

04B : PATH, flux et outils de texte

420-3A6-ST : Programmation système

Dany Gagnon

Cégep de Saint-Jean-sur-Richelieu

Mercredi 16 septembre 2026

Le parcours de la séance

- ① Comprendre comment Bash trouve une commande avec `PATH`.
- ② Identifier l'entrée standard, la sortie standard et la sortie d'erreur.
- ③ Rediriger ces flux vers des fichiers ou un autre programme.
- ④ Sélectionner des lignes avec `grep` et composer quelques outils.
- ⑤ Pratiquer sur des fichiers et des scripts courts.

PATH : les répertoires où chercher un exécutable

`PATH` est une variable d'environnement contenant une liste de répertoires séparés par `:`.

```
printf '%s\n' "$PATH"
```

Exemple de valeur :

```
/home/alex/.local/bin:/usr/local/bin:/usr/bin
```

Pour chercher un programme externe par son nom, Bash parcourt ces répertoires dans cet ordre et retient le premier exécutable trouvé.

Identifier la commande qui sera utilisée

`command -v bash`

`type -a printf`

- `command -v` indique ce que Bash utiliserait pour ce nom.
- `type -a` affiche les différentes définitions connues : alias, fonction, commande interne ou exécutable.
- `printf` existe comme commande interne de Bash ; son appel habituel ne nécessite aucune recherche dans `PATH`.

Un nom de commande ou un chemin explicite

saluer-cours

./bin/saluer-cours

- `saluer-cours` : Bash doit résoudre le nom ; pour notre script, il cherche dans `PATH`.
- `./bin/saluer-cours` : le chemin contient `/` ; Bash utilise directement ce chemin.
- Le répertoire courant est consulté par `PATH` seulement s'il y figure.

Ajouter le dossier de nos scripts à PATH

Depuis le dossier d'essai préparé avec le README, ouvrir un shell :

```
bash
export PATH="$PWD/bin:$PATH"
command -v saluer-cours
saluer-cours
exit
```

`$PWD/bin` est ajouté au début ; `$PATH` conserve la liste existante. `export` transmet la valeur aux programmes lancés ensuite.

`exit` ferme le shell d'essai : le shell parent conserve son ancien `PATH`.

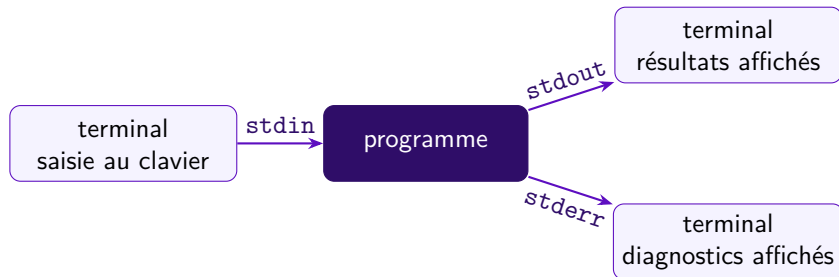
Un flux est une suite de données lues ou écrites

Un programme peut lire des données depuis une source et écrire des données vers une destination. On appelle ces échanges des **flux**.

Numéro	Nom	Sens	Usage habituel
0	<code>stdin</code>	vers le programme	données à lire
1	<code>stdout</code>	depuis le programme	résultats
2	<code>stderr</code>	depuis le programme	diagnostics

Ces trois flux sont normalement ouverts au lancement d'une commande.

Les connexions habituelles au terminal



Les deux sorties s'affichent au même endroit, mais restent distinctes. Bash peut changer la source ou la destination de chaque flux.

cat lit puis recopie sur la sortie standard

Sans nom de fichier, `cat` lit son entrée standard :

```
cat
```

Saisir `Salut`, puis Entrée : la ligne est recopiée. Sur une ligne vide, **Ctrl+D** signale la fin de l'entrée à `cat`.

Avec un argument, `cat` ouvre le fichier indiqué :

```
cat messages.txt
```

Dans les deux cas, le contenu lu est écrit sur `stdout`.

Écrire un résultat et un diagnostic

Le fichier fourni `flux.sh` contient :

```
#!/usr/bin/env bash
printf '3 fichiers traites\n'
printf 'Attention : fichier ignore\n' >&2
```

`printf` écrit normalement sur `stdout`. La redirection `>&2` envoie le second message sur `stderr`. Lancer `bash flux.sh` : les deux lignes apparaissent au terminal. Nous allons ensuite enregistrer chaque sortie séparément.

Rediriger la sortie avec > et >>

```
printf 'Lundi\n' > jours.txt  
printf 'Mercredi\n' >> jours.txt  
cat jours.txt
```

- > crée le fichier ou vide son contenu avant d'écrire.
- >> crée le fichier au besoin et ajoute à la fin.

Le fichier contient maintenant deux lignes : **Lundi**, puis **Mercredi**.

Rediriger l'entrée avec <

```
cat < messages.txt
```

Bash ouvre `messages.txt` et le relie à `stdin`. `cat` reçoit son contenu par l'entrée standard.

```
cat < messages.txt > copie.txt
```

Ici, `stdin` vient de `messages.txt` et `stdout` va vers `copie.txt`.

Enregistrer stdout et stderr séparément

```
bash flux.sh > resultat.txt 2> erreurs.txt  
cat resultat.txt  
cat erreurs.txt
```

Fichier	Contenu
resultat.txt	3 fichiers traites
erreurs.txt	Attention : fichier ignore

2> redirige le flux numéro 2. 2>> ajoute ses messages à la fin d'un fichier.

Réunir les deux sorties avec `2>&1`

```
bash flux.sh > execution.log 2>&1
```

Bash traite les redirections de gauche à droite :

- ➊ `> execution.log` dirige `stdout` vers le fichier.
- ➋ `2>&1` donne à `stderr` la destination actuelle de `stdout`.

Les deux messages sont enregistrés dans `execution.log`.

Les redirections à reconnaître

Syntaxe	Effet
<code>< fichier</code>	fournir le fichier à l'entrée standard
<code>> fichier</code>	écrire stdout dans un fichier créé ou vidé
<code>>> fichier</code>	ajouter stdout à la fin d'un fichier
<code>2> fichier</code>	écrire stderr dans un fichier créé ou vidé
<code>2>> fichier</code>	ajouter stderr à la fin d'un fichier
<code>2>&1</code>	donner à stderr la destination actuelle de stdout

Le shell effectue ces connexions avant de lancer la commande.

Le pipeline | relie deux commandes

`commande_gauche | commande_droite`

La sortie standard de gauche devient l'entrée standard de droite.

`printf 'Lundi\nMercredi\n' | wc -l`

`wc -l` compte les caractères de fin de ligne reçus. L'exemple affiche 2.

Un pipeline transmet stdout

```
bash flux.sh | wc -l
```

`wc` reçoit seulement 3 fichiers traites suivi d'un saut de ligne. Il affiche 1. Le diagnostic apparaît encore au terminal.

```
bash flux.sh 2>&1 | wc -l
```

Cette fois, les deux messages passent dans le tube : `wc` affiche 2.

grep affiche les lignes qui correspondent

Forme générale :

```
grep [options] motif [fichier...]
```

```
grep -F 'ERREUR' journal.log
```

-F recherche le texte donné littéralement. Dans le journal fourni, cet appel affiche :

```
ERREUR photos
```

```
ERREUR musique
```

grep peut lire son entrée standard

Sans argument de fichier, `grep` lit les données reçues sur `stdin` :

```
grep -F 'ERREUR' < journal.log  
cat journal.log | grep -F 'ERREUR'
```

Ces deux appels sélectionnent les mêmes lignes que `grep -F 'ERREUR' journal.log`.

Le pipeline sert ici à observer la connexion. Quand le fichier est déjà connu, l'argument de fichier suffit.

Les options courantes de grep

Option	Effet
-F	chercher un texte littéral
-n	afficher le numéro de chaque ligne retenue
-i	ignorer les différences de majuscules et minuscules
-v	retenir les lignes qui ne correspondent pas
-c	afficher le nombre de lignes retenues
-x	exiger une correspondance sur toute la ligne

```
grep -nF 'ERREUR' journal.log
```

L'exemple affiche `2:ERREUR photos` et `5:ERREUR musique`.

Décrire un motif avec une expression régulière

`grep -E` interprète le motif comme une expression régulière étendue.

Syntaxe	Sens
<code>^</code>	début de ligne
<code>\$</code>	fin de ligne
<code>.</code>	un caractère
<code>.*</code>	une suite de zéro ou plusieurs caractères
<code>[0-9]</code>	un chiffre de 0 à 9

```
grep -E '^INFO .*cours$' journal.log
```

Deux lignes identiques sont affichées : `INFO cours`. Les apostrophes transmettent le motif à `grep`.

Le motif de fichier et le motif de texte

```
grep -E '^ERREUR' *.log
```

- ❶ Bash développe `*.log` en noms de fichiers du répertoire courant.
- ❷ `grep` reçoit ces noms en arguments et ouvre les fichiers.
- ❸ `grep` interprète `'^ERREUR'` et sélectionne les lignes qui commencent par `ERREUR`.

Le code de sortie de grep

Pour les appels montrés avec GNU `grep` :

Code	Signification
0	au moins une ligne sélectionnée
1	aucune ligne sélectionnée
2	erreur, par exemple un fichier inaccessible

```
grep -F 'ABSENT' journal.log  
printf 'Statut : %s\n' "$?"
```

L'exemple n'affiche aucune ligne du journal, puis affiche `Statut : 1`.

Afficher, compter et trier du texte

Commande	Action
<code>cat fichier</code>	afficher le contenu
<code>head -n 3 fichier</code>	afficher les trois premières lignes
<code>wc -l < fichier</code>	compter les caractères de fin de ligne
<code>sort fichier</code>	trier les lignes selon la locale active
<code>sort fichier uniq</code>	trier, puis supprimer les doublons consécutifs

```
grep -F 'INFO' journal.log | sort | uniq
```

Construire et vérifier un pipeline par étapes

```
grep -F 'ERREUR' journal.log  
grep -F 'ERREUR' journal.log | wc -l  
grep -F 'ERREUR' journal.log | wc -l > compte.txt
```

- ❶ Vérifier les deux lignes retenues par `grep`.
- ❷ Vérifier que `wc` affiche 2.
- ❸ Enregistrer ce nombre dans `compte.txt`.

find sélectionne des chemins

```
find documents -maxdepth 1 -type f -name '.*' -print
```

- `documents` : point de départ de la recherche.
- `-maxdepth 1` : examiner le premier niveau.
- `-type f` : retenir les fichiers réguliers.
- `-name '.*'` : retenir les noms commençant par un point.

Dans le dossier fourni, le résultat est `documents/.notes`.

Atelier : expliquer puis exécuter

Depuis le dossier de démonstration `04B-flux-de-texte` :

- ➊ Dans un shell d'essai, rendre `saluer-cours` accessible par son nom ; vérifier le chemin choisi.
- ➋ Exécuter `flux.sh` en séparant résultat et diagnostic dans deux fichiers.
- ➌ Extraire les lignes `ERREUR` du journal, puis compter ces lignes.
- ➍ Trouver les fichiers réguliers cachés de `documents`.
- ➎ Exporter une variable de démonstration et enregistrer sa valeur avec `printenv` et une redirection.

Les notions à réutiliser dans les prochains programmes

- `PATH` permet de retrouver un exécutable à partir de son nom.
- Les arguments sont donnés au lancement ; `stdin` fournit les données à lire pendant l'exécution.
- `stdout` contient les résultats ; `stderr` permet de garder les diagnostics séparés.
- Les redirections et les pipelines choisissent la source ou la destination de ces flux.

Un programme C qui lit `stdin` pourra recevoir une saisie au terminal, le contenu d'un fichier ou la sortie d'une autre commande.