Programmation et projet encadré - L7TI005

Shell Unix

Crédits supports : Serge Fleury

Yoann Dupont prenom.nom@sorbonne-nouvelle.fr
Pierre Magistry pierre.magistry@inalco.fr

2024-2025

Université Sorbonne-Nouvelle
INALCO
Université Paris-Nanterre

Introduction

Avant propos

- Unix ?
- Linux ?
- GNU/Linux ?
- Système d'exploitation
  - kernel
  - shell
  - applications
- logiciels et licences

Philosphie derrière Unix

Existence et importance du système de fichiers

Tout est fichier

- dans les années 60, ce n'était pas évident,
- même les programmes, les périphériques, le réseau, ...
- simplicité $\rightarrow$ on ne pense qu'en termes de fichiers et dossiers,
- d'où l'importance de bien les organiser !

Philosphie derrière Unix

Existence et importance du système de fichiers

Tout est fichier

- dans les années 60, ce n'était pas évident,
- même les programmes, les périphériques, le réseau, ...
- simplicité $\rightarrow$ on ne pense qu'en termes de fichiers et dossiers,
- d'où l'importance de bien les organiser !

Un ensemble de commandes pour manipuler les fichiers et leur contenu.

(depuis **la ligne de commandes**)

- Chaque commande doit faire **une** chose et la faire **bien** (K.I.S.S.)
- Chaque commande est pensée pour pouvoir interagir avec les autres.
- On fait des choses complexes en combinant des commandes simples. (*pipelines* ou *scripts*)

Le Système de fichier

Mots clefs de la section

à définir

- fichier
- dossier (ou répertoire)
- dossier «parent»
- arborescence
- racine
- dossier personnel
- dossier courant ou «de travail» (*working directory*)
- chemin absolu
- chemin relatif
- caractères de remplacement (jokers ou *wildcards*),

Le système de fichier – L'arborescence

Points clefs

- les **fichiers** sont dans un **dossier**
- pouvant eux même être dans un **dossier** « parent »
- ce qui forme une **arborescence**

Le système de fichiers – Les chemin

L'ensemble des données sur la machine sont regroupées en une seule **arborescence**, il est intéressant de pouvoir identifier les fichiers et les dossiers par un **chemin** dans l'arbre.

- **/** désigne la **racine** de l'arbre
- **~/** désigne le dossier personnel de l'utilisateur ("*HOME*")
- **./** désigne le dossier courant (*working directory*)
- **../** désigne le dossier parent

un **chemin** est formé par une suite de noms de dossiers séparés par des /, pouvant se finir par le nom d'un fichier.

Le système de fichiers – chemins absolus et relatifs

un chemin absolu indique la position d'un fichier en partant de la **racine**.

ex: `/home/pierre/PPE1`

un chemin relatif indique la position d'un fichier en partant du dossier courant.

ex: `../../dev/input/mouse3`

$\rightarrow$ habituez vous à toujours être capable de donner le chemin relatif et absolu vers les fichiers que vous manipulez.

Le système de fichier – Jokers ! (*wildcards*)

ou « caractères de remplacement »

Dans un chemin, caractères **?** et ***** ont un comportement spécial.

? peut remplacer n'importe quel caractère (unique)

***** peut remplacer n'importe quelle suite de caractères

$\rightarrow$ le chemin peut alors désigner plusieurs fichiers !

À tester avec la commande `ls`.

Mots clefs de la section

à retenir

- fichier
- dossier (ou répertoire)
- dossier «parent»
- arborescence
- racine
- dossier personnel
- dossier courant ou «de travail» (*working directory*)
- chemin absolu
- chemin relatif
- caractères de remplacement (jokers ou *wildcards*),

Confirmez que vous maîtrisez ces notions dans votre journal, ou exprimez y vos doutes. Et posez des questions la semaine prochaine !

Les Commandes

Mots clefs de la section

à découvrir

- commande
- options
- arguments
- page de man(uel)
- et moultes commandes à découvrir

Les commandes

Les commandes sont des fichiers comme les autres,

- qui ont la propriété d'être exécutables
- qui sont placés par convention dans un dossier où le système sait les trouver (typiquement `/usr/bin/`)

La syntaxe d'une commande

```
nom [-options...]  [arguments...]
```

- les **options** peuvent avoir une forme courte (`-o` avec un seul tiret)
- ou une forme longue (`--option` avec deux tirets).
- Les **arguments** sont typiquement des chemins vers des fichiers (mais pas toujours).
- La première action de l'interpréteur de commande est de *tokeniser* la ligne pour découper le nom de la commande, les options et les arguments.

$\rightarrow$ habituez vous à anticiper comment les lignes de commandes que vous saisissez vont être découper et à repérer nom, options et arguments, comme l'interpréteur

Quelques commandes à connaître

`cd`, `ls`, `pwd`, `cat`, `less`, `wc`, `echo`, `head`, `tail`, `mkdir`, `cp`, `mv`, `rm`, `rmdir`, `file...`

Et surtout: `man`

La plupart des commandes ont aussi une option `--help` qui permet d'obtenir une description concise du fonctionnement et des option de la commande

Se promener dans l'arbre

pwd ***p**rint **w**orking **d**irectory*

ls ***lis**ter le contenu d'un dossier*

cd ***c**hange **d**irectory*

copier, déplacer, supprimer ou créer des fichiers

cp **cop**ier

mv ***mov***e déplacer

rm ***rem***ove supprimer

mkdir ***mk dir***ectory créer un dossier

touch crée un fichier (effet de bord bien pratique)

zip compresser une archive `zip`

unzip décompresser une archive `zip`

tar manipuler les archives `tar`

voir et manipuler le contenu des fichiers

file donne des informations sur le type de fichier

cat lit le contenu d'un ou plusieurs fichiers

head lit le début d'un fichier

tail lit la fin d'un fichier

cut sélectionne une ou plusieurs colonnes dans un fichier tabulé

less lecteur (interactif)

Mots clefs de la section

à retenir

- commande
- options
- arguments
- page de man(uel)
- et moultes commandes à découvrir

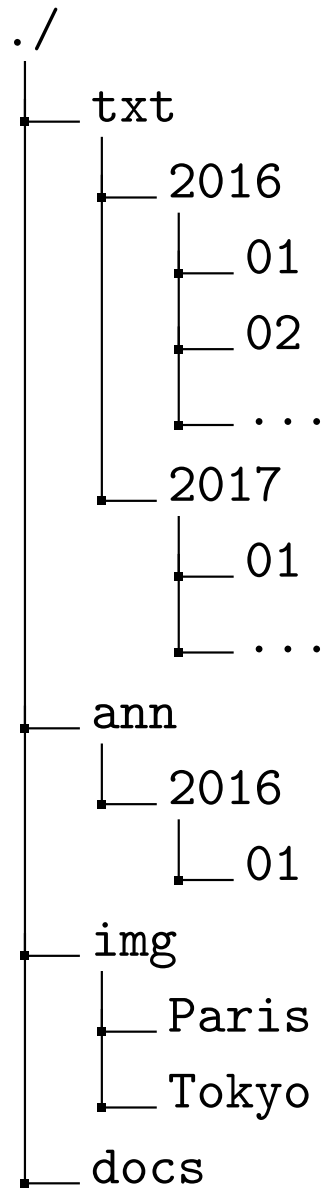
N'hésitez pas à notez les commandes et les options les plus utiles dans votre journal (les pages man sont parfois très longues !)

Exercice

L'adresse d'une archive zip a été ajoutée sur le tableur contenant la liste des inscrits au cours (un fichier par personne, en fin de chaque ligne). copier cette adresse qui sera à saisir sur icampus.
Puis dans un terminal:

- créer un dossier "Exercice1" en sous-dossier de votre répertoire personnel avec la **commande** `mkdir`
- rendez-vous ce dossier (par exemple avec la **commande** `cd ~/Exercice1`)
- Télécharger l'archive `fichiers.zip` avec la **commande** `wget` en lui donnant l'url du zip en **argument**.
- utiliser la commande `unzip` pour décompresser l'archive. (consultez le **man**uel si besoin).
- créer une arborescence pour classer les documents
  - par type de fichier (txt, ann, img, docs)
  - puis par date pour les txt et ann
  - puis par lieux pour les photos.

Exercice



Consignes pour le rendu

- L'arborescence à obtenir peut être schématiser comme ci-contre.
- créer une nouvelle archive contenant les fichiers triés
- obtenez l'historique des commandes avec la commande **history** et copier son contenu (à partir du DEBUT de l'exercice) dans un fichier texte
- le nouveau zip et le fichier texte d'historique sont à déposer sur iCampus avant lundi 30 à 23h59.

Les Pipelines

Flux d'entrées-sorties

Généralités

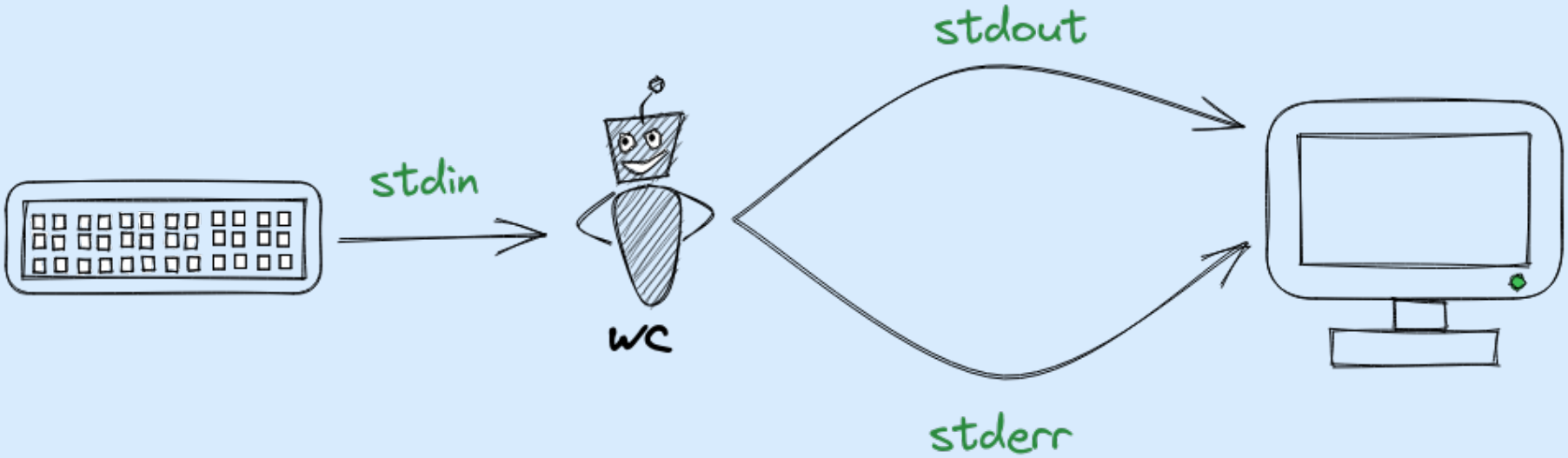
Toutes les commandes communiquent via trois flux de données

0	**stdin**	l'entrée standard (par défaut le clavier)
1	**stdout**	la sortie standard (par défaut l'écran)
2	**stderr**	la sortie d'erreurs standard (par défaut l'écran)

Par défaut, les commandes échangent dans le terminal via le clavier et l'écran, mais on peut **rediriger** ces flux.

wc

Situation par défaut



note: pour indiquer la fin de notre saisie au clavier, on utilise la combinaison de touches **Ctrl-D**

Flux d'entrées-sorties

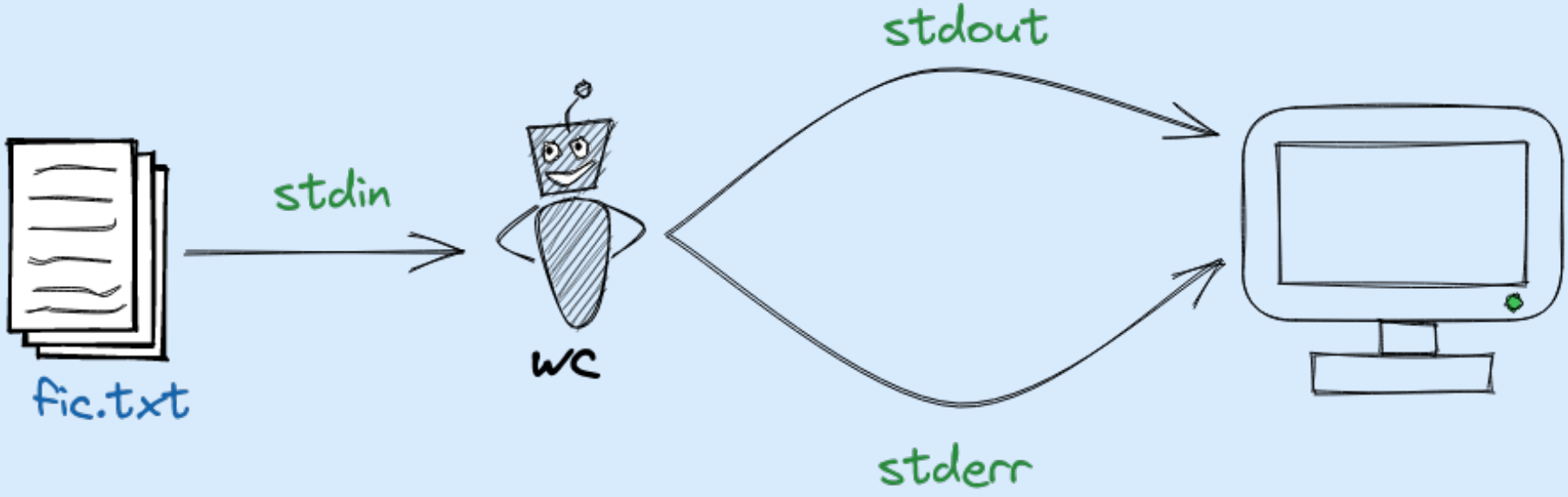
Redirections vers et depuis des fichiers

<	remplace le clavier par le contenu d'un fichier
1> ou >	écrit stdout dans un fichier
2>	écrit stderr dans un fichier
&>	écrit stdout et stderr dans un fichier

En écriture, si on double le chevron (>>, >>&, 2>>), on écrit en ajoutant la sortie à la fin d'un fichier.
ATTENTION: les chevrons simples (>, >&, 2>) écrasent le fichier si il existe déjà.

`wc < fic.txt`

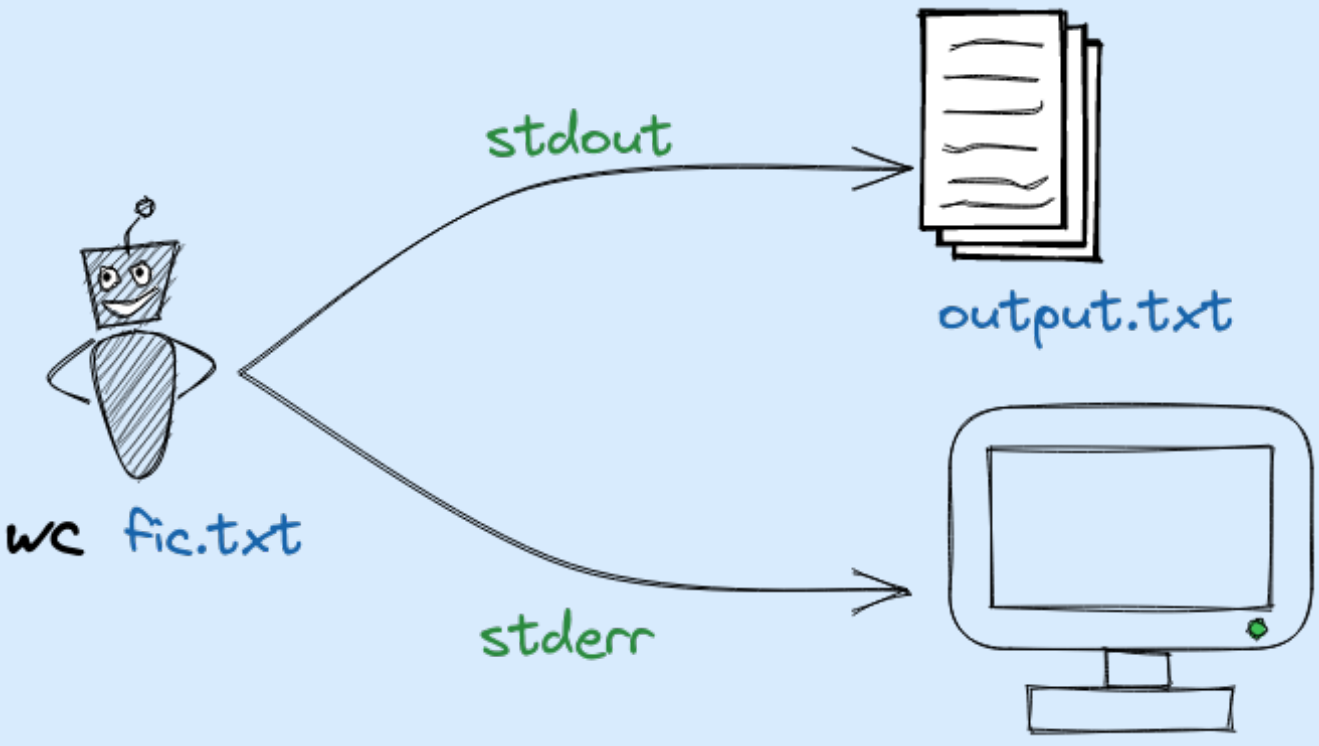
Redirection du contenu d'un fichier dans `stdin`



ATTENTION: ici `fic.txt` n'est PAS un argument.

`wc fic.txt > output.txt`

Redirection de `stderr` dans un fichier.

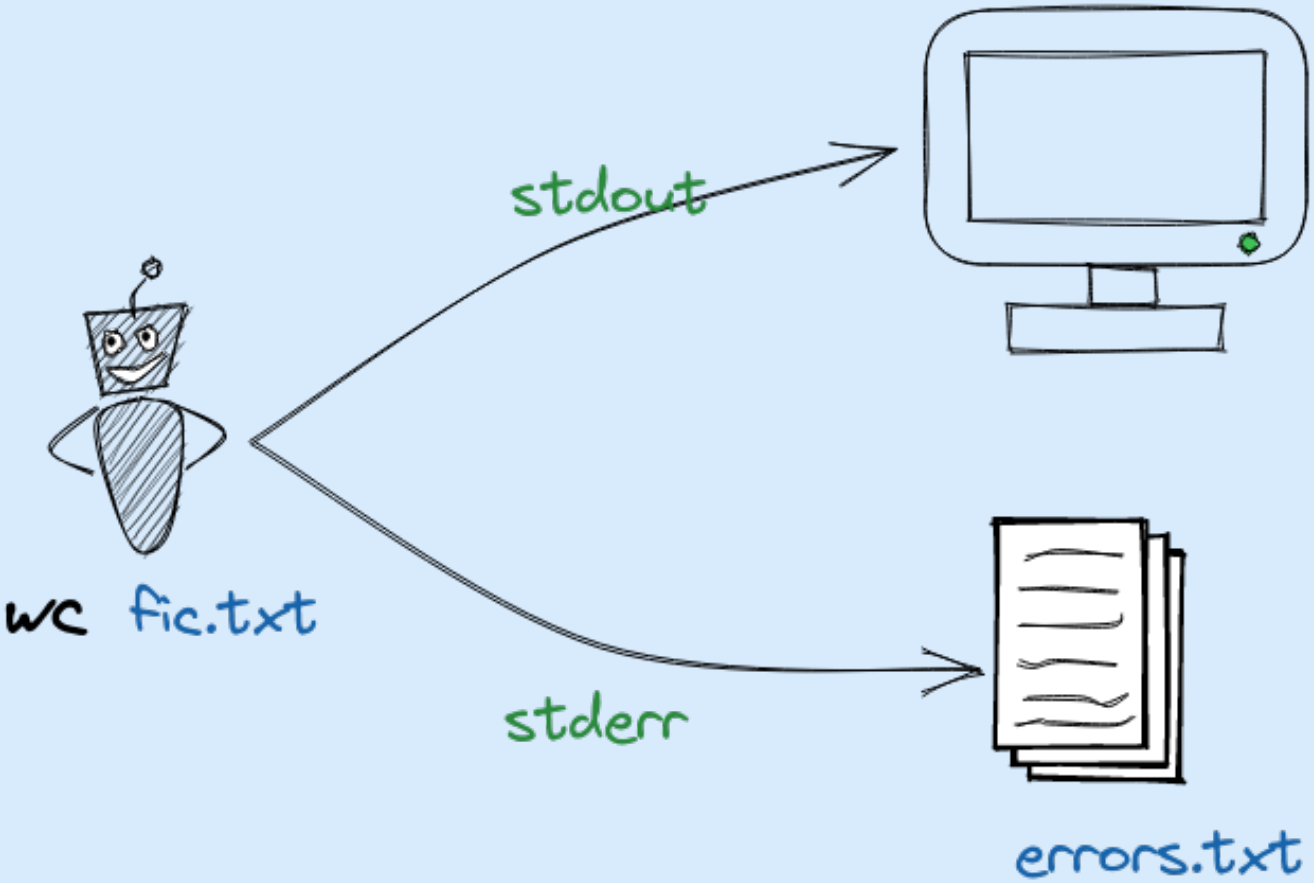


ATTENTION: ici `fic.txt` est donné en argument (et `wc` ignore l'entrée standard).

La sortie standard reste l'écran par défaut.

`wc fic.txt 2> errors.txt`

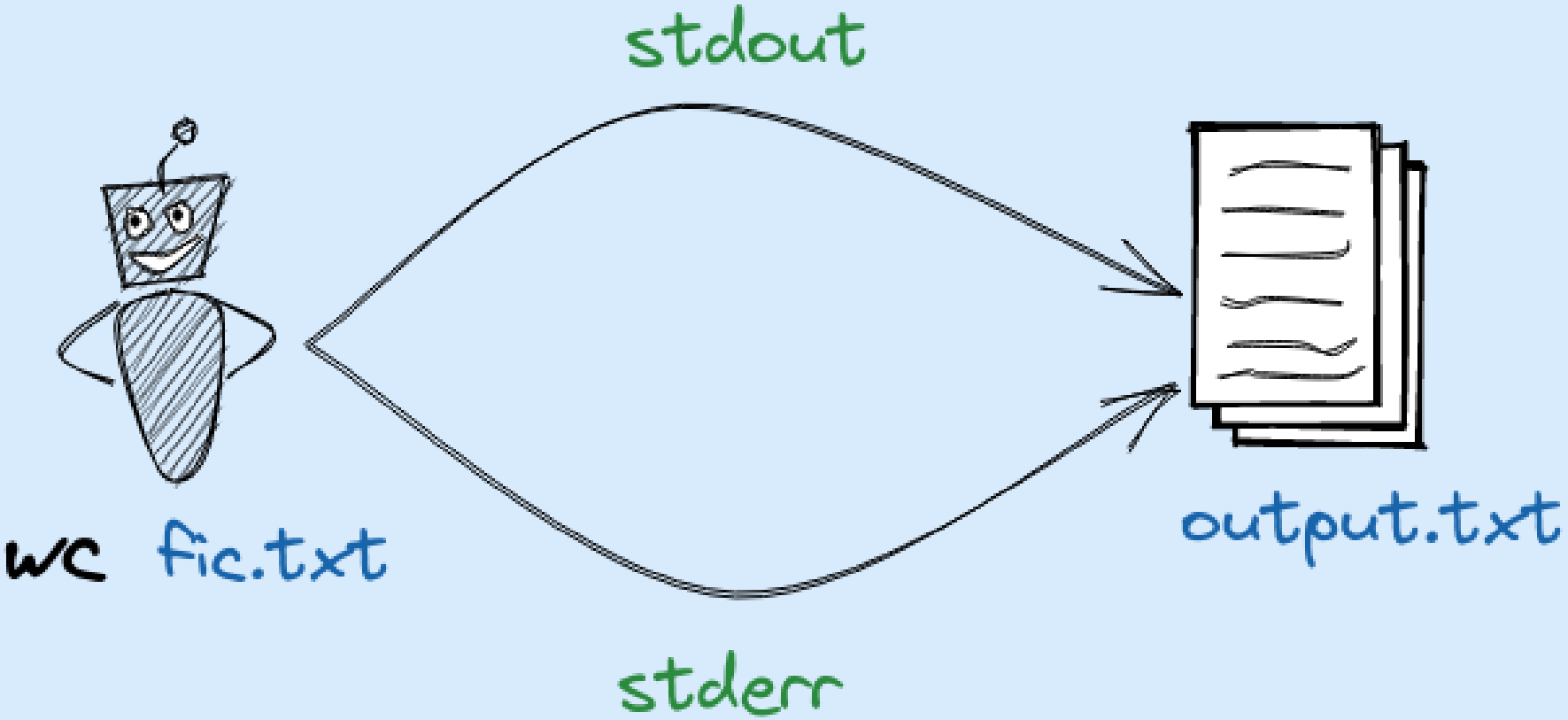
Redirection de `stderr` dans un fichier.



ATTENTION: ici `fic.txt` est donné en argument (et `wc` ignore l'entrée standard).

`wc fic.txt &> output.txt`

Redirection des sorties `stdin` et `stdout` dans un fichier.



ATTENTION: ici `fic.txt` est donné en argument
(et `wc` ignore l'entrée standard).

Flux d'entrées-sorties

Redirections entre les commandes

Plusieurs commandes peuvent communiquer en connectant les sorties aux entrées aux moyen du caractère «pipe» |

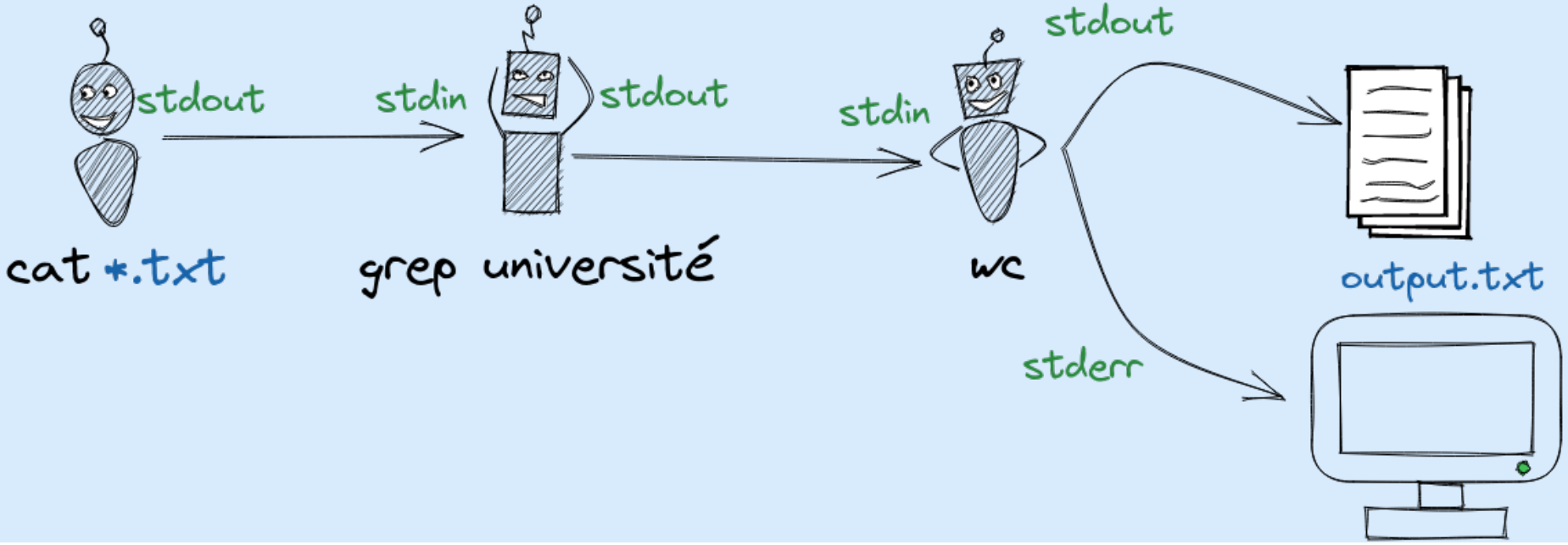
`cmd1 | cmd2` la sortie standard (stdout) de cmd1 est envoyée en stdin de cmd2.

`cmd1 |& cmd2` les sorties stdout et stderr de cmd1 sont toutes deux redirigées vers stdin de cmd2.

flux d'entrées-sorties

```
cat *.txt | grep université | wc > output.txt
```

Redirection de `stderr` dans un fichier.



Exercices

Quelques commandes de plus

grep recherche de motifs dans l'entrée (ou dans des fichiers)

sort trier des lignes

uniq supprimer les lignes qui se répètent

echo affiche un texte (pour formater vos résultats)

cut selectionne des colonnes d'un fichier tabulaire

tail ne garde que les dernières lignes d'un flux ou d'un fichier.

avec les fichiers `*.ann`, Sauriez vous... ?

- Compter le nombre d'annotations par année (2016, 2017 et 2018),
- limiter ce comptage aux lieux (Location),
- sauvegarder ces résultats dans un (seul) fichier,
- établir le classement des lieux les plus cités.
- trouver les annotations les plus fréquentes pour votre mois de naissance, toutes années confondues.

Introduction aux scripts

Un premier script

la commande bash

Une commande comme les autres qui **interprète des commandes**

- depuis l'entrée standard
- ou depuis un fichier → un **script** !

voir `premier_script.sh`

Il suffit donc d'écrire des commandes dans un fichier texte pour obtenir un script.
On peut :

- ajouter des commentaires #
- ajouter un *shebang* `#!/usr/bin/bash`
- rendre le fichier exécutable (avec **chmod +x**)

À vous !

- écrire un script qui donne le nombre de Location par année
- ajoutez-le à votre git

Arguments et variables

Arguments d'un scripts

Les commandes peuvent recevoir des arguments

- Ça permet de ne pas faire toujours la même chose
- souvent le même traitement sur des données différentes
- ou un traitement légèrement différent

Dans un script bash

Par convention, au début de l'exécution d'un script, l'interpréteur donne la valeur des arguments aux **variables** $1 $2 $3 ...
dans l'ordre d'apparition sur la ligne de commande.

- On peut les utiliser dans notre script
  (ex: faites un script qui contient uniquement la commande `echo $2` et testez son comportement)
- Un bon usage est de créer de nouvelles variables avec un nom plus explicite.
  example: `FICHIER_URLS=$1`

Les variables en bash

Affectation

On donne une valeur à une variable avec le signe =

- sans espace `FICHIER=urls.txt`
- avec des " ou ' si il y a des espaces
  `MSG="Bonjour tout le monde"`
- on peut aussi stocker le resultat d'une commande en l'écrivant dans $()
  ex: `NB_LIGNES=$(wc -l $FICHIER)`

Utilisation

On fait référence à la valeur d'une variable en prefixant son nom d'un $

- sera remplacé dans une chaîne entre " (doubles)
- mais pas entre ' (simples)

(tout ça marche aussi bien dans un script que en mode interactif).

Arguments d'un scripts – exercices

Exercice 1

- Écrire un script qui compte les entités pour une année un type d'entité donnés en argument du programme.
- Écrire un second script qui lance le script précédent trois fois, une fois pour chaque années, en prenant le type d'entité en argument.

Exercice 2

- créer un script pour établir le classement des lieux les plus cités.
- prendre en argument l'année, le mois et le nombre de lieux à afficher
- accepter * pour l'année et le mois.

Pensez à faire des **commit** et **push** vers vos dépôt git au fur et à mesure !

Instructions de contrôle

Instructions conditionnelles

Besoin

On doit être capable d'effectuer certains traitements seulement si une condition particulière est vérifiée

- Si telle condition est vraie alors on fait le traitement A, sinon on fait le traitement B
- Si les arguments sont corrects, je lance le programme, sinon j'émets un message d'erreur et j'arrête le script.

Instructions conditionnelles

Besoin

On doit être capable d'effectuer certains traitements seulement si une condition particulière est vérifiée

- Si telle condition est vraie alors on fait le traitement A, sinon on fait le traitement B
- Si les arguments sont corrects, je lance le programme, sinon j'émets un message d'erreur et j'arrête le script.

l'instruction if

```
if [ condition ]
then
    echo "la condition est valide"
else
    echo "la condition n'est pas valide"
fi
```

Conditions possibles

Sur les chemins

-f fichier	vrai si le fichier existe
-d dossier	vrai si le dossier existe
-s fichier	vrai si le fichier existe et n'est pas vide

Sur des chaînes de caractères

= ou !=	tester si deux chaînes sont identiques (=) ou différentes (!=)
< ou >	pour déterminer si in chaîne est avant ou après une autre dans l'ordre alphabétique
-n chaine	vrai si la chaîne n'est pas vide
-z	vrai si la chaîne est vide (ex: argument non fourni)

Conditions possibles (suite)

Sur les entiers

a -eq b si a est égal à b (**eq**ual)

a -ne b si a est différent de b (**n**ot **e**qual)

a -lt b si a est plus petit que b (**l**ess **t**han)

a -gt b si a est plus grand que b (**g**reater **t**han)

a -le b si a est inférieur ou égal à b

a -ge b si a supérieur ou égal à b

Conditions possibles (suite)

Avec des doubles crochets, il est possible d'utiliser des expressions régulières pour tester des chaînes

example

```
if [[ $1 =~ bon(jour|soir) ]]
then
  echo "salut"
fi
```

→ Testons tout ça ensemble...

Validation des arguments

Une bonne habitude

Le premier usage (pas le seul) de ces tests est de vérifier que toutes les conditions sont réunies pour que le traitement se passe bien avant de lancer les calculs. Et d'informer l'utilisateur de tout problème

- Les fichiers attendus existent-ils ?
- Les arguments ont-ils le bon format ?
- on peut arrêter l'execution du script à tout moment avec la commande **exit**

Validation des arguments

Une bonne habitude

Le premier usage (pas le seul) de ces tests est de vérifier que toutes les conditions sont réunies pour que le traitement se passe bien avant de lancer les calculs. Et d'informer l'utilisateur de tout problème

- Les fichiers attendus existent-ils ?
- Les arguments ont-ils le bon format ?
- on peut arrêter l'execution du script à tout moment avec la commande **exit**

Exercice

- Modifier vos programmes pour qu'ils valident leurs arguments et se terminent si il y a un problème.
- synchroniser votre git !

Les boucles

Répéter des actions sans répéter le code

- appliquer le même traitement
- à des données (variables) différentes
- en séquence

Les boucles FOR (« pour tout élément, faire... »)

```
N=0
for ELEMENT in a b c d e
do
    N=$(expr $N + 1)
    echo "le $N ieme élément est $ELEMENT"
done
```

Les boucles FOR

remarques

- le choix du nom de la variable est libre
- la commande **expr** est une calculatrice
- on utilise souvent une commande pour générer la liste d'éléments (à tester)

Les boucles FOR (« pour tout élément, faire... »)

```
N=0
for ELEMENT in a b c d e
do
    N=$(expr $N + 1)
    echo "le $N ieme élément est $ELEMENT"
done
```

Les boucles WHILE

Les boucles WHILE (« tant qu'une *condition* est vraie, on recommence... »)

```
while [ condition ];
do
    echo "je continue à boucler";
done
```

remarques

- les conditions sont similaires à celles des **IF**
- la commande **read** est souvent utilisée avec **WHILE** («tant qu'il y a quelque chose à lire, on le traite») → démo.
- attention aux boucles infinies ! (CTRL-C pour arrêter brutalement le programme).

Example: lire et expliquer ce code

```
#!/usr/bin/bash
if [ $# -ne 1 ]
then
  echo "ce programme demande un argument"
    exit
fi
FICHIER_URLS=$1
OK=0
NOK=0
while read -r LINE;
do
  echo "la ligne: $LINE"
  if [[ $LINE =~ ^https?:// ]]
  then
    echo "ressemble à une URL valide"
    OK=$(expr $OK + 1)
  else
    echo "ne ressemble pas à une URL valide"
    NOK=$(expr $NOK + 1)
  fi
done < $FICHIER_URLS
echo "$OK URLs et $NOK lignes douteuses"
```