R1.04 - Systèmes - TP 4
Redirections & Tubes

Canaux

Tout processus dispose, dès son lancement, de 3 canaux de communication :

- Un canal d'entrée, appelé **`STDIN`** (**ST**andar**D IN**put)
- Deux canaux de sortie appelés **`STDOUT`** (**ST**andar**D OUT**put) et **`STDERR`** (**ST**andar**D ERR**or output)

Chaque canal est aussi connu par un *numéro* :

- **`STDIN`** = numéro **`0`**
- **`STDOUT`** = numéro **`1`**
- **`STDERR`** = numéro **`2`**

STDIN

Par défaut[1], le canal d'entrée (**`STDIN`**) est le clavier.

C'est sur ce canal que la fonction C **`scanf(...)`** lit ses données.

On dit que **`STDIN`** est attaché au clavier.

STDOUT

Par défaut, le canal de sortie standard (**`STDOUT`**) est l'écran. Plus précisément, il s'agit d'un **`TTY`**[2], qui se trouve être, dans un environnement graphique, une fenêtre, donc une portion de l'écran. Si vous êtes dans un environnement texte, c'est généralement l'écran complet.

`STDOUT` est utilisé pour les affichages d'un programme, d'un processus, dans le cadre d'un fonctionnement normal. Par exemple l'affichage du résultat d'une commande, comme **`ls`** qui affiche une liste d'objets du Système de fichiers.

C'est sur ce canal que la fonction C **`printf(...)`** écrit ses données.

On dit que **`STDOUT`** est attaché à l'écran.

[1] Dans un Terminal.
[2] Voir TP 3 - Processus.

STDERR

Par défaut, le canal de sortie des erreurs (**STDERR**) est aussi l'écran (même explication que pour **STDOUT**).

STDERR est utilisé pour l'affichage des messages d'erreur, quand quelque chose ne se déroule pas normalement. Par exemple, faire un **mkdir** alors qu'un dossier de même nom existe déjà va provoquer une erreur. Le message d'erreur sera écrit sur **STDERR**.

A ce stade, vous n'avez pas encore vu, en Algo/Prog, comment écrire sur cet autre canal en C. Pour info, on utilise **fprintf(stderr, ...)** dont la syntaxe est très proche de celle d'un **printf(...)** mais sur **STDERR** au lieu de **STDOUT**.

On dit aussi que **STDERR** est attaché à l'écran.

Comme **STDOUT** et **STDERR** ont, par défaut, la même destination : l'écran, on ne se rend pas compte au 1er abord qu'il y a deux canaux différents, en fonction de la nature des affichages (normaux vs erreurs). On va voir dans ce TP comment on peut les distinguer.

A noter qu'il est aussi possible d'écrire sur **STDOUT** en utilisant une écriture similaire, à l'aide d'un **fprintf(stdout, ...)** qui est une autre forme d'écriture du **printf(...)** que vous connaissez déjà en Algo/Prog.

Redirections

En ligne de commande, on peut rediriger les trois canaux depuis et vers des fichiers.

Le 1er usage des redirections est la redirection des sorties standard (**STDOUT**) et d'erreur (**STDERR**).

Rediriger signifie qu'au lieu d'avoir des affichages à l'écran, ce sont des fichiers qui vont servir de réceptacles et plus rien ne va s'afficher dans la fenêtre du Terminal.

L'intérêt premier est de pouvoir conserver une trace du résultat produit par une commande, pour pouvoir le consulter ultérieurement, parce qu'il est très long (nombre de lignes affichées), parce que le traitement dure très longtemps et qu'on ne va pas rester devant l'écran, parce que le traitement se passe la nuit en mode automatique et sans aucun écran connecté, etc. Les raisons sont diverses et variées. L'expérience vous montrera que les raisons valables et pratiques ne manquent pas.

STDOUT

Pour rediriger le **STDOUT** d'une commande on utilise la syntaxe suivante :

```
commande [options] [paramètres] > fichier
```

A l'instar des Jokers qui sont traités en amont par le Shell, les redirections sont aussi traitées par le Shell.

Cela signifie qu'une commande, un programme quelconque, notamment un programme que vous pouvez produire dans un TP d'Algo/Prog, n'a pas à prévoir de gérer ces redirections. Toutes les commandes, tous les programmes sont compatibles nativement et automatiquement avec les redirections, sans rien avoir à faire de spécial.

C'est totalement transparent et le programme ne sait même pas s'il écrit sur l'écran où dans un fichier en cas de redirection. Le programme fait des **`printf(...)`** et c'est tout en ce qui le concerne. Le Shell s'occupe de l'éventuelle partie redirection.

Expérimentez en lançant ceci :

```
cd
ls -l
```

Rappel : une commande **`cd`**, sans aucun paramètre derrière, permet de vous déplacer dans votre *Home*.

Observez ce qu'affiche cette commande. A ce stade vous devriez déjà le savoir, rien de bien nouveau.

Lancez maintenant ceci dans le même Terminal :

```
ls -l > mes_fichiers
```

Il est important de respecter les espaces devant et derrière le caractère : **`>`**

Q.1 Qu'affiche cette commande cette fois-ci ? Il s'agit pourtant bien de la même commande (avant le **`>`**), mais que s'est-il passé ?

Pour confirmer votre intuition (qu'on espère être bonne, mais si vous n'en avez aucune, faites appel à votre enseignant.e), vérifiez la présence d'un nouveau fichier nommé **`mes_fichiers`**. Que contient-il ? Comprenez-vous ce qui s'est passé avec la commande précédente ?

Essayez maintenant ceci :

```
ls -l zorglub > autre_fichier
```

Q.2 Cette fois-ci, avez-vous un affichage ? Quelle est la nature de ce qui s'est affiché si tel est le cas ? Est-ce que le fichier **autre_fichier** a été créé ? Que contient-il ?

STDERR

Pour rediriger le **STDERR** (qui est le canal n°**2**[3]) d'une commande on utilise la syntaxe suivante :

```
ls -l zorglub 2> other_file
```

> Il est important de respecter l'absence d'espace entre les caractères : **2** et **>**. Il est impératif qu'ils soient collés l'un à l'autre, et comme on l'a déjà dit, on respecte aussi les espaces autour de ce **2>**.

Q.3 Mêmes questions : cette fois-ci, avez-vous un affichage ? Est-ce que le fichier **other_file** a été créé ? Que contient-il ?

Avec un **2>** on ne redirige pas **STDOUT** mais **STDERR**, la sortie des erreurs de la commande.

En laissant la commande utiliser l'écran comme canaux de sortie, on ne discerne pas de différence entre l'affichage normal d'une commande (ce que la commande produit comme résultat) et l'affichage de ses éventuelles erreurs. Les deux types d'informations sont envoyés à l'écran, dans le Terminal, et s'entremêlent même parfois.

En redirigeant chaque canal séparément, on confirme déjà l'existence de ces deux canaux, leur différence de fonction et on peut ainsi séparer le résultat de ce que produit la commande et les informations périphériques, des erreurs dans le cas présent.

Ces deux types de redirection sont combinables. Testez ceci :

[3] On l'a indiqué en introduction de ce TP.

```
cat /etc/passwd /etc/shadow > list_users 2> errors
```

Attention à bien orthographier : c'est **/etc** pas **/ect** !

Pour comprendre, il faut savoir que les fichiers **/etc/passwd** et **/etc/shadow** sont deux fichiers système qui contiennent des informations sur les utilisateurs qui peuvent se connecter à l'ordinateur. Le 1er fichier est lisible par quiconque, mais vous n'avez pas le droit de lire le second.

Q.4 Avez-vous eu un affichage ? Consultez le contenu des fichiers **list_users** et **errors**. Comprenez-vous ce que vous y trouvez ?

Essayez cette autre façon d'écrire (inversion de l'ordre d'apparition de **>** et **2>**)

```
cat /etc/passwd /etc/shadow 2> errors > list_users
```

Q.5 S'est-il passé quelque chose de différent ? Consultez le contenu des fichiers **list_users** et **errors**.

On notera au passage que **STDOUT** étant le canal n°**1**, il serait donc tout à fait légitime d'écrire **1>** au lieu d'un simple **>**, mais pour **STDOUT** ce n'est pas nécessaire.

STDOUT + STDERR

Il peut arriver qu'on souhaite envoyer **STDOUT** et **STDERR** dans le même fichier de destination, un peu comme ce qui s'affiche à l'écran, un entremêlement des informations produites par la commande et des informations périphériques que sont les erreurs. Après tout, chacun peut décider de la façon dont il souhaite obtenir le résultat d'une commande.

Testez cet exemple :

```
cat /etc/passwd /etc/shadow > list_users 2>&1
```

Il est important de respecter l'absence d'espace dans l'écriture de la partie : **2>&1**. Il est impératif que tous les caractères soient collés les uns aux autres.

Q.6 Consultez le contenu du fichier **list_users**. Comparez son contenu avec ce qui s'affiche avec seulement :

```
cat /etc/passwd /etc/shadow
```

Est-ce identique ?

Voici comment il faut comprendre la syntaxe : **> nom_fic 2>&1**

- Chaque partie est à prendre séquentiellement de gauche à droite
- Le **>** doit se lire ainsi : *Envoie le canal **1 (STDOUT)** vers …* et ce qui suit donne la destination : ici il s'agit d'un fichier nommé **nom_fic**
- Le **2>** doit se lire ainsi : *Envoie le canal **2 (STDERR)** vers …* et ce qui suit donne la destination : ici il s'agit de **&1**, une façon de dire qu'il s'agit du même canal que le n°**1**. Dans notre exemple, le canal **1** (**STDOUT**) vient juste d'être redirigé vers le fichier **nom_fic**, il en sera donc de même pour le canal **2** (**STDERR**)

Q.7 Mais saurez-vous comprendre cette commande et pourquoi le résultat n'est pas le même cette fois-ci ? Faites-vous confirmer et expliquer la différence par l'enseignant.e. Suivez l'explication donnée ci-dessus pour vous y aider :

```
cat /etc/passwd /etc/shadow 2>&1 > list_users
```

Notez que ce **&** dans **2>&1** n'a rien à voir avec le **&** d'exécution en arrière plan[4].

[4] Voir TP 3 - Processus

Concaténation

> Attention, les redirections des sorties **STDOUT** et **STDERR** provoquent l'écrasement du fichier de redirection s'il existe déjà !
>
> Le fichier de sortie est créé quel que soit le comportement de la commande qui précède le symbole de redirection de sortie, y compris si la commande n'existe pas, le fichier sera quand même créé ou écrasé.

Il est possible de ne pas écraser le fichier de redirection, en ajoutant simplement à la suite de ce qui est déjà présent dans le fichier. Au travers d'un exemple, voici la syntaxe à utiliser :

```
echo DEBUT > resultat
ls >> resultat
echo FIN >> resultat
```

A noter que, comme avec un **>**, si le fichier n'existe pas il sera créé aussi avec un **>>**

Il existe aussi une version pour **STDERR** :

```
cat /etc/passwd /etc/shadow > resultat 2> errors
cat /etc/group /etc/gshadow >> resultat 2>> errors
```

STDIN

Après la redirection des sorties (**STDOUT** et **STDERR**) d'un processus, nous allons parler du dernier canal, qui est le 1er par son numéro (**0**). Il s'agit de **STDIN**, l'unique canal d'entrée de tout processus. Par défaut, dans un Terminal, il s'agit du clavier.

Pour rediriger le **STDIN** d'une commande on utilise la syntaxe suivante :

```
commande [options] [paramètres] < fichier
```

Mais, que signifie *rediriger le **STDIN** depuis un fichier* ?

Rediriger un canal de sortie d'un processus *vers un fichier*, c'est écrire ses **`printf(...)`** ou **`fprintf(...)`** *dans un fichier* au lieu de l'écran.

Rediriger un canal d'entrée d'un processus *depuis un fichier*, c'est lire ses **`scanf(...)`** *depuis un fichier* au lieu du clavier.

Attention à ne pas confondre le sens des **<** et **>**. Un **>** créera ou écrasera le fichier, un **<** lira dans le fichier, qui doit donc exister.

Ces chevrons[5] sont à comprendre, à interpréter, comme des flèches qui indiquent le sens des données : soit du fichier vers le processus pour le **<**, soit du processus vers un fichier pour le **>**

Un mauvais sens et c'est le fichier qui est définitivement écrasé et perdu alors qu'on voulait se servir de son contenu pour alimenter le **`STDIN`** d'une commande !

A partir du moment où on redirige **`STDIN`**, le processus n'a plus accès au clavier. D'une certaine manière, le clavier c'est désormais le contenu du fichier.

Rediriger **`STDIN`** nécessite de se projeter dans le futur. Que va demander le processus ? Quelles vont être ses lectures au clavier ?

C'est donc un exercice moins immédiat et moins évident qu'une redirection des canaux de sortie. Pour les redirections sortantes, on n'a pas réellement besoin de connaître le comportement du processus, de la commande. Pour la redirection entrante, il est nécessaire de bien connaître son comportement attendu.

Voici un code source qui pourrait être issu d'un exercice d'Algo/Prog. **Contrairement au TP3, on vous demande de comprendre, dans les grandes lignes, ce qu'il fait**. Il est récupérable sur Moodle. Un symbole ⤶ indique que la ligne se poursuit sur la ligne suivante (ne formant ainsi qu'une seule et même ligne de code).

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main() {
    int val, somme = 0, nb_vals = 0, ret;
    bool fini = false;

    while (fini == false) {
        printf("Entrez une valeur entière strictement positive ou ⤶
```

[5] C'est le nom typographique des symboles **<** et **>**

```
0 pour terminer : ");
       scanf("%d", &val);
       if (val > 0) {
           somme = somme + val;
           nb_vals = nb_vals + 1;
       } else if (val == 0) {
           fini = true;
       } else {
           fprintf(stderr, "%d est une valeur erronée,  ↵
recommencez\n", val);
       }
   }
   if (nb_vals > 0) {
       printf("La moyenne vaut %.2f\n", (float)somme /  ↵
(float)nb_vals);
       ret = EXIT_SUCCESS;
   } else {
       fprintf(stderr, "Impossible de calculer la moyenne !\n");
       ret = EXIT_FAILURE;
   }
   return ret;
}
```

Compilez-le pour produire un exécutable **moy**, puis exécutez-le et testez-le :

```
./moy
```

Si vous n'avez pas écrit un tel code en TP d'Algo/Prog, vous avez certainement écrit des choses très similaires (lecture des valeurs en boucle jusqu'à rencontrer une valeur de fin).

Au passage, vous noterez l'usage des **fprintf(stderr, ...)** qui permettent d'écrire les erreurs sur **STDERR**. Même si vous n'avez sans doute pas encore vu ça en Algo/Prog, on profite ici de l'occasion pour en apprendre un peu plus, c'est cadeau !

Après un test normal, testez le programme avec des valeurs erronées pour voir comment il réagit. Testez-le aussi en saisissant directement une valeur **0** dès le départ.

Vous allez maintenant remplacer vos saisies au clavier par des données provenant d'un fichier texte. C'est donc un peu comme si on simulait des frappes au clavier. Vous allez voir que c'est finalement très pratique et super efficace quand on est amené à tester de

nombreuses fois un programme en cours de développement (en contrôle TP par exemple ?).

Créez un fichier texte nommé **`vals.txt`**, avec ce contenu (en respectant les sauts de ligne). Vous pouvez utiliser un éditeur de texte (vim, nano, Geany, VSC, etc.) :

```
5
12
8
20
7
0
```

Testez maintenant la commande suivante :

```
./moy < vals.txt
```

Q.8 Avez-vous vu s'afficher des demandes de saisie d'une valeur au clavier ? Combien de fois ? Avez-vous réellement eu l'occasion de saisir ne serait-ce qu'une valeur ? Le programme a-t-il quand même fonctionné et affiché un résultat ? Le résultat correspond-il à la moyenne des valeurs du fichier **`vals.txt`** ?

Vous devez avoir compris que les saisies au clavier ont été remplacées par chacune des lignes du fichier **`vals.txt`**

Vous pouvez tester avec d'autres valeurs et vous pouvez aussi saisir beaucoup plus de lignes et observer ainsi que :

- L'exécution est immédiate
- Le procédé est réitérable et duplicable à l'envi, ce qui est appréciable pour faire des tests. Vous pouvez ainsi avoir plusieurs jeux de test (plusieurs fichiers de données) sans avoir à ressaisir de nombreuses fois de longues séries de valeurs au moment de l'exécution de votre programme.
- L'affichage des demandes de saisie de valeurs est un peu bizarre, les lignes s'affichent les unes derrière les autres et non pas l'une sous l'autre. Faites-vous expliquer cela par l'enseignant.e si vous ne comprenez pas pourquoi[6].

[6] Ce n'est pas évident à deviner.

Q.9 Reprenez l'exercice 3 du TP 3 d'Algo/Prog, qui se prête bien à cet exercice et essayez de le faire fonctionner à l'aide d'une redirection entrante au lieu de saisies au clavier.

Enfin, vous pouvez aussi combiner tout ce qu'on vient de voir de cette façon :

```
./moy < vals.txt > result 2> errors
```

Vous devez alors avoir le résultat et les éventuelles erreurs dans les fichiers **`result`** et **`errors`**. Testez avec des valeurs dans **`vals.txt`** permettant de provoquer des erreurs.

Tubes

Maintenant que vous maîtrisez à la perfection les redirections d'entrée (**`STDIN`**) et de sorties (**`STDOUT`** et **`STDERR`**) des processus, nous allons voir un mécanisme qui s'appelle les tubes et qui exploite ces redirections.

Sans tube

Commençons par une expérience. Voici une nouvelle commande qui permet d'afficher, à l'écran, une séquence de valeurs. Testez-la :

```
seq 1 1000
```

Toutes les valeurs de l'intervalle mathématique **`[1, 1000]`**[7] s'affichent à l'écran, une valeur par ligne.

Modifiez la commande en ajoutant un paramètre **`17`** entre le **`1`** et le **`1000`** :

```
seq 1 17 1000
```

Toutes les valeurs de l'intervalle mathématique **`[1, 1000]`** s'affichent à l'écran mais cette fois-ci par saut de **`17`**, une valeur par ligne.

On aimerait savoir combien il y a de valeurs au total dans cette séquence.

[7] Il s'agit bien d'un intervalle mathématique cette fois-ci. Absolument rien à voir avec les listes de symboles qu'on a vues dans le TP3 sur les Jokers.

Avant de vous lancer dans l'aventure, voici une commande qui compte le nombre de lignes dans un fichier texte :

```
wc -l < nom_fic
```

Testez cette commande **wc** sur le fichier **/etc/passwd** qui contient la liste des utilisateurs du système.

Q.10 Maintenant, avec vos connaissances actuelles sur les redirections, et uniquement avec elles[8], ainsi qu'avec la commande **wc** qu'on vient juste de voir, proposez une façon de compter le nombre de valeurs de la séquence affichée par : **seq 1 17 1000**

Astuce : ça se fait sans doute en plus d'une étape.

Faites-vous confirmer votre solution par l'enseignant.e.

Avec tube

La solution que vous avez trouvée est certainement de passer par un fichier intermédiaire, temporaire, portant le nom que vous voulez, disons **seq.txt** :

- Création d'un fichier **seq.txt** avec la liste des valeurs de l'intervalle **[1, 1000]** par saut de **17** en utilisant une redirection de **STDOUT** à l'aide de **>**
- Puis comptage des lignes écrites dans **seq.txt** à l'étape précédente, à l'aide de la commande **wc** et d'une redirection de **STDIN** depuis ce fichier **seq.txt** à l'aide de **<**

Volontairement, le détail des deux commandes n'est pas donné ici pour vous laisser un peu de travail quand même ! Donc, si vous n'avez pas réussi la Q.9 de cette façon, demandez l'aide de l'enseignant.e afin de continuer.

Ce qui nous amène maintenant à la solution *avec tube*, en passant par l'analogie suivante :

- La commande **seq** *produit* des nombres, comme un robinet *produit* de l'eau.
- Vous avez choisi d'envoyer ces nombres *dans un fichier* **seq.txt**, comme on remplirait *un seau* avec l'eau du robinet.
- Enfin, vous avez alimenté la commande **wc** à partir du fichier **seq.txt**, comme on *alimenterait* une cuve avec le contenu du seau.

N'y a-t-il pas plus simple que de passer par un seau ? Relier un robinet à la cuve ne peut-il pas se faire plutôt au moyen d'un tuyau ? Tuyau ? Tube ? On y est enfin !

[8] On est dans le chapitre sur les tubes, mais pour le moment on n'a pas encore vu ce que sont réellement les tubes. Vous ne devez donc pas en faire usage si vous savez déjà ce que c'est, ni chercher sur Internet comment les utiliser. On y vient un peu plus loin.

Voici comment on relie deux commandes sans passer par un fichier intermédiaire. On relie une commande qui *produit quelque chose* (ici ce sont des *nombres* avec **seq**) à une commande qui *consomme des choses* (ici ce sont des *lignes de texte* avec **wc**)[9].

Ce qui nous donne :

```
seq 1 17 1000 | wc -l
```

Le tube est représenté par un **|** qu'on appelle le caractère *pipe* (**AltGr 6** sur un clavier de PC et **Option-Shift-L** sur Mac). On l'appelle aussi *barre verticale* mais le nom de *pipe* (*tuyau* en anglais) est plutôt significatif dans le cas présent. On en fait aussi un verbe et on peut dire : *je lance un **seq** pipé sur un **wc***

Q.11 Essayez maintenant de calculer la moyenne des **100** premiers nombres à l'aide de **seq** et de votre programme **moy** créé précédemment.

Quel problème avez-vous ? Pourquoi ? Comment pourriez-vous le résoudre ?

---

less

Nous savons[10] que **less** permet d'afficher un fichier en mode paginé, si le contenu du fichier est plus long qu'une hauteur de fenêtre ou d'écran (si pas d'interface graphique).

Nous avons vu une syntaxe comme celle-ci :

```
less nom_fichier
```

Mais, comme beaucoup de commandes qui traitent des fichiers texte en lecture (cat, more, et beaucoup d'autres qu'on verra plus tard), il existe une autre syntaxe qui est la suivante :

```
less < nom_fichier
```

Avec cette syntaxe, la commande ne lit pas un fichier mais lit **STDIN**, qui se trouve être alimenté par le contenu d'un fichier au moyen d'une redirection entrante.

Le résultat est en tout point identique du point de vue de la commande.

[9] Autrement dit, on relit une commande qui fait des **printf(...)** à une commande qui fait des **scanf(...)**. Du producteur au consommateur !
[10] TP 1 - Premiers pas.

Cette particularité de **less** de pouvoir s'alimenter depuis **STDIN** prend un grand intérêt maintenant qu'on connaît les tubes car elle va nous permettre d'enchaîner des commandes de cette façon :

```
une_commande_qui_produit_beaucoup_de_lignes | less
```

et avoir ainsi l'affichage paginé que **une_commande_qui_produit_beaucoup_de_lignes** aurait affiché au kilomètre[11] sinon !

Q.12 Un exemple de commande qui produit beaucoup de lignes est un **ls** avec l'option récursive. Elle va afficher tous les dossiers et pour chacun tous les sous-dossiers et ainsi de suite, en profondeur. Ça peut représenter énormément de lignes. Voici ce que vous devez faire, dans l'ordre :

- Trouver, dans le manuel, l'option qui fait un **ls** récursif
- Exécuter un **ls** récursif de votre *Home*, à l'aide de cette option. Vous pouvez aussi ajouter une option **-l** pour afficher, en même temps, plus de détails avec **ls**.
- Adapter votre commande pour un affichage paginé, tel qu'on vient de l'apprendre.

Quelques exemples utiles

Voici quelques exemples classiques que vous pouvez reproduire et décliner de votre côté en cherchant à bien en comprendre leur fonctionnement et leur utilité :

- Nombre de lignes de code dans un TP d'Algo/Prog :

```
cd dossier_TP
cat *.c | wc -l
```

- Nombre d'images PNG dans un dossier :

```
cd un_dossier_images
ls *.png | wc -l
```

- Affichage paginé d'un long fichier texte + numérotation des lignes

```
cat -n long_fichier.txt | less
```

Le TP suivant, sur les filtres, va faire la part belle au mécanisme des tubes. Il faut bien comprendre ce mécanisme important d'Unix.

[11] Au kilomètre signifie : en bloc sans s'interrompre sur des dizaines, des centaines voire des milliers de lignes dans le Terminal.