

Parfait, tu as structuré ton projet pipex proprement et il respecte les demandes du sujet 42. Voici une **explication complète** de ton code, son fonctionnement global, les **fonctions impliquées**, leur **ordre d'appel**, et comment tout cela répond aux exigences du sujet.

Objectif du projet

Simuler le comportement suivant du shell :

```
< file1 cmd1 | cmd2 > file2
```

Ton exécutable s'utilise ainsi :

```
./pipex infile "cmd1 arg1 arg2 ..." "cmd2 arg1 ..." outfile
```

Il doit :

1. Lire depuis infile
2. Lancer cmd1 et lui passer infile en entrée
3. Passer la sortie de cmd1 en entrée à cmd2
4. Rediriger la sortie de cmd2 dans outfile

Architecture générale

Fichiers principaux :

- pipex.c : fonction principale main() et fonction pipex()
- child_process.c : création et gestion des deux processus child1 et child2
- exec_cmd.c : exécution d'une commande via execve()
- get_path.c : résolution du PATH d'une commande
- file_utils.c : gestion des fichiers (ouverture, erreurs)
- pipe_utils.c : création de pipe, fork, et gestion d'erreur associée
- utils.c : fonctions utilitaires (ft_strdup, ft_strjoin, etc.)
- ft_split.c : équivalent custom de split()
- cleanup.c : libération mémoire, fermeture de file descriptors

Ordre d'exécution / appel

main() → pipex()

pipex.c

```
int main(int argc, char **argv, char **envp)
{
    if (argc != 5)
        exit_error("Usage: ./pipex file1 cmd1 cmd2 file2");
    return pipex(argv, envp);
}
```

pipex() – cœur du programme

```
int pipex(char **av, char **envp)
{
    int pipe_fd[2];

    create_pipe(pipe_fd);      // pipe()
    pid_t pid1 = create_process(); // fork()
    if (pid1 == 0)
        child1(pipe_fd, av, envp); // processus enfant 1

    pid_t pid2 = create_process(); // fork()
    if (pid2 == 0)
        child2(pipe_fd, av, envp); // processus enfant 2

    close(pipe_fd[0]);
    close(pipe_fd[1]);
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
    return 0;
}
```

child1() – Premier processus

Fichier : child_process.c

1. Ouvre infile
2. Redirige stdin avec dup2(fd_in, STDIN_FILENO)
3. Redirige stdout vers le pipe : dup2(pipe_fd[1], STDOUT_FILENO)

4. Appelle `execute_cmd(av[2], envp)`

`child2()` – Deuxième processus

1. Ouvre outfile
2. Redirige stdin depuis le pipe : `dup2(pipe_fd[0], STDIN_FILENO)`
3. Redirige stdout vers fd_out : `dup2(fd_out, STDOUT_FILENO)`
4. Appelle `execute_cmd(av[3], envp)`

`execute_cmd()`

1. Appelle `ft_split(cmd, ' ')` pour séparer les arguments
2. Appelle `get_path()` pour résoudre le chemin de la commande
3. Utilise `execve(path, args, envp)` pour lancer la commande

Si `execve` échoue → `exit_error("command not found")`

`get_path()`

- Récupère la variable `PATH` depuis `envp`
- Coupe les chemins avec `ft_split()`
- Teste chaque chemin + cmd avec `access(path, X_OK)`
- Retourne le premier chemin exécutable valide

`file_utils.c`

- `open_input_file()` → ouvre file1, gère les erreurs
- `open_output_file()` → ouvre file2 avec `O_CREAT | O_TRUNC`
- `handle_dup_error()` → en cas d'échec de `dup2()`

`pipe_utils.c`

- `create_pipe(int *pipe_fd)` → appelle `pipe()`, gère l'échec
- `create_process()` → appelle `fork()`, gère l'échec

cleanup.c






- `ft_free_array()` → libère un tableau de strings (ex. résultat de `ft_split`)

ft_split.c, utils.c

Ces fichiers contiennent des fonctions libft utilisées localement pour éviter d'importer une lib externe :

- `ft_strlen`, `ft_strdup`, `ft_strjoin`, `ft_strchr`, `ft_strncmp`, `ft_memcpy`

☒ Ce que ton code respecte parfaitement du sujet

Exigence	État
<code>./pipex file1 cmd1 cmd2 file2</code>	
Utilisation de pipe, fork, dup2, execve, waitpid	
Gestion des erreurs (fichiers, execve, dup2, etc.)	
Pas de leaks mémoire (à tester avec valgrind)	 à confirmer
Code organisé, lisible, modulaire	
Makefile avec règles obligatoires	