Assignment A1: Intro C

Introduction

For this assignment, you will be writing two command-line utilities: one that prints the number of files with a specific set of permissions that exceed a specified size and one that verifies Canadian Social Insurance Numbers (SINs). Both programs will require that you use command-line arguments and arrays. The first will also require that you process standard input (using `scanf`).

Please remember that we are using testing scripts to evaluate your work. As a result, it's very important that (a) all of your files are named correctly and located in the specified locations in your repository and (b) the output of your programs match the expected output precisely.

Part 0: Getting Started (0%)

Follow the instructions carefully, so that we receive your work correctly.

Your first step should be to log into MarkUs and navigate to the **a1: Intro C** assignment. Like for the labs, this triggers the starter code for this assignment to be committed to your repository. You will be working alone on this assignment, and the URL for your Git repository can be found on the right hand side of the page.

Pull your git repository. There should be a new folder named `a1`. All of your code for this assignment should be located in this folder. Starter code, including a `Makefile` that will compile your code and provide a sanity check, has already been placed in this folder. To use the `Makefile`, type "*make test_part1*" or "*make test_part2*". That will compile the corresponding program and run our sanity checks. To remove .o files and the executable, type "*make clean*".

Once you have pushed files to your repository, you can use MarkUs to verify that what you intended to submit was actually submitted. The Submissions tab for the assignment will show you what is in the repository, and will let you know if you named the files correctly.

Part 1: `count_large.c` (3%)

Your first task is to write a C program called `count_large`. The program will inspect the output from the `ls -l` command and print the number of files larger than a specified size.

Details

From standard input, your program reads input in the same format as produced by the `ls` program with the `-l` option when run on the teaching lab machines.

Your program prints to standard output, the number of regular files in the the current working directory that are larger than the cutoff size supplied by the user in the first command-line argument. Directories are not considered to be regular files.

Your program optionally accepts a second command-line argument which is a string of 9 characters that represent the required file permissions. Each position is either `r`,`w`,x or - as would be appropriate in that location in the file-permission field of the long format of `ls` output. If called with this option, your program will only count the files that have at least the permissions specified by this argument.

For example, given the output of `ls -l`:

```
total 329
-rwx------  1 reid  staff    1734 Jun 22 14:52 prog
-rw-------  1 reid  staff   21510 Apr  6 12:10 tmp.txt
-rwxr-xr-x  1 reid  staff    8968 Feb  1 2013 xyz
-rw-r--r--  1 reid  staff      88 Feb 15 2013 xyz.c
```

Running `count_large 1000 rwx------`

will print `2` (followed by a single newline `\n` character) because `prog` and `xyz` are both larger than 1000 bytes and have at least the permissions specified.

Running `count_large 1000` will print `3` (followed by a single newline `\n` character) because there are three files larger than 1000 bytes. Do not print any extra text - just the number.

Notice that the first line of output from running `ls -l` is the total line. Your program should expect this line and handle it appropriately. That is, you should read and then ignore it.

Requirements

You are required to write (and use) a helper function named `check_permissions` that takes two 9-element character arrays as arguments and returns an integer. (The prototype for this function is in the starter code.) The first array will represent the a permission field of a file and the second will represent the permissions that are required. The function will return `0` if the file has all the required permissions and `1` otherwise. The arrays do not have to be identical for a `0` to be returned. For example, if the first array holds the characters `rwxr-x---`and the second array holds the characters `r-x------`, the function should return `0`. Often when we pass arrays as parameters to functions, we also pass a size so that the function can work on arrays of varying lengths. In this case we don't need to pass a size because both arrays will always have nine elements.

Assumptions You May Make

For this assignment only, you may assume the following:

1. The user will provide at most two command-line arguments. If provided, arguments will be the correct type and in the correct order. In other words, if only one command-line argument is present, it is an integer. If two are provided, the first is an integer and the second is the required permissions string.
2. The input listing contains only regular files and directories. (There are no links or other unusual file types.)
3. Names of things (filenames, usernames, groupnames, ...) will be at most 31 characters.
4. An integer is large enough to hold any file sizes for this assignment.
5. Filenames do not have spaces in them.

In general these are not valid assumptions -- either in the real world or in later CSC209 assignments -- but making them puts the focus of this assignment on basic C syntax and use of arrays, rather than command line parsing.

Command-line arguments and return codes

Assumption 1 above is particularly unrealistic for real C programs. We would normally want to have many different options and would design our programs to behave like other C tools where options can be specified in any order and grouped together. In the future, you will learn about the `getopt` library that makes processing command-line arguments easy. But for now, you are not expected to use `getopt`. Feel free to try it on your own, but for this assignment, you are not expected to use it.

For this program, the user might still forget command-line arguments altogether. When this happens, your program shouldn't crash. Instead you should print to standard error the message "`USAGE: count_large size [permissions]`" (followed by a single newline `\n` character), and return from `main` with return code `1`. The starter code actually implements this behaviour for you already. Following the standard C conventions, `main` should return `0` when the program runs successfully.

Strong Suggestions

You should use `scanf` to read the input. By the time the assignment is submitted, you will have learned more about strings in C and about tokenizing strings. However, the assignment is designed specifically so that you can do the input parsing with the C functions that you've learned in the first two weeks of class.

Don't try to save the entire listing in memory. That isn't necessary and is a very poor design. It will also make the program much harder to write.

Since your program reads from standard input, it would be possible to type all the input to your program from the keyboard. But typing in a listing that looks like the output from `ls -l` even once would be awful. Instead you could run `ls -l` and **pipe** its output to your program. Make sure you understand that last sentence and try it out. Even better, you should run `ls -l` and **redirect** its output to another file. Then edit that file to create different versions that thoroughly test your program. Now run your program and **redirect** standard input to read from one of the test input files that you just created.

Reminder

Your program must compile on `teach.cs` using `gcc` with the `-Wall` option and should not produce any error messages. Programs that do not compile, will get 0. You can still get part marks by submitting something that doesn't completely work but does some of the job -- but it **must at least compile** to get any marks at all. Also check that your output messages are **exactly** as specified in this handout.

Part 2: `validate_sin.c` and `sin_helpers.c` (2%)

You will write a C program called `validate_sin.c` that could be used to check whether a Canadian Social Insurance Number (SIN) is valid.

Details

Your program reads a single command line argument representing a candidate SIN number. Your program then prints one of two messages (followed by a single newline `\n`character) to standard output : "`Valid SIN`", if the value given is a valid Canadian SIN number, or "`Invalid SIN`", if it is invalid.

Requirements

For this assignment, you are required to write (and use) two helper functions. (These helper functions are to be implemented in the file `sin_helpers.c`. The main function is in `validate_sin.c`.) The first helper function is named `populate_array`. It takes an integer and an integer array as its arguments, and returns an integer. This function's job is to populate the given integer array so that it contains the 9 digits of the given integer, in the same order as in the integer. Hint: use `% 10` and `/ 10` to calculate the digits. The function must return `0` when it completes successfully, and `1` if the given integer is not 9 digits.

The second helper function is named `check_sin` and it takes an 9 element integer array representing a candidate SIN number. It returns `0` if the number given is a valid Canadian SIN number, and `1` otherwise.

Your function should implement the Luhn algorithm (a process for validating Canadian SIN numbers) to perform this task. The Luhn algorithm is outlined in this Wikipedia page. Here is an example of how to validate a candidate SIN number:

```
810620716 <--- A candidate SIN number
121212121 <--- Multiply each digit in the top number by the digit below it.


If the product is a one-digit number, put it in the corresponding position of
the result.
If the product is a two-digit number, add the digits together and put the sum
in the
corresponding position of the result. For example, in the fourth column, the
product of 6
multiplied by 2 is 12. Add the digits together (1 + 2 = 3) and put 3 in the
fourth column of result.

The result is:
820320726

Next, sum the digits of the result to produce a total:
8+2+0+3+2+0+7+2+6=30

If the SIN is valid, the total will be divisible by 10.
```

Note: as a simplification, we will only consider numbers that begin with a non-zero digit to be valid. Therefore, numbers such as 046454286 and 000000026 are considered invalid.

Command line arguments and return codes

For this program, you may assume that if exactly one command-line argument is provided, it will have the correct format of a candiate SIN number. If the user calls the program with too few or too many arguments, the program should not print anything, but should return from `main` with return code `1`. Following the standard C conventions, `main` should return `0`when the program runs successfully.

Reminder

Your program must compile on `teach.cs` using `gcc` with the `-std=gnu99 -Wall` options and should not produce any error messages. Programs that do not compile, will get 0. You can still get part marks by submitting something that doesn't completely work but does some of the job -- but it **must at least compile** to get any marks at all. Also check that your output messages are **exactly** as specified in this handout.

Submission

We will be looking for the following files in the a1 directory of your repository:

- `count_large.c`
- `sin_helpers.c`
- `validate_sin.c`

Do not commit `.o` files or executables to your repository.