Guide d’implémentation du builtin cd dans un minishell

Introduction

Dans ce guide, nous allons implémenter pas à pas la commande interne (builtin) cd dans un minishell en C. Cette commande permet de **changer de répertoire courant** dans le système de fichiers, un peu comme on se déplace d’un dossier à un autre sur son ordinateur. L’objectif est de comprendre clairement la logique derrière les chemins de fichiers (chemins absolus, relatifs, etc.) et le comportement attendu de cd dans différents cas (cd seul, cd .., cd -, cd chemin_relatif, cd /chemin/absolu, etc.), ainsi que la gestion des erreurs possibles.

Ce guide s’adresse à une débutante qui connaît les bases du C et du Shell, mais qui souhaite un accompagnement pédagogique pour construire progressivement la fonctionnalité cd. Nous aborderons d’abord quelques notions essentielles sur les chemins, puis les fonctions C utiles, et enfin l’implémentation par étapes avec des exemples de code et des tests à réaliser à chaque étape.

1. Notions de base : chemins absolus, relatifs et spéciaux

Avant d’écrire du code, il est important de bien comprendre comment fonctionnent les **chemins de fichiers** sous Unix/Linux :

- **Chemin absolu** : un chemin commençant par / (le répertoire racine). C’est l’adresse complète d’un fichier ou dossier dans l’arborescence. *Analogie* : c’est comme donner l’adresse complète d’une maison (pays, ville, rue, numéro). Par exemple, /home/alice/documents est un chemin absolu qui mène au dossier documents, en partant de la racine puis en traversant home puis alice.
- **Chemin relatif** : un chemin par rapport au répertoire courant, *sans* commencer par /. Il dépend de l’endroit où l’on se trouve actuellement. *Analogie* : c’est comme dire « à partir d’ici, avance de deux pas puis tourne à gauche ». Par exemple, si le répertoire courant est /home/alice, le chemin relatif documents fera référence à /home/alice/documents. De même, ../bob depuis /home/alice indiquerait le chemin /home/bob (on est monté d’un niveau puis entré dans bob).
- **. et ..** : ce sont des répertoires spéciaux. . représente le répertoire courant lui-même, et .. représente le répertoire parent (celui juste au-dessus dans l’arborescence). Par exemple, depuis n’importe quel dossier, cd . vous laissera au même endroit, et cd .. vous fera remonter d’un cran vers le dossier parent. Si vous êtes dans /home/alice/documents, faire cd .. vous ramènera dans /home/alice. On peut enchaîner plusieurs .. pour remonter plusieurs niveaux (par ex. ../../ pour monter de deux niveaux).

- **Le répertoire HOME (~)** : En shell, le symbole ~ est utilisé comme raccourci pour votre répertoire personnel (défini par la variable d'environnement HOME). Ainsi, cd ~ vous amène dans votre dossier personnel[juliend.github.io](). Dans un programme C, il faudra lire la variable d'environnement HOME pour obtenir ce chemin (nous y reviendrons).
- **Le répertoire précédent (-)** : La commande cd - permet de retourner au **dernier répertoire courant** dans lequel vous vous trouviez[juliend.github.io](). Le shell mémorise l'ancien répertoire dans une variable d'environnement spéciale (OLDPWD). Nous verrons comment gérer cela lors de l'implémentation.

En résumé, l'**arborescence du système de fichiers** s'organise comme un arbre avec une racine /. Un **chemin absolu** donne l'emplacement exact d'une ressource à partir de la racine, tandis qu'un **chemin relatif** indique un emplacement en partant du répertoire courant. Les notations spéciales . et .. permettent respectivement de rester au même endroit ou de monter d'un niveau dans l'arborescence.

Quelques exemples pour illustrer ces notions (commande cd en usage) [juliend.github.io]() :

- cd / – vous place à la racine du système de fichiers (le dossier /) [juliend.github.io]().
- cd home/alice – si vous êtes à la racine, ce chemin relatif vous amène dans /home/alice.
- cd ../bob – remonte d'un dossier puis va dans bob. Par exemple, depuis /home/alice, cette commande vous positionnera dans /home/bob.
- cd ~ – vous amène dans votre répertoire personnel (défini par la variable HOME)[juliend.github.io]().
- cd - – vous ramène au répertoire où vous étiez précédemment[juliend.github.io]().

2. Fonctions C utiles pour la commande cd

Pour implémenter cd en C, nous allons utiliser des fonctions systèmes (appels système ou fonctions de la libc) autorisées dans le cadre du projet minishell. Les principales fonctions utiles sont :

- **chdir(const char *path)** – C'est l'appel système qui permet de changer le répertoire courant du processus. Il prend en argument le chemin vers le nouveau répertoire (absolu ou relatif) et retourne 0 en cas de succès, ou -1 en cas d'erreur (en plaçant le code d'erreur dans errno). C'est la fonction *centrale* pour implémenter cd[stackoverflow.com](stackoverflow.com).
- **getcwd(char *buf, size_t size)** – Cette fonction remplit le buffer buf avec le chemin absolu du répertoire de travail actuel (current working directory). Elle est utile par exemple pour obtenir le chemin complet après un chdir (afin de mettre à jour la variable d'environnement PWD, ou pour afficher le chemin courant). Si le buffer passé est NULL et la taille 0, getcwd alloue automatiquement un buffer de la bonne taille (qu'il faudra libérer plus tard avec free). En cas d'erreur, elle renvoie NULL.
- **getenv(const char *name)** – Permet de lire la valeur d'une variable d'environnement. Par exemple, getenv("HOME") retourne le chemin du répertoire personnel de l'utilisateur (ou NULL si la variable n'existe pas). De même, getenv("OLDPWD") donne le dernier répertoire courant connu. On utilisera getenv pour récupérer HOME et OLDPWD lors du traitement de cd sans argument et de cd -.
- **stat(const char *path, struct stat *buf)** et **lstat(const char *path, struct stat *buf)** – Ces fonctions remplissent une structure stat avec des informations sur un fichier ou dossier. En particulier, après un stat, on peut utiliser des macros comme S_ISDIR(buf.st_mode) pour tester si le chemin correspond à un répertoire, ou S_ISREG(buf.st_mode) pour un fichier régulier, etc. Dans le contexte de cd, stat peut servir à vérifier qu'une cible existe et est un dossier avant de tenter le chdir, afin de fournir un message d'erreur approprié (par exemple distinguer « Aucun fichier ou dossier » d'un « N'est pas un dossier »). lstat est similaire à stat mais ne suit pas les liens symboliques (utile si on veut vérifier la nature d'un lien sans suivre sa cible).
- **access(const char *path, int mode)** – Permet de vérifier les droits d'accès à un fichier ou dossier. Par exemple, access(path, F_OK) vérifie que le fichier/dossier existe, et access(path, X_OK) vérifie qu'on a le droit d'exécution sur ce dossier (droit d'y entrer). On pourrait utiliser access pour anticiper certaines erreurs (existence, permission) avant de faire chdir. Cependant, ce n'est pas strictement nécessaire si on gère correctement l'erreur de chdir (car chdir échouera de toute façon si le dossier n'existe pas ou si on n'a pas les droits).
- **Fonctions d'affichage d'erreur** : perror(const char *msg) affiche un message d'erreur correspondant au dernier appel système échoué (basé sur errno), préfixé par la chaîne msg. Par exemple, après un échec de chdir, appeler

perror("minishell: cd") pourrait afficher minishell: cd: No such file or directory (en anglais par défaut). Alternativement, strerror(errno) retourne une chaîne avec le message d'erreur pour le code dans errno. Ces fonctions aident à informer l'utilisateur du problème en cas de commande invalide.

Note : cd est une commande **interne** du shell (builtin), ce qui signifie qu'elle est exécutée au sein du processus du shell lui-même (il n'y a pas de programme /bin/cd). Il faut donc que notre shell mette à jour son **environnement**. En particulier, les variables d'environnement PWD (répertoire courant) et OLDPWD (ancien répertoire) doivent être modifiées manuellement lors d'un cd stackoverflow.com. Après chaque changement de dossier réussi, on attribuera OLDPWD à l'ancienne valeur de PWD, et on mettra PWD à jour avec le nouveau chemin (souvent obtenu via getcwd). Nous verrons dans les étapes comment effectuer ces mises à jour.

Maintenant que ces bases sont posées, passons à l'implémentation de cd étape par étape.

3. Implémentation de cd étape par étape

Nous allons construire notre fonction ft_cd (par exemple) petit à petit. À chaque étape, nous ajouterons une fonctionnalité et nous testerons son comportement avant de passer à la suivante.

Étape 1 : cd sans argument (aller dans HOME)

Objectif : Gérer la commande cd lorsqu'elle est tapée **sans argument**. Dans Bash, cd tout court renvoie l'utilisateur dans son répertoire **HOME** personnel juliend.github.io.

1. **Récupérer le chemin du HOME** – via la variable d'environnement HOME. En C, on peut utiliser getenv("HOME"). Cela nous donne une chaîne de caractères avec le chemin absolu du dossier home de l'utilisateur (par ex. /home/alice).
2. **Vérifier que HOME est défini** – Il est possible que HOME ne soit pas défini (même si c'est rare dans un shell classique). Si getenv("HOME") renvoie NULL, on ne peut pas effectuer cd et on affichera un message d'erreur du style minishell: cd: HOME non défini. Dans ce cas, on ne change pas de répertoire. (Bash afficherait bash: cd: HOME not set et resterait dans le dossier courant.)
3. **Changer de répertoire vers HOME** – si on a obtenu un chemin pour HOME, on appelle chdir(home_path). Si chdir retourne 0, le changement est réussi. Sinon, il y a une erreur (par ex, si le dossier n'existe pas ou n'est pas accessible, ce qui serait inhabituel pour HOME). En cas d'erreur, on peut utiliser perror("minishell: cd") pour afficher le problème (ou un message personnalisé en utilisant errno).
4. **Mettre à jour PWD/OLDPWD** – Lors du tout premier cd, on n'a pas encore de OLDPWD défini. On peut décider de ne pas le mettre à jour dans ce cas, ou alors d'initialiser OLDPWD avec le répertoire courant initial. Pour simplifier, on peut procéder ainsi après un chdir réussi :
   - Stocker l'ancien répertoire courant (par exemple obtenu via getcwd avant de changer) dans une variable temporaire (ce sera la nouvelle valeur de OLDPWD).
   - Une fois dans HOME, récupérer le nouveau répertoire courant (avec getcwd) et mettre à jour la variable d'environnement PWD avec ce chemin.
   - Mettre à jour OLDPWD avec l'ancienne valeur sauvegardée.
     (Si c'est le tout premier cd de la session et que OLDPWD n'existait pas, il faut créer cette variable d'environnement.)

Exemple de code (Étape 1) :

```
char *home = getenv("HOME");

if (home == NULL) {

    fprintf(stderr, "minishell: cd: HOME non défini\n");
```

```
    } else {
        if (chdir(home) != 0) {
            perror("minishell: cd");
        } else {
            // Succès : mettre à jour PWD et OLDPWD
            char old_pwd[PATH_MAX];
            char new_pwd[PATH_MAX];
            // old_pwd était le PWD actuel avant changement
            if (getcwd(new_pwd, sizeof(new_pwd)) != NULL) {
                update_env("OLDPWD", old_pwd);
                update_env("PWD", new_pwd);
            }
        }
    }
```

(Dans ce pseudo-code, update_env représente une fonction utilitaire qui met à jour la valeur d'une variable d'environnement dans notre shell. Par exemple, elle pourrait utiliser setenv ou modifier une structure interne représentant l'environnement. La variable old_pwd doit avoir été remplie avec la valeur de PWD avant le chdir – par exemple via getcwd ou une variable globale.)

Tests pour l'Étape 1 :

- **Cas nominal :** Depuis n'importe quel répertoire courant, tapez cd sans argument. Après l'exécution, utilisez la commande pwd (si implémentée, pour afficher le répertoire courant) ou vérifiez le chemin courant. **Vous devriez vous retrouver dans votre HOME** (par exemple /home/alice).
- **HOME non défini :** Si possible, lancez votre minishell après avoir supprimé la variable d'environnement HOME (ou modifiez TEMPORAIREMENT son contenu pour un test). Essayez cd sans argument. Le shell devrait afficher une erreur indiquant que HOME n'est pas défini et rester dans le répertoire courant.
- **Conservation du répertoire courant :** Si la commande réussit, vérifiez que le répertoire courant du processus a bien changé (en listant le contenu, en faisant pwd, ou en regardant le prompt si vous y affichez le dossier courant). Aucune autre action ne doit se produire (pas d'impression imprévue, pas de crash). Si chdir échoue (ce qui serait surprenant pour HOME), assurez-vous qu'un message d'erreur s'affiche et que le répertoire courant n'a pas changé.

Étape 2 : cd - (retourner au répertoire précédent)

Objectif : Gérer la commande cd - qui permet de revenir à l'**ancien répertoire courant** (celui stocké dans OLDPWD).

1. **Mémoriser l'ancien PWD à chaque cd** – Pour que cd - fonctionne, il faut avoir conservé le chemin du précédent répertoire à chaque fois qu'on change de dossier. Dans l'étape 1, nous avons suggéré d'utiliser une variable (ou directement OLDPWD) pour stocker l'ancienne valeur de PWD avant de la modifier. Assurez-vous que ce mécanisme est en place : avant chaque changement de répertoire réussi, faites OLDPWD = PWD (sauvegarde de l'actuel), puis après le changement, mettez à jour PWD.
2. **Récupérer OLDPWD** – Lorsque l'utilisateur tape cd -, on doit retrouver le chemin stocké dans OLDPWD. Si vous avez mis à jour la variable d'environnement OLDPWD aux étapes précédentes, récupérez sa valeur avec getenv("OLDPWD"). Sinon, utilisez la variable globale où vous l'avez stockée. Si OLDPWD n'existe pas ou est vide (par exemple aucune commande cd n'a été exécutée auparavant), on affiche une erreur :
   minishell: cd: OLDPWD non défini
   et on ne change pas de répertoire.
3. **Se déplacer vers OLDPWD** – Si OLDPWD contient un chemin valide, on appelle chdir(oldpwd_path). S'il y a une erreur (par ex. le dossier n'existe plus), on l'affiche comme d'habitude et on ne change pas de dossier. En cas de succès, une particularité de cd - est que le shell **affiche le chemin** où l'on se rend. Bash fait cela pour que l'utilisateur sache où il atterrit. Vous pouvez donc faire par exemple printf("%s\n", oldpwd_path) après un changement réussi, ce qui affichera le chemin.
4. **Mettre à jour PWD/OLDPWD après cd -** – Supposons que le répertoire courant était A et que OLDPWD valait B (votre ancien dir). Quand vous tapez cd -, vous allez vous retrouver dans B. Il faut alors :
   - Mettre OLDPWD à la valeur précédente de PWD (c'est-à-dire A, où vous étiez juste avant le cd -).
   - Mettre PWD à la valeur de B (le nouveau répertoire courant).
     En pratique, cela revient à échanger les deux. Vous pouvez stocker l'actuel PWD dans une variable temporaire, puis faire chdir(oldpwd) et ensuite OLDPWD = temp et PWD = oldpwd.

Exemple de code (Étape 2) :

```
if (strcmp(args[1], "-") == 0) {

    char *oldpwd = getenv("OLDPWD");

    if (!oldpwd) {

        fprintf(stderr, "minishell: cd: OLDPWD non défini\n");

    } else {
```

```
            char current[PATH_MAX];
            if (getcwd(current, sizeof(current)) == NULL) {
                perror("minishell: cd");
                return;
            }
            if (chdir(oldpwd) != 0) {
                perror("minishell: cd");
            } else {
                printf("%s\n", oldpwd);           // affiche le nouveau répertoire
courant
                update_env("OLDPWD", current);    // l'ancien courant devient
l'OLDPWD
                update_env("PWD", oldpwd);        // PWD prend la valeur de oldpwd
            }
        }
    }
```

(Ici on compare l'argument à "-" pour détecter la commande cd -. On récupère OLDPWD, on utilise getcwd pour connaître le chemin actuel (avant le changement) puis on fait le chdir. En cas de succès, on imprime le nouveau dossier et on met à jour les variables. On a fait simple : utiliser getcwd deux fois. On aurait aussi pu conserver la valeur de PWD dans une variable globale plutôt que rappeler getcwd.)

Tests pour l'Étape 2 :

- **Retour au précédent :** Placez-vous dans un répertoire, par exemple /usr. Tapez cd /tmp pour changer de dossier. Ensuite, tapez cd -. Vous devriez revenir à /usr et le shell devrait afficher /usr suite à la commande (c'est la valeur de OLDPWD où il est allé).
- **Allers-retours successifs :** Enchaînez plusieurs changements et utilisations de cd - : par exemple, depuis votre home (/home/alice), faites cd /etc puis cd /var puis cd - (devrait vous ramener à /etc), puis encore cd - (retour à /var). Vérifiez à chaque fois le chemin courant et la valeur affichée, pour vous assurer que le mécanisme alterne bien entre les deux derniers dossiers visités.

- **OLDPWD non défini :** Testez le cas où vous démarrez le shell et tapez immédiatement cd - alors qu'aucun cd précédent n'a été fait. Le shell doit indiquer que OLDPWD n'est pas défini, sans essayer de changer de dossier.
- **Arguments inattendus :** Assurez-vous que cd - ne prend pas d'argument supplémentaire. Par exemple, si l'utilisateur tape cd - quelquechose, votre implémentation devrait soit ignorer quelquechose, soit afficher une erreur "trop d'arguments" (voir gestion des erreurs à l'étape 5). Le comportement standard serait de considérer tout argument supplémentaire comme une erreur.

Étape 3 : cd chemin_relatif, cd ./chemin et cd .. (chemins relatifs)

Objectif : Gérer les déplacements vers des répertoires spécifiés par un chemin relatif, y compris les notations . (répertoire courant) et .. (répertoire parent).

L'utilisateur fournit ici un argument qui est un chemin *relatif*. Cela signifie que :

- Le chemin **ne commence pas par '/'** (sinon ce serait un chemin absolu, que l'on traitera à l'étape 4).
- Ce chemin peut être simple (ex: dossier), contenir des sous-répertoires (ex: dossier/sousdossier), ou inclure des références ./.. pour naviguer dans l'arborescence.

Comment le gérer : La bonne nouvelle, c'est que nous n'avons pas besoin de "calculer" manuellement le chemin final. L'appel chdir() sait interpréter les chemins relatifs par rapport au répertoire courant du processus. Par exemple, si le répertoire courant est /home/alice et qu'on appelle chdir("documents"), le système va automatiquement nous emmener dans /home/alice/documents. De même, chdir("..") va nous faire monter d'un cran (si possible), et chdir("../bob") depuis /home/alice/documents nous amènera dans /home/alice/bob. La page de manuel de chdir nous rappelle d'ailleurs que *« le répertoire de travail courant est le point de départ pour interpréter les noms de chemin relatifs (ceux ne commençant pas par '/') »*man7.org.

Il est quand même utile de comprendre ce qu'il se passe en coulisses :

- Le système de fichiers possède une notion de **répertoire courant** (*current working directory*) pour chaque processus. chdir met à jour cette notion.
- Quand on passe un chemin relatif à chdir, le noyau l'interprète en relatif au chemin courant et *normalise* le résultat en gérant les . et ... Par exemple, si on est dans /home/alice et qu'on appelle chdir("docs/../pics"), le système va interpréter le chemin comme /home/alice/docs/../pics, qu'il simplifiera en /home/alice/pics (car docs/.. s'annule et revient à /home/alice).
- Si on le souhaitait, on pourrait construire le chemin absolu nous-même en C (en préfixant le chemin relatif par le PWD courant, puis en supprimant les . et en traitant les .. dans la chaîne), mais ce n'est pas nécessaire puisque le système d'exploitation le fait déjà pour nous.

Implémentation :

- Vérifier que l'argument n'est ni vide, ni -, ni un chemin absolu. Ici, on suppose qu'on traite ce cas dans un bloc du genre else if (path[0] != '/') après avoir géré les cas précédents.
- Appeler chdir(path_rel).
- Si chdir échoue, gérer l'erreur (voir étape 5). Par exemple, afficher que le dossier n'existe pas, ou autre message approprié.
- Si chdir réussit, mettre à jour OLDPWD et PWD comme d'habitude (OLDPWD = ancien chemin, PWD = nouveau chemin obtenu via getcwd).

Les cas particuliers à prendre en compte :

- cd . : doit ne rien changer (on reste dans le même répertoire). chdir(".") n'a aucun effet sauf éventuellement mettre à jour quelques méta-données. PWD devrait rester identique.
- cd .. : doit placer l'utilisateur dans le dossier parent. Exemple : de /home/alice/documents vers /home/alice. Si on est déjà à la racine / et qu'on fait cd .., on reste à / (le parent de la racine est elle-même).
- Chemins combinés avec . et .. : ex. cd .././../dossier. Il faut que le résultat final soit cohérent (dans cet exemple, on monte deux fois puis on redescend dans dossier). Le *chdir* du système gérera cela correctement si les dossiers existent.
- Chemin relatif multi-niveaux : ex. cd dir1/dir2 depuis un certain répertoire courant. Ça doit amener dans le sous-dossier dir2.
- Chemin avec des slashs consécutifs (ex. cd dir1//dir2) : les // sont interprétés comme un seul /, donc cela revient au même que cd dir1/dir2. (Le système ignore les chemins vides entre deux /.)

Exemple de code (Étape 3) :

```
char *path = args[1];

if (path != NULL && path[0] != '\0') {

    if (path[0] != '/') {

        // Chemin relatif (y compris "." ou "..")

        char oldpwd[PATH_MAX];

        if (getcwd(oldpwd, sizeof(oldpwd)) == NULL) {

            perror("minishell: cd");

            return;

        }

        if (chdir(path) != 0) {
```

```
            perror("minishell: cd");
        } else {
            char newpwd[PATH_MAX];
            if (getcwd(newpwd, sizeof(newpwd)) != NULL) {
                update_env("OLDPWD", oldpwd);
                update_env("PWD", newpwd);
            }
        }
    }
}
```

(Ici on ne traite que le cas où le chemin n'est pas absolu – ce pourrait être un chemin relatif. On sauvegarde le répertoire courant dans oldpwd puis on tente le chdir. Après succès, on récupère le nouveau chemin dans newpwd et on met à jour les variables. En cas d'échec, on utilise perror pour afficher l'erreur par défaut.)

Tests pour l'Étape 3 :
Imaginons une arborescence de test dans votre home, par exemple /home/alice/test contenant un sous-dossier foo qui lui-même contient bar.

- Depuis /home/alice/test, tapez cd foo. Vous devriez vous retrouver dans /home/alice/test/foo. Vérifiez avec pwd ou en listant les fichiers.
- Depuis /home/alice/test/foo, tapez cd bar (un niveau de plus), puis cd ... Vous devriez revenir dans /home/alice/test/foo après le cd .. (car on est monté de bar à foo).
- Testez cd .. de nouveau pour remonter dans /home/alice/test. Puis cd .. encore depuis /home/alice/test pour remonter dans /home/alice. Chaque .. enlève un niveau du chemin. Si vous êtes déjà dans votre home et que vous faites cd .., vous irez à /home (dossier parent de votre home).
- Testez cd ../../ en partant de ~/test/foo (c'est-à-dire /home/alice/test/foo). ../../ devrait vous faire remonter deux niveaux, c'est-à-dire vous ramener à /home/alice.
- Assurez-vous que cd . ne change rien du tout (vous restez dans le même dossier, PWD ne bouge pas).
- Testez un chemin combiné : par exemple, depuis /home/alice/test, faites cd foo/./bar/../bar. Ici, foo/./bar revient à foo/bar, puis /../bar remonte d'un cran (à foo) et redescend dans bar. Le résultat final devrait être d'entrer dans foo/bar (donc le même dossier au final). Vérifiez que vous y êtes bien.

- Essayez un chemin relatif invalide : par exemple `cd nope` alors que `nope` n'existe pas dans le répertoire courant. Vous devriez obtenir une erreur du type *Aucun fichier ou dossier de ce type* (ou l'équivalent en anglais) et rester dans le répertoire courant initial.

Étape 4 : cd /chemin/absolu (chemins absolus)

Objectif : Gérer le changement de répertoire lorsque l'utilisateur fournit un **chemin absolu** en argument de cd.

La bonne nouvelle ici aussi, c'est que chdir gère déjà les chemins absolus sans effort supplémentaire. Si on lui passe une chaîne commençant par /, il va directement naviguer à partir de la racine vers la destination indiquée.

Implémentation :

- Détecter que l'argument commence par '/'. (Par exemple, dans un else if (path[0] == '/').)
- Appeler chdir(path_absolu).
- Gérer les erreurs éventuelles (de la même façon que précédemment : message et pas de changement de PWD en cas d'échec).
- En cas de succès, mettre à jour OLDPWD et PWD (comme aux étapes précédentes). Ici, on peut directement utiliser le chemin absolu fourni comme nouvelle valeur de PWD, **mais attention** : il est préférable d'utiliser getcwd pour obtenir le chemin canonique. En effet, si le chemin fourni contient des .. ou des liens symboliques, getcwd donnera le chemin réel final. (Par exemple, si on fait cd /usr/../etc, le répertoire courant réel sera /etc.)

En somme, le code sera très similaire à l'étape 3, sauf qu'on n'a pas besoin de manipuler le chemin, juste à utiliser celui donné.

Tests pour l'Étape 4 :

- **Vers la racine :** Depuis n'importe où, faites cd / pour aller à la racine. Vérifiez que vous êtes bien à la racine (pwd doit afficher /).
- **Vers un dossier existant :** Depuis /, essayez d'aller dans un chemin absolu comme /bin ou /tmp ou tout autre répertoire existant. Vérifiez que ça marche (listez le contenu pour vous en assurer, ou pwd).
- **Chemin inexistant :** Essayez un chemin absolu inexistant : cd /dossier/introuvable. Vous devriez obtenir une erreur du style *Aucun fichier ou dossier de ce type* et rester dans le répertoire courant.
- **Chemin avec .. :** Testez un chemin absolu contenant des .. (même si c'est peu courant de saisir cela manuellement) : par exemple cd /usr/../home/alice. Cela devrait vous mener à /home/alice (puisque /usr/.. revient à /). Vérifiez que votre PWD final est bien /home/alice.
- **Lien symbolique :** Si vous avez un lien symbolique connu vers un dossier, testez cd en utilisant son chemin absolu. Par exemple, si /tmp/monlien pointe vers /var/log, cd /tmp/monlien devrait vous amener à /var/log. (Le test des liens symboliques est également mentionné en cas limite plus bas.)

Étape 5 : Gestion des erreurs et usages incorrects

Objectif : Prendre en compte les différents cas d'erreur ou d'utilisation incorrecte de cd afin de rendre notre builtin robuste et informatif.

À ce stade, toutes les fonctionnalités principales de cd sont implémentées. Il est crucial de bien gérer les **erreurs** pour que le shell soit fiable et qu'il informe correctement l'utilisateur en cas de problème. Voici les principaux cas d'erreur ou d'usage invalide à considérer :

- **Le répertoire n'existe pas** : Si l'utilisateur tape cd vers un chemin qui n'existe pas (ex: cd dossier_inconnu), chdir va échouer avec errno = ENOENT (No such file or directory). Le shell doit intercepter cette erreur et afficher un message. Typiquement :
  minishell: cd: dossier_inconnu: Aucun fichier ou dossier de ce type
  (En anglais : *No such file or directory*).
- **Le chemin est un fichier (pas un dossier)** : Si l'utilisateur indique un chemin qui existe mais est un fichier régulier et non un dossier, chdir échouera avec errno = ENOTDIR. Dans ce cas, le message d'erreur devrait indiquer que ce n'est **pas un dossier**. Par exemple :
  minishell: cd: monFichier.txt: n'est pas un dossier
  (Bash affiche : *Not a directory*).
- **Permission refusée** : Si le dossier existe mais que l'on n'a pas le droit d'y entrer (droit d'exécution manquant sur ce dossier ou l'un de ses parents), chdir échouera avec errno = EACCES. Le message type :
  minishell: cd: secret: Permission non accordée
  (Bash : *Permission denied*).
- **Trop d'arguments** : La commande cd ne prend qu'au plus un argument (mis à part les options -L/-P que nous n'implémentons pas ici). Si l'utilisateur tape par exemple cd dir1 dir2, c'est une erreur d'utilisation. Bash afficherait bash: cd: trop d'arguments. Vous devriez détecter ce cas en vérifiant le nombre d'arguments fournis à votre fonction ft_cd. Si argc > 2 (c'est-à-dire plus d'un argument après "cd"), n'effectuez aucun chdir et affichez un message du genre :
  minishell: cd: trop d'arguments.
- **Variables non définies (OLDPWD, HOME)** : Comme déjà vu, si HOME ou OLDPWD sont nécessaires mais absents de l'environnement, il faut le signaler clairement à l'utilisateur (et ne pas tenter de chdir). On a traité ces cas aux étapes 1 et 2 avec des messages HOME non défini ou OLDPWD non défini.
- **Chemin vide** : Si par erreur un argument vide est passé (par ex cd ""), vous pouvez le traiter comme un cd sans argument (certains shells le font, considérant une chaîne vide comme $HOME), ou bien afficher un message d'erreur. Ce cas est rare, mais à documenter si besoin.

Comment implémenter la gestion d'erreur :
Après chaque appel à chdir qui échoue, utilisez le code d'erreur dans errno pour

déterminer le problème. Par exemple :

```
    if (chdir(path) != 0) {
        if (errno == ENOENT) {
            fprintf(stderr, "minishell: cd: %s: Aucun fichier ou dossier de ce
    type\n", path);
        } else if (errno == ENOTDIR) {
            fprintf(stderr, "minishell: cd: %s: n'est pas un dossier\n", path);
        } else if (errno == EACCES) {
            fprintf(stderr, "minishell: cd: %s: Permission non accordée\n", path);
        } else {
            // autre erreur générique
            perror("minishell: cd");
        }
    }
```

(Les codes d'erreur ENOENT, ENOTDIR, EACCES sont définis dans ‹errno.h›. Vous pouvez ajouter d'autres else if pour gérer ENAMETOOLONG etc., mais les trois principaux cas ci-dessus couvrent la majorité des erreurs de cd. La branche finale avec perror couvrira toute erreur imprévue.)

Il est également important de vérifier le nombre d'arguments avant de décider quoi faire :

```
    if (arg_count > 2) {
        fprintf(stderr, "minishell: cd: trop d'arguments\n");
        return;
    }
```

Ainsi, vous évitez d'avoir à gérer des situations ambiguës avec plusieurs arguments.

Et souvenez-vous : en cas d'échec du cd, **ne changez pas** les variables PWD et OLDPWD (puisque le répertoire courant du shell n'a pas changé).

Tests pour l'Étape 5 (erreurs) :

- **Dossier inexistant :** cd /chemin/qui/nexistepas ou cd dossier_inconnu depuis un répertoire courant. Attendez-vous à un message d'erreur *Aucun fichier ou dossier de ce type* et vérifiez que vous êtes toujours dans le répertoire d'origine.
- **Fichier au lieu d'un dossier :** Créez un fichier texte, par ex. touch monfichier. Puis tentez cd monfichier. Le shell doit indiquer que ce n'est pas un dossier (message *n'est pas un dossier* attendu). Assurez-vous de toujours être resté dans le même dossier après cela.
- **Permission refusée :** Sur Linux/Unix, créez un dossier et retirez les droits d'exécution :
- bash
- CopierModifier
- mkdir secret
- chmod -rwx secret # plus aucun droit pour personne
- Puis essayez cd secret. Vous devriez voir *Permission non accordée*. Vérifiez que vous n'êtes pas entré dans le dossier. (Après le test, pensez à re-grant les droits ou à supprimer le dossier de test : chmod +rwx secret && rmdir secret.)
- **Trop d'arguments :** Essayez cd / tmp (avec un espace, donc deux arguments / et tmp) ou cd foo bar. Vous devez voir une erreur du style *trop d'arguments* et rester dans votre répertoire courant initial.
- **HOME/OLDPWD non défini :** Ces cas ont déjà été couverts : testez cd sans HOME ou cd - sans OLDPWD, pour s'assurer que les messages sont clairs et que rien ne plante.

5. Cas limites et tests supplémentaires

Pour s'assurer que notre implémentation de cd est complète et robuste, examinons quelques cas limites et vérifions le comportement du shell dans ces situations particulières :

- **Chemins avec slash multiples :** Le système ignore les slashs consécutifs dans les chemins. Par exemple, la commande cd ////usr//local/// doit vous amener dans /usr/local (les // supplémentaires sont traités comme un seul /). De même, cd foo//bar est équivalent à cd foo/bar. Normalement, vous n'avez rien de spécial à coder pour cela : en passant la chaîne telle quelle à chdir, le noyau s'en occupe. Ce test sert surtout à vérifier que vous ne faites pas une mauvaise gestion de chaîne de votre côté.
- **Chemin vers un fichier au lieu d'un dossier :** Ce cas produit l'erreur que nous avons gérée. Par exemple, cd /etc/passwd (qui est un fichier) doit afficher une erreur *n'est pas un dossier* et ne pas changer le répertoire courant. Vérifiez qu'aucune confusion n'est possible (ex: votre code pourrait confondre un fichier existant avec "dossier inexistant" s'il teste mal les erreurs – d'où l'intérêt d'utiliser errno ou stat pour différencier ENOTDIR de ENOENT).
- **Dossier sans droits d'accès :** Si un dossier existe mais que le shell n'a pas le droit d'y entrer, cd doit échouer avec *Permission non accordée*. On a testé avec l'exemple du dossier secret. Un autre exemple : essayer cd /root en tant qu'utilisateur normal. Vous obtiendrez probablement *Permission non accordée* car le dossier /root (home de l'admin) n'est pas accessible aux autres utilisateurs.
- **Lien symbolique :** Les liens symboliques (symlinks) sont des "raccourcis" vers d'autres fichiers ou dossiers. Si on cd vers un lien symbolique qui pointe sur un dossier, le shell doit nous amener dans le dossier cible du lien. Par exemple, si mylink est un lien vers /usr/share/doc, alors cd mylink doit vous placer dans /usr/share/doc. Après ce cd, la variable PWD contiendra le chemin réel (par exemple /usr/share/doc, pas mylink). En effet, chdir suit les liens symboliques par défaut.
  Test : créez un lien symbolique vers un dossier existant, par exemple ln -s /usr mylink dans le répertoire courant. Puis faites cd mylink. Vous devriez vous retrouver dans /usr. La commande pwd devrait afficher /usr (et non pas .../mylink).
  Si le lien symbolique pointe vers un fichier ou vers une cible inexistante, le comportement sera : *Not a directory* dans le premier cas, *No such file or directory* dans le second, conformément aux erreurs gérées plus haut.
- **Répertoire courant supprimé :** Un scénario extrême : si le répertoire courant du shell est supprimé ou renommé à l'insu du shell (par exemple via un autre terminal), cela peut poser problème. Si vous essayez ensuite de faire cd .. depuis ce répertoire supprimé, il se peut que chdir échoue car le chemin parent n'est plus référencé correctement. Le comportement exact dépend du système de fichiers, mais en général le dossier courant devient une référence "dangling" (orpheline). Dans un shell réel, si le dossier courant a été

supprimé, pwd peut échouer (puisqu'il n'y a plus de chemin valide) et cd .. peut donner *No such file or directory*.

Solution simple : si vous détectez que getcwd renvoie NULL (par exemple, errno == ENOENT après une suppression du dossier courant), vous pourriez informer l'utilisateur que son répertoire courant n'existe plus. Une approche est alors de le renvoyer vers un dossier sûr comme HOME ou /. Cependant, pour un minishell éducatif, vous n'êtes pas obligé de gérer ce cas manuellement. L'important est que votre programme ne crash pas et reste cohérent.

Test : Allez dans un dossier temporaire, puis via un autre moyen supprimez ce dossier. De retour dans votre minishell, essayez cd ... Observez si une erreur est affichée. Tant que le shell ne se ferme pas inopinément, c'est acceptable. (Bash par exemple vous laissera dans un état où pwd renvoie une erreur jusqu'à ce que vous quittiez le dossier supprimé.)

- **Longueur du chemin :** Assurez-vous de prévoir un buffer suffisamment grand pour stocker les chemins (PATH_MAX est défini dans ‹limits.h›, souvent à 4096 octets sur Linux). Si vous utilisez getcwd, passez-lui un buffer de cette taille. Si un chemin est trop long, getcwd peut renvoyer NULL avec errno = ENAMETOOLONG. Ce cas est rare, mais par sécurité vous pouvez le gérer (par exemple en allouant dynamiquement le buffer si besoin, ou en affichant une erreur si vraiment un chemin dépasse vos limites).

En suivant ces étapes et en testant chaque scénario, vous devriez obtenir une implémentation fiable de la commande cd dans votre minishell. N'oubliez pas de toujours tester dans différentes conditions et de vérifier que les variables d'environnement PWD et OLDPWD sont correctement mises à jour à chaque fois (d'autres fonctionnalités du shell, comme l'affichage du prompt ou la commande pwd, en dépendront).

Bon codage et bon courage ! Ce guide vous a fait parcourir la logique de cd de façon progressive et détaillée. N'hésitez pas à revenir sur les sections précédentes si un comportement n'est pas celui attendu, et à ajouter des printf de debug pour voir les valeurs de PWD/OLDPWD lors de vos tests. Avec une bonne implémentation du builtin cd, votre minishell se comportera de façon beaucoup plus proche d'un shell réel, ce qui est très satisfaisant pendant l'utilisation.
juliend.github.io