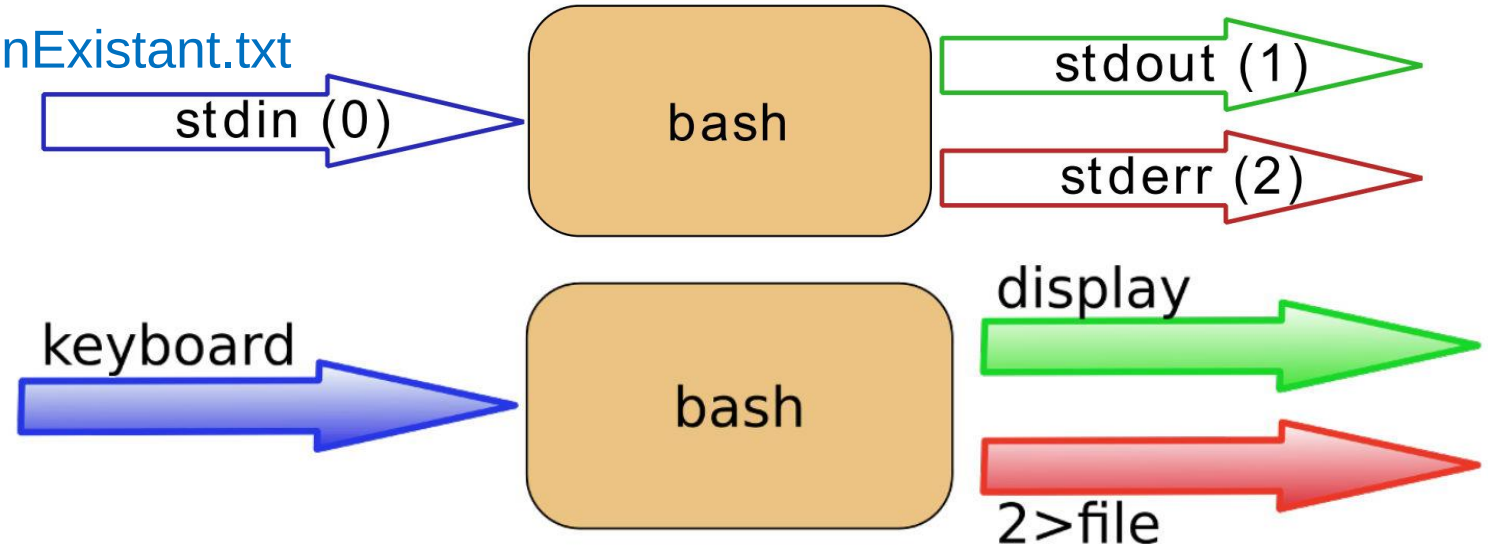


Redirection (2) – Sortie d'erreur

- La sortie d'erreur standard peut être également redirigée vers un autre fichier.
- Cette fois-ci, il faudra préciser le numéro du canal (qui peut être omis pour les canaux 0 et 1) :

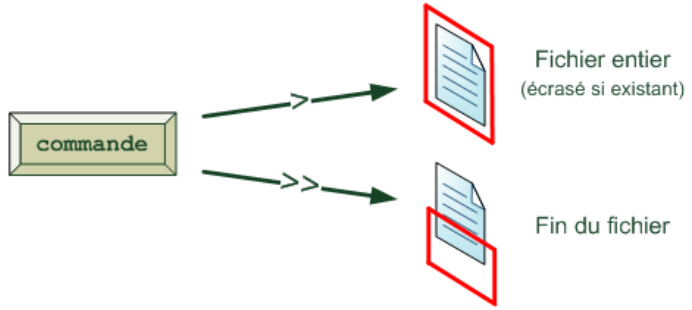
commande 2**>** *file*

Exemple : ls 2> listeFichiersNonExistant.txt



- Lorsque les 2 flux de sortie sont redirigés, aucune information n'est affichée à l'écran.
- Pour utiliser à la fois la redirection de sortie et conserver l'affichage, il faudra utiliser la commande tee.

Redirection (3)



Redirection des Flux dans Linux

- La redirection des flux est une fonctionnalité puissante dans les systèmes Linux, permettant de manipuler les entrées et sorties des commandes.

Commandes Fondamentales

> : Redirige la sortie standard (stdout) vers un fichier et écrase le contenu existant.

>> : Redirige la sortie standard (stdout) vers un fichier et ajoute le contenu à la fin du fichier existant.

Exemple d'utilisation de > :

ls > fichier.txt

➔ Cela prend la sortie de la commande et l'écrit dans le fichier.txt, écrasant le contenu existant.

Exemple d'utilisation de >>

ls >> fichier.txt

➔ Cela ajoute la sortie de la commande à la fin du fichier.txt sans supprimer le contenu existant.

Redirection (4) – Entrée standard

La redirection de l'entrée standard consiste à fournir des données à un programme à partir d'une source autre que le clavier.

< Redirection de Fichier : utilisé pour rediriger l'entrée standard d'une commande à partir d'un fichier.

- Syntaxe : commande **<** fichier.txt
- Exemple : cat < fichier.txt

<< : Utilisé pour permettre la saisie interactive d'une séquence de commandes ou de données jusqu'à ce qu'un marqueur spécifié soit rencontré.

- Syntaxe : commande << MARQUEUR

- Exemple :
  commande << MARQUEUR
  Texte à fournir en entrée MARQUEUR

Exercice

1. Redirigez la sortie de la commande **ls** vers le fichier **Exercice1** tout en conservant le contenu initial.
2. Testez la commande **ls Vide**.
3. Affichez le code d'erreur généré par la commande précédente.
4. Redirigez **la sortie d'erreur** de la commande **ls vide** vers le fichier **Exercice1** tout en conservant le contenu initial.
5. Affichez le contenu du fichier **Exercice1**.

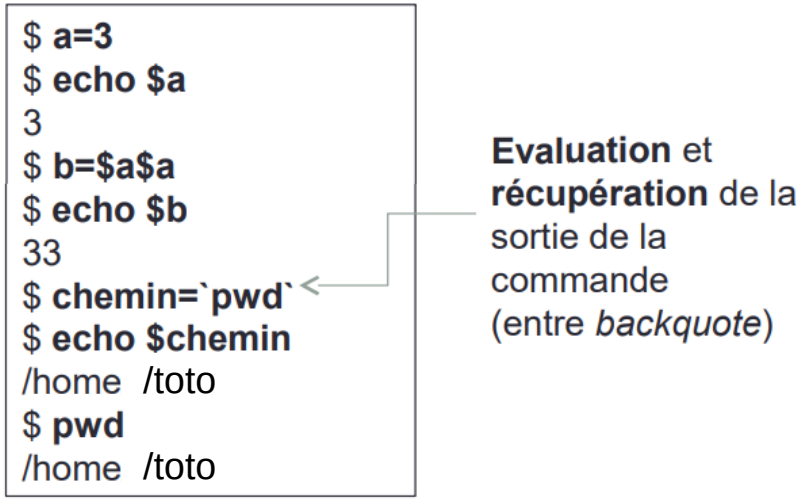
Variables (1)

• En bash, toutes les variables sont de type chaîne de caractères
• Par défaut, elles sont locales au processus courant

Affectation et références

• Affectation : nom=valeur
• Référence : $nom
• Destruction : unset nom

Par défaut, une variable non affectée = chaine vide
Parfois indispensable de mettre entre accolade un nom de variable ex ${a1}
• Lecture au clavier de la valeur d'une variable grâce à la commande read



Variables (2)

Opérations arithmétiques sur des variables

- La commande expr permet de réaliser des opérations arithmétiques

expr var1 op var2

- renvoie à l'écran le résultat de l'opération op entre var1 et var2
- Opérations disponibles op : +,-, *, /, %

- Ecriture alternative : la double parenthèse

((var1 op var2))

```
$ a=3
$ b=8
$ expr $a + $b
11
$ c=`expr $a \* $b`
$ echo $c
24
$ c=$(expr $c + 1)
$ echo $c
25
```

```
$ a=3456
$ expr length $a
4
$ expr length a
1
```

```
$ a=3
$ b=8
$ echo $(($a + $b))
11
$ c=$(($a * $b))
$ echo $c
24
$ c=$(($c + 1))
$ echo $c
25
```

Variables (3)

- Substituer, c'est remplacer une variable par sa valeur

- Substitution autorisée par défaut

```
$ a=pwd
$ echo $a
pwd
$ $a
/home  /toto
```

- **Autorisation** de substitution : " "

```
$ echo "$a"
pwd
$ "$a"
/home  /toto
```

- **Interdiction** de substitution : ' '

```
$ echo '$a'
$a
$ '$a'
$a: not found
```

Variables (4)

Variables d'Environnement dans Linux

- Une variable qui détient des informations sur l'environnement du système

Variables d'Environnement Courantes

- **USER : Contient le nom de l'utilisateur actuel.**

```
echo $USER
```

- **HOME : Indique le répertoire personnel de l'utilisateur.**

```
echo $HOME
```

- **PATH : Spécifie les répertoires où le système recherche les exécutables.**

```
echo $PATH
```

Définir une Variable d'Environnement

Syntaxe : export NOM_VARIABLE="Valeur"

Script - Introduction

- Un script Linux est un fichier texte contenant une série d'instructions ou de commandes que le système d'exploitation Linux peut exécuter dans un shell (l'interface de ligne de commande).

- Les scripts sont utilisés pour

  - Automatiser des tâches,
  - Simplifier des processus
  - Gérer des configurations.

Caractéristiques des scripts Linux

- **Syntaxe** :
  Les scripts utilisent des commandes et des syntaxes spécifiques au shell, comme Bash, Zsh ou Sh.

- **Extensibilité** :
  Les scripts peuvent contenir des boucles, des conditions, et des fonctions, ce qui permet d'écrire des programmes plus complexes.

- **Exécution** :
  Pour exécuter un script, il faut généralement lui donner des permissions d'exécution avec la commande `chmod +x nom_du_script.sh`, puis le lancer avec `./nom_du_script.sh`.

- **Interprétation** :
  Les scripts sont interprétés ligne par ligne par le shell, ce qui signifie qu'ils peuvent être modifiés facilement et exécutés immédiatement.

Exemples

Voici un exemple de script qui affiche "Bonjour, monde !" :

- Le `#!/bin/bash` au début du fichier indique quel interpréteur doit être utilisé pour exécuter le script.

```
#!/bin/bash
echo "Bonjour, monde !"
```

```
#!/bin/bash

echo "La date actuelle est : $(date)"

echo "Voici les fichiers dans le répertoire courant :" ls –

mkdir nouveau_dossier echo "Le répertoire 'nouveau_dossier' a été créé."
```

Liste de variables prépositionnées

Ces variables sont très utilisées lors la création de scripts :

- $0 : nom du script. Plus précisément, il s'agit du paramètre 0 de la ligne de commande, équivalent de argv[0]
- $1, $2, …, $9 : respectivement premier, deuxième, …, neuvième paramètre de la ligne de commande
- $* : tous les paramètres vus comme un seul mot
- $@ : tous les paramètres vus comme des mots séparés : "$@" équivaut à "$1" "$2" …
- $# : nombre de paramètres sur la ligne de commande
- $- : options du shell
- $? : code de retour de la dernière commande
- $$: PID du shell
- $! : PID du dernier processus lancé en arrière-plan
- $_ : dernier argument de la commande précédente

Exemple:

```
#!/bin/bash
# script1.sh
echo "Nom du script $0"
echo "premier paramètre $1"
echo "second paramètre $2"
echo "PID du shell " \$\$
echo "code de retour $?"
exit
```

Exercice :

1. Créer un script qui affiche « Bonjour »
2. Créer un script qui affiche le nom de l'utilisateur (paramètre de la commande)
3. Créer un script qui permet l'addition de deux variables passées en paramètres.

IF

- L'instruction `if` dans les scripts shell permet de prendre des décisions basées sur des conditions.
- C'est un élément essentiel pour contrôler le flux d'exécution du script.

Syntaxe 1 :

```
if [ condition ] ;
then
   action1
fi
```

Syntaxe 2 :

```
if [ condition ] ;
then
   action1
else
   action2
fi
```

Syntaxe 3 :

```
if [ condition ] ;
then
        Action1
elif [ autre_condition ] ;

then
        Action2
else
        Action3
fi
```

Opérateurs de comparaison (1)

Tests sur les entiers :

- entier1 -eq entier2 Vrai si entier1 est égal à entier2
- entier1 -ge entier2 Vrai si entier1 est supérieur ou égal à entier2
- entier1 -gt entier2 Vrai si entier1 est strictement supérieur à entier2
- entier1 -le entier2 Vrai si entier1 est inférieur ou égal à entier2
- entier1 -lt entier2 Vrai si entier1 est strictement inférieur à entier2
- entier1 -ne entier2 Vrai si entier1 est différent de entier2

Opérateurs de comparaison (2)

Tests sur les chaînes de caractères :

• -n "chaîne"	Vrai si la chaîne n'est pas vide
• -z "chaîne"	Vrai si la chaîne est vide
• "chaine1" = "chaine2"	Vrai si les deux chaînes sont identiques
• "chaine1" != "chaine2"	Vrai si les deux chaînes sont différentes

Opérateurs de comparaison (3)

Tests sur les fichiers (et sur les répertoires, suite) :

- -c nom — Vrai si nom représente un périphérique (pseudo-fichier) de type caractère (terminaux, modems et port parallèles par exemple)
- -d nom — Vrai si nom représente un répertoire
- -f nom — Vrai si nom représente un fichier
- -L nom — Vrai si nom représente un lien symbolique
- -p nom — Vrai si nom représente un tube nommé
- f1 -nt f2 — Vrai si les deux fichiers existent et si f1 est plus récent que f2
- f1 -ot f2 — Vrai si les deux fichiers existent et si f1 est plus ancien que f2
- f1 -ef f2 — Vrai si les deux fichiers représentent un seul et même fichier

Opérateurs de comparaison (4)

Tests sur les fichiers (et sur les répertoires) :

- -e fichier — Vrai si le fichier/répertoire existe
- -s fichier — Vrai si le fichier à une taille supérieure à 0
- -z fichier — Vrai si le fichier fait 0 octet (donc si il est vide)
- -r fichier — Vrai si le fichier/répertoire est lisible
- -w fichier — Vrai si le fichier/répertoire est modifiable
- -x fichier — Vrai si le fichier est exécutable ou si le répertoire est accessible
- -O fichier — Vrai si le fichier/répertoire appartient à l'utilisateur
- -G fichier — Vrai si le fichier/répertoire appartient au groupe de l'utilisateur
- -b nom — Vrai si nom représente un périphérique (pseudo-fichier) de type bloc (disques et partitions de disques généralement)

Exemple 1 : Vérifier si un fichier existe

=> un script qui vérifie si un fichier existe et affiche un message en conséquence :

- `-e` : Vérifie si un fichier existe

Exemple 1 : Vérifier si un fichier existe

Voici un script qui vérifie si un fichier existe et affiche un message en conséquence :

- `-e` : Vérifie si un fichier existe

```
#!/bin/bash

# Nom du fichier à vérifier
fichier="mon_fichier.txt"

# Vérifier si le fichier existe
if [ -e "$fichier" ]; then
    echo "Le fichier '$fichier' existe."
else
    echo "Le fichier '$fichier' n'existe pas."
fi
```

Exemple 2 : Comparaison de nombres

=> Un script compare deux nombres et affiche lequel est le plus grand :

- -gt : Vérifie si le premier nombre est plus grand que le second.

- -lt : Vérifie si le premier nombre est plus petit que le second.

Exemple 2 : Comparaison de nombres

Ce script compare deux nombres et affiche lequel est le plus grand :

- -gt : Vérifie si le premier nombre est plus grand que le second.
- -lt : Vérifie si le premier nombre est plus petit que le second.

```
#!/bin/bash

# Définir deux nombres
a=10
b=20

# Comparer les nombres
if [ $a -gt $b ]; then
   echo "$a est plus grand que $b."
elif [ $a -lt $b ]; then
   echo "$a est plus petit que $b."
else
   echo "$a est égal à $b."
fi
```

Exemple 3 : Vérifier un mot de passe

Ce script demande à l'utilisateur d'entrer un mot de passe et vérifie s'il est correct :

== : Vérifie si deux chaînes de caractères sont égales.

- **read -sp** : Utilisé pour demander une entrée utilisateur sans affichage à l'écran (utile pour les mots de passe).

- **echo -e** : Permet d'activer les séquences d'échappement comme les retours à la ligne (\n) ou les tabulations (\t).

Exemple 3 : Vérifier un mot de passe

Ce script demande à l'utilisateur d'entrer un mot de passe et vérifie s'il est correct :

== : Vérifie si deux chaînes de caractères sont égales.

- **read -sp** : Utilisé pour demander une entrée utilisateur sans affichage à l'écran (utile pour les mots de passe).

- **echo -e** : Permet d'activer les séquences d'échappement comme les retours à la ligne (\n) ou les tabulations (\t).

```
#!/bin/bash

# Définir le mot de passe correct
mot_de_passe_correct="secret"

# Demander à l'utilisateur d'entrer le mot de passe
read -sp "Entrez votre mot de passe : " mot_de_passe

# Vérifier le mot de passe
if [ "$mot_de_passe" == "$mot_de_passe_correct" ]; then
    echo -e "\nAccès accordé."
else
    echo -e "\nAccès refusé."
fi
```

Exercice 1

Objectif : Écrire un script qui demande à l'utilisateur d'entrer un nombre, puis utilise l'instruction `if` pour vérifier si ce nombre est **pair** ou **impair**.

Étapes à suivre :

Demander à l'utilisateur d'entrer un nombre.

Vérifier si le nombre est divisible par 2.

Afficher "Le nombre est pair" si c'est le cas.

Sinon, afficher "Le nombre est impair".

- **read -p** : Demande à l'utilisateur d'entrer un nombre.
- **$(($nombre % 2))** : Calcule le reste de la division du nombre par 2.
- **if [... -eq 0]** : Vérifie si le reste est égal à 0 (ce qui signifie que le nombre est pair).

- # Exercice 2 : Vérification des droits d'accès d'un fichier

Objectif : Écrire un script qui prend en entrée le nom d'un fichier, puis vérifie s'il existe et s'il a les permissions de lecture, d'écriture et d'exécution. Le script affichera des messages indiquant quels droits d'accès sont disponibles pour ce fichier.

Étapes à suivre :

Demander à l'utilisateur d'entrer le nom d'un fichier.

Vérifier si le fichier existe.

Si le fichier existe :

- Vérifier si l'utilisateur a la permission de **lecture**.
- Vérifier si l'utilisateur a la permission d'**écriture**.
- Vérifier si l'utilisateur a la permission d'**exécution**.

Afficher un message pour chaque permission : "Oui" ou "Non".

Si le fichier n'existe pas, afficher un message d'erreur.

Exercice 2

`read -p` : Demande à l'utilisateur d'entrer le nom du fichier.

`if [ -e "$fichier" ]` : Vérifie si le fichier existe.

`if [ -r "$fichier" ]` : Vérifie si le fichier a la permission de lecture.

`if [ -w "$fichier" ]` : Vérifie si le fichier a la permission d'écriture.

`if [ -x "$fichier" ]` : Vérifie si le fichier a la permission d'exécution.

La boucle `for`

- La boucle `for` dans les scripts shell est utilisée pour parcourir une série d'éléments et exécuter des commandes pour chaque élément.

- Elle est très utile pour automatiser des tâches répétitives, comme itérer sur une liste de fichiers, des nombres ou des chaînes.

```
for variable in liste_d_éléments; do
   # commandes à exécuter pour chaque élément
done
```

`variable` : Représente l'élément en cours dans la liste.

`liste_d_éléments` : Une liste d'éléments (fichiers, chaînes, nombres, etc.).

`do ... done` : Délimite le bloc de commandes qui sera exécuté pour chaque élément de la liste.

Exemple 1: Utiliser une boucle for pour itérer sur une séquence de nombres

la boucle parcourt une séquence de nombres et affiche chaque nombre

```
#!/bin/bash

# Parcourir les nombres de 1 à 5
for i in {1..5}; do
   echo "Nombre : $i"
done
```

Sortie du script :

```
Nombre : 1
Nombre : 2
Nombre : 3
Nombre : 4
Nombre : 5
```

Exemple 2 : Boucle avec une plage de nombres et un pas

Pour parcourir une plage de nombres avec un pas spécifique (par exemple, de 2 en 2), tu peux utiliser la commande seq.

```
#!/bin/bash

# Parcourir les nombres de 1 à 10 avec un pas de 2
for i in $(seq 1 2 10); do
    echo "Nombre : $i"
done
```

Sortie du script

```
Nombre : 1
Nombre : 3
Nombre : 5
Nombre : 7
Nombre : 9
```

Exemple 3 : Boucle imbriquée (boucles dans des boucles)

Dans cet exemple, une boucle est imbriquée dans une autre pour parcourir deux listes (1,2,3) et (a,b,c) .

Sortie du script

```
Combinaison : 1a
Combinaison : 1b
Combinaison : 1c
Combinaison : 2a
Combinaison : 2b
Combinaison : 2c
Combinaison : 3a
Combinaison : 3b
Combinaison : 3c
```

Exemple 3 : Boucle imbriquée (boucles dans des boucles)

Dans cet exemple, une boucle est imbriquée dans une autre pour parcourir deux listes (1,2,3) et (a,b,c) .

```
#!/bin/bash

# Parcourir deux listes de manière imbriquée
for i in 1 2 3; do
   for lettre in a b c; do
      echo "Combinaison : $i$lettre"
   done
done
```

Sortie du script

```
Combinaison : 1a
Combinaison : 1b
Combinaison : 1c
Combinaison : 2a
Combinaison : 2b
Combinaison : 2c
Combinaison : 3a
Combinaison : 3b
Combinaison : 3c
```

Exemple 1 : Parcourir une liste de chaînes

Dans cet exemple, on parcourt une liste de noms de fruits et on les affiche un par un.

Sortie du script :

```
J'aime les pomme.
J'aime les orange.
J'aime les raisin.
```

Exemple 1 : Parcourir une liste de chaînes

Dans cet exemple, on parcourt une liste de noms de fruits et on les affiche un par un.

```
#!/bin/bash

# Définir une liste de fruits
for fruit in pomme orange raisin; do
    echo "J'aime les $fruit."
done
```

Sortie du script :

```
J'aime les pomme.
J'aime les orange.
J'aime les raisin.
```

Exemple 2 : Parcourir une liste de fichiers

Ce script parcourt tous les fichiers `.txt` dans un répertoire et affiche leur nom.

```
#!/bin/bash

# Parcourir tous les fichiers .txt dans le répertoire courant
for fichier in *.txt; do
    echo "Traitement de $fichier"
done
```

Dans ce cas, le script liste et traite tous les fichiers qui se terminent par `.txt` dans le répertoire où le script est exécuté.

Exercice (for)

un script qui calcule la somme de tous les éléments entiers compris entre deux nombres a et b.

Opérations arithmétiques sur des variables :
((var1 op var2))

Exercice (for)

un script qui calcule la somme de tous les éléments entiers compris entre deux nombres a et b.

```
#!/bin/bash

# Lire les entiers a et b
read -p "Entrez a: " a
read -p "Entrez b: " b

# Calculer la somme des entiers entre a et b
sum=0
for i in $(seq $a $b); do
  sum=$((sum + i))
done

# Afficher le résultat
echo "La somme des éléments entre $a et $b est: $sum"
```

While

Instruction `while`

```
while [ condition ]
do
    commandes
done
```

Exemple :

```
while [ "$var1" != "fin" ]
do
  echo "Variable d'entrée #1 (quitte avec fin) "
  read var1
  echo "variable #1 = $var1"
  echo
done
```

Case (1)

Instruction case

```
case valeur_de_variable in
  val1)
    commandes
    ;;
  val2)
    commandes
    ;;
    ...
  *)
    commandes
esac
```

Case (2)

Exemple #2 :

```
echo "Voulez vous continuer le programme ?"
read reponse
case $reponse in
  [yYoO]*) echo "Ok, on continue";;
  [nN]*) echo "$0 arrete"
         exit 0;;
  *) echo "ERREUR de saisie"
     exit 1;;
esac
```

Case (3)

Exemple #1 :

```
case $# in
  0) echo "aucun parametre"
     echo "Syntaxe :  $0 <nom d'utilisateur>";;
  1) echo "1 parametre passe au programme : $1";;
  2) echo "2 parametres passes au programme : $1 et $2";;
  *) echo "TROP DE PARAMETRES !"
esac
```

```
#!/bin/bash

# Demander à l'utilisateur d'entrer le nom d'un fichier
read -p "Entrez le nom du fichier à vérifier : " fichier

# Vérifier si le fichier existe
if [ -e "$fichier" ]; then
    echo "Le fichier '$fichier' existe."

    # Vérifier les permissions de lecture
    if [ -r "$fichier" ]; then
        echo "Vous avez la permission de lire le fichier."
    else
        echo "Vous n'avez pas la permission de lire le fichier."
    fi

    # Vérifier les permissions d'écriture
    if [ -w "$fichier" ]; then
        echo "Vous avez la permission d'écrire dans le fichier."
    else
        echo "Vous n'avez pas la permission d'écrire dans le fichier."
    fi
```

```
    # Vérifier les permissions d'exécution
    if [ -x "$fichier" ]; then
        echo "Vous avez la permission d'exécuter le
fichier."
    else
        echo "Vous n'avez pas la permission d'exécuter le
fichier."
    fi
else
    echo "Le fichier '$fichier' n'existe pas."
fi
```

```
#!/bin/bash
# Demander à l'utilisateur d'entrer un nombre
read -p "Entrez un nombre : " nombre

# Vérifier si le nombre est pair ou impair
if [ $(($nombre % 2)) -eq 0 ]; then
    echo "Le nombre est pair."
else
    echo "Le nombre est impair."
fi
```

Filtres sur les fichiers

- Buts : afficher / trier / analyser / rechercher / remplacer dans des fichiers

Description	Commande	Exemple
Afficher le contenu des fichiers	**cat**	cat *fichier*
Afficher page par page	**more**	more *fichier*
Trier par ordre alphabétique, numérique (-n)	**sort**	sort *fichier*
Sélectionner des colonnes / champs d'un fichier	**cut**	cut -c1-10 *fichier*
Rechercher dans un fichier et afficher les lignes contenant une chaine de caractères	**grep**	grep *.txt *fichier_liste*
Remplacer des chaines de caractères / supprimer des lignes particulières dans un fichier …	**sed**	sed -i '/*mon*/d' brassens
Rechercher et afficher des lignes communes / spécifiques entre 2 fichiers	**comm**	comm -1 *fichier1 fichier2*

Astérisques (*)

- L'astérisque (*) est un caractère générique utilisé pour faire correspondre n'importe quel ensemble de caractères

Exemple : $ **ls *.txt**

➔ Affiche tous les fichiers se terminant par .txt

Utilisation de la Tabulation (Tab Completion) :

- Lorsque vous tapez le début d'un nom de dossier et appuyez sur la touche "Tab", le système tentera d'auto-compléter le nom en fonction des fichiers et dossiers existants.

Commande	Action
$ wc -l < /etc/passwd	compte le nombre de lignes dans */etc/passwd*
$ wc –c < /etc/passwd	compte le nombre de caractère dans */etc/passwd*

Commande	Action
$ grep -i "home" < /etc/passwd	affiche les lignes contenant *home* sans tenir compte des majuscules et minuscules
$ grep -v "home" < /etc/passwd	affiche les lignes ne contenant pas *home*

Commande	Action
$ cat /etc/passwd | tr : "\t"	remplace les caractères : par une tabulation
$ cat /etc/passwd | tr -d [A-Z]	supprime tous les caractères majuscule de A à Z
$ last | tr [:lower:] [:upper:]	remplace toutes les minuscules par des majuscules

Question : Possibilité de lancer plusieurs commandes à la fois ?

Ressources en ligne

- **Les pages man !**
- **http://www.linux-france.org/article/ memo/node80.html**
- **https://www.univ-orleans.fr/lifo/membres/Yannick.Parmentier/linux/support2.pdf**
- **http://tvaira.free.fr/bts-sn/linux/cours/cours-scripts.pdf**

Les fonctions

- Les fonctions dans les scripts shell permettent de regrouper des ensembles de commandes réutilisables.
- Elles facilitent l'organisation, la modularité et la réutilisation du code.
- Une fonction peut prendre des arguments, exécuter des commandes, et renvoyer des valeurs ou des statuts d'exécution.
- Syntaxe de base des fonctions

```
nom_de_fonction() {
   # corps de la fonction (les commandes à exécuter)
}
```

Exemple 1 : Fonction simple sans arguments

```
#!/bin/bash

# Définir une fonction
bonjour() {
    echo "Bonjour, monde !"
}

# Appeler la fonction
bonjour
```

Exemple 2 : Fonction avec des arguments

- Les fonctions dans les scripts shell peuvent recevoir des arguments que tu peux utiliser dans la fonction.
- Ces arguments sont accessibles via des variables spéciales comme $1, $2, etc., où $1 est le premier argument, $2 le second, et ainsi de suite.

```
#!/bin/bash

# Définir une fonction qui prend des arguments
saluer() {
    echo "Bonjour, $1 !"
    echo "Comment allez-vous, $2 ?"
}

# Appeler la fonction avec deux arguments
saluer "Alice" "aujourd'hui"
```

Exemple 3 : Variables locales dans une fonction

- Par défaut, les variables définies dans une fonction sont globales dans le script, c'est-à-dire qu'elles peuvent être utilisées en dehors de la fonction.
- Cependant, tu peux rendre une variable locale à la fonction en utilisant le mot-clé `local`.

```
#!/bin/bash
# Fonction avec des variables locales
calculer_somme() {
    local a=$1
    local b=$2
    local somme=$(($a + $b))
    echo "La somme de $a et $b est : $somme"
}

# Appeler la fonction
calculer_somme 3 7
```