컴퓨터학부 20211407 김지호

1. 개요

linux 시스템 상에서 사용자가 원하는 디렉토리에 대한 tree 생성 및 다양한 옵션을 가진 정렬을 수행하는 프로그램

2. 기능 및 프로토타입

2-1. main.c (exit 내장명령어 기능 포함)

2-1-1. 개요

main 모듈은 프로그램의 시작점으로, 사용자 입력을 받아 cmd_process 함수를 통해 명령어를 분기 처리한다.
명령어 종류는 tree, arrange, help, exit 등이 있으며, 각각의 하위 함수를 호출하여 동작을 수행한다.

2-1-2. 함수 프로토타입 및 설명

int main(void);
- 프로그램의 진입점.
- 사용자에게 프롬프트(20211407>)를 출력하고, 반복문을 통해 사용자의 입력을 받아 처리한다.
- 입력받은 명령어를 cmd_process로 전달하여 해당 명령어에 맞는 함수를 호출한다.
- exit 명령어가 입력되면 프로그램이 종료된다.

*void cmd_process(char input);
- 사용자가 입력한 문자열(input)을 공백 단위로 파싱하여, 첫 번째 토큰을 명령어로 인식한다.
- 명령어가 tree, arrange, help, exit 중 무엇인지 판별하고, 해당하는 함수를 호출한다.
- 그 외의 명령어는 cmd_help 함수를 호출하여 도움말을 출력한다.

void cmd_help(void);
- help 명령어를 처리하는 함수.
- 인자가 없으면 전체 명령어(tree, arrange, help, exit)에 대한 사용법을 출력한다.

void cmd_exit(void);
- exit 명령어를 처리하는 함수.
- exit(0)를 호출하여 프로그램을 정상 종료한다

2-2. tree.c

2-2-1. 개요

tree 모듈은 디렉터리 구조를 트리 형태로 출력하는 기능을 담당한다.
사용자는 tree <DIR_PATH> 형태로 명령어를 입력하며, 추가 옵션(-s, -p)을 통해 파일 크기, 파일 권한 정보를 함께 출력할 수 있다. 내부적으로 재귀 함수를 사용하여 디렉터리를 순회하고, FileNode 구조를 이용해 트리 형태를 구성한 뒤 이를 출력한다.

2-2-2. 함수 프로토타입 및 설명

*void cmd_tree(char path, int opt_size, int opt_permission);
- tree 명령어의 메인 함수.
- 입력된 path를 실제 경로(realpath)로 확인하고, HOME 디렉터리 내부 여부, 디렉터리 여부를 검사한다.
- 문제 없으면 get_root로 루트 노드를 만들고, print_tree로 최종 출력한다.
- 트리 구성 후, 디렉터리/파일 개수를 count_nodes로 계산하여 결과를 함께 표시한다.

**FileNode *create_node(const char path, const char name, int is_root, int is_last);
- 파일(또는 디렉터리)을 나타내는 FileNode 구조체를 동적 할당하여 초기화한다.
- path, name을 저장하고, lstat로 파일 상태 정보를 읽어 is_dir 등을 설정한다.
- is_root와 is_last 플래그로 루트 노드 여부, 마지막 형제 노드 여부를 구분한다.

**FileNode *get_root(const char abs_path, const char origin_path);
- 트리의 루트 노드를 생성하기 위한 함수.
- create_node를 통해 루트 노드를 만든 뒤, build_tree를 호출하여 전체 트리를 구성한다.
- 루트 노드의 이름은 사용자가 입력한 origin_path 그대로 사용한다.

*void build_tree(FileNode parent);
- 주어진 parent 노드가 디렉터리라면, scandir를 이용하여 하위 항목들을 읽고, 각 항목을 create_node로 생성한다.
- 형제 노드(sibling)로 연결하며, 재귀적으로 하위 디렉터리를 탐색하여 트리 구조를 만든다.

**void print_tree(FileNode node, const char prefix, int opt_size, int opt_permission);
- 재귀적으로 트리를 탐색하면서, 들여쓰기(prefix)를 이용해 트리 구조를 시각적으로 출력한다.
- opt_size가 참이면 파일 크기를, opt_permission이 참이면 파일 권한을 추가로 표시한다.
void print_permissions(mode_t mode);
- 파일(또는 디렉터리)의 mode 값을 해석하여 [drwxr-xr-x] 형태로 권한을 출력한다.
*int is_inside_home(const char path);
- 주어진 path가 $HOME 경로 내부인지 검사한다.
- $HOME 문자열과 path의 앞부분을 비교하여 일치하면 1, 아니면 0을 반환한다.
**void count_nodes(FileNode *node, int dir_count, int file_count);
- 재귀적으로 트리를 순회하면서, 디렉터리와 파일 개수를 센다(루트 노드는 제외).
- 결과를 dir_count, file_count에 누적한다.
**int cmp(const struct dirent **a, const struct dirent b);
- scandir에서 사용하는 정렬 함수.
- (*a)->d_name과 (*b)->d_name을 strcmp로 비교하여 알파벳 순으로 정렬한다.
**int expand_tilde(const char input_path, char expanded_path);
- 경로가 ~로 시작하면 $HOME 디렉터리로 치환하여 expanded_path에 저장한다.
- 변환이 이루어지면 1, 아니면 0을 반환한다.

2-3. arrange.c

2-3-1. 개요

arrange 모듈은 디렉터리 내의 파일들을 확장자별로 분류하여 복사·정리하는 기능을 담당한다. arrange <DIR_PATH> 형태로 명령어가 주어지며, -d(출력 디렉터리 지정), -t(수정 시간 제한), -x(제외 디렉터리), -e(허용 확장자) 등의 옵션을 통해 동작을 제어할 수 있다. 내부적으로 재귀 함수를 통해 디렉터리를 순회(traverse_directory)하며, 파일 확장자별(ExtGroup) 및 동일 파일명(ConflictGroup)을 그룹화하고, 필요 시 충돌(handle_conflicts)을 해결한다.

2-3-2. 함수 프로토타입 및 설명

*void cmd_arrange(char path);
- arrange 명령어의 메인 함수.
- 사용자 입력을 토큰화하여 Options 구조체에 옵션을 파싱(parse_options)하고, 경로 유효성(realpath), 디렉터리 여부 등을 확인한다.
- 문제 없으면 traverse_directory로 파일들을 확장자별·파일명별 그룹화하고, handle_conflicts로 충돌 파일을 해결한 뒤, copy_selected_files로 결과 디렉터리에 복사한다.
- 마지막으로 <basename> arranged라는 완료 메시지를 출력한다.
**void parse_options(int argc, char argv[], Options opts);
- arrange 명령어에 전달된 인자들을 분석하여, Options 구조체(예: opts->result_dir, opts->exclude_dirs, opts->extensions, opts->time_limit 등)에 저장한다.
- -d, -x, -e, -t 옵션을 처리한다.
*void free_options(Options opts);
- parse_options 과정에서 동적 할당한 exclude_dirs, extensions 배열과 그 내부 문자열들을 free()하여 메모리를 해제한다.
*int validate_path(const char path);
- 경로가 NULL인지, 길이가 MAX_PATH를 초과하지 않는지 검사한다.
- 유효하지 않을 경우 -1을, 정상인 경우 0을 반환한다.
*int is_directory(const char path);
- stat 함수를 사용하여 path가 디렉터리인지 검사한다.
- 디렉터리이면 1, 아니면 0을 반환한다.
**void traverse_directory(const char *path, ExtGroup *ext_list, Options opts);
- 주어진 디렉터리를 재귀적으로 순회하며, 파일마다 확장자를 기준으로 ExtGroup에 분류한다.
- -x 옵션(제외 디렉터리), -t 옵션(시간 제한), -e 옵션(확장자 필터)을 적용하여 파일을 건너뛰거나 포함한다.
- 같은 파일 이름은 ConflictGroup으로 묶여 충돌로 관리된다.

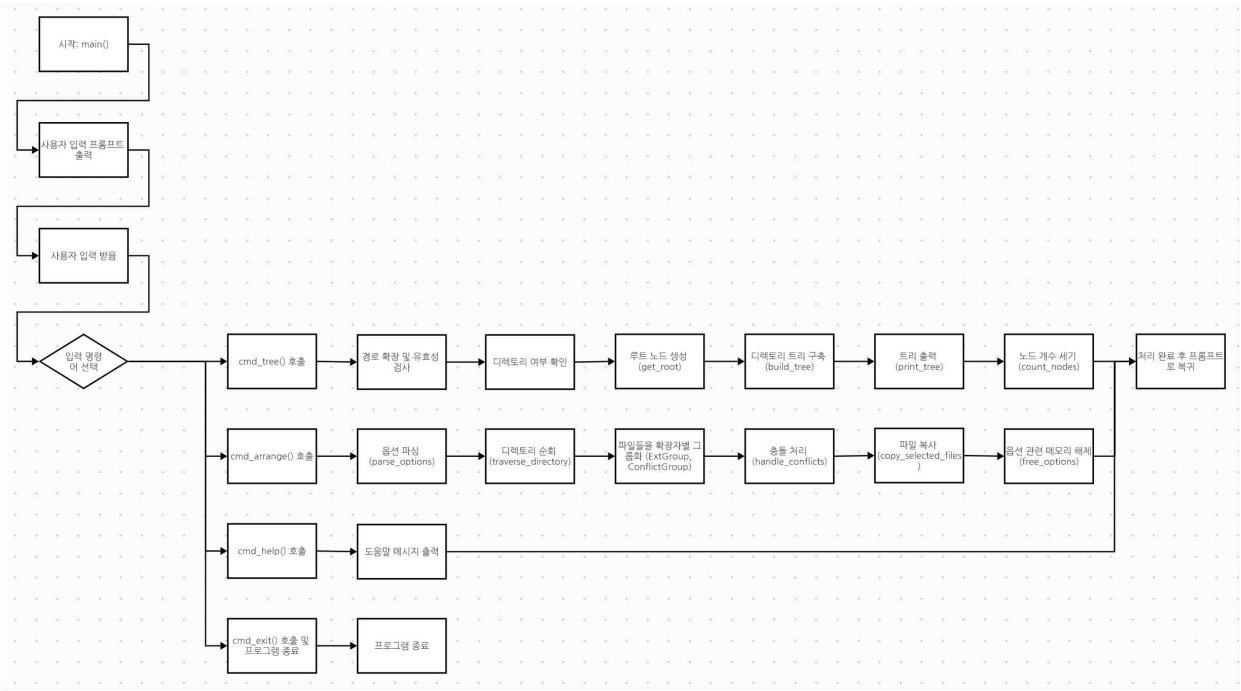
**int should_skip_dir(const char dirname, Options opts);
- -x 옵션으로 지정된 디렉터리 목록(opts->exclude_dirs)과 비교하여, dirname이 해당 목록에 있으면 1, 없으면 0을 반환한다.
**int is_valid_extension(const char ext, Options opts);
- -e 옵션으로 지정된 확장자 목록(opts->extensions)에 ext가 포함되어 있으면 1, 아니면 0을 반환한다.
*void handle_conflicts(ExtGroup ext_list);
- 확장자 그룹(ExtGroup)마다 파일 이름이 같은 ConflictGroup에 속한 파일이 2개 이상인 경우, 사용자에게 select/diff/vi/do not select 등의 옵션을 제공하여 충돌을 해결한다.
**void copy_selected_files(ExtGroup ext_list, const char dest_dir);
- 충돌이 해소된 파일들을 확장자별로 디렉터리를 생성(예: <dest_dir>/<확장자>)하고, copy_file를 통해 복사한다.
- 이미 존재하는 파일은 덮어쓰며, 각 파일 복사 후 완료 메시지를 따로 출력하지 않고, 마지막에 <basename> arranged만 출력한다.
**ExtGroup create_ext_group(const char extension);
- 특정 확장자를 담는 ExtGroup 노드를 동적 할당하여 생성하고, extension 문자열을 복사해 저장한다.
**ConflictGroup create_conflict_group(const char filename);
- 특정 파일 이름을 담는 ConflictGroup 노드를 동적 할당하여 생성하고, filename을 복사해 저장한다.
**ExtGroup *find_ext_group(ExtGroup head, const char extension);
- 연결 리스트로 구성된 확장자 그룹(ExtGroup)에서, 해당 extension을 가진 노드를 검색해 반환한다.
- 없으면 NULL을 반환한다.
**ConflictGroup *find_conflict_group(ConflictGroup head, const char filename);
- 연결 리스트로 구성된 충돌 그룹(ConflictGroup)에서, filename을 가진 노드를 검색해 반환한다.
- 없으면 NULL을 반환한다.
**void add_file_to_groups(ExtGroup **ext_list, const char *filepath, const char filename, const char extension, time_t mtime);
- extension에 해당하는 ExtGroup 노드를 찾거나 새로 만들고, 그 안에서 filename 충돌 그룹을 찾거나 새로 만든 뒤, 파일 정보를 추가한다.
- 내부적으로 add_file_to_conflict_group를 호출하여 실제 파일(ConflictFile)을 배열에 저장한다.

2-4. 공통 유틸리티 함수

**char get_extension(const char filename);
- 파일 이름에서 마지막 '.' 뒤의 문자열을 반환하며, 확장자가 없으면 빈 문자열을 반환한다.
**char *concat_path(const char dir, const char file);
- 디렉터리 경로(dir)와 파일 이름(file)을 합쳐 새로운 문자열을 동적 할당하여 반환한다.
**int copy_file(const char src, const char dst);
- open/read/write 등의 저수준 I/O를 사용하여 src에서 dst로 파일을 복사한다.
- 성공 시 0, 실패 시 -1을 반환한다.

3. 상세설계

1) flow chart



2) 전체 flow

(1) 프로그램 시작 (main)

- 프로그램이 실행되면 main 함수가 호출
- 초기 프롬프트("20211407> ")를 출력하고, 사용자의 입력을 대기

(2) 사용자 입력 대기 및 처리 (main 내 반복문)

- fgets로 한 줄을 입력받아, 문자열 끝의 개행 문자(\n)를 제거
- 입력된 문자열이 비어 있지 않다면 cmd_process(input) 함수를 호출하여 명령어를 처리
- 명령어 처리가 끝나면 다시 프롬프트를 출력하고 다음 입력을 기다리는 과정을 반복

(3) cmd_process: 명령어 파싱 및 분기

- 입력된 문자열을 공백 단위로 토큰화하여, 첫 번째 토큰(명령어)을 cmd에 저장
- cmd 값에 따라 분기:

"tree": 다음 토큰을 디렉터리 경로로 간주하고, 추가 옵션(-s, -p 등)을 파싱한 뒤 cmd_tree 함수로 이동

"arrange": 다음 토큰을 디렉터리 경로로 간주하고, 추가 옵션(-d, -x, -e, -t 등)을 파싱한 뒤 cmd_arrange 함수로 이동

"help": 전체 사용법을 출력

"exit": cmd_exit 호출 → 프로그램 종료

기타: 잘못된 명령어이므로 cmd_help를 호출하여 전체 사용법 출력

(4) tree 명령어 처리 (cmd_tree)

- 입력된 경로를 실경로(realpath)로 변환하고, HOME 내부 여부, 디렉터리 여부 등을 확인
- get_root 함수를 호출하여 트리의 루트 노드를 생성하고, 내부적으로 build_tree를 통해 재귀적으로 하위 디렉터리를 탐색하며 노드들을 연결
- 구성된 트리 구조를 print_tree를 통해 재귀적으로 출력. 옵션(-s, -p)에 따라 파일 크기와 권한 정보를 함께 표시
- count_nodes 함수를 사용하여 디렉터리와 파일 개수를 세고, 그 결과를 출력한 뒤 종료

(5) arrange 명령어 처리 (cmd_arrange)

- 옵션 파싱

사용자가 입력한 <DIR_PATH> 뒤에 오는 토큰들을 parse_options로 분석하여, Options 구조체(opts)에 저장

예: -d로 출력 디렉터리 지정, -x로 제외할 디렉터리 목록 지정, -e로 허용 확장자 지정, -t로 수정 시간 제한 설정 등.

- 경로 검증 및 디렉터리 확인

realpath로 실제 경로를 확인하고, 실패하면 <path> does not exist를 출력 후 종료.

is_directory로 디렉터리인지 확인하고, 아니면 <path> is not a directory 출력 후 종료.

출력 디렉터리가 따로 지정되지 않았다면 <DIR_PATH>_arranged 형태로 자동 설정.

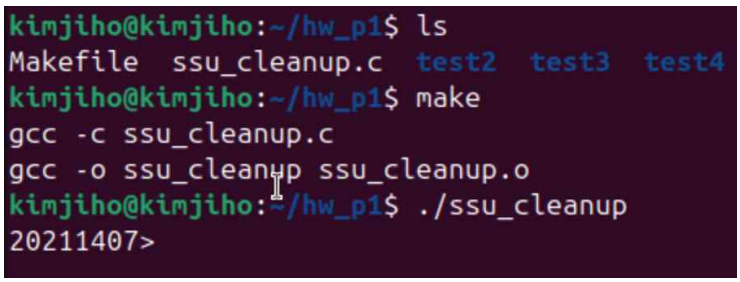
- 디렉터리 트래버스 및 파일 그룹화
  traverse_directory 함수에서, 재귀적으로 디렉터리를 순회하며, 파일을 확장자별로 ExtGroup에 분류
  이 과정에서 -x 옵션(제외 디렉터리), -t 옵션(시간 제한), -e 옵션(확장자 필터)을 적용
  파일 이름이 동일하면 ConflictGroup으로 묶여 “충돌 파일”로 관리
- 충돌 처리 (handle_conflicts)
  확장자 그룹 내에서 파일 이름이 동일(충돌)한 경우, 사용자와 상호작용하여 어떤 파일을 남길지/버릴지, diff/vi로 비교·편집할지 결정
- 파일 복사 (copy_selected_files)
  충돌이 해소된(또는 충돌이 없던) 파일들을 확장자별 하위 디렉터리(예: <결과디렉터리>/<확장자>)에 복사.
  모든 파일 복사가 끝나면 <basename> arranged를 출력.
  여기서 <basename>은 사용자 입력 디렉터리 경로의 마지막 컴포넌트(opts.dir_path에서 추출).

(6) help 명령어 (cmd_help)
- 추가 인자(예: "tree", "arrange")가 있으면 해당 명령어의 사용법만,
- 인자가 없으면 전체 명령어에 대한 사용법을 출력합니다.

(7) exit 명령어 (cmd_exit)
- main 루프가 끝나거나 exit 명령어를 통해 종료되면 프로그램이 종료됩니다.

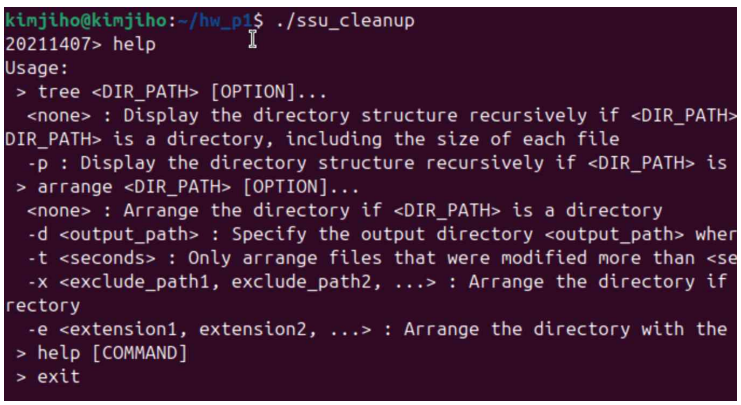
4. 실행결과

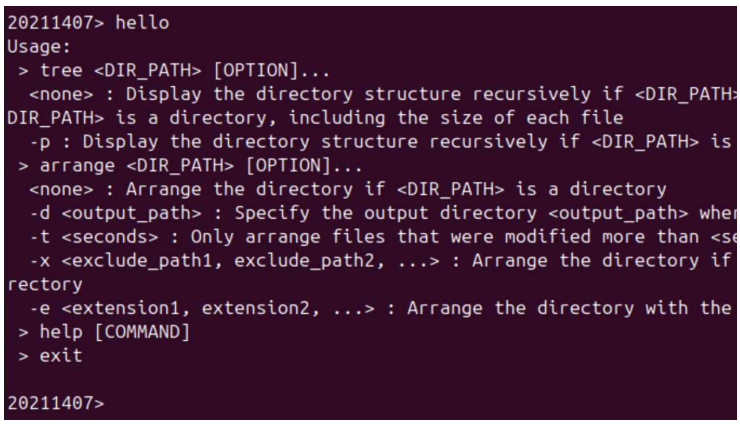
1) ssu_cleanup



- ls 결과 : 현재 디렉토리의 상태를 볼 수 있음
- make : Makefile을 통해 ssu_cleanup 컴파일
- $ ./ssu_cleanup 으로 실행하였더니, 프롬프트 나온 것을 볼 수 있음

2) help





- help 입력시 Usage 출력
- 지원하지 않은 명령어 입력시 help 출력

3) tree

(1) tree 출력
- 절대주소와 상대주소를 사용하여 tree 출력

(2) -s option

- -s 옵션으로 파일의 크기를 볼 수 있음.

```
20211407> tree /home/kimjiho/hw_p1/test2
/home/kimjiho/hw_p1/test2
├── a/
│   ├── b/
│   ├── c.c
│   └── d/
├── e.c
└── f/
    ├── g/
    │   └── h.c
    └── i.c

6 directories, 4 files
```

```
20211407> tree .
.
├── Makefile
├── ssu_cleanup
├── ssu_cleanup.c
├── ssu_cleanup.o
├── test2/
│   ├── a/
│   │   ├── b/
│   │   ├── c.c
│   │   └── d/
│   ├── e.c
│   └── f/
│       ├── g/
│       │   └── h.c
│       └── i.c
├── test3/
│   ├── a/
│   │   ├── b/
│   │   ├── c.txt
│   │   └── d/
│   ├── e.txt
│   └── f/
│       ├── g/
│       │   └── h.c
│       └── i.c
└── test4/
    ├── a/
    │   ├── abc.txt
    │   └── hello.txt
    └── b/
        └── hello.txt

16 directories, 15 files
```

```
20211407> tree /home/kimjiho/hw_p1/test2 -s
/home/kimjiho/hw_p1/test2
├── [4096] a/
│   ├── [4096] b/
│   ├── [0] c.c
│   └── [4096] d/
├── [0] e.c
└── [4096] f/
    ├── [4096] g/
    │   └── [0] h.c
    └── [0] i.c

6 directories, 4 files
```

```
20211407> tree . -s
.
├── [272] Makefile
├── [73488] ssu_cleanup
├── [30890] ssu_cleanup.c
├── [32472] ssu_cleanup.o
├── [4096] test2/
│   ├── [4096] a/
│   │   ├── [4096] b/
│   │   ├── [0] c.c
│   │   └── [4096] d/
│   ├── [0] e.c
│   └── [4096] f/
│       ├── [4096] g/
│       │   └── [0] h.c
│       └── [0] i.c
├── [4096] test3/
│   ├── [4096] a/
│   │   ├── [4096] b/
│   │   ├── [0] c.txt
│   │   └── [4096] d/
│   ├── [0] e.txt
│   └── [4096] f/
│       ├── [4096] g/
│       │   └── [0] h.c
│       └── [0] i.c
└── [4096] test4/
    ├── [4096] a/
    │   ├── [0] abc.txt
    │   └── [0] hello.txt
    └── [4096] b/
        └── [0] hello.txt

16 directories, 15 files
```

(3) -p option

- 절대주소와 상대주소 모두 -p로 권한을 볼 수 있음

(4) -sp option

```
20211407> tree /home/kimjiho/hw_p1/test2 -p
/home/kimjiho/hw_p1/test2
├ [drwxrwxr-x] a/
│   ├ [drwxrwxr-x] b/
│   ├ [-rw-rw-r--] c.c
│   └ [drwxrwxr-x] d/
├ [-rw-rw-r--] e.c
└ [drwxrwxr-x] f/
    ├ [drwxrwxr-x] g/
    │   └ [-rw-rw-r--] h.c
    └ [-rw-rw-r--] i.c

6 directories, 4 files
```

```
20211407> tree . -p
.
├ [-rw-rw-r--] Makefile
├ [-rwxrwxr-x] ssu_cleanup
├ [-rw-rw-r--] ssu_cleanup.c
├ [-rw-rw-r--] ssu_cleanup.o
├ [drwxrwxr-x] test2/
│   ├ [drwxrwxr-x] a/
│   │   ├ [drwxrwxr-x] b/
│   │   ├ [-rw-rw-r--] c.c
│   │   └ [drwxrwxr-x] d/
│   ├ [-rw-rw-r--] e.c
│   └ [drwxrwxr-x] f/
│       ├ [drwxrwxr-x] g/
│       │   └ [-rw-rw-r--] h.c
│       └ [-rw-rw-r--] i.c
├ [drwxrwxr-x] test3/
│   ├ [drwxrwxr-x] a/
│   │   ├ [drwxrwxr-x] b/
│   │   ├ [-rw-rw-r--] c.txt
│   │   └ [drwxrwxr-x] d/
│   ├ [-rw-rw-r--] e.txt
│   └ [drwxrwxr-x] f/
│       ├ [drwxrwxr-x] g/
│       │   └ [-rw-rw-r--] h.c
│       └ [-rw-rw-r--] i.c
└ [drwxrwxr-x] test4/
    ├ [drwxrwxr-x] a/
    │   ├ [-rw-rw-r--] abc.txt
    │   └ [-rw-rw-r--] hello.txt
    └ [drwxrwxr-x] b/
        └ [-rw-rw-r--] hello.txt

16 directories, 15 files
```

```
20211407> tree /home/kimjiho/hw_p1/test2 -sp
/home/kimjiho/hw_p1/test2
├ [drwxrwxr-x] [4096] a/
│   ├ [drwxrwxr-x] [4096] b/
│   ├ [-rw-rw-r--] [0] c.c
│   └ [drwxrwxr-x] [4096] d/
├ [-rw-rw-r--] [0] e.c
└ [drwxrwxr-x] [4096] f/
    ├ [drwxrwxr-x] [4096] g/
    │   └ [-rw-rw-r--] [0] h.c
    └ [-rw-rw-r--] [0] i.c

6 directories, 4 files
```

```
20211407> tree . -sp
.
├ [-rw-rw-r--] [272] Makefile
├ [-rwxrwxr-x] [73488] ssu_cleanup
├ [-rw-rw-r--] [30890] ssu_cleanup.c
├ [-rw-rw-r--] [32472] ssu_cleanup.o
├ [drwxrwxr-x] [4096] test2/
│   ├ [drwxrwxr-x] [4096] a/
│   │   ├ [drwxrwxr-x] [4096] b/
│   │   ├ [-rw-rw-r--] [0] c.c
│   │   └ [drwxrwxr-x] [4096] d/
│   ├ [-rw-rw-r--] [0] e.c
│   └ [drwxrwxr-x] [4096] f/
│       ├ [drwxrwxr-x] [4096] g/
│       │   └ [-rw-rw-r--] [0] h.c
│       └ [-rw-rw-r--] [0] i.c
├ [drwxrwxr-x] [4096] test3/
│   ├ [drwxrwxr-x] [4096] a/
│   │   ├ [drwxrwxr-x] [4096] b/
│   │   ├ [-rw-rw-r--] [0] c.txt
│   │   └ [drwxrwxr-x] [4096] d/
│   ├ [-rw-rw-r--] [0] e.txt
│   └ [drwxrwxr-x] [4096] f/
│       ├ [drwxrwxr-x] [4096] g/
│       │   └ [-rw-rw-r--] [0] h.c
│       └ [-rw-rw-r--] [0] i.c
└ [drwxrwxr-x] [4096] test4/
    ├ [drwxrwxr-x] [4096] a/
    │   ├ [-rw-rw-r--] [0] abc.txt
    │   └ [-rw-rw-r--] [0] hello.txt
    └ [drwxrwxr-x] [4096] b/
        └ [-rw-rw-r--] [0] hello.txt

16 directories, 15 files
```

- 절대주소와 상대주소 모두 -sp로 권한과 파일크기 한 번에 볼 수 있음

(5) 예외처리

4) arrange

(1) 일반 arrange

```
20211407> tree hello
Usage : tree <DIR_PATH> [OPTION]...

20211407> tree /
/ is outside the home directory

20211407> tree . -a
Usage: tree <DIR_PATH> [-s] [-p]
```

- 존재하지 않는 경로라면 Usage 출력
- home 디렉토리를 벗어나면 에러 출력
- 지원하지 않는 옵션 사용하면 Usage 출력

```
20211407> arrange test2
test2 arranged

20211407> tree test2_arranged
test2_arranged
└─ c/
   ├─ c.c
   ├─ e.c
   ├─ h.c
   └─ i.c

2 directories, 4 files
```

- 해당 경로의 파일이 정렬된 것을 볼 수 있음

(2) conflict arrange

```
20211407> arrange test4
1. test4/b/hello.txt
2. test4/a/hello.txt

choose an option:
0. select [num]
1. diff [num1] [num2]
2. vi [num]
3. do not select
20211407> vi 1

choose an option:
0. select [num]
1. diff [num1] [num2]
2. vi [num]
3. do not select
20211407>
```

- 이름이 겹치는 파일 존재하여 option 2번을 선택하면, vi 가 실행됨

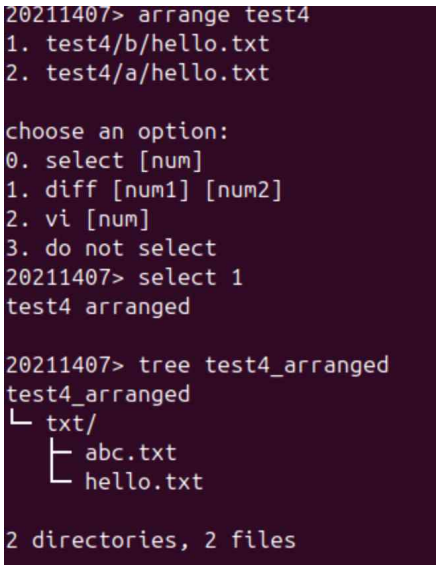
```
kimjiho@kimjiho:~/hw_p1$ ./ssu_cleanup
20211407> arrange test4
1. test4/b/hello.txt
2. test4/a/hello.txt

choose an option:
0. select [num]
1. diff [num1] [num2]
2. vi [num]
3. do not select
20211407> do not select
test4 arranged

20211407> tree test4_arranged
test4_arranged
└─ txt/
   └─ abc.txt

2 directories, 1 files
```

- 3번 옵션 선택과 그 결과

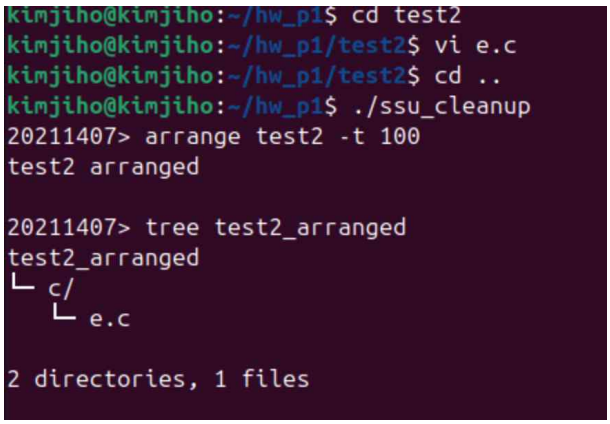


- 0번 옵션 선택시, 여러개의 파일 중 한 개만 선택할 수 있음

(3) -d 옵션

- -d 옵션으로 저장할 디렉토리의 이름을 바꿀 수 있음

(4) -t 옵션



- -t 옵션으로 최근 <sec>초 이내 수정했던 파일만 분류함

(5) -x 옵션

```
kimjiho@kimjiho:~/hw_p1$ ./ssu_cleanup
20211407> arrange test2 -x f
test2 arranged

20211407> tree test2_arranged
test2_arranged
└ c/
   ├ c.c
   └ e.c

2 directories, 2 files
```

- -x 옵션으로 해당 디렉토리의 파일은 무시하고 분류함

(6) -e 옵션

```
20211407> arrange test3 -e txt
test3 arranged

20211407> tree test3_arranged
test3_arranged
└ txt/
   ├ c.txt
   └ e.txt

2 directories, 2 files
```

- -e 옵션으로 해당 확장자 파일만 분류함

(7) 예외처리

```
20211407> arrange
Usage : arrange <DIR_PATH> [OPTION]

20211407> arrange hello
hello does not exist

20211407> arrange test2/e.c
test2/e.c is not a directory
```

- 경로 지정하지 않을 시, Usage 출력
- 존재하지 않는 경로 입력 시, 에러 출력
- 디렉토리가 아닌 파일을 입력 시, 에러 출력

5) exit

```
20211407> exit
kimjiho@kimjiho:~/hw_p1$
```

- exit 으로 프로그램 종료

5. 소스코드

```
#include <stdio.h>     // Standard I/O
#include <stdlib.h>    // malloc, free, exit, etc.
#include <string.h>    // strcpy, strcat, strcmp, etc.
#include <unistd.h>    // realpath, fork, execvp, etc.
#include <dirent.h>    // opendir, readdir, closedir, scandir
#include <sys/stat.h> // stat, lstat, mkdir
#include <sys/types.h>
#include <time.h>
#include <pwd.h>       // user info
#include <grp.h>       // group info
#include <fcntl.h>     // open flags
```

```
#include <errno.h>     // Error number definitions
#include <sys/wait.h> // waitpid

#define MAX_USER_INPUT 1024
#define MAX_FILENAME    256
#define MAX_PATH        4096
#define MAX_CONFLICTS   1000
#define MAX_GROUPS      100

// Represents a file or directory node in the tree
typedef struct FileNode {
    char name[MAX_FILENAME];    // File or directory name
    char path[MAX_PATH];        // Absolute path
    int is_dir;                 // 1 if directory, 0 if file
    int is_root;                // 1 if this node is the root of the tree
    int is_last;                // 1 if this node is the last sibling
    struct stat st;             // Stat structure for file info
    struct FileNode *child;     // Pointer to first child (if directory)
    struct FileNode *sibling;   // Pointer to next sibling
} FileNode;

// Structure for directory linked list (if needed)
typedef struct DirNode {
    char path[MAX_PATH];
    struct DirNode *next;
} DirNode;

// Structure to represent a file in a conflict group (files with the same name)
typedef struct ConflictFile {
    char *filepath; // Full path to the file
    char *filename; // File name only
    time_t mtime;   // Last modification time
} ConflictFile;

// Structure for a group of conflicting files (same filename)
typedef struct ConflictGroup {
    char *filename;             // Conflicting filename
    ConflictFile **files;       // Array of pointers to ConflictFile
    int count;                  // Number of files in the group
    int capacity;               // Allocated capacity of the array
    struct ConflictGroup *next; // Pointer to next conflict group
} ConflictGroup;

// Structure for grouping files by extension
typedef struct ExtGroup {
    char *extension;            // File extension
    ConflictGroup *conflicts;  // Linked list of conflict groups within this extension
    struct ExtGroup *next;      // Pointer to next extension group
```

```
} ExtGroup;

// Structure for arrange command options
typedef struct Options {
    char *dir_path;         // Directory to be arranged
    char *result_dir;       // Output directory name (-d option)
    char **exclude_dirs;    // Array of directories to exclude (-x option)
    int exclude_count;      // Number of exclude directories
    char **extensions;      // Array of allowed extensions (-e option)
    int ext_count;          // Number of allowed extensions
    int time_limit;         // Time limit in seconds (-t option)
} Options;

// Function prototypes
void cmd_process(char *input);
void cmd_help(void);
void cmd_exit(void);
void cmd_arrange(char *path);    // Subroutine for "arrange" command

void cmd_tree(char *path, int opt_size, int opt_permission);
FileNode *create_node(const char *path, const char *name, int is_root, int is_last);
FileNode *get_root(const char *abs_path, const char *origin_path);
void build_tree(FileNode *parent);
void print_tree(FileNode *node, const char *prefix, int opt_size, int opt_permission);
void print_permissions(mode_t mode);
int is_inside_home(const char *path);
void count_nodes(FileNode *node, int *dir_count, int *file_count);
int cmp(const struct dirent **a, const struct dirent **b);
int expand_tilde(const char *input_path, char *expanded_path);

void print_usage(const char *progname);
int validate_path(const char *path);
int is_directory(const char *path);
void parse_options(int argc, char *argv[], Options *opts);
void free_options(Options *opts);

ExtGroup *create_ext_group(const char *extension);
ConflictGroup *create_conflict_group(const char *filename);
void add_file_to_groups(ExtGroup **ext_list, const char *filepath, const char *filename, const char *extension,
time_t mtime);
ConflictGroup *find_conflict_group(ConflictGroup *head, const char *filename);
ExtGroup *find_ext_group(ExtGroup *head, const char *extension);

void traverse_directory(const char *path, ExtGroup **ext_list, Options *opts);
int should_skip_dir(const char *dirname, Options *opts);
int is_valid_extension(const char *ext, Options *opts);

void handle_conflicts(ExtGroup *ext_list);
```

```
void copy_selected_files(ExtGroup *ext_list, const char *dest_dir);

char *get_extension(const char *filename);
char *concat_path(const char *dir, const char *file);
int copy_file(const char *src, const char *dst);

// Main function: prints prompt and processes user input
int main(void) {
    char input[MAX_USER_INPUT];
    printf("20211407> ");
    while (fgets(input, MAX_USER_INPUT, stdin)) {
        input[strcspn(input, "\n")] = '\0';
        if (strlen(input) > 0)
            cmd_process(input);
        printf("\n20211407> ");
    }
    return 0;
}

// Processes the user input command
void cmd_process(char *input) {
    // Tokenize the input using space as delimiter
    char *cmd = strtok(input, " ");
    if (!cmd) return;

    if (strcmp(cmd, "tree") == 0) {
        // Process the "tree" command
        char *path = strtok(NULL, " ");
        int on_filesize = 0, on_permission = 0;
        char *option;
        while ((option = strtok(NULL, " ")) != NULL) {
            if (strcmp(option, "-s") == 0)
                on_filesize = 1;
            else if (strcmp(option, "-p") == 0)
                on_permission = 1;
            else if (strcmp(option, "-sp") == 0)
                on_filesize = on_permission = 1;
            else {
                printf("Usage: tree <DIR_PATH> [-s] [-p]\n");
                return;
            }
        }
        if (!path) {
            printf("Usage: tree <DIR_PATH> [-s] [-p]\n");
            return;
        }
        cmd_tree(path, on_filesize, on_permission);
    }
```

```
    else if (strcmp(cmd, "arrange") == 0) {
        // Process the "arrange" command
        char *path = strtok(NULL, " ");
        if (!path) {
            printf("Usage : arrange <DIR_PATH> [OPTION]\n");
            return;
        }
        // Call the arrange subroutine with the directory path
        cmd_arrange(path);
    }
    else if (strcmp(cmd, "help") == 0) {
        cmd_help();
    }
    else if (strcmp(cmd, "exit") == 0) {
        cmd_exit();
    }
    else {
        cmd_help();
    }
}

// Prints help message for available commands
void cmd_help(void) {
    printf("Usage:\n");
    printf(" > tree <DIR_PATH> [OPTION]...\n");
    printf("   <none> : Display the directory structure recursively if <DIR_PATH> is a directoryn");
    printf("   -s : Display the directory structure recursively if <DIR_PATH> is a directory, including the size of each file\n");
    printf("   -p : Display the directory structure recursively if <DIR_PATH> is a directory, including the permissions of each directory and file\n");
    printf(" > arrange <DIR_PATH> [OPTION]...\n");
    printf("   <none> : Arrange the directory if <DIR_PATH> is a directory\n");
    printf("   -d <output_path> : Specify the output directory <output_path> where <DIR_PATH> will be arranged if <DIR_PATH> is a directory\n");
    printf("   -t <seconds> : Only arrange files that were modified more than <seconds> seconds ago\n");
    printf("   -x <exclude_path1, exclude_path2, ...> : Arrange the directory if <DIR_PATH> is a directory except for the files inside <exclude_path> directory\n");
    printf("   -e <extension1, extension2, ...> : Arrange the directory with the specified extension <extension1, extension2, ...>\n");
    printf(" > help [COMMAND]\n");
    printf(" > exit\n");
}

// Exits the program
void cmd_exit(void) {
    exit(0);
}
```

```
// Subroutine for "arrange" command: parses options, traverses directory, handles conflicts, and copies files
void cmd_arrange(char *path)
{
    // Build a temporary argv array: argv[0] is the target directory, followed by optional arguments.
    char *argv_arr[50];
    int argc = 0;
    argv_arr[argc++] = path;
    char resolved[MAX_PATH];
    char *token;
    while ((token = strtok(NULL, " ")) != NULL) {
        argv_arr[argc++] = token;
    }

    Options opts;
    memset(&opts, 0, sizeof(Options));
    parse_options(argc, argv_arr, &opts);

    // Validate the target directory path and check if it is a directory.
    if(!realpath(path, resolved)){
        printf("%s does not exist\n", path);
        free_options(&opts);
        return;
    }
    if(!is_directory(opts.dir_path)) {
        printf("%s is not a directory\n", opts.dir_path);
        free_options(&opts);
        return;
    }
    // Set the default result directory if not provided: <DIR_PATH>_arranged
    char default_result[MAX_FILENAME];
    if (!opts.result_dir) {
        snprintf(default_result, sizeof(default_result), "%s_arranged", opts.dir_path);
        opts.result_dir = default_result;
    }

    // Traverse the directory and group files by extension with conflict grouping.
    ExtGroup *ext_list = NULL;
    traverse_directory(opts.dir_path, &ext_list, &opts);

    // Handle conflicts: interactively resolve files with the same name using diff/vi.
    handle_conflicts(ext_list);

    // Copy the selected files into the result directory.
    copy_selected_files(ext_list, opts.result_dir);

    free_options(&opts);
    // Note: Freeing ext_list and related dynamic memory should be implemented as needed.
```

```
    // complete
    printf("%s arranged\n", path);
}

// Processes the "tree" command to display the directory structure recursively.
void cmd_tree(char *path, int opt_size, int opt_permission) {
    char real_path[MAX_PATH];
    struct stat statbuf;

    // Expand '~' to HOME if needed.
    char expanded_path[MAX_PATH];
    if (expand_tilde(path, expanded_path)) {
        path = expanded_path;
    }

    // Resolve the real path.
    if (!realpath(path, real_path)) {
        //fprintf(stderr, "%s does not exist\n", path);
        fprintf(stderr, "Usage : tree <DIR_PATH> [OPTION]...\n");
        return;
    }
    // Check if the path is inside HOME.
    if (!is_inside_home(real_path)) {
        fprintf(stderr, "%s is outside the home directory\n",path);
        return;
    }
    // Check if the path is a directory.
    if (lstat(real_path, &statbuf) == -1) {
        perror("lstat");
        return;
    }
    if (!S_ISDIR(statbuf.st_mode)) {
        fprintf(stderr, "%s is not a directory.\n", path);
        return;
    }
    // Build the file tree.
    FileNode *root = get_root(real_path, path);
    if (!root) {
        fprintf(stderr, "Error: cannot build tree.\n");
        return;
    }
    // Print the file tree.
    print_tree(root, "", opt_size, opt_permission);

    // Count directories and files (excluding the root) and print counts.
    int dir_count = 0, file_count = 0;
    count_nodes(root, &dir_count, &file_count);
    printf("\n%d directories, %d files\n", dir_count + 1, file_count);
```

```
}

// Expands '~' at the beginning of the input path to the HOME directory.
int expand_tilde(const char *input_path, char *expanded_path) {
    if (input_path[0] == '~') {
        char *home = getenv("HOME");
        if (!home) {
            fprintf(stderr, "Error: HOME not set.\n");
            return 0;
        }
        // If path is "~" or "~/something", prepend home.
        if (input_path[1] == '/' || input_path[1] == '\0') {
            snprintf(expanded_path, MAX_PATH, "%s%s", home, input_path + 1);
            return 1;
        }
    }
    return 0;
}

// Allocates and initializes a new FileNode with the given path and name.
FileNode *create_node(const char *path, const char *name, int is_root, int is_last) {
    FileNode *node = malloc(sizeof(FileNode));
    if (!node) {
        fprintf(stderr, "Error: malloc failed.\n");
        exit(1);
    }
    strncpy(node->name, name, MAX_FILENAME);
    strncpy(node->path, path, MAX_PATH);
    lstat(path, &node->st);
    node->is_dir = S_ISDIR(node->st.st_mode);
    node->child = NULL;
    node->sibling = NULL;
    node->is_root = is_root;
    node->is_last = is_last;
    return node;
}

// Recursively builds the tree by scanning the directory using scandir and sorting entries.
void build_tree(FileNode *parent) {
    if (!parent->is_dir) return;
    struct dirent **namelist = NULL;
    int n = scandir(parent->path, &namelist, NULL, cmp);
    if (n < 0) {
        fprintf(stderr, "opendir failed: %s\n", parent->path);
        return;
    }
    FileNode *first_child = NULL;
    FileNode *prev = NULL;
```

```
    for (int i = 0; i < n; i++) {
        struct dirent *entry = namelist[i];
        if (!entry) continue;
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            free(entry);
            continue;
        }
        if (strlen(entry->d_name) > MAX_FILENAME) {
            fprintf(stderr, "Error: Too long file name.\n");
            free(entry);
            continue;
        }
        char child_path[MAX_PATH];
        int len = snprintf(child_path, sizeof(child_path), "%s/%s", parent->path, entry->d_name);
        if (len < 0 || len >= (int)sizeof(child_path)) {
            fprintf(stderr, "Error: Too long path.\n");
            free(entry);
            continue;
        }
        FileNode *child = create_node(child_path, entry->d_name, 0, 0);
        if (!first_child)
            first_child = child;
        if (prev)
            prev->sibling = child;
        prev = child;
        free(entry);
    }
    free(namelist);
    if (prev)
        prev->is_last = 1;
    parent->child = first_child;
    // Recursively build the tree for each child.
    for (FileNode *c = parent->child; c != NULL; c = c->sibling)
        build_tree(c);
}

// Creates the root FileNode and builds the tree.
FileNode *get_root(const char *abs_path, const char *origin_path) {
    FileNode *root = create_node(abs_path, origin_path, 1, 0);
    build_tree(root);
    return root;
}

// Prints file permissions in the format [drwxr-xr-x].
void print_permissions(mode_t mode) {
    printf("[");
    printf((S_ISDIR(mode)) ? "d" : "-");
    printf((mode & S_IRUSR) ? "r" : "-");
```

```
    printf((mode & S_IWUSR) ? "w" : "-");
    printf((mode & S_IXUSR) ? "x" : "-");
    printf((mode & S_IRGRP) ? "r" : "-");
    printf((mode & S_IWGRP) ? "w" : "-");
    printf((mode & S_IXGRP) ? "x" : "-");
    printf((mode & S_IROTH) ? "r" : "-");
    printf((mode & S_IWOTH) ? "w" : "-");
    printf((mode & S_IXOTH) ? "x" : "-");
    printf("]");
}

// Checks whether the given path is within the user's HOME directory.
int is_inside_home(const char *path) {
    char *home = getenv("HOME");
    if (!home) {
        fprintf(stderr, "Error: HOME not set.\n");
        return 0;
    }
    return strncmp(path, home, strlen(home)) == 0;
}

// Recursively prints the tree structure with optional file size and permissions.
void print_tree(FileNode *node, const char *prefix, int opt_size, int opt_permission) {
    if (!node) return;
    if (node->is_root) {
        // Print the root exactly as given by the user.
        printf("%s\n", node->name);
    } else {
        printf("%s", prefix);
        printf("%s", node->is_last ? " └── " : " ├── ");
        if (opt_permission) {
            print_permissions(node->st.st_mode);
            printf(" ");
        }
        if (opt_size) {
            printf("[%ld] ", node->st.st_size);
        }
        printf("%s", node->name);
        if (node->is_dir)
            printf("/");
        printf("\n");
    }
    char new_prefix[MAX_PATH];
    if (node->is_root)
        strcpy(new_prefix, "");
    else {
        strcpy(new_prefix, prefix);
        if (node->is_last)
```

```
            strcat(new_prefix, "    ");
        else
            strcat(new_prefix, " |    ");
    }
    for (FileNode *child = node->child; child != NULL; child = child->sibling)
        print_tree(child, new_prefix, opt_size, opt_permission);
}

// Recursively counts directories and files (excluding the root node).
void count_nodes(FileNode *node, int *dir_count, int *file_count) {
    if (!node) return;
    if (!node->is_root) {
        if (node->is_dir)
            (*dir_count)++;
        else
            (*file_count)++;
    }
    for (FileNode *c = node->child; c; c = c->sibling)
        count_nodes(c, dir_count, file_count);
}

// Custom comparator for scandir to sort directory entries alphabetically.
int cmp(const struct dirent **a, const struct dirent **b) {
    return strcmp((*a)->d_name, (*b)->d_name);
}

// Parses the command-line-like arguments for the arrange command.
// argv[0] is the target directory; subsequent tokens are options (-d, -x, -e, -t).
void parse_options(int argc, char *argv[], Options *opts) {
    opts->dir_path = argv[0];
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-d") == 0 && i + 1 < argc) {
            opts->result_dir = argv[++i];
        } else if (strcmp(argv[i], "-x") == 0 && i + 1 < argc) {
            char *token = strtok(argv[++i], ",");
            while (token) {
                opts->exclude_dirs = realloc(opts->exclude_dirs, sizeof(char*) * (opts->exclude_count + 1));
                opts->exclude_dirs[opts->exclude_count] = strdup(token);
                opts->exclude_count++;
                token = strtok(NULL, ",");
            }
        } else if (strcmp(argv[i], "-e") == 0 && i + 1 < argc) {
            char *token = strtok(argv[++i], ",");
            while (token) {
                opts->extensions = realloc(opts->extensions, sizeof(char*) * (opts->ext_count + 1));
                opts->extensions[opts->ext_count] = strdup(token);
                opts->ext_count++;
                token = strtok(NULL, ",");
```

```
            }
        } else if (strcmp(argv[i], "-t") == 0 && i + 1 < argc) {
            opts->time_limit = atoi(argv[++i]);
        } else {
            fprintf(stderr, "Unknown option or missing argument: %s\n", argv[i]);
        }
    }
}

// Frees any dynamically allocated memory in the Options structure.
void free_options(Options *opts) {
    if (opts->exclude_dirs) {
        for (int i = 0; i < opts->exclude_count; i++) {
            free(opts->exclude_dirs[i]);
        }
        free(opts->exclude_dirs);
        opts->exclude_dirs = NULL;
    }
    if (opts->extensions) {
        for (int i = 0; i < opts->ext_count; i++) {
            free(opts->extensions[i]);
        }
        free(opts->extensions);
        opts->extensions = NULL;
    }
}

// Creates a new ExtGroup node for a given file extension.
ExtGroup *create_ext_group(const char *extension) {
    ExtGroup *group = (ExtGroup *)calloc(1, sizeof(ExtGroup));
    if (!group) return NULL;
    group->extension = strdup(extension);
    group->conflicts = NULL;
    group->next = NULL;
    return group;
}

// Creates a new ConflictGroup node for a given filename.
ConflictGroup *create_conflict_group(const char *filename) {
    ConflictGroup *cg = (ConflictGroup *)calloc(1, sizeof(ConflictGroup));
    if (!cg) return NULL;
    cg->filename = strdup(filename);
    cg->count = 0;
    cg->capacity = 0;
    cg->files = NULL;
    cg->next = NULL;
    return cg;
}
```

```
// Searches for an ExtGroup with the specified extension.
ExtGroup *find_ext_group(ExtGroup *head, const char *extension) {
    ExtGroup *cur = head;
    while (cur) {
        if (strcmp(cur->extension, extension) == 0)
            return cur;
        cur = cur->next;
    }
    return NULL;
}

// Searches for a ConflictGroup with the specified filename.
ConflictGroup *find_conflict_group(ConflictGroup *head, const char *filename) {
    ConflictGroup *cur = head;
    while (cur) {
        if (strcmp(cur->filename, filename) == 0)
            return cur;
        cur = cur->next;
    }
    return NULL;
}

// Adds a file (with its path, name, and modification time) to a ConflictGroup.
static void add_file_to_conflict_group(ConflictGroup *cg, const char *filepath, const char *filename, time_t
mtime) {
    if (cg->count == cg->capacity) {
        int new_cap = (cg->capacity == 0) ? 4 : (cg->capacity * 2);
        cg->files = realloc(cg->files, sizeof(ConflictFile*) * new_cap);
        cg->capacity = new_cap;
    }
    ConflictFile *cf = (ConflictFile *)malloc(sizeof(ConflictFile));
    cf->filepath = strdup(filepath);
    cf->filename = strdup(filename);
    cf->mtime = mtime;
    cg->files[cg->count++] = cf;
}

// Adds a file into an ExtGroup based on its extension and organizes conflicts by filename.
void add_file_to_groups(ExtGroup **ext_list, const char *filepath, const char *filename, const char *extension,
time_t mtime) {
    ExtGroup *eg = find_ext_group(*ext_list, extension);
    if (!eg) {
        eg = create_ext_group(extension);
        eg->next = *ext_list;
        *ext_list = eg;
    }
    ConflictGroup *cg = find_conflict_group(eg->conflicts, filename);
```

```
    if (!cg) {
        cg = create_conflict_group(filename);
        cg->next = eg->conflicts;
        eg->conflicts = cg;
    }
    add_file_to_conflict_group(cg, filepath, filename, mtime);
}

// Recursively traverses the directory tree starting at 'path' and groups files by extension according to Options.
void traverse_directory(const char *path, ExtGroup **ext_list, Options *opts) {
    DIR *dir = opendir(path);
    if (!dir) {
        fprintf(stderr, "opendir error: %s\n", path);
        return;
    }
    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
            continue;
        char *fullpath = concat_path(path, entry->d_name);
        if (!fullpath) {
            fprintf(stderr, "concat_path error\n");
            continue;
        }
        struct stat st;
        if (stat(fullpath, &st) < 0) {
            free(fullpath);
            continue;
        }
        if (S_ISDIR(st.st_mode)) {
            if (!should_skip_dir(entry->d_name, opts))
                traverse_directory(fullpath, ext_list, opts);
        } else if (S_ISREG(st.st_mode)) {
            if (opts->time_limit > 0) {
                time_t now = time(NULL);
                if (difftime(now, st.st_mtime) > opts->time_limit) {
                    free(fullpath);
                    continue;
                }
            }
            char *ext = get_extension(entry->d_name);
            if (opts->ext_count > 0) {
                if (!is_valid_extension(ext, opts)) {
                    free(fullpath);
                    continue;
                }
            }
```

```
            add_file_to_groups(ext_list, fullpath, entry->d_name, ext, st.st_mtime);
        }
        free(fullpath);
    }
    closedir(dir);
}

// Checks if a directory should be skipped based on the -x option.
int should_skip_dir(const char *dirname, Options *opts) {
    for (int i = 0; i < opts->exclude_count; i++) {
        if (strcmp(dirname, opts->exclude_dirs[i]) == 0)
            return 1;
    }
    return 0;
}

// Checks if the file's extension is allowed based on the -e option.
int is_valid_extension(const char *ext, Options *opts) {
    for (int i = 0; i < opts->ext_count; i++) {
        if (strcmp(ext, opts->extensions[i]) == 0)
            return 1;
    }
    return 0;
}

// Returns the file extension (substring after the last '.'); returns empty string if none.
char *get_extension(const char *filename) {
    const char *dot = strrchr(filename, '.');
    if (!dot || dot == filename)
        return "";
    return (char *)(dot + 1);
}

// Concatenates a directory and file name into a full path; returns a newly allocated string.
char *concat_path(const char *dir, const char *file) {
    char buf[MAX_PATH];
    if (snprintf(buf, sizeof(buf), "%s/%s", dir, file) >= (int)sizeof(buf))
        return NULL;
    return strdup(buf);
}

// Copies a file from src to dst using low-level I/O; returns 0 on success, -1 on error.
int copy_file(const char *src, const char *dst) {
    int fd_src = open(src, O_RDONLY);
    if (fd_src < 0) {
        perror("open src");
        return -1;
    }
```

```
    int fd_dst = open(dst, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd_dst < 0) {
        perror("open dst");
        close(fd_src);
        return -1;
    }
    char buffer[4096];
    ssize_t nread;
    while ((nread = read(fd_src, buffer, sizeof(buffer))) > 0) {
        write(fd_dst, buffer, nread);
    }
    close(fd_src);
    close(fd_dst);
    return 0;
}

// Handles conflicts when multiple files with the same name exist.
// For each conflict group, prompts the user to choose an option:
// 1) select [num]: keep only the specified file
// 2) diff [num1] [num2]: compare two files using diff (fork/execvp)
// 3) vi [num]: open the file in vi (fork/execvp)
// 4) do not select: ignore all files in this group
void handle_conflicts(ExtGroup *ext_list) {
    ExtGroup *eg = ext_list;
    while (eg) {
        ConflictGroup *cg = eg->conflicts;
        while (cg) {
            if (cg->count > 1) {
                // Print each conflicting file with numbering starting at 1.
                for (int i = 0; i < cg->count; i++) {
                    printf("%d. %s\n", i + 1, cg->files[i]->filepath);
                }
                int resolved = 0;
                while (!resolved) {
                    printf("\nchoose an option:\n");
                    printf("0. select [num]\n");
                    printf("1. diff [num1] [num2]\n");
                    printf("2. vi [num]\n");
                    printf("3. do not select\n");
                    printf("20211407> ");
                    char input[128];
                    if (!fgets(input, sizeof(input), stdin)) {
                        break;
                    }
                    char cmd[16];
                    int num = 0, num2 = 0;
                    cmd[0] = '\0';
                    int tokens = sscanf(input, "%s %d %d", cmd, &num, &num2);
```

```
                if (strcmp(cmd, "select") == 0 && tokens >= 2) {
                    if (num > 0 && num <= cg->count) {
                        for (int i = 0; i < cg->count; i++) {
                            if (i != (num - 1)) {
                                free(cg->files[i]->filepath);
                                free(cg->files[i]->filename);
                                cg->files[i]->filepath = NULL;
                                cg->files[i]->filename = NULL;
                            }
                        }
                        resolved = 1;
                    } else {
                        printf("Invalid file number.\n");
                    }
                }
                else if (strcmp(cmd, "diff") == 0 && tokens >= 3) {
                    if (num > 0 && num2 > 0 && num <= cg->count && num2 <= cg->count) {
                        pid_t pid = fork();
                        if (pid == 0) {
                            char *args[] = {"diff", cg->files[num - 1]->filepath, cg->files[num2 - 1]->filepath,
NULL};

                            execvp("diff", args);
                            perror("execvp diff");
                            exit(EXIT_FAILURE);
                        } else if (pid > 0) {
                            int status;
                            waitpid(pid, &status, 0);
                        } else {
                            perror("fork");
                        }
                    } else {
                        printf("Invalid file number(s) for diff.\n");
                    }
                }
                else if (strcmp(cmd, "vi") == 0 && tokens >= 2) {
                    if (num > 0 && num <= cg->count) {
                        pid_t pid = fork();
                        if (pid == 0) {
                            char *args[] = {"vi", cg->files[num - 1]->filepath, NULL};
                            execvp("vi", args);
                            perror("execvp vi");
                            exit(EXIT_FAILURE);
                        } else if (pid > 0) {
                            int status;
                            waitpid(pid, &status, 0);
                        } else {
                            perror("fork");
                        }
```

```
                        } else {
                            printf("Invalid file number for vi.\n");
                        }
                    }
                    else if (strcmp(cmd, "do") == 0 || strcmp(cmd, "do not") == 0) {
                        resolved = 1;
                        for (int i = 0; i < cg->count; i++) {
                            free(cg->files[i]->filepath);
                            free(cg->files[i]->filename);
                            cg->files[i]->filepath = NULL;
                            cg->files[i]->filename = NULL;
                        }
                    }
                    else {
                        printf("Invalid command.\n");
                    }
                }
            }
            cg = cg->next;
        }
        eg = eg->next;
    }
}

// Creates the result directory (and subdirectories for each extension) and copies selected files.
void copy_selected_files(ExtGroup *ext_list, const char *dest_dir) {
    if (mkdir(dest_dir, 0755) < 0) {
        if (errno != EEXIST) {
            perror("mkdir dest_dir");
            return;
        }
    }
    ExtGroup *eg = ext_list;
    while (eg) {
        char ext_dir[MAX_PATH];
        snprintf(ext_dir, sizeof(ext_dir), "%s/%s", dest_dir, eg->extension);
        if (mkdir(ext_dir, 0755) < 0) {
            if (errno != EEXIST) {
                perror("mkdir ext_dir");
                eg = eg->next;
                continue;
            }
        }
        ConflictGroup *cg = eg->conflicts;
        while (cg) {
            for (int i = 0; i < cg->count; i++) {
                ConflictFile *cf = cg->files[i];
                if (!cf->filepath)
```

```
                    continue;
                char *dst_path = concat_path(ext_dir, cf->filename);
                if (!dst_path)
                    continue;
                copy_file(cf->filepath, dst_path);
                free(dst_path);
            }
            cg = cg->next;
        }
        eg = eg->next;
    }
}

// Checks whether the given path is valid (e.g., not too long).
int validate_path(const char *path) {
    if (!path) return -1;
    if (strlen(path) >= MAX_PATH) {
        fprintf(stderr, "Error: path is too long.\n");
        return -1;
    }
    return 0;
}

// Checks if the given path refers to a directory.
int is_directory(const char *path) {
    struct stat st;
    if (stat(path, &st) < 0) {
        perror("stat");
        return 0;
    }
    return S_ISDIR(st.st_mode);
}
```