



TELECOM NANCY

---

## **Projet de réseaux et Systèmes "Find The Cat"**

---

*Auteurs :*

Malo DAMIEN

Jules BRUNET

*Responsable de module :*

Maiwenn RACOUCHOT



18 décembre 2022

# Table des matières

<b>1</b>	<b>Présentation &amp; Conception</b>	<b>1</b>
1.1	Introduction au sujet . . . . .	1
1.2	Conception . . . . .	1
1.3	Répartition du temps de travail . . . . .	1
<b>2</b>	<b>Implémentation</b>	<b>2</b>
2.1	Structure globale du programme . . . . .	2
2.2	Structure du parser . . . . .	2
2.3	Structure path_list . . . . .	2
2.4	Implémentation de execute.c . . . . .	3
2.5	Différentes options de la commande ftc . . . . .	3
2.5.1	-name . . . . .	3
2.5.2	-size . . . . .	3
2.5.3	-date . . . . .	3
2.5.4	-mime . . . . .	3
2.5.5	-dir . . . . .	3
2.5.6	-ctc . . . . .	4
<b>3</b>	<b>Extensions</b>	<b>5</b>
3.1	Options . . . . .	5
3.1.1	-color . . . . .	5
3.1.2	-date avec mots clés . . . . .	5
3.1.3	-perm . . . . .	5
3.2	Difficulté . . . . .	5

# Chapitre 1

## Présentation & Conception

### 1.1 Introduction au sujet

Durant ce semestre, nous avons eu à réaliser un programme en C "ftc" permettant à un utilisateur de trouver un chemin dans une arborescence. Ce programme reprend les fonctionnalités principales de la commande "find" sur Linux, et certaines de ses expansions comme -size, -name,... D'éventuelles extensions peuvent être ajoutées, comme l'écriture du texte en couleur, ou d'autres expansions comme -perm par exemple.

### 1.2 Conception

Avant de nous lancer dans la programmation, il a fallu passer par une phase de conception. Elle s'est faite via deux réunions, séparées par des discussions avec d'autres groupes sur la manière d'approcher le problème. Durant cette phase, nous avons décidé :

- De la manière dont la commande serait parsée (détaillée plus bas) : il fallait qu'elle permette de détecter les différentes options et leur valeur, et donc faire la différence entre nouveau flag et valeur du flag précédent ;
- De la manière dont le programme traiterait les différents flags, et comment il discriminerait les fichiers invalides (execute et path\_list expliqués plus bas) ;

Notons que nous n'avons pas pensé individuellement à la manière de réaliser chacune des options durant cette étape, car cela nous semblait plus simple à implémenter directement maintenant que la structure générale du programme était établie.

### 1.3 Répartition du temps de travail

Notons que nous avons rassemblé la partie implémentation et la partie tests, car ces derniers ont été réalisés en parallèle de la période de développement.

Membre	Conception	Implémentation et tests	Rapport
Malo DAMIEN	6 h	25 h	2 h
Jules BRUNET	6 h	19 h	3 h

FIGURE 1.1 – Tableau récapitulatif du temps de travail de chacun

# Chapitre 2

## Implémentation

### 2.1 Structure globale du programme

Afin d'avoir le code le plus lisible et fonctionnel possible, nous avons décidé de diviser notre programme en différents sous programmes. Nous avons un fichier main qui permet de lancer le programme, puis un fichier parser et execute qui permettent respectivement de stocker sous forme de structure adéquate la ligne de commande et d'exécuter cette ligne de commande. Enfin, nous avons un fichier par commande afin de s'y retrouver rapidement et de garder le code clair et le plus concis possible.

### 2.2 Structure du parser

Afin de réaliser la lecture de la ligne de commande pour lire les flags et effectuer les bonnes opérations, nous avons implémenté une structure de parser que nous allons détailler ici.

```
26
27 typedef struct _token_item {
28     token_t token;
29     char * value;
30     int position;
31 } token_item;
32
33 typedef struct _token_list {
34     token_item *data;
35     int ptr;
36     int size;
37 } token_list;
```

FIGURE 2.1 – Structure des tokens

Un token dans notre code est considéré comme un flag accompagné de sa valeur, ce qui correspond à une option de la ligne de commande.

Un "Token Item" stocke le type de flag (via le type token\_t qui est un enum de tous les types de flag possibles -name, -size ...), sa valeur sous forme d'une chaîne de caractères et également sa position dans la ligne de commande (ce qui va nous permettre éventuellement de repérer une erreur ou encore de placer correctement les tokens dans notre structure de token\_list).

La token\_list, quant à elle, va stocker tous nos tokens sous forme d'une liste dynamique (à taille variable) avec un pointeur de liste et une variable indiquant sa taille actuelle. Toutes les fonctions de gestion de cette liste sont regroupées dans le fichier "token.c".

Avec ces deux structures implémentées, la principale tâche de notre structure de parsing est de construire la token\_list avec des token\_item en lisant l'argv. Pour cela, nous avons utilisé un switch case en fonction du type de token considéré.

### 2.3 Structure path\_list

Nous avons opter pour une structure de liste de chaîne de caractères pour sauvegarder les chemins correspondant au fichiers "trouvés" par la commande find the cat. Au début, nous prenons l'ensemble des chemins accessibles depuis le starting point indiqué en deuxième argument. Puis, nous supprimons au fur et à mesure les chemins ne remplissant pas les conditions de la des options passées en paramètres. C'est ici qu'utiliser une liste à taille variable est utile.

A la fin de notre exécution, il suffit simplement de print dans la stdout cette liste de chemins pour afficher le(s) résultat(s) de la commande.

## 2.4 Implémentation de `execute.c`

Ainsi, notre programme s'organise en deux grandes étapes : le parsing de la ligne de commande et l'exécution des différentes options. C'est pourquoi, dans `execute.c`, nous avons deux fonctions `exec_parser()` et `exec_find()` qui gèrent chacune ces étapes :

- `exec_parser()` : Initialise le parser et la liste des paths. Teste pour des erreurs de syntaxe dans la commande. S'occupe d'orienter le type de recherche (directory ou file), test mode on ou off,... ;
- `exec_find()` : Met à jour la liste des paths en itérant sur la liste de Tokens. Teste pour des erreurs de sémantique dans la commande.

## 2.5 Différentes options de la commande `ftc`

### 2.5.1 `-name`

La fonction `name` permet de trouver un fichier via son nom. Dans un premier temps, le sujet nous demandait de chercher avec le nom exact, puis de chercher en utilisant une expression régulière. Cependant, cela pose certains problèmes car la combinaison de ces deux fonctionnalités ne permet plus de faire des recherches "exactes". En effet, l'utilisation d'une expression régulière nous autorise à matcher seulement une partie du nom du fichier pour le trouver. Cela constitue en soit une ambiguïté du sujet qui nous a posé quelques problèmes lors de la conception. Nous avons préféré laisser l'utilisation classique de l'expression régulière car elle passait tous les tests d'intégration imposés tout en étant la plus efficace.

### 2.5.2 `-size`

La fonction `-size` permet de retrouver les fichiers dont la taille est inférieure (ou supérieure) à la valeur passée en paramètre. Pour ce faire, on commence par parser la valeur passée pour voir si elle est conforme à la syntaxe de la commande. Puis on regarde, pour chaque fichier (ou directory), si sa taille est conforme ou non. On utilise la structure `stat` pour récupérer les données liées à chaque fichier.

Notons que comme cette taille est ultimement comparée en bytes, il y a des problèmes d'int overflow sur des valeurs de l'ordre du gigabytes. Ce problème persiste sur les long int, et nous considérons que l'on n'utilisera pas `-size` avec une taille supérieure à 1G. De plus, ne mettre qu'une lettre (par exemple `-size +G`) est équivalent à `+0G`.

### 2.5.3 `-date`

La fonction `-date` permet de retrouver les fichiers dont la date de dernière modification est antérieure (ou postérieure) à la valeur passée en paramètre. Son fonctionnement est similaire à celui de `size` : on parse la valeur donnée et on trie les paths de la liste à l'aide des fonctions `time()` et `difftime()`, toujours en utilisant la struct `stat`.

Notons que l'on compare ces dates avec des temps d'une précision de l'ordre de la seconde.

### 2.5.4 `-mime`

La fonction `mime` permet de retrouver un fichier en fonction de son type ou de son sous-type. Pour retrouver le type et sous type d'un fichier en fonction de son chemin, nous avons utilisé son extension. En effet, connaissant l'extension d'un fichier, nous pouvons retrouver son type mime. Cette méthode peut cependant poser problème car il suffit qu'une personne modifie l'extension du fichier en le renommant tout simplement pour qu'il ait un type mime différent. Une autre technique aurait été d'utiliser la signature électronique d'un fichier pour retrouver son type mime. On peut penser à l'utilisation de magic numbers par exemple.

Pour effectuer la liaison extension-type mime, nous avons copié un dictionnaire regroupant les extensions les plus communes et leur extensions associé. Notre dictionnaire reconnaît 738 extensions différentes.

### 2.5.5 `-dir`

`-dir` permet de travailler non plus sur des fichiers, mais sur des dossiers. En effet, jusqu'à présent, on ne traitait que les paths des fichiers. On testait si ces paths correspondaient à des fichiers ou non dans chacune des différentes fonctions, ce qui en terme de conception n'était pas très bien pensé. Il a donc fallu séparer le teste du type de path du reste de ces fonctions. Pour ce faire, lors du parsing de la commande et de la création initiale de la liste des paths, on détecte la présence du flag `-dir` ou non, et on effectue un premier trie préalable sur le type de path à inclure. Ensuite, on ré utilise la fonction `name` déjà implémentée pour faire la discrimination sur la valeur passée en paramètre.

Notons qu'à la base, nous pensions que `-dir` était un flag sans valeur et qu'il indiquait seulement qu'on travaillait sur des dossiers (un peu comme son équivalent avec `find -type "d"`). Ce n'est que plus tard qu'on s'est rendu compte de notre erreur pour le faire fonctionner comme demandé dans le sujet, c'est à dire un mix entre `-name` et `-type d`.

### 2.5.6 -ctc

Cette fonction est similaire à la fonction 'grep' de linux. Elle permet de trouver tous les fichiers qui contiennent une chaîne de caractères matchée par l'expression régulière passée en valeur du flag -ctc. Pour cela, on ouvre les fichiers présents dans notre path list, on stocke leur contenu dans un buffer et on vérifie si l'expression régulière matche une chaîne de caractères dans le fichier grâce à notre fonction regex\_check().

```
FILE * file = fopen(path, "r");
char * buffer = malloc(sizeof(char) * (st.st_size+1));
fread(buffer, sizeof(char), st.st_size, file);
fclose(file);
buffer[st.st_size] = '\0';

if (regex_check(buffer, value) == 0) {
    //printf("Found file %s.\n", path);
}
else{
    delete_path(pl, path);
    incr = 0;
}
free(buffer);
```

FIGURE 2.2 – Ouverture d'un fichier et stockage dans un buffer

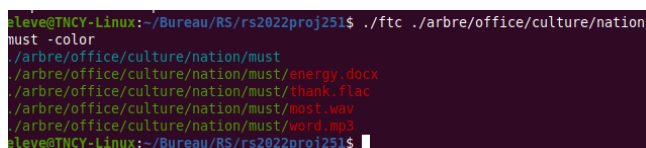
## Chapitre 3

# Extensions

### 3.1 Options

#### 3.1.1 -color

Comme indiqué dans le sujet, `-color` permet d'afficher séparément les noms des fichiers et leur path, et de distinguer les fichiers des dossiers. Simplement, on s'est contenté de détecter au parsing la présence du flag `-color`, et dans ce cas d'utiliser une fonction de print différente qui gère les couleurs. Cette fonction s'appuie sur les escapes code Ansi pour afficher les couleurs dans le terminal.



```
steve@TNCY-Linux:~/Bureau/RS/rs2022proj251$ ./ftc ./arbre/office/culture/nation/
must -color
./arbre/office/culture/nation/must
./arbre/office/culture/nation/must/energy.docx
./arbre/office/culture/nation/must/thank.flac
./arbre/office/culture/nation/must/most.wav
./arbre/office/culture/nation/must/world.mp3
steve@TNCY-Linux:~/Bureau/RS/rs2022proj251$
```

FIGURE 3.1 – Exemple d'utilisation de `-color`

#### 3.1.2 -date avec mots clés

Pour améliorer `-date`, on a décidé d'accepter les mots-clés "today", "yesterday", "this month" et "now" pour faciliter la recherche.

On effectue un test préalable dans `date` pour voir si la valeur passée en flag est l'un de ces mots (d'ailleurs on a dû autoriser `-date` à prendre jusqu'à 2 flags, pour le mot clé "this month"), et dans ce cas on utilise la structure `tm` (qui fournit des données plus précises vis-à-vis des dates) pour pouvoir discriminer les paths valides ou non. Notons que la commande `./ftc . -date +today` est possible pour sélectionner tous les fichiers modifiés avant aujourd'hui, par exemple : on gère l'utilisation du "+" et "-" avec ces mots clés.

#### 3.1.3 -perm

Afin de réaliser la recherche de fichier par permission, nous avons utilisé la structure `stat` qui nous donne des informations sur les permissions d'un fichier. En utilisant le fait que les permissions en linux sont représentées par un nombre octal de 3 chiffres, on peut recréer la permission du fichier et ainsi la comparer à celle qui est passée en valeur du flag `perm`. Par exemple, on part avec une permission de 0, on trouve que l'utilisateur à la droite d'écriture, de lecture et d'exécution ; on va faire  $perm = perm + 4 * 100 + 2 * 100 + 1 * 100$ . Si ensuite on trouve que le groupe a la permission de lecture et d'exécution et que les autres (others) ont seulement le droit d'exécution on va faire  $perm = perm + 4 * 10 + 1$ .

### 3.2 Difficulté

Nous n'avons pas pu implémenter la fonction OU car elle nécessitait trop de modifications dans notre structure en un temps limité. En effet, pour construire notre liste, on considère tous les paths possibles et on la réduit au fur et à mesure des flags testés. Or la gestion du OU aurait nécessité l'inverse, et il aurait fallu repenser toute la structure du programme pour y parvenir : construire une liste annexe et y ajouter au fur et à mesure les fichiers valides, or cela implique de totalement revoir le fonctionnement de `execute_find()`, sans parler des fonctions des flags en elles-mêmes qui gèrent, pour le moment et de manière autonome, la suppression des fichiers non valides. En clair, par manque de temps et de motivation, nous n'avons pas pu réaliser cette option.