SYSTÈMES D'EXPLOITATION

LA PROGRAMMATION AVEC LE SHELL/UNIX

3A - Cursus Ingénieurs CentraleSupelec 2023/2024

Idir AIT SADOUNE
idir.aitsadoune@centralesupelec.fr

PLAN

Retour au plan - Retour à l'accueil

PLAN

Retour au plan - Retour à l'accueil

LES VARIABLES DANS LE SHELL

LES VARIABLES DANS LE SHELL

- **Variables** de l'interpréteur de commandes :

LES VARIABLES DANS LE SHELL

- **Variables** de l'interpréteur de commandes :
  👉 non déclarées

LES VARIABLES DANS LE SHELL

- **Variables** de l'interpréteur de commandes :
  👉 non déclarées
  👉 non typées a priori → **chaînes de caractères**

LES VARIABLES DANS LE SHELL

- **Variables** de l'interpréteur de commandes :
  - 👉 non déclarées
  - 👉 non typées a priori → **chaînes de caractères**
  - 👉 pas d'héritage par les processus fils

LES VARIABLES DANS LE SHELL

- **Variables** de l'interpréteur de commandes :
  - 👉 non déclarées
  - 👉 non typées a priori → **chaînes de caractères**
  - 👉 pas d'héritage par les processus fils

- Syntaxe de l'affectation (sans espace autour du signe =) :

```
variable=valeur
```

LES VARIABLES DANS LE SHELL

- **Variables** de l'interpréteur de commandes :
  - 👉 non déclarées
  - 👉 non typées a priori → **chaînes de caractères**
  - 👉 pas d'héritage par les processus fils

- Syntaxe de l'affectation (sans espace autour du signe =) :

```
variable=valeur
```

- Référence à la valeur de la variable :

```
$variable
${variable}
```

EXEMPLES

EXEMPLES

Déclaration et initialisation de variables

```
$ alpha=toto
$ b=35
$ c2=3b
```

EXEMPLES

Déclaration et initialisation de variables

```
$ alpha=toto
$ b=35
$ c2=3b
```

Affichage du contenu des variables

```
$ echo alpha, b, c2 contiennent ${alpha}, ${b}, ${c2}
alpha, b, c2 contiennent toto, 35, 3b
```

EXEMPLES

Déclaration et initialisation de variables

```
$ alpha=toto
$ b=35
$ c2=3b
```

Affichage du contenu des variables

```
$ echo alpha, b, c2 contiennent ${alpha}, ${b}, ${c2}
alpha, b, c2 contiennent toto, 35, 3b
```

Affichage des noms et valeurs des variables du **Shell**

```
$ set | grep alpha
alpha=toto
```

EVALUATION DES EXPRESSIONS

EVALUATION DES EXPRESSIONS

Evaluation d'une expression ?

```
$ b=35
$ bb=${b}+${b}
$ echo b vaut ${b}, bb vaut ${bb}
b vaut 35, bb vaut 35+35
```

EVALUATION DES EXPRESSIONS

Evaluation d'une expression ?

```
$ b=35
$ bb=${b}+${b}
$ echo b vaut ${b}, bb vaut ${bb}
b vaut 35, bb vaut 35+35
```

Pas d’arithmétique directement avec le **shell**

EVALUATION DES EXPRESSIONS

Evaluation d'une expression ?

```
$ b=35
$ bb=${b}+${b}
$ echo b vaut ${b}, bb vaut ${bb}
b vaut 35, bb vaut 35+35
```

Pas d'arithmétique directement avec le **shell**

👉 utiliser `expr` ou `$(())`

EVALUATION DES EXPRESSIONS

Evaluation d'une expression ?

```
$ b=35
$ bb=${b}+${b}
$ echo b vaut ${b}, bb vaut ${bb}
b vaut 35, bb vaut 35+35
```

Pas d'arithmétique directement avec le **shell**

👉 utiliser `expr` ou `$(())`

👉 les arguments de `expr` doivent être séparés par au moins un espace

EVALUATION DES EXPRESSIONS

Evaluation d'une expression ?

```
$ b=35
$ bb=${b}+${b}
$ echo b vaut ${b}, bb vaut ${bb}
b vaut 35, bb vaut 35+35
```

Pas d'arithmétique directement avec le **shell**

👉 utiliser `expr` ou `$(())`

👉 les arguments de `expr` doivent être séparés par au moins un espace

```
$ expr $b + $b
70
```

EVALUATION DES EXPRESSIONS

Evaluation d'une expression ?

```
$ b=35
$ bb=${b}+${b}
$ echo b vaut ${b}, bb vaut ${bb}
b vaut 35, bb vaut 35+35
```

Pas d'arithmétique directement avec le **shell**

👉 utiliser `expr` ou `$(())`

👉 les arguments de `expr` doivent être séparés par au moins un espace

```
$ expr $b + $b
70
```

```
$ bb=$(($b+$b))
$ echo ${bb}
70
```

ÉTENDRE LA PORTÉE D’UNE VARIABLE

ÉTENDRE LA PORTÉE D'UNE VARIABLE

- `export` : étend la portée d'une variable aux processus du **Shell** courant

ÉTENDRE LA PORTÉE D'UNE VARIABLE

- `export` : étend la portée d'une variable aux processus du **Shell** courant

```
export [options] [name[=value] ...]
```

ÉTENDRE LA PORTÉE D'UNE VARIABLE

- `export` : étend la portée d'une variable aux processus du **Shell** courant

```
export [options] [name[=value] ...]
```

- **Principales options**

ÉTENDRE LA PORTÉE D'UNE VARIABLE

- `export` : étend la portée d'une variable aux processus du **Shell** courant

```
export [options] [name[=value] ...]
```

- **Principales options**
  - `-p` liste des variables exportées dans le shell courant

ÉTENDRE LA PORTÉE D'UNE VARIABLE

- `export` : étend la portée d'une variable aux processus du **Shell** courant

```
export [options] [name[=value] ...]
```

- **Principales options**
  - `-p` liste des variables exportées dans le shell courant
  - `-n` supprime la variable de la liste exportée

ÉTENDRE LA PORTÉE D'UNE VARIABLE

- `export` : étend la portée d'une variable aux processus du **Shell** courant

```
export [options] [name[=value] ...]
```

- **Principales options**
  - `-p` liste des variables exportées dans le shell courant
  - `-n` supprime la variable de la liste exportée
  - `-f` exporte une fonction

ÉTENDRE LA PORTÉE D'UNE VARIABLE

- `export` : étend la portée d'une variable aux processus du **Shell** courant

```
export [options] [name[=value] ...]
```

- **Principales options**
  - `-p` liste des variables exportées dans le shell courant
  - `-n` supprime la variable de la liste exportée
  - `-f` exporte une fonction

- **Exemple**

```
$ export JAVA_HOME="/usr/local/jdk"
$ export -p
...
JAVA_HOME=/usr/local/jdk
```

LES VARIABLES D’ENVIRONNEMENT

LES VARIABLES D'ENVIRONNEMENT

- Les **variables d'environnement** sont systématiquement héritées par tous les processus.

LES VARIABLES D'ENVIRONNEMENT

- Les **variables d'environnement** sont systématiquement héritées par tous les processus.
- La commande env donne la liste des variables d'environnement et leurs valeurs.

```
$ env
```

LES VARIABLES D'ENVIRONNEMENT

- Les **variables d'environnement** sont systématiquement héritées par tous les processus.
- La commande env donne la liste des variables d'environnement et leurs valeurs.

```
$ env
```

- Quelques **variables d'environnement** :

LES VARIABLES D'ENVIRONNEMENT

- Les **variables d'environnement** sont systématiquement héritées par tous les processus.
- La commande env donne la liste des variables d'environnement et leurs valeurs.

```
$ env
```

- Quelques **variables d'environnement** :
  - SHELL : interpréteur de commandes utilisé (bash, zsh ...)

LES VARIABLES D'ENVIRONNEMENT

- Les **variables d'environnement** sont systématiquement héritées par tous les processus.
- La commande `env` donne la liste des variables d'environnement et leurs valeurs.

```
$ env
```

- Quelques **variables d'environnement** :
  - `SHELL` : interpréteur de commandes utilisé (`bash`, `zsh` ...)
  - `TERM` : type de terminal utilisé (`vt100`, `xterm`, ...)

LES VARIABLES D'ENVIRONNEMENT

- Les **variables d'environnement** sont systématiquement héritées par tous les processus.
- La commande `env` donne la liste des variables d'environnement et leurs valeurs.

```
$ env
```

- Quelques **variables d'environnement** :
  - `SHELL` : interpréteur de commandes utilisé (`bash`, `zsh` ...)
  - `TERM` : type de terminal utilisé (`vt100`, `xterm`, ...)
  - `HOME` : répertoire d'accueil

LES VARIABLES D'ENVIRONNEMENT

- Les **variables d'environnement** sont systématiquement héritées par tous les processus.
- La commande `env` donne la liste des variables d'environnement et leurs valeurs.

```
$ env
```

- Quelques **variables d'environnement** :
  - `SHELL` : interpréteur de commandes utilisé (`bash`, `zsh` ...)
  - `TERM` : type de terminal utilisé (`vt100`, `xterm`, ...)
  - `HOME` : répertoire d'accueil
  - `USER` : identifiant (nom) de l'utilisateur

LES VARIABLES D'ENVIRONNEMENT

- Les **variables d'environnement** sont systématiquement héritées par tous les processus.
- La commande `env` donne la liste des variables d'environnement et leurs valeurs.

```
$ env
```

- Quelques **variables d'environnement** :
  - `SHELL` : interpréteur de commandes utilisé (`bash`, `zsh` ...)
  - `TERM` : type de terminal utilisé (`vt100`, `xterm`, ...)
  - `HOME` : répertoire d'accueil
  - `USER` : identifiant (nom) de l'utilisateur
  - `PATH` : liste des chemins de recherche des commandes

LA VARIABLE PATH

- PATH : liste des chemins de recherche des commandes

LA VARIABLE PATH

- PATH : liste des chemins de recherche des commandes

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

LA VARIABLE PATH

- PATH : liste des chemins de recherche des commandes

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

- **Deux méthodes d'exécution** de commandes/exécutables :

LA VARIABLE PATH

- PATH : liste des chemins de recherche des commandes

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

- **Deux méthodes d'exécution** de commandes/exécutables :

  👉 par le nom du script/la commande:

```
$ mon_script.sh
```

  la recherche se fait dans les répertoires listés dans la variable PATH en respectant l'ordre

LA VARIABLE PATH

- PATH : liste des chemins de recherche des commandes

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

- **Deux méthodes d'exécution** de commandes/exécutables :

  👉 par le nom du script/la commande:

```
$ mon_script.sh
```

  la recherche se fait dans les répertoires listés dans la variable PATH en respectant l'ordre

  👉 par le chemin explicite vers le script/la commande :

```
$ ./mes_bins/mon_script.sh
```

INITIALISATION DU SHELL

INITIALISATION DU SHELL

- Quelques fichiers d’initialisation du **shell** :

INITIALISATION DU SHELL

- Quelques fichiers d'initialisation du **shell** :
  - `/etc/profile` : pour tous les utilisateurs **à la connexion**

INITIALISATION DU SHELL

- Quelques fichiers d'initialisation du **shell** :
  - `/etc/profile` : pour tous les utilisateurs **à la connexion**
  - `${HOME}/.zshenv`, `${HOME}/.zprofile` ... : pour le **Shell** courant

INITIALISATION DU SHELL

- Quelques fichiers d'initialisation du **shell** :
  - `/etc/profile` : pour tous les utilisateurs **à la connexion**
  - `${HOME}/.zshenv`, `${HOME}/.zprofile` ... : pour le **Shell** courant

- **Exemple** de modifications à mettre dans `${HOME}/.zshenv` (recherche des commandes dans le répertoire `${HOME}/Scripts`)

```
...
PATH="${HOME}/Scripts:${PATH}" # le répertoire Scripts dans home
PATH=".:${PATH}" # regarder aussi le répertoire courant
export PATH
...
```

PLAN

Retour au plan - Retour à l'accueil

SUBSTITUTION DE COMMANDE

SUBSTITUTION DE COMMANDE

- Récupération du **résultat d'une commande** (sa sortie standard)

SUBSTITUTION DE COMMANDE

- Récupération du **résultat d'une commande** (sa sortie standard)
  👉 chaîne de caractères stocké dans une variable

SUBSTITUTION DE COMMANDE

- Récupération du **résultat d'une commande** (sa sortie standard)
  👉 chaîne de caractères stocké dans une variable
  👉 repris comme argument d'une autre commande

SUBSTITUTION DE COMMANDE

- Récupération du **résultat d'une commande** (sa sortie standard)
  👉 chaîne de caractères stocké dans une variable
  👉 repris comme argument d'une autre commande

```
$(commande)
```

SUBSTITUTION DE COMMANDE

- Récupération du **résultat d'une commande** (sa sortie standard)
  👉 chaîne de caractères stocké dans une variable
  👉 repris comme argument d'une autre commande

```
$(commande)
```

- **Utilisations** :

SUBSTITUTION DE COMMANDE

- Récupération du **résultat d'une commande** (sa sortie standard)

  👉 chaîne de caractères stocké dans une variable

  👉 repris comme argument d'une autre commande

```
$(commande)
```

- **Utilisations** :

  👉 paramétrage de `shell-scripts`

SUBSTITUTION DE COMMANDE

- Récupération du **résultat d'une commande** (sa sortie standard)
  👉 chaîne de caractères stocké dans une variable
  👉 repris comme argument d'une autre commande

```
$(commande)
```

- **Utilisations** :
  👉 paramétrage de `shell-scripts`
  👉 calculs sur les entiers avec la commande `expr`

EXEMPLES

EXEMPLES

Récupération des résultats d'une commande

```
$ qui=$(whoami)
$ echo ${qui}
idiraitsadoune
```

EXEMPLES

Récupération des résultats d'une commande

```
$ qui=$(whoami)
$ echo ${qui}
idiraitsadoune
```

```
$ echo aujourd_hui on est $(date)
aujourd_hui on est Mar 17 oct 2023 05:45:25 CEST
```

EXEMPLES

Récupération des résultats d'une commande

```
$ qui=$(whoami)
$ echo ${qui}
idiraitsadoune
```

```
$ echo aujourd_hui on est $(date)
aujourd_hui on est Mar 17 oct 2023 05:45:25 CEST
```

Sauvegarder le résultat d'une expression arithmétique

```
$ s1=$(expr 12 + 2)
$ echo 12 + 2 = ${s1}
12 + 2 = 14
```

EXEMPLES

Récupération des résultats d'une commande

```
$ qui=$(whoami)
$ echo ${qui}
idiraitsadoune
```

```
$ echo aujourd_hui on est $(date)
aujourd_hui on est Mar 17 oct 2023 05:45:25 CEST
```

Sauvegarder le résultat d'une expression arithmétique

```
$ s1=$(expr 12 + 2)
$ echo 12 + 2 = ${s1}
12 + 2 = 14
```

```
$ s2=$(($((12+2))+1))
$ echo ${s2}
15
```

L’INTERPRÉTATION D'UNE COMMANDE

L’INTERPRÉTATION D'UNE COMMANDE

- L'**interprétation d'une commande** passe par deux étapes

L'INTERPRÉTATION D'UNE COMMANDE

- L'**interprétation d'une commande** passe par deux étapes
  1. Le **shell** interprète la commande
     (options, caractères jokers, redirections, variables, ...)

L'INTERPRÉTATION D'UNE COMMANDE

- L'**interprétation d'une commande** passe par deux étapes
  1. Le **shell** interprète la commande
     (options, caractères jokers, redirections, variables, ...)
  2. La **commande** interprète les arguments
     (paramètres, expressions régulières, ...)

L'INTERPRÉTATION D'UNE COMMANDE

- L'**interprétation d'une commande** passe par deux étapes
  1. Le **shell** interprète la commande
     (options, caractères jokers, redirections, variables, ...)
  2. La **commande** interprète les arguments
     (paramètres, expressions régulières, ...)

- Pour éviter d'exposer les métacaractères à l'interprétation par le **Shell**, on peut utiliser les protections suivantes :

L'INTERPRÉTATION D'UNE COMMANDE

- L'**interprétation d'une commande** passe par deux étapes
  1. Le **shell** interprète la commande
     (options, caractères jokers, redirections, variables, ...)
  2. La **commande** interprète les arguments
     (paramètres, expressions régulières, ...)

- Pour éviter d'exposer les métacaractères à l'interprétation par le **Shell**, on peut utiliser les protections suivantes :
  - \ : protection individuelle du caractère suivant (**backslash**)

L'INTERPRÉTATION D'UNE COMMANDE

- L'**interprétation d'une commande** passe par deux étapes
  1. Le **shell** interprète la commande
     (options, caractères jokers, redirections, variables, ...)
  2. La **commande** interprète les arguments
     (paramètres, expressions régulières, ...)

- Pour éviter d'exposer les métacaractères à l'interprétation par le **Shell**, on peut utiliser les protections suivantes :
  - `\` : protection individuelle du caractère suivant (**backslash**)
  - `'...'` : protection forte (aucune interprétation)

L'INTERPRÉTATION D'UNE COMMANDE

- L'**interprétation d'une commande** passe par deux étapes
  1. Le **shell** interprète la commande
     (options, caractères jokers, redirections, variables, ...)
  2. La **commande** interprète les arguments
     (paramètres, expressions régulières, ...)

- Pour éviter d'exposer les métacaractères à l'interprétation par le **Shell**, on peut utiliser les protections suivantes :
  - `\` : protection individuelle du caractère suivant (**backslash**)
  - `'...'` : protection forte (aucune interprétation)
  - `"..."` : protection faible (substitution de paramètres/commandes)

EXEMPLES

EXEMPLES

Affiche les lignes de `fic.txt` comportant au moins un chiffre

```
$ grep '[0-9][0-9]*' fic.txt
```

EXEMPLES

Affiche les lignes de `fic.txt` comportant au moins un chiffre

```
$ grep '[0-9][0-9]*' fic.txt
```

Affectation d'une chaîne comportant des blancs à une variable

```
$ v1="avec blanc1" ; v2='avec blanc2' ; v3=avec\ blanc3
```

EXEMPLES

Affiche les lignes de `fic.txt` comportant au moins un chiffre

```
$ grep '[0-9][0-9]*' fic.txt
```

Affectation d'une chaîne comportant des blancs à une variable

```
$ v1="avec blanc1" ; v2='avec blanc2' ; v3=avec\ blanc3
```

Lecture du contenu d'une variable

```
$ echo ${TERM} "${TERM}" '${TERM}'
xterm-256color xterm-256color ${TERM}
```

EXEMPLES

Affiche les lignes de `fic.txt` comportant au moins un chiffre

```
$ grep '[0-9][0-9]*' fic.txt
```

Affectation d'une chaîne comportant des blancs à une variable

```
$ v1="avec blanc1" ; v2='avec blanc2' ; v3=avec\ blanc3
```

Lecture du contenu d'une variable

```
$ echo ${TERM} "${TERM}" '${TERM}'
xterm-256color xterm-256color ${TERM}
```

Lecture du résultat d'une commande

```
$ echo Je suis $(whoami) - "Je suis $(whoami)" - 'Je suis $(whoami)'
Je suis idiraitsadoune - Je suis idiraitsadoune - Je suis $(whoami)
```

PLAN

Retour au plan - Retour à l'accueil

FICHIERS DE COMMANDES

FICHIERS DE COMMANDES

- **Shell-scripts** : fichier texte contenant des commandes

FICHIERS DE COMMANDES

- **Shell-scripts** : fichier texte contenant des commandes
- **Trois méthodes d'exécution** :

FICHIERS DE COMMANDES

- **Shell-scripts** : fichier texte contenant des commandes
- **Trois méthodes d'exécution** :
  1. utiliser **la commande** bash

```
$ bash script.sh
```

FICHIERS DE COMMANDES

- **Shell-scripts** : fichier texte contenant des commandes
- **Trois méthodes d'exécution** :
  1. utiliser **la commande** bash

```
$ bash script.sh
```

  2. rendre le script exécutable et utiliser le **chemin vers le script**

```
$ chmod +x script.sh
$ cp script.sh mes_bins/Scripts
$ mes_bins/Scripts/script.sh
```

FICHIERS DE COMMANDES

- **Shell-scripts** : fichier texte contenant des commandes
- **Trois méthodes d'exécution** :
  1. utiliser **la commande** `bash`

```
$ bash script.sh
```

  2. rendre le script exécutable et utiliser le **chemin vers le script**

```
$ chmod +x script.sh
$ cp script.sh mes_bins/Scripts
$ mes_bins/Scripts/script.sh
```

  3. rendre le script exécutable et le mettre dans un **dossier du `PATH`** (`${HOME}/Scripts` doit être ajouté au `PATH`)

```
$ chmod +x script.sh
$ cp script.sh ${HOME}/Scripts
$ script.sh
```

LE PREMIER SCRIPT

LE PREMIER SCRIPT

Récupérer des informations sur l'utilisateur et la machine

LE PREMIER SCRIPT

Récupérer des informations sur l'utilisateur et la machine

```
1 #!/bin/sh
2 # file name : my_script.sh
3 echo nous sommes le $(date)
4 echo mon login est $(whoami)
5 echo "le calculateur est $(hostname)"
```

LE PREMIER SCRIPT

Récupérer des informations sur l'utilisateur et la machine

```
1 #!/bin/sh
2 # file name : my_script.sh
3 echo nous sommes le $(date)
4 echo mon login est $(whoami)
5 echo "le calculateur est $(hostname)"
```

```
$ bash my_script.sh
nous sommes le Mar 17 oct 2023 06:40:29 CEST
mon login est idiraitsadoune
le calculateur est MBP-de-Idir
```

LES PARAMÈTRES DES SCRIPTS

LES PARAMÈTRES DES SCRIPTS

Les **variables positionnées** lors du lancement d'une commande :

LES PARAMÈTRES DES SCRIPTS

Les **variables positionnées** lors du lancement d'une commande :

- `$0` : nom du fichier de la commande (spécifié lors de l'appel)

LES PARAMÈTRES DES SCRIPTS

Les **variables positionnées** lors du lancement d'une commande :

- `$0` : nom du fichier de la commande (spécifié lors de l'appel)
- `$1`, `$2`, ... , `$9`, `${10}`, ... les paramètres (arguments) de la commande

LES PARAMÈTRES DES SCRIPTS

Les **variables positionnées** lors du lancement d'une commande :

- `$0` : nom du fichier de la commande (spécifié lors de l'appel)
- `$1`, `$2`, ... , `$9`, `${10}`, ... les paramètres (arguments) de la commande
- `$*` : chaîne formée par les paramètres d'appel (`"$1 $2 $3 ..."`)

LES PARAMÈTRES DES SCRIPTS

Les **variables positionnées** lors du lancement d'une commande :

- `$0` : nom du fichier de la commande (spécifié lors de l'appel)
- `$1`, `$2`, ... , `$9`, `${10}`, ... les paramètres (arguments) de la commande
- `$*` : chaîne formée par les paramètres d'appel (`"$1 $2 $3 ..."`)
- `$@` : liste des paramètres d'appel (`["$1", "$2", "$3", ..."]`)

LES PARAMÈTRES DES SCRIPTS

Les **variables positionnées** lors du lancement d'une commande :

- `$0` : nom du fichier de la commande (spécifié lors de l'appel)
- `$1`, `$2`, ... , `$9`, `${10}`, ... les paramètres (arguments) de la commande
- `$*` : chaîne formée par les paramètres d'appel (`"$1 $2 $3 ..."`)
- `$@` : liste des paramètres d'appel (`["$1", "$2", "$3", ..."]`)
- `$#` : nombre de paramètres lors de l'appel

LES PARAMÈTRES DES SCRIPTS

Les **variables positionnées** lors du lancement d'une commande :

- `$0` : nom du fichier de la commande (spécifié lors de l'appel)
- `$1`, `$2`, ... , `$9`, `${10}`, ... les paramètres (arguments) de la commande
- `$*` : chaîne formée par les paramètres d'appel (`"$1 $2 $3 ..."`)
- `$@` : liste des paramètres d'appel (`["$1", "$2", "$3", ..."]`)
- `$#` : nombre de paramètres lors de l'appel
- `$$` : numéro du processus lancé (`pid`)

LES PARAMÈTRES DES SCRIPTS

Les **variables positionnées** lors du lancement d'une commande :

- `$0` : nom du fichier de la commande (spécifié lors de l'appel)
- `$1`, `$2`, ... , `$9`, `${10}`, ... les paramètres (arguments) de la commande
- `$*` : chaîne formée par les paramètres d'appel (`"$1 $2 $3 ..."`)
- `$@` : liste des paramètres d'appel (`["$1", "$2", "$3", ..."]`)
- `$#` : nombre de paramètres lors de l'appel
- `$$` : numéro du processus lancé (`pid`)
- `$?` : le code d'erreur de la dernière commande exécutée

EXEMPLE UTILISANT LES PARAMÈTRES DES SCRIPTS

```
1 #!/bin/sh
2 # file name : prog.sh
3 echo la procedure $0
4 echo a ete appelee avec $# parametres
5 echo le premier parametre est $1
6 echo la liste des parametres est $*
7 echo le numero du processus lance est $$
```

EXEMPLE UTILISANT LES PARAMÈTRES DES SCRIPTS

```
1 #!/bin/sh
2 # file name : prog.sh
3 echo la procedure $0
4 echo a ete appelee avec $# parametres
5 echo le premier parametre est $1
6 echo la liste des parametres est $*
7 echo le numero du processus lance est $$
```

```
$ ./prog.sh p1 p2 p3
la procedure ./prog.sh
a ete appelee avec 3 parametres
le premier parametre est p1
la liste des parametres est p1 p2 p3
le numero du processus lance est 2960
```

UN AUTRE EXEMPLE

Concatener deux fichiers ($1 et $2) dans le fichier $3

```
1  #!/bin/sh
2  file_in1=$1
3  file_in2=$2
4  file_out=$3
5  echo '----------------------------------------' > $file_out
6  echo \| $file_in1 \| >> $file_out
7  echo '----------------------------------------' >> $file_out
8  cat $file_in1 >> $file_out
9  echo '----------------------------------------' >> $file_out
10 echo \| $file_in2 \| >> $file_out
11 echo '----------------------------------------' >> $file_out
12 cat $file_in2 >> $file_out
13 echo termine
14 exit 0
```

PLAN

Retour au plan - Retour à l'accueil

LA COMMANDE `test`

LA COMMANDE `test`

- La commande `test` permet de **vérifier** les attributs d'un fichier ou le contenu d'une variable.

LA COMMANDE `test`

- La commande `test` permet de **vérifier** les attributs d'un fichier ou le contenu d'une variable.
- La commande `test` renvoie le code 0 ou 1 (vrai ou faux) qui est sauvegardé dans la variable `$?`.

LA COMMANDE `test`

- La commande `test` permet de **vérifier** les attributs d'un fichier ou le contenu d'une variable.
- La commande `test` renvoie le code 0 ou 1 (vrai ou faux) qui est sauvegardé dans la variable `$?`.
- La commande `test` propose deux syntaxes équivalentes :

LA COMMANDE `test`

- La commande `test` permet de **vérifier** les attributs d'un fichier ou le contenu d'une variable.
- La commande `test` renvoie le code 0 ou 1 (vrai ou faux) qui est sauvegardé dans la variable `$?`.
- La commande `test` propose deux syntaxes équivalentes :

```
$ test expression
```

LA COMMANDE `test`

- La commande `test` permet de **vérifier** les attributs d'un fichier ou le contenu d'une variable.
- La commande `test` renvoie le code 0 ou 1 (vrai ou faux) qui est sauvegardé dans la variable `$?`.
- La commande `test` propose deux syntaxes équivalentes :

```
$ test expression
```

```
$ [expression]
```

LA COMMANDE `test`

- La commande `test` permet de **vérifier** les attributs d'un fichier ou le contenu d'une variable.
- La commande `test` renvoie le code 0 ou 1 (vrai ou faux) qui est sauvegardé dans la variable `$?`.
- La commande `test` propose deux syntaxes équivalentes :

```
$ test expression
```

```
$ [expression]
```

- Consultez ce lien pour découvrir la commande `test`
- Consultez ce lien pour comparer `test` à `[[ ... ]]`

STRUCTURE STRUCTURES `if ... fi`

STRUCTURE STRUCTURES `if ... fi`

- L'instruction `if` permet d'effectuer des opérations **si** une condition est vérifiée.

STRUCTURE STRUCTURES `if ... fi`

- L'instruction `if` permet d'effectuer des opérations **si** une condition est vérifiée.

```
if condition
    then instruction(s)
fi
```

STRUCTURE STRUCTURES `if ... fi`

- L'instruction `if` permet d'effectuer des opérations **si** une condition est vérifiée.

```
if condition
    then instruction(s)
fi
```

- L'instruction `if` peut aussi inclure une instruction `else` dans le cas ou la condition n'est pas vérifiée.

STRUCTURE STRUCTURES `if ... fi`

- L'instruction `if` permet d'effectuer des opérations **si** une condition est vérifiée.

```
if condition
    then instruction(s)
fi
```

- L'instruction `if` peut aussi inclure une instruction `else` dans le cas ou la condition n'est pas vérifiée.

```
if condition
    then instruction(s)
else instruction(s)
fi
```

EXEMPLE

EXEMPLE

Vérifier si un utilisateur est connecté

```
1 #!/bin/sh
2 if who | grep "^$1 "
3     then
4     echo $1 est connecte
5 fi
```

EXEMPLE

Vérifier si un utilisateur est connecté

```
1 #!/bin/sh
2 if who | grep "^$1 "
3     then
4     echo $1 est connecte
5 fi
```

```
$ ./prog.sh idiraitsadoune
idiraitsadoune   console      18 oct 07:22
idiraitsadoune   ttys001      18 oct 07:40
idiraitsadoune est connecte
```

UN AUTRE EXEMPLE

UN AUTRE EXEMPLE

Vérifier le nombre de paramètres d'une commande

```
1 #!/bin/sh
2 if test $# -eq 0
3     then
4     echo commande lancee sans parametres
5 else
6     echo commande lancee avec au moins un parametre
7 fi
```

UN AUTRE EXEMPLE

Vérifier le nombre de paramètres d'une commande

```
1 #!/bin/sh
2 if test $# -eq 0
3     then
4     echo commande lancee sans parametres
5 else
6     echo commande lancee avec au moins un parametre
7 fi
```

```
$ ./prog.sh p1 p2 p3
commande lancee avec au moins un parametre
```

UN AUTRE EXEMPLE

Vérifier le nombre de paramètres d'une commande

```
1 #!/bin/sh
2 if test $# -eq 0
3     then
4     echo commande lancee sans parametres
5 else
6     echo commande lancee avec au moins un parametre
7 fi
```

```
$ ./prog.sh p1 p2 p3
commande lancee avec au moins un parametre
```

```
$ ./prog.sh
commande lancee sans parametres
```

STRUCTURES `if` IMBRIQUÉES : `elif`

STRUCTURES `if` IMBRIQUÉES : `elif`

- Il est possible d'imbriquer des `if` dans d'autres `if`

STRUCTURES `if` IMBRIQUÉES : `elif`

- Il est possible d'imbriquer des `if` dans d'autres `if`

```
if condition1
    then instruction(s)
else
    if condition2
        then instruction(s)
    else
        if condition3
            ...
        fi
    fi
fi
```

STRUCTURES `if` IMBRIQUÉES : `elif`

- Pour permettre d'alléger ce type de code, `ksh` fournit un raccourci d'écriture

STRUCTURES `if` IMBRIQUÉES : `elif`

- Pour permettre d'alléger ce type de code, `ksh` fournit un raccourci d'écriture

```
if condition1
    then instruction(s)
elif condition2
    then instruction(s)
elif condition3
    ...
fi
```

EXEMPLE

Vérifier si une commande a des paramètres avant de l'utiliser

EXEMPLE

Vérifier si une commande a des paramètres avant de l'utiliser

```
1  #!/bin/sh
2  if test $# -eq 0
3      then
4      echo Relancer la cmde en ajoutant un parametre
5  elif who | grep "^$1 " > /dev/null
6      then
7      echo $1 est connecte
8  else
9      echo "$1 n\'est pas connecte"
10 fi
```

EXEMPLE

Vérifier si une commande a des paramètres avant de l'utiliser

```
 1 #!/bin/sh
 2 if test $# -eq 0
 3     then
 4     echo Relancer la cmde en ajoutant un parametre
 5 elif who | grep "^$1 " > /dev/null
 6     then
 7     echo $1 est connecte
 8 else
 9     echo "$1 n\'est pas connecte"
10 fi
```

```
$ ./prog.sh
Relancer la cmde en ajoutant un parametre
```

EXEMPLE

Vérifier si une commande a des paramètres avant de l'utiliser

```
 1 #!/bin/sh
 2 if test $# -eq 0
 3     then
 4     echo Relancer la cmde en ajoutant un parametre
 5 elif who | grep "^$1 " > /dev/null
 6     then
 7     echo $1 est connecte
 8 else
 9     echo "$1 n\'est pas connecte"
10 fi
```

```
$ ./prog.sh
Relancer la cmde en ajoutant un parametre
```

```
$ ./prog.sh marc
marc n'est pas connecte
```

ÉNUMÉRATION DE MOTIFS

ÉNUMÉRATION DE MOTIFS

- case **compare** une valeur avec une liste de valeurs et exécute des instructions si une des valeurs de la liste correspond.

ÉNUMÉRATION DE MOTIFS

- case **compare** une valeur avec une liste de valeurs et exécute des instructions si une des valeurs de la liste correspond.

```
case valeur_testee in
    valeur1) instruction(s);;
    valeur2) instruction(s);;
    valeur3) instruction(s);;
    * ) instruction_else(s);;
esac
```

EXEMPLE

EXEMPLE

Comparer une réponse à des expressions régulières

EXEMPLE

Comparer une réponse à des expressions régulières

```
#! /bin/sh
echo ecrivez OUI
read reponse
case ${reponse} in
    OUI)            echo bravo
                    echo merci infiniment ;;

    [Oo][Uu][Ii])   echo merci beaucoup ;;

    o*|O*)          echo un petit effort ! ;;

    n*|N*)          echo vous etes contrariant ;;

    *)              echo "ce n'est pas malin"
                    echo recommencez ;;
esac
```

LA STRUCTURE `for ... do ... done`

LA STRUCTURE `for ... do ... done`

- **La boucle** `for` permet de parcourir une liste de valeurs,
  elle effectue un nombre d'itérations qui est connu à l'avance.

LA STRUCTURE `for ... do ... done`

- **La boucle** `for` permet de parcourir une liste de valeurs,
  elle effectue un nombre d'itérations qui est connu à l'avance.

```
for variable [in liste_valeurs]
    do instruction(s)
done
```

LA STRUCTURE `for ... do ... done`

- **La boucle** `for` permet de parcourir une liste de valeurs,
  elle effectue un nombre d'itérations qui est connu à l'avance.

```
for variable [in liste_valeurs]
    do instruction(s)
done
```

- **La boucle** `for` possède une deuxième syntaxe :

LA STRUCTURE `for ... do ... done`

- **La boucle** `for` permet de parcourir une liste de valeurs,
  elle effectue un nombre d'itérations qui est connu à l'avance.

```
for variable [in liste_valeurs]
    do instruction(s)
done
```

- **La boucle** `for` possède une deuxième syntaxe :

```
for ((e1;e2;e3))
    do instruction(s)
done
```

LA STRUCTURE `for ... do ... done`

- **La boucle** `for` permet de parcourir une liste de valeurs,
  elle effectue un nombre d'itérations qui est connu à l'avance.

```
for variable [in liste_valeurs]
    do instruction(s)
done
```

- **La boucle** `for` possède une deuxième syntaxe :

```
for ((e1;e2;e3))
    do instruction(s)
done
```

  - commence par exécuter `e1`, puis tant que `e2` $\neq$ `0`
    le bloc d'instructions est exécuté et `e3` également.

EXEMPLE I

EXEMPLE I

Parcourir une liste de valeurs

EXEMPLE I

Parcourir une liste de valeurs

```
1 #! /bin/sh
2 for mot in 1 5 10 2 "la fin"
3 do
4     echo mot vaut ${mot}
5 done
```

EXEMPLE I

Parcourir une liste de valeurs

```
1 #! /bin/sh
2 for mot in 1 5 10 2 "la fin"
3 do
4     echo mot vaut ${mot}
5 done
```

```
$ ./my_prog.sh
mot vaut 1
mot vaut 5
mot vaut 10
mot vaut 2
mot vaut la fin
```

EXEMPLE II

EXEMPLE II

Parcourir les paramètres d'une commande

EXEMPLE II

Parcourir les paramètres d'une commande

```
1 #! /bin/sh
2 for param in "$*"
3 do
4     echo -${param}-
5 done
```

EXEMPLE II

Parcourir les paramètres d'une commande

```
1 #! /bin/sh
2 for param in "$*"
3 do
4     echo -${param}-
5 done
```

```
$ ./my_prog.sh a b c d
-a-
-b-
-c-
-d-
```

EXEMPLE III

EXEMPLE III

Parcourir une liste de fichiers correspondant à un motif

EXEMPLE III

Parcourir une liste de fichiers correspondant à un motif

```
1 #!/bin/sh
2 for fichier in *.f90
3 do
4     echo fichier ${fichier}
5 done
```

EXEMPLE III

Parcourir une liste de fichiers correspondant à un motif

```
#!/bin/sh
for fichier in *.f90
do
    echo fichier ${fichier}
done
```

```
#!/bin/sh
motif=$1
for fic in $(grep -l ${motif} *)
do
    echo le fichier $fic contient le motif $motif
done
```

LA STRUCTURE `while ... do ... done`

LA STRUCTURE `while ... do ... done`

- **La boucle** `while` exécute un bloc d'instructions tant qu'une certaine condition est satisfaite

LA STRUCTURE `while ... do ... done`

- **La boucle** `while` exécute un bloc d'instructions tant qu'une certaine condition est satisfaite

```
while condition
    do instruction(s)
done
```

EXEMPLE

EXEMPLE

Script qui boucle jusqu’à ce qu’un utilisateur se déconnecte

EXEMPLE

Script qui boucle jusqu'à ce qu'un utilisateur se déconnecte

```
1 #!/bin/sh
2 utilisateur=$1
3 while who | grep "^${utilisateur} " > /dev/null
4 do
5     echo ${utilisateur} est connecte
6     sleep 2
7 done
8 echo "${utilisateur} n'est pas connecte"
```

LA STRUCTURE `until ... do ... done`

LA STRUCTURE `until ... do ... done`

- **La boucle** `until` est exécutée tant que la condition est fausse.

LA STRUCTURE `until ... do ... done`

- **La boucle** `until` est exécutée tant que la condition est fausse.

```
until condition
    do instruction(s)
done
```

EXEMPLE

EXEMPLE

Script qui boucle jusqu’à ce qu’un utilisateur se connecte

EXEMPLE

Script qui boucle jusqu'à ce qu'un utilisateur se connecte

```
1 #!/bin/sh
2 utilisateur=$1
3 until who | grep "^${utilisateur} " > /dev/null
4 do
5     echo "${utilisateur} n'est pas connecte"
6     sleep 2
7 done
8 echo ${utilisateur} est connecte
```

LA COMMANDE `exit`

LA COMMANDE `exit`

- `exit` arrête l'exécution du script et rend un `statut` (0 par défaut) à l'appelant (accessible via `$?`)

```
exit [statut]
```

LA COMMANDE `exit`

- `exit` arrête l'exécution du script et rend un `statut` (0 par défaut) à l'appelant (accessible via `$?`)

```
exit [statut]
```

- Utilisé également pour arrêter le traitement en cas d'erreur
  👉 rendre alors un `statut ≠ 0`

LA COMMANDE `exit`

- `exit` arrête l'exécution du script et rend un `statut` (0 par défaut) à l'appelant (accessible via `$?`)

```
exit [statut]
```

- Utilisé également pour arrêter le traitement en cas d'erreur

  👉 rendre alors un `statut ≠ 0`

```
1 #!/bin/sh
2 if [ $# -lt 1 ] # test sur le nb d'arguments
3 then
4     echo "il manque les arguments" >&2 # sur la sortie d'erreur
5     exit 1 # sortie avec code d'erreur
6 fi
```

LA COMMANDE break

LA COMMANDE break

- La commande break permet de sortir d'une boucle avant sa fin.

```
break
```

LA COMMANDE break

- La commande break permet de sortir d'une boucle avant sa fin.

```
break
```

- La commande break n permet de sortir des n boucles les plus intérieures.

```
break n
```

LA COMMANDE `break`

- La commande `break` permet de sortir d'une boucle avant sa fin.

```
break
```

- La commande `break n` permet de sortir des `n` boucles les plus intérieures.

```
break n
```

- Nécessaire dans les boucles infinies.

  👉 `while true` ou `until false`

LA COMMANDE `break`

- La commande `break` permet de sortir d'une boucle avant sa fin.

```
break
```

- La commande `break n` permet de sortir des `n` boucles les plus intérieures.

```
break n
```

- Nécessaire dans les boucles infinies.

  👉 `while true` ou `until false`

  👉 insérée dans un bloc conditionnel pour arrêter la boucle.

EXEMPLE

EXEMPLE

Répéter une boucle jusqu'à ce qu'une valeur soit lue

EXEMPLE

Répéter une boucle jusqu'à ce qu'une valeur soit lue

```
1  #!/bin/sh
2  while true # boucle infinie
3  do
4      echo "entrer un chiffre (0 pour finir)"
5      read i
6      if [ "$i" -eq 0 ]
7      then
8          echo '**' sortie de boucle par break
9          break # sortie de boucle
10     fi
11     echo vous avez saisi $i
12 done
13 echo "fin du script"
14 exit 0
```

LA COMMANDE `continue`

LA COMMANDE `continue`

- La commande `continue` saute les commandes suivantes dans la boucle et de reprendre l'exécution en début de boucle

```
continue
```

LA COMMANDE `continue`

- La commande `continue` saute les commandes suivantes dans la boucle et de reprendre l'exécution en début de boucle

```
continue
```

- La commande `continue n` sort des `n-1` boucles les plus intérieures et reprend au début de la `n ième` boucle.

```
continue n
```

LA COMMANDE `continue`

- La commande `continue` saute les commandes suivantes dans la boucle et de reprendre l'exécution en début de boucle

```
continue
```

- La commande `continue n` sort des `n-1` boucles les plus intérieures et reprend au début de la `n ième` boucle.

```
continue n
```

- Utilisée dans un bloc conditionnel pour court-circuiter les instruction de la fin de boucle.

EXEMPLE

EXEMPLE

Afficher les 4 premières lignes d'un fichier s'il est lisible

EXEMPLE

Afficher les 4 premières lignes d'un fichier s'il est lisible

```
#!/bin/sh
for fic in *.sh
do
    echo "**************************"
    echo "< fichier ${fic} >"
    if [ ! -r "${fic}" ] # tester si le fichier existe et est lisible
        then
        echo "fichier ${fic} non lisible"
        continue # sauter la commande head
    fi
    head -n 4 ${fic}
done
exit 0
```

PLAN

Retour au plan - Retour à l'accueil

SYNTHÈSE DU COURS OS

SYNTHÈSE DU COURS OS

À l'issue de ce cours, vous êtes capables de :

SYNTHÈSE DU COURS OS

À l'issue de ce cours, vous êtes capables de :

- Comprendre le **fonctionnement d'un système informatique**

SYNTHÈSE DU COURS OS

À l'issue de ce cours, vous êtes capables de :

- Comprendre le **fonctionnement d'un système informatique**
- Résoudre des problèmes de **gestion de processus** ou threads concurrents partageant des ressources

SYNTHÈSE DU COURS OS

À l'issue de ce cours, vous êtes capables de :

- Comprendre le **fonctionnement d'un système informatique**
- Résoudre des problèmes de **gestion de processus** ou threads concurrents partageant des ressources
- Comprendre le fonctionnement de **la mémoire**

SYNTHÈSE DU COURS OS

À l'issue de ce cours, vous êtes capables de :

- Comprendre le **fonctionnement d'un système informatique**
- Résoudre des problèmes de **gestion de processus** ou threads concurrents partageant des ressources
- Comprendre le fonctionnement de **la mémoire**
- Comprendre le fonctionnement d'**un système de fichier**

SYNTHÈSE DU COURS OS

À l'issue de ce cours, vous êtes capables de :

- Comprendre le **fonctionnement d'un système informatique**
- Résoudre des problèmes de **gestion de processus** ou threads concurrents partageant des ressources
- Comprendre le fonctionnement de **la mémoire**
- Comprendre le fonctionnement d'**un système de fichier**
- Manipuler et **programmer** à l'aide du **shell Unix/Linux**

FIN

- Retour à l'accueil
- Retour au plan