Vous **devez** utiliser l'outil `make` pour compiler vos programmes. Les options `-Wall -W -Werror` devront être utilisées.

Le but de ce TP est d'implémenter un sous ensemble des fonctionnalités de la commande `find`. Pour celà, vous aurez besoin d'utiliser les appels systèmes suivants :

- `opendir`
- `closedir`
- `readdir`
- `stat`
- `lstat`

La commande `find` permet, comme son nom l'indique, de faire des recherches de fichiers. La commande effectue une recherche récursive dans un répertoire donné en paramètre et affiche les fichiers qui vérifient une expression donnée.

1 Ligne de commande

La forme générale de la ligne de commande que nous allons implémenter est la suivante :

```
find [-L|-P] [chemin] [expression]
```

- Le `chemin` correspond au répertoire dans lequel la recherche va être effectuée. S'il est absent, la recherche s'effectue à partir du répertoire courant ;
- l'`expression` permet de donner les conditions qu'un fichier doit vérifier pour être affiché ;
- les options `-L` et `-P` contrôlent le comportement de la commande vis à vis des liens symboliques. Si l'option `-P` est utilisée, la commande ne suit **pas** les liens symboliques et considère le lien en tant que tel. C'est donc le lien en lui même qui devra vérifier la condition définie par l'`expression`. À l'inverse, si l'option `-L` est utilisée, la commande suit le lien et c'est le fichier pointé par le lien qui devra vérifier la condition. Si ni `-L` ni `-P` ne sont présent, c'est l'option `-P` qui est utilisée par défaut.

2 Expressions

Les expressions que nous allons implémenter sont les suivantes :

- `-name <nom>` : recherche un fichier dont le nom est `nom`
- `-type <type>` : recherche un fichier dont le type est `type`. Le type est défini par un caractère qui peut être :
  - `f` : fichier régulier,
  - `d` : répertoire,
  - `l` : lien symbolique,
  - `b` : fichier spécial de type bloc,
  - `c` : fichier spécial de type caractère,
  - `p` : tube nommé (FIFO),
  - `s` : socket.
- `-executable` : le fichier possède des droits permettant de l'exécuter (droit `x`) ;
- `-empty` : le fichier est vide
- `-anewer <fichier_reference>` : le fichier recherché doit avoir été modifié plus récemment que le fichier `fichier_reference`

3 Implémentation

3.1 Parcours de répertoire

Q 1. Implémentez la fonction :

```
void traiter_fichier(const char *chemin);
```

Dans un premier temps, cette fonction affichera simplement le chemin qui lui est passé en paramètre. Nous modifierons cette fonction plus tard.

Q 2. Implémentez la fonction :

```
void parcourir_repertoire(const char *chemin);
```

qui parcours récursivement le répertoire dont le chemin est passé en paramètre et traite chacun des fichiers qu'elle rencontre. Cette fonction doit donc :

1. ouvrir le répertoire (`opendir`) ;
2. parcourir les fichiers contenus dans le répertoire (`readdir`) et, pour chaque fichier :
   (a) appeler la fonction `traiter_fichier` en lui passant en paramètre le chemin vers le fichier ;
   (b) si le fichier est un répertoire, appeler récursivement la fonction `parcourir_repertoire` [1]

3.2 Vérification des expressions

Q 3. Implémentez la fonction

int `nom_correspond(const char *chemin, const char *motif);`

qui retourne vrai si le nom du fichier indiqué par `chemin` correspond au motif indiqué par le paramètre `motif`. Attention, il faut tester uniquement le nom du fichier, pas le chemin complet.

Q 4. Implémentez la fonction

int `type_correspond(const struct stat *buf, char type);`

qui retourne vrai si le type du fichier indiqué dans la structure pointée par `buf` correspond au type `type`. La valeur du paramètre `type` est une des valeurs possible pour l'expression `-type`.

Q 5. Implémentez la fonction

int `executable(const struct stat *buf);`

qui retourne vrai si la structure pointée par `buf` décrit un fichier exécutable.

Q 6. Implémentez la fonction

int `vide(const struct stat *buf);`

qui retourne vrai si la structure pointée par `buf` décrit un fichier vide.

Q 7. Implémentez la fonction

int `plus_recent(const struct stat *buf, const char *fichier_reference);`

qui retourne vrai si la structure pointée par `buf` décrit un fichier dont la date de dernière modification est plus récente que celle du fichier `fichier_reference`.

3.3 Traitement des fichiers

Pour décrire une expression nous allons utiliser les déclarations suivantes :

```
enum
{
   NAME,
   TYPE,
   EXEC,
   EMPTY,
   NEWER,
   INVALID,
};

typedef struct
{
   int type;
   union
   {
      const char *motif;
      const char *chemin;
      char        type_fichier;
   } operande;
} expression;
```

1. Attention aux répertoire `.` et `..`, évitez les récursions infinies...

La structure `expression` permet de décrire complètement une expression. Elle utilise une notion nouvelle, l'union.

Une union ressemble à une structure dans la façon de la déclarer, mais, à la différence d'une structure, les champs ne sont pas situés les uns à la suite des autres en mémoire.

Dans une union, la taille de l'union sera la taille du plus grand de ces champs et tous les champs sont stockés au même endroit en mémoire. En d'autres termes, dans l'union que l'on va utiliser pour décrire l'opérande de l'expression, les adresses des champs `motif`, `chemin` et `type_fichier` sont les mêmes.

Le but d'une union n'est donc pas de mémoriser plusieurs informations, comme dans le cas d'une structure. Le but est d'avoir un type qui mémorise une seule information, mais cette information peut être d'un type différent suivant le contexte. On utilise donc qu'un champ à la fois. L'intérêt et tout simplement une économie mémoire et un code clair à l'utilisation.

Ce qui va nous indiquer le champ à utiliser dans l'union `operande` est le champ `type` de la structure `expression`. Si le champ `type` vaut

— `NAME`, seul le champ `motif` sera valable ;
— `TYPE`, seul le champ `type_fichier` sera valable ;
— `EXEC` ou `EMPTY`, aucun champ ne sera utilisé ;
— `NEWER`, seul le champ `chemin` sera valable.

Par exemple, pour décrire l'expression `-name fichier.txt` on utilisera un code C similaire au code suivant :

```
expression exp;

exp.type = NAME;
exp.operande.motif = "fichier.txt";
```

Pour décrire l'expression `-type d` on utilisera le code suivant :

```
expression exp;
exp.type = TYPE;
exp.operande.type_fichier = 'd';
```

Q 8. Modifiez la fonction `traiter_fichier` pour que son prototype soit maintenant le suivant :

```
void traiter_fichier(const char *chemin, const expression *exp, int suivre_lien);
```

La fonction doit donc maintenant n'afficher le chemin du fichier **que** si le fichier vérifie l'expression passée en paramètre. Vous allez bien évidemment devoir utiliser les fonctions implémentées dans la section 3.2. L'utilisation de la fonction `stat` ou `lstat` sera conditionné par le paramètre `suivre_lien`.

Q 9. Modifiez la fonction `parcourir_repertoire` pour que son prototype soit maintenant le suivant :

```
void parcourir_repertoire(const char *chemin, const expression *exp, int suivre_lien);
```

qui va donc maintenant appeler la fonction `traiter_fichier` avec les bons paramètres.

4 Interprétation de la ligne de commande

Vous pouvez maintenant terminer l'implémentation de votre commande en analysant la ligne de commande pour appeler correctement la fonction `parcourir_repertoire`.