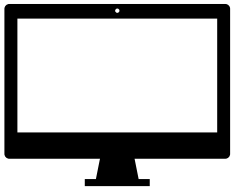
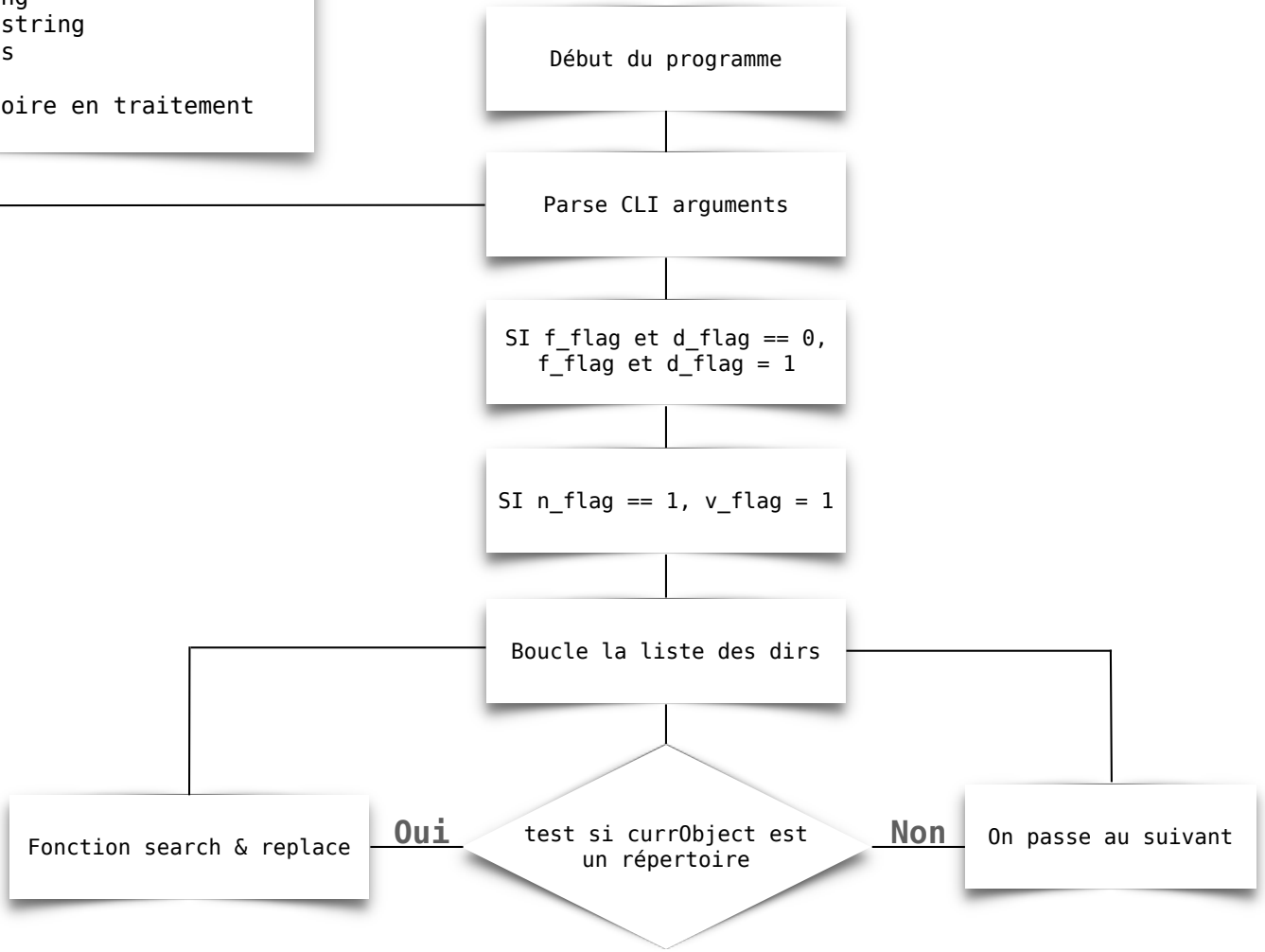
```
Variables:
     flags:
          f_flag: -f
          d_flag: -d
          r_flag: -r, --recursive
          i_flag: -i, --include
          I_flag: -I, --IgnoreCase
          n_flag: -n, --simulate
          v_flag: -v, --verbose
     autres:
          ptrn: pattern string
          repl: replacement string
          paths: list of dirs

          currObject: répertoire en traitement
```

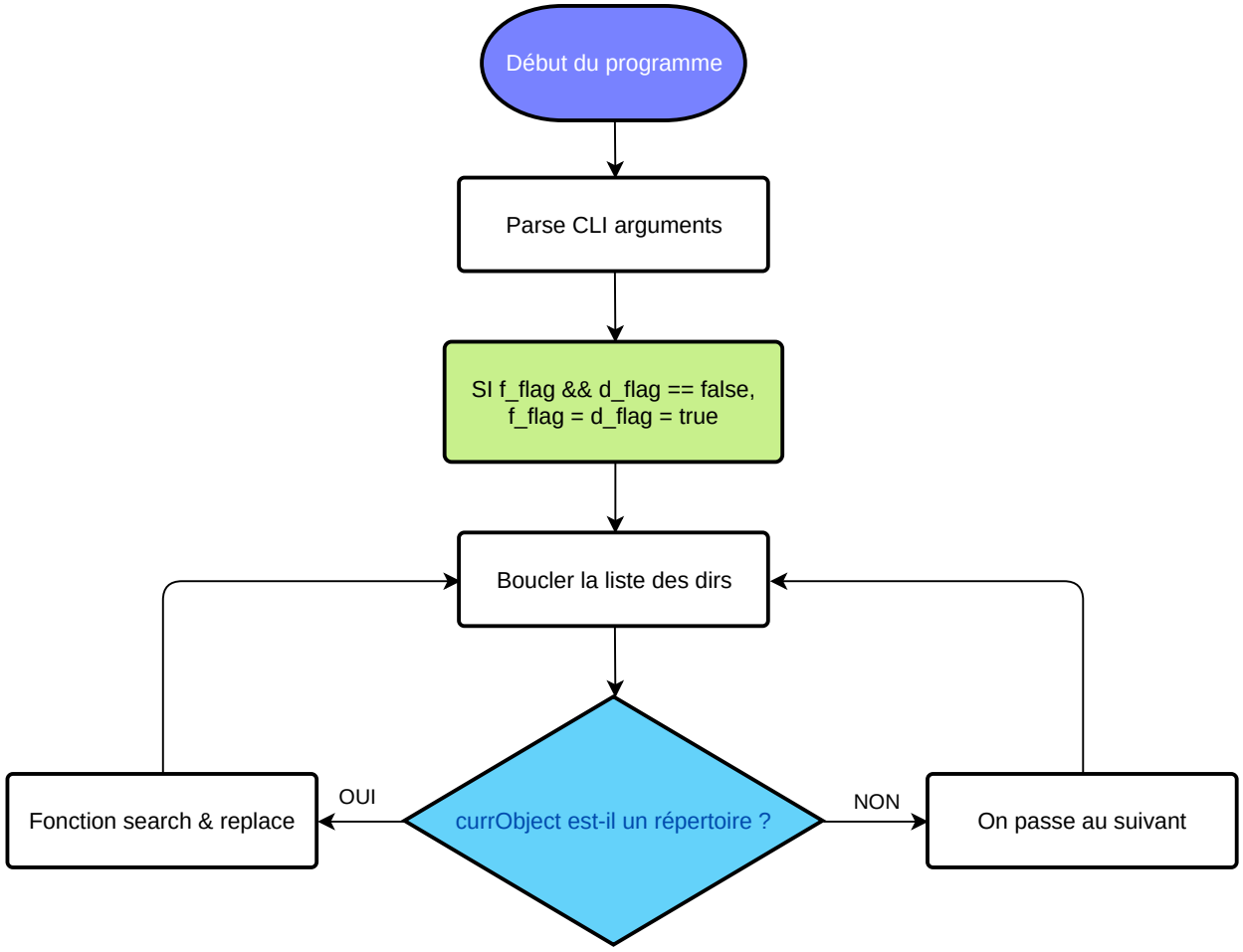




File: **/Users/danv/bin/python/mren.py**

```python
#! /usr/bin/env python3

import os
import re
import argparse
import unicodedata as ud

from push_pop import pushd, popd

def my_usage(name: str):
    return name + ''' [-f|-d] [-riInv] <regex pattern> <remplacement> <dirname ...>'''

parser = argparse.ArgumentParser(description='Renommage de fichier(s) de répertoire(s) selon un certain pattern.',
                                 usage=my_usage('%(prog)s'))
parser.add_argument('-ver', '--version', action='version',
                    version='%(prog)s 2023-07-30 - Renommage multiple à partir d\'un pattern.',
                    help='Renommage multiple à partir d\'un pattern')

group1 = parser.add_mutually_exclusive_group()
group1.add_argument('-f', dest='f_flag', action="store_true",
                    help='N\'agit que sur les fichiers')
group1.add_argument("-d", dest='d_flag', action="store_true",
                    help='N\'agit que sur les répertoires')

parser.add_argument("-r", '--recursif', dest='r_flag', action="store_true",
                    help='Procède de façon récursive sur les répertoires')
parser.add_argument("-i", '--include', dest='i_flag', action="store_true",
                    help='En mode récusif, inclu le dossier en ligne de commande')
parser.add_argument("-I", '--IgnoreCase', dest='I_flag', action="store_true",
                    help='Fait une recherche en ignorant la case')
parser.add_argument("-n", '--simulate', dest='n_flag', action="store_true",
                    help='Simule les opérations demandées - Fichiers affectés en VERT')
parser.add_argument("-v", '--verbose', dest='v_flag', action="store_true",
                    help='Donne des détails sur le(s) fichier(s) traité(s) - Fichiers affectés en ROUGE')

parser.add_argument(dest='ptrn', type=str,
                     metavar="<regex pattern>",
                     help="Pattern à chercher: Mettre entre '...'")
parser.add_argument(dest='repl', type=str,
                     metavar="<remplacement>",
                     help="Chaîne de remplacement")
parser.add_argument(dest='paths', type=str,
                     metavar="<dirname>", nargs='+',
                     help="Répertoires de recherche")
args = parser.parse_args()

# -----------------------------------------------------------
# Ensemble des fonctions individuelles.
'''
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKCYAN = '\033[96m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
'''

def prepTrouve():
    global currObject
    levelOne: list = []

    if pushd(levelOne, currObject):
        repertoire = os.listdir('.')
        repertoire.sort()
        if repertoire:
            for file in repertoire:
                currObject = file
                trouveMatch()
        popd(levelOne)

def trouveMatch():
    global currObject
    cmp_ptrn = NFD_ptrn if ud.is_normalized('NFD', currObject) else NFC_ptrn

    if args.I_flag: m = re.search(cmp_ptrn, currObject, flags=re.I)
    else: m = re.search(cmp_ptrn, currObject)
```

```
    if m:
        print('\033[92m', end='') if args.n_flag else print('\033[91m', end='')

        if args.I_flag: newfile = re.sub(cmp_ptrn, args.repl, currObject, flags=re.I)
        else: newfile = re.sub(cmp_ptrn, args.repl, currObject)

        if os.path.isfile(currObject) and args.f_flag:
            if args.v_flag: print(f"    {currObject} ==> {newfile}")
            if not args.n_flag: os.rename(currObject, newfile)

        if os.path.isdir(currObject) and args.d_flag:
            if args.v_flag: print(f"....==>>    /{currObject}/ ==> {newfile}")
            if not args.n_flag:
                os.rename(currObject, newfile)
                currObject = newfile

        print('\033[0m', end='')

pushRep = []

def fc_recursif():
    global currObject

    if args.r_flag > 50: return # Quand t'es rendu à 50 dirs de creux !!!
    if args.i_flag or args.r_flag >= 2: trouveMatch()

    if pushd(pushRep, currObject):
        if args.v_flag: print(f'----- {os.path.abspath(os.path.curdir)} ===>>')
        args.r_flag += 1
        fl = os.listdir('.')
        fl.sort()
        if fl:
            for obj in fl:
                currObject = obj
                if os.path.isfile(obj):
                    trouveMatch()
                if os.path.isdir(obj):
                    fc_recursif()
            if args.v_flag: print(f'----- {os.path.abspath(os.path.curdir)} <<===')
        popd(pushRep)
        args.r_flag -= 1
    else:
        print(f"Impossible d'entrer dans le répertoire «{obj}».")

# --------------  Début du programme  -------------------------
# Si aucun des flags n'a été choisi, c'est qu'on veut les deux.
if args.f_flag == False and args.d_flag == False:
    args.f_flag = args.d_flag = True

# Si on veut simuler, faut accepter d'être verbose.
if args.n_flag: args.v_flag = True

# Convert pattern to NFD et vice-et-versa
if ud.is_normalized('NFC', args.ptrn):
    NFD_ptrn = ud.normalize('NFD', args.ptrn)
    NFC_ptrn = args.ptrn
else:
    NFD_ptrn = args.ptrn
    NFC_ptrn = ud.normalize('NFC', args.ptrn)

for path in args.paths:
    currObject = path

    if os.path.isdir(path):
        currObject = os.path.abspath(path)
        if args.r_flag:
            fc_recursif()
        else:
            if args.v_flag: print(currObject)
            prepTrouve()
    elif os.path.isfile(path):
        print(f"«{path}» est un fichier, il me faut un répertoire.")
    elif not os.path.exists(path):
        print(f"«{path}» n'existe pas")
    else:
        print(f"«{path}» n'est ni un répertoire, ni un fichier.")
```



mren.cpp

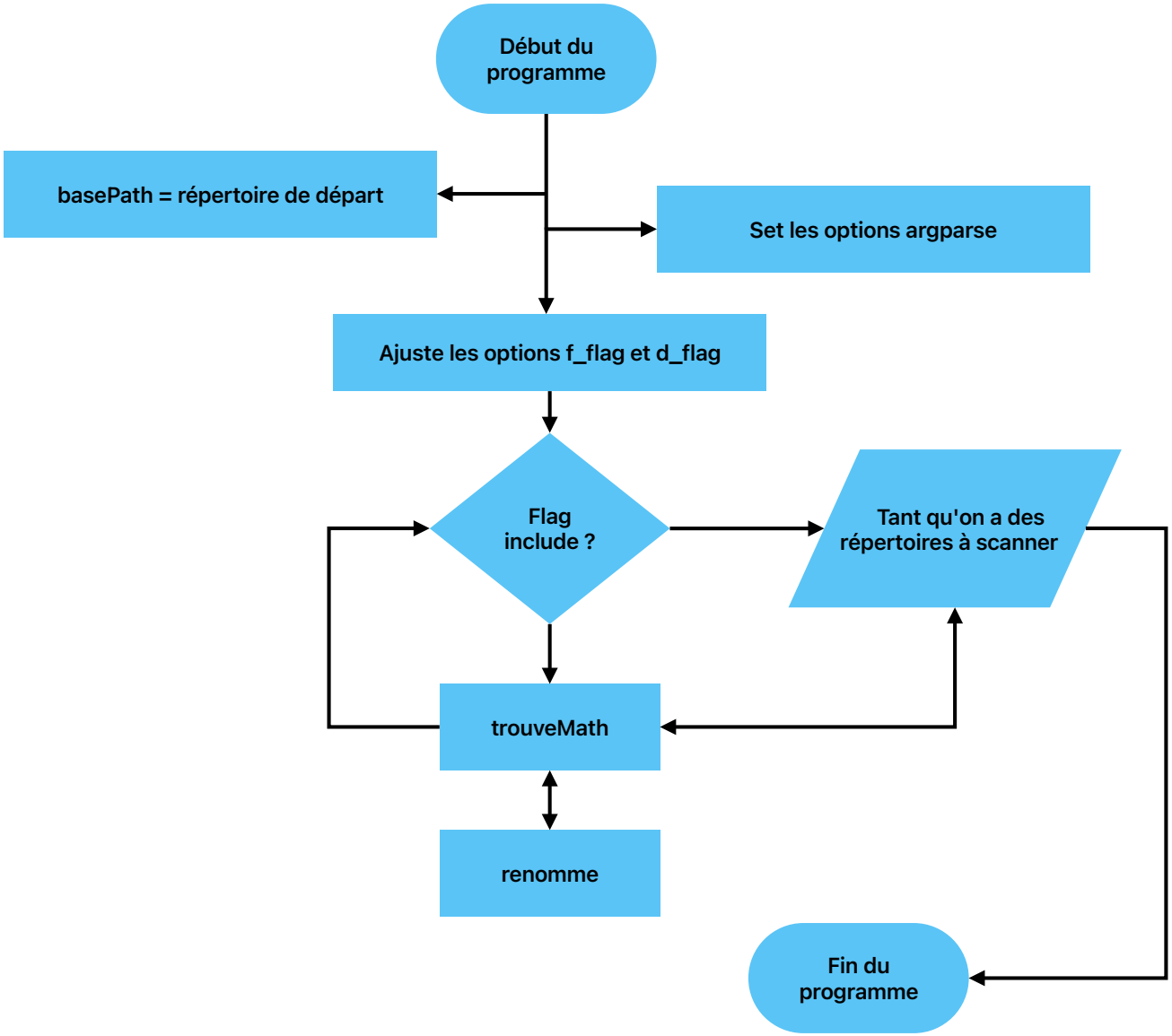
```cpp
#include <iostream> // cout ...
#include <sstream>  // Pour stringStream : fonction récursive : Capturer cout
#include <vector>   // dname, fname
#include <unistd.h> // chdir(), getcwd()
#include <regex>    // ...
#include <dirent.h> // Pour récupérer le contenu d'un répertoire
#include <sys/stat.h>   // Récupère les info d'un fichier/dir
#include <cctype>   // std::tolower
#include "argparse.hpp" // Parse la ligne de commande

#ifdef __APPLE__
    #include "minNFC2NFD.hpp"
#endif

// Pour compiler: g++ -std=c++11 mren.cpp -o mren

struct flags // all flags and default values
{
    bool f_flag = false;
    bool d_flag = false;
    bool r_flag = false;
    bool i_flag = false;
    bool I_flag = false;
    bool n_flag = false;
    bool v_flag = false;
};

struct couleurTerm
{
    std::string FGROUGE = "\033[91m";
    std::string FGVERT = "\033[92m";
    std::string FGJAUNE = "\033[93m";
    std::string FGBLEU = "\033[94m";
    std::string FGMAGENTA = "\033[95m";
    std::string FGCYAN = "\033[96m";
    std::string RESET = "\033[0m";
    std::string GRAS = "\033[1m";
    std::string ITALIQUE = "\033[3m";
    std::string UNDERLINE = "\033[4m";
};

struct flags fl;
struct couleurTerm coul;

std::string ptrn;

#ifdef __APPLE__
    std::string ptrn_NFD;
#endif

std::string repl;
std::string currObject;

auto regexOptionI = std::regex_constants::ECMAScript;

std::string message_usage = // option -h
    "usage: mren [-f|-d] [-riInv] <regex pattern> <remplacement> [dirname ...]";

std::string message_description =
    "Renommage multiple selon un certain modèle";

std::string message_version = "version 2024-02-20";

std::string message_aide = // option --help
    message_usage+"\n""\n"+message_description+"\n""\n"
    "Arguments en position:\n"
    "  <regex pattern>   Modèle à chercher: Mettre entre '...'\n"
    "  <remplacement>    Chaîne de remplacement\n"
    "  [dirname ...]     Répertoire(s) de recherche\n"
    "\n"
    "Options:\n"
    "  -f                N'agit que sur les fichiers\n"
    "  -d                N'agit que sur les répertoires\n"
    "  -r, --recursive   Procède de façon récursive sur les répertoires\n"
    "  -i, --include     En mode récusif, inclu le dossier en ligne de commande\n"
    "  -I, --ignoreCase  Fait une recherche en ignorant la case\n"
    "  -n, --simulate    Simule les opérations demandées - Fichiers affectés en VERT\n"
    "  -v, --verbose     Donne des détails sur le(s) fichier(s) traité(s) - Fichiers affectés en ROUGE\n"
    "  -ver, --version   Renommage multiple à partir d'un modèle\n"
    "  -h, --help        Montre ce message d'aide et termine";

std::string message_erreur =
    coul.FGROUGE+"- - ATTENTION - - ATTENTION - - ATTENTION - - ATTENTION - - ATTENTION - -"+coul.RESET+"\n"
```

```
    "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n"
    "Il faut au minimum: <modèle> <remplacement>\n"
    "en ligne de commande: "+coul.FGROUGE+"ex. '\\.jpeg' '.jpg'"+coul.RESET+"\n"
    "Plus les flags de contrôle le cas échéant.\n"
    "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n"+message_aide;

std::string message_exclude =
    "Pour les flags «f|d», il sont mutuellement exclusif. C'est un ou c'est l'autre.";

std::string get_working_path()
{
    char pathBuf[PATH_MAX];

    if (getcwd(pathBuf, PATH_MAX) == 0) {

        int error = errno;

        switch (error) {
            // EINVAL can't happen - size argument > 0

            // PATH_MAX includes the terminating nul,
            // so ERANGE should not be returned

            case EACCES:
                std::cout << "Access denied" << std::endl;

            case ENOMEM:
                // I'm not sure whether this can happen or not
                std::cout << "Insufficient storage" << std::endl;

            default: {
                std::cout << "Unrecognised error" << std::endl;
            }
        }
    }
    return std::string(pathBuf);
}

bool renomme(std::string nom, std::string nouveauNom, std::string indent)
{
    if(fl.n_flag) {
        std::cout << indent << nom
        << coul.FGVERT << " ==> Deviendrait ==> " << coul.RESET
        << nouveauNom << '\n';
        return(false);
    }
    else {
        if (std::rename(nom.c_str(), nouveauNom.c_str())) {
            std::cout << coul.FGROUGE << "Je ne peux renommer "
                << coul.RESET << nom << " en " << nouveauNom << std::endl;
            return(false);
        }
        if (fl.v_flag) std::cout << indent << nom
                << coul.FGROUGE << " ==> est devenu ==> " << coul.RESET
                << nouveauNom << '\n';
        return(true);
    }
}

int iteration = 0;
bool trouveMatch()
{
    bool match = false;
    bool recursive_match = false;

    if (iteration > 100) return false;  // Pas normal, on doit être pris dans une boucle

    DIR *dh;
    struct dirent* contents;
    std::string prevPath = get_working_path();
    std::string space = "▶ "; // Pour créer une indentation en récursif
    std::string rspace = "◀ "; // Pour créer une indentation en récursif

    // Pour créer l'indentation necessaire
    for (int i = 0; i < iteration; i++) {
        space.insert(0, "  ");
        rspace.insert(0, "  ");
    }
    if ( (dh = opendir (".")) != NULL) {
        std::string name;
        std::vector<std::string> fname; // Un vector pour les fichiers
        std::vector<std::string> dname; // Un vector pour les répertoires

        // Récupère la liste de tout les fichiers en 2 vector
        // dname: répertoires, fname: les fichiers.
        while ((contents = readdir( dh )) != NULL)
```

```
        {
            name = contents->d_name;

            if ((name == ".") || (name == "..")) continue;  // On en veut pas

            if (contents->d_type == DT_DIR) dname.push_back(name);
            else if (contents->d_type == DT_REG) fname.push_back(name);
            else continue;
        }
        closedir (dh);

        // Manipulation pour trier les vectors dname et fname.
        struct {
            bool operator()(std::string a, std::string b) const {
                return std::tolower(a[0]) < std::tolower(b[0]);
            }
        }
        customLess;
        std::sort(fname.begin(), fname.end(), customLess);
        std::sort(dname.begin(), dname.end(), customLess);


        // lambda function pour éviter la répétition de code
        auto find_rename = [&] (std::regex& sr) {
        //                  ^         ^ Argument passé en référence à lambda
        //                  | pour récupérer les variable du bloc parent
            // Scan chaque fichier pour un match.
            if (fl.f_flag) {
                std::string new_f;
                for ( auto &f : fname ) {
                    new_f = std::regex_replace(f, sr, repl);
                    if (new_f != f) {
                        match = true;
                        renomme(f, new_f, space);
                    }
                }
            }
            // Scan chaque répertoire pour un match.
            if (fl.d_flag) {
                std::string new_d;
                for ( auto &d : dname ) {
                    new_d = std::regex_replace(d, sr, repl);
                    if (new_d != d) {
                        match = true;
                        if (renomme(d, new_d, space)) d = new_d;
                    }
                }
            }
        };

//  BLOC QUI S'EXÉCUTE QUAND IL N'Y A PAS DE CARACTÈRES SPÉCIAUX
        std::regex self_replace(ptrn, regexOptionI);
        find_rename(self_replace);

//  FIN DU BLOC
#ifdef __APPLE__
//  BLOC QUI S'EXÉCUTE QUAND IL Y A DES CARACTÈRES SPÉCIAUX
        if( ptrn != ptrn_NFD) // Si il ne sont pas pareil refaisont le tour.
        {
            std::regex self_replace(ptrn_NFD, regexOptionI);
            find_rename(self_replace);
        }
//  FIN DU BLOC
#endif
        // Quand on veut y aller récursivement.
        if (fl.r_flag == true) {
            if (! match && dname.empty()) return false;

            for ( auto d : dname ) {
                recursive_match = false;
                iteration++;
                if (chdir(d.c_str()) == 0) {
                    if (fl.v_flag) {
                        std::stringstream ss;
                        ss << space << coul.FGBLEU << d << coul.RESET << " ▶" << std::endl;

                        auto orig = std::cout.rdbuf(ss.rdbuf());    //Capture le stream de cout
                        // Si il n'y a pas de match, je ne veux pas l'afficher
                        if (trouveMatch()) recursive_match = true;
                        else recursive_match = recursive_match|false;   // Pour passer le match au parent

                        ss << rspace << coul.FGBLEU << d << coul.RESET << " ◀" << std::endl;

                        std::cout.rdbuf(orig);                      // Reset le buffer à cout
                        if (recursive_match) std::cout << ss.str() << std::flush;
                    }
```

```
                    else {
                        if (trouveMatch()) recursive_match = true;
                        else recursive_match = recursive_match|false;   // Pour passer le match au parent
                    }
                    chdir(prevPath.c_str());
                }
                else { std::cout << coul.FGROUGE <<"---" << coul.RESET << " Je ne peux "
                                 << coul.FGROUGE << "«chdir()»" << coul.RESET << " vers le répertoire "
                                 << coul.FGROUGE << d << " ---" << coul.RESET << std::endl; }
                match = match|recursive_match;
                iteration--;
            }
        }
    }
    else {
        std::cout << "Ne peut lire le répertoire " << get_working_path() << std::endl;
        return false;
    }
    return match|recursive_match;
}

int main(int argc, char *argv[])
{
    char basePath [ PATH_MAX ];
    if ( getcwd(basePath, PATH_MAX) == 0) exit(1); // Si tu peux pas connaître ton cwd: ça va mal

    // Set les éléments pour parser avec argparse.hpp
    argparse arg({.version = message_version, .description = message_description,
                                  .usage = message_usage, .helpMsg = message_aide});

    if (! arg.addOption({.varPtr = &fl.f_flag, .shortOption = "-f", .varType = def::BOOL})) return 1;
    if (! arg.addOption({.varPtr = &fl.d_flag, .shortOption = "-d", .varType = def::BOOL})) return 1;
    if (! arg.addOption({.varPtr = &fl.r_flag, .shortOption = "-r", .longOption = "--recursive", .varType = def::BOOL})) return 1;
    if (! arg.addOption({.varPtr = &fl.i_flag, .shortOption = "-i", .longOption = "--include", .varType = def::BOOL})) return 1;
    if (! arg.addOption({.varPtr = &fl.I_flag, .shortOption = "-I", .longOption = "--ignoreCase", .varType = def::BOOL})) return 1;
    if (! arg.addOption({.varPtr = &fl.n_flag, .shortOption = "-n", .longOption = "--simulate", .varType = def::BOOL})) return 1;
    if (! arg.addOption({.varPtr = &fl.v_flag, .shortOption = "-v", .longOption = "--verbose", .varType = def::BOOL})) return 1;

    int ret = arg.parse(argc, argv);
    if (ret == retcode::ERROR) return 1;
    if (ret == retcode::HELP_VERSION) return 0;

    if (arg.argPos_c < 2) { std::cout << message_erreur << std::endl; return 1; }

    // On ajuste les flags f_flag et d_flag pour être cohérent
    if ((fl.f_flag == false) && (fl.d_flag == false)) fl.f_flag = fl.d_flag = true;
    else if ((fl.f_flag == true) && (fl.d_flag == true)) {
        std::cout << message_exclude << std::endl;
        return 1;
    }

    if (fl.n_flag == true) fl.v_flag = true; // Verbose si une simulation
    if (fl.I_flag == true) regexOptionI = regexOptionI | std::regex_constants::icase;

    // Le premier sera toujours ptrn et le deuxième repl.
    ptrn = arg.argPos_v[0];

#ifdef __APPLE__
    ptrn_NFD = ptrn;
    if (! isPlainText(ptrn))
    {
        ptrn_NFD = minNFC2NFD(ptrn);
    }
#endif
    repl = arg.argPos_v[1];

    if (arg.argPos_c < 3) { arg.argPos_v.push_back("./"); arg.argPos_c++; }
    // Si nous avons que 2 argPos_v, on force le troisième au current dir.

    for (int i = 2; i < arg.argPos_c; i++) // À partir du troisième ce sont des dirs.
    {
        struct stat sb; // Teste pour un répertoire valide
        if (!(stat(arg.argPos_v[i].c_str(), &sb) == 0 && (sb.st_mode & S_IFDIR))) {
            std::cout << "+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +" << std::endl;
            std::cout << coul.FGROUGE << arg.argPos_v[i] << coul.RESET
                      << " n'est pas un répertoire valide." << std::endl;
            continue;
        }
        // À partir d'ici nous avons un répertoire valide.
        char fullPath [ PATH_MAX ];
        realpath(arg.argPos_v[i].c_str(), fullPath);
        currObject = fullPath;

        if (fl.v_flag) {
            if (i > 2) std::cout << "+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +" << std::endl;
            std::cout << "À partir du répertoire: "
```

```
                        << coul.FGBLEU << currObject << coul.RESET << "\n"
                        << "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -" << std::endl;
        }
        if (fl.i_flag && fl.d_flag) { // Si on a le flag -i, inclure le rep source.
            std::string baseName = currObject.substr(currObject.find_last_of("/\\") + 1);
            std::string pathName = currObject.substr(0, currObject.find_last_of("/\\"));

            // lambda function pour éviter la répétition de code
            auto find_rename = [&] (std::regex& sr) {
            //                   ^         ^ Argument passé en référence à lambda
            //                   | pour récupérer les variable du bloc parent
                std::string new_baseName = std::regex_replace(baseName, sr, repl);
                if (baseName != new_baseName) {
                    if ( ! chdir(pathName.c_str()) ) {
                        if (renomme(baseName, new_baseName, "⊙ ")) {
                            // currObject a changé de nom.
                            currObject = pathName + "/" + new_baseName;
                        }
                        std::cout << "- - - - - - - - - - - - - - - - - - - - - - - - - - - - -" << std::endl;
                    }
                }
            };

//  BLOC QUI S'EXÉCUTE QUAND IL N'Y A PAS DE CARACTÈRES SPÉCIAUX
            std::regex self_replace(ptrn, regexOptionI);
            find_rename(self_replace);
//  FIN DU BLOC
#ifdef __APPLE__
//  BLOC QUI S'EXÉCUTE QUAND IL Y A DES CARACTÈRES SPÉCIAUX
            if (ptrn != ptrn_NFD) {
                std::regex self_replace(ptrn_NFD, regexOptionI);
                find_rename(self_replace);
            }
//  FIN DU BLOC
#endif

        }
        if ( ! chdir(currObject.c_str()) ) {
            if (! trouveMatch()) std::cout << "Pas de correspondance pour ce répertoire ..." << std::endl;
            chdir(basePath); // À cause de la boucle, toujours revenir au dossier de base
        }
        else {
            std::cout << "Impossible de travailler avec le répertoire "
                      << coul.FGROUGE << arg.argPos_v[i] << coul.RESET << std::endl;
            continue;
        }
    }
}

```


Début du programme
basePath = répertoire de départ
Set les options argparse
Ajuste les options f_flag et d_flag
Flag include ?
Tant qu'on a des répertoires à scanner
trouveMath
renomme
Fin du programme