How you could've designed `ls` yourself

An interactive guide to building your own `ls` in C

41247044S

Contents

1 Introduction

You'll find explanations and C code snippets. This is a rudimentary, simple and barebones start to an `ls` implementation

2 Building from scratch → listing files

Alright, so first up, if you just type `ls` without anything else, it should show you what's in the current folder. If you give it a path to a file, it should just print that file's name. If it's a folder, it should show what's inside

To start, we'll use a few tools from the C library that are POSIX compliant:
We use POSIX functions:

- `stat()` - to detect file vs directory
- `stat()` - helps us figure out if something is a file or a folder
- `opendir()` - to peek into a directory
- `readdir()` - reads out the names of things (files and other folders) inside the directory
- `closedir()` - like closing a file, so closes the directory when we're done

Here's what it looks like now:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>

void list_directory(const char *path) {
    DIR *dir = opendir(path);
    if (!dir) {
        perror("Oops, couldn't open directory");
        return;
    }

    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
            continue;
        if (entry->d_name[0] == '.') continue;
        printf("%s\\n", entry->d_name);
    }

    closedir(dir);
}

int main(int argc, char *argv[]) {
    const char *path = (argc > 1) ? argv[1] : ".";
    struct stat st;
    if (stat(path, &st) == 0 && !S_ISDIR(st.st_mode)) {
        printf("%s\\n", path);
    } else {
        list_directory(path);
    }
    return 0;
}
```

So, this little program opens the folder you tell it to (or the current one), reads everything inside, skips `.` and `..` (those just mean "this folder" and "the parent folder"), and also skips hidden files for now. Then it just prints out the names. No sorting or anything fancy yet just plain, unordered file names.

3 Handling Options with `getopt`

You know how `ls` can do different things with options like `-l` (long format), `-a` (show all), or `-R` (recursive)? We need a way to understand those. The C library has a function called `getopt()` to accomplish this.

```c

#include <unistd.h>   // getopt lives here
#include <stdbool.h>  // So we can use true/false

bool show_long_format = false;
bool show_all_files = false;
bool go_recursive = false;

// ... stick this inside your main function, before you decide the path_to_look_at
    ...
int opt;
// The "alR" string tells getopt which letters are valid options
while ((opt = getopt(argc, argv, "alR")) != -1) {
    switch (opt) {
        case 'a':
            show_all_files = true;
            break;
        case 'l':
            show_long_format = true; // We'll use this flag later
            break;
        case 'R':
            go_recursive = true;
            break;
        default: /* '?' */
            // getopt already printed an error message
            fprintf(stderr, "Try this: %s [-alR] [file...]\n", argv[0]);
            exit(1); // Exit with an error code
    }
}
// After this loop, 'optind' (another thing from getopt) tells us
// where the actual file/directory names start in argv.
```

Ok, Now we have some `true/false` flags (`show_all_files`, `show_long_format`, `go_recursive`) that we can check later to make our `ls` do different tricks and stuff.

4 Sorting

Usually, `ls` shows files in alphabetical order. To do that, we can't just print filenames as soon as we find them. We gotta:

1. Read all the filenames we care about into a list (an array that can grow)
2. Sort that list
3. Then print everything from the sorted list

We'll use `realloc` to make our array bigger as needed, and `qsort` to do the actual sorting. To sort text properly, especially if you have weird characters or different languages, `strcoll` is a good fit

```c

#include <stdlib.h>
#include <string.h>
#include <locale.h> // For strcoll (fancy string comparison)

// ... inside your list_directory function ...
char **names_list = NULL; // Our list of names
size_t names_count = 0;   // How many names are in the list

// Make sure setlocale is called once, maybe at the start of main()
// setlocale(LC_COLLATE, ""); // This tells strcoll to use the system's sorting rules

while ((entry = readdir(dir)) != NULL) {
    // Your filtering logic for '.', '..', and hidden files (based on show_all_files) goes here
    if ((!show_all_files && entry->d_name[0] == '.') ||
        strcmp(entry->d_name, ".") == 0 ||
        strcmp(entry->d_name, "..") == 0) {
        // If we're not showing all, and it's hidden, skip.
        // Always skip . and .. for the main listing (unless you *really* want them for -a)
        // The behavior of -a with . and .. can be a bit specific,
        // for now, let's just focus on user files.
        if (entry->d_name[0] == '.' && !show_all_files) continue;
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) continue; // Generally skip these
    }

    // Add the name to our list
    names_list = realloc(names_list, (names_count + 1) * sizeof(char*));
    if (!names_list) { perror("realloc failed, oh no!"); exit(1); }
    names_list[names_count++] = strdup(entry->d_name); // strdup makes a copy, remember to free it!
}

// This is how qsort knows how to compare our strings
int compare_names(const void *a, const void *b) {
    // We need to cast 'a' and 'b' back to what they really are: pointers to char pointers
    return strcoll(*(const char **)a, *(const char **)b);
}

qsort(names_list, names_count, sizeof(char*), compare_names);

// Now print 'em and clean up
for (size_t i = 0; i < names_count; i++) {
    printf("%s\n", names_list[i]);
    free(names_list[i]); // Free the copy strdup made
}
free(names_list); // Free the list itself
// ...
```

5 The Long Format (-l)

The `-l` option gives you that detailed, multi-column view: permissions, owner, size, date, etc. This means we need to dig up more info about each file using `stat()` (or `lstat()` if we want info about a symbolic link itself, not what it points to). For each file, we're interested in:

- File type and permissions: We get this from a field in `stat` called `st_mode`

- Number of hard links: `st_nlink`
- Owner name: `st_uid` (user ID), which we can turn into a name using `getpwuid()`
- Group name: `st_gid` (group ID), turned into a name with `getgrgid()`
- File size: `st_size`
- Modification time: `st_mtime`, which we can make human-readable with `strftime()`
- Filename: And if it's a symlink, we use `readlink()` to show what it's pointing to

Here's a rough idea of a `print_in_long_format` function:

```c

#include <sys/stat.h>
#include <pwd.h>      // For getpwuid (user ID to name)
#include <grp.h>      // For getgrgid (group ID to name)
#include <time.h>     // For formatting dates
#include <unistd.h>   // For readlink

void print_in_long_format(const char *directory_path, const char *filename) {
    char full_item_path[1024]; // Careful with fixed buffer sizes!
                               // A safer way is to calculate needed size and
                                   malloc.
    snprintf(full_item_path, sizeof(full_item_path), "%s/%s", directory_path,
        filename);

    struct stat file_stats;
    if (lstat(full_item_path, &file_stats) != 0) { // lstat doesn't follow symlinks
        perror("lstat failed");
        return;
    }

    // 1. Figure out file type (d, l, -, etc.) and permissions (rwx)
    //    Example: (S_ISDIR(file_stats.st_mode) ? 'd' : '-')
    //    Example: (file_stats.st_mode & S_IRUSR ? 'r' : '-') ... and so on

    // 2. Print link count (file_stats.st_nlink)

    // 3. Get and print owner name
    //    struct passwd *pwd = getpwuid(file_stats.st_uid);
    //    printf("%s ", (pwd ? pwd->pw_name : "unknown_user"));

    // 4. Get and print group name
    //    struct group *grp = getgrgid(file_stats.st_gid);
    //    printf("%s ", (grp ? grp->gr_name : "unknown_group"));

    // 5. Print size (file_stats.st_size)

    // 6. Format and print modification time
    //    char time_buffer[100];
    //    strftime(time_buffer, sizeof(time_buffer), "%b %d %H:%M", localtime(&
        file_stats.st_mtime));
    //    printf("%s ", time_buffer);

    // 7. Print the filename
    //    printf("%s", filename);

    // 8. If it's a symlink (S_ISLNK(file_stats.st_mode)), show target
    //    char link_target[1024];
    //    ssize_t len = readlink(full_item_path, link_target, sizeof(link_target)
        -1);
    //    if (len != -1) {
    //        link_target[len] = '\0';
    //        printf(" -> %s", link_target);
    //    }

    printf("\n"); // Each entry on a new line
}
```

You'd call this function from your main loop if `show_long_format` is `true`, instead of the simple `printf("%s\n", names_list[i])`.

6 Recursive Listing (-R)

To make `-R` work (listing subdirectories too), our `list_directory` function (or a new version of it) needs to be brave enough to call itself whenever it finds another directory.

```c

// Let's imagine a function signature like this:
void list_contents(const char *current_dir_path, bool use_long_format, bool
    show_hidden, bool be_recursive) {
    // ... (open directory, read entries into a list, sort them, as before) ...

    printf("\n%s:\n", current_dir_path); // Show which directory we're listing

    // Loop through your sorted names_list
    // size_t names_count; // Assume this is set after reading/sorting
    // char **names_list; // Assume this is populated
    for (size_t i = 0; i < names_count; i++) {
        if (use_long_format) {
            // print_in_long_format(current_dir_path, names_list[i]);
        } else {
            // printf("%s\n", names_list[i]);
        }
    }

    if (be_recursive) {
        for (size_t i = 0; i < names_count; i++) {
            char item_path_buffer[1024];
            snprintf(item_path_buffer, sizeof(item_path_buffer), "%s/%s",
                current_dir_path, names_list[i]);
            struct stat item_stats;

            if (lstat(item_path_buffer, &item_stats) == 0 && S_ISDIR(item_stats.
                st_mode)) {
                // Super important: Make sure it's not '.' or '..' to avoid
                    infinite loops!
                if (strcmp(names_list[i], ".") != 0 && strcmp(names_list[i], "..")
                    != 0) {
                    // Dive in!
                    list_contents(item_path_buffer, use_long_format, show_hidden,
                        be_recursive);
                }
            }
        }
    }

    // ... (free your names_list and its contents, close the directory) ...
}
```

a quick note here is that a real `ls` is smart enough to not get stuck in infinite loops if you have symbolic links pointing back to parent directories. Our simple version here isn't that clever, but it's a big thing to keep in mind for a production tool.

7 Revealing Hidden Files (-a)

simple, but an easier implement than what we just did. Here, we already decided to skip files starting with . earlier. Now, we just make that decision based on our `show_all_files` flag.

```c

// Inside your readdir loop, when deciding whether to add 'entry->d_name' to your
    list:
// bool show_all_files; // Assume this is set
// struct dirent *entry; // Assume this is current entry
bool is_hidden = (entry->d_name[0] == '.');
bool is_dot_or_dotdot = (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "
    ..") == 0);

if (is_dot_or_dotdot) {
    // POSIX `ls -a` *does* show . and ..
    // If you want to strictly follow that:
    if (show_all_files) {
        // Add to list
    } else {
        // continue; // Skip . and .. if not -a
    }
} else if (is_hidden && !show_all_files) {
    // continue; // Skip other hidden files if not -a
} else {
    // Add to list
}
```

The exact rules for . and .. with -a vs -A (another common option) can be a bit finicky. For a basic -a, showing everything starting with . (including . and ..) is a good start.

8 Coloring

Colors make `ls` output way easier to read (directories in blue, executables in green, etc.). This isn't a POSIX must-have, but it's a nice touch. We can do it by printing special "ANSI escape codes" before and after filenames. But, we should only do this if the output is actually going to a terminal screen. If someone's piping your `ls` output to another program or a file, those color codes will just look like garbage. We use `isatty()` to check if we're talking to a terminal.

```c

#include <unistd.h> // For isatty
// #include <sys/stat.h> // For S_ISDIR, S_IXUSR

// ... when you're about to print a filename ...
// Assume you've already done a stat or lstat to get 'file_stats' for the item
// struct stat file_stats; // Assume this is populated
// const char *filename_to_print; // Assume this is set
// bool show_long_format; // Or some other flag to enable colors

bool output_is_to_terminal = isatty(STDOUT_FILENO);

if (output_is_to_terminal && (show_long_format /* or some other color flag */)) {
    if (S_ISDIR(file_stats.st_mode)) {
        printf("\x1b[1;34m"); // Bold Blue for directories
    } else if (file_stats.st_mode & S_IXUSR) { // Check if it's executable by user
        printf("\x1b[1;32m"); // Bold Green for executables
    }
    // Add more else ifs for other types: symlinks, etc.
}

printf("%s", filename_to_print); // The actual filename
```

```c

if (output_is_to_terminal && (show_long_format /* or some other color flag */)) {
    printf("\x1b[0m"); // Reset color back to normal
}
// Then print your newline if needed
printf("\n");
```

9 Wrapping Up

Further features

We've hit the main points, but a full implement of `ls` (like the one that comes with Linux or macOS) has tons more features, such as,

- JSON or XML Output: useful if you want other programs to easily use your `ls` output
- POSIX-Compliance: Try to use POSIX functions so your `ls` works on different Unix-like systems. Also, `ls` often pays attention to things like the `LC_COLLATE` environment variable for sorting.

Building your own `ls` is a great C project: from basic listing to options, sorting, details, recursion, hidden files, and colors.