CS 376 Lab 11: Scripting Task

In this lab, you will write a script that performs a "real-world" task. It will require you draw upon the knowledge that you have acquired during the labs throughout the semester.

This lab has a total of 100 possible points.

Note: for the purposes of this lab, 'whitespace' means "spaces and tabs".
Use Linux VM for this lab.

Objectives

Upon completion of the laboratory exercise, you will have gained experience using a variety of shell and Linux utilities to perform a task. These will likely include:

- Shell variables, conditionals and loops
- Pipes
- Return status of commands
- Unix commands such as `grep`, `sed`, `echo`, `ls`, `cd` and `tr`.

Background: Repeated inclusion of a .h file

It is not uncommon for one .h file to #include another one. For example, one .h file may require some type definitions or symbolic constants that are defined in another. This gives rise to the possibility that one .h file might be #included more than once; this is a problem because this will almost certainly produce "multiply defined symbol" errors.

Consider the following example:

- `definitions.h` defines a symbol, `SIZE` (e.g., `#define SIZE 100`)
- `test1.h` includes `definitions.h`
- `main.c` includes both `test1.h` and `definitions.h`
- The author (aka developer) of `main.c` does not know whether `test1.h` includes `definitions.h`. We'd like the program to compile correctly either way. (The author/developer of `test1.h` might change their mind about whether to include `definitions.h`.)

In the above scenario, the following will occur when `main.c` is compiled:

- `test1.h` will be included
  - Because `test1.h` includes `definitions.h`, this will cause `definitions.h` to be included. Therefore the symbol `SIZE` will be defined by the time the compiler has finished processing the include of `test1.h` in `main.c`.
- `definitions.h` will be included (this time, directly from `main.c`)
  - When the definition of `SIZE` is encountered, a "multiply-defined name" error will be flagged; this will cause the compilation to abort.

To prevent this from happening, the following include file convention has been adopted. Every #include file should be "wrapped" in a *header guard*, as in:

```
#ifndef TEST1_H
#define TEST1_H
#define SIZE 100
#endif
```

(The convention is that the symbol be the name of the file with all characters uppercased, and any character that is not a digit or a letter is replaced with a '_'.)

A header guard prevents the body of the include file from being processed more than once, because the second time it is included, the symbol will be defined—thereby preventing the main part of the file from being processed. The header guard's symbol is derived from the file name so that two headers do not accidentally use the same guard.

In the example above, the effect would be that when `definitions.h` was included directly from `main.c`, the symbol `TEST1_H` would have already been defined. So the `#ifndef` would prevent the body of the file (including the definition of `SIZE`) from being processed again.

(Note that for simplicity, many of the include files in CS376 thus far have not employed header guards.)

Overview

In this lab, you will get started (the last steps are optional in interest of time) on the process of examining all .h files in a directory and modifying those that do not follow this convention so that they do follow it.

In order to prevent files from getting trashed due to a buggy script, the modified versions of the files will be created in a new directory. The task of your script, therefore will be to:

- delete any existing version of the new directory
- create a clean, empty version of the new directory
- examine each file in the existing directory:
    - each file that is not a .h file is copied, unchanged, to the new directory
    - each .h file that already follows the convention is copied, unchanged to the new directory
    - each .h file that does not follow the convention is copied to the new directory, but with the appropriate preprocessor directives prepended and appended.

Writing this script "all at once" is likely to be a bit daunting. You will therefore be asked to do it in stages. At many of these stages, you will simply be printing something to the console to give you confirmation that you're on the right track. In the next stage, you will typically remove the print statements, and instead do the task for the next stage.

The major (checkpoint) stages will be as follows:

- `script1`, a new script:
  - Creates a directory called '`new`' (deleting existing one, if any), copying all files from the '`orig`' directory to '`new`'
- `script2`, continuation of `script1`:
  - Like `script1`, but when any .h file is copied, its contents are uppercased. (We will not be uppercasing in the final product, but this step proves that we can distinguish .h files.)
- `script3`, a new script:
  - Print the first line that has something other than //-style comments and whitespace; print the last line of the file that has something other than //-style comments and whitespace
- `script4`, continuation of `script3`
  - Tell whether a given file needs to be "wrapped" in a header guard.

Optional:

- `script5`, continuation of `script4`
  - Print an exact copy of the given file to the console, except that if it is not properly guarded, a guarded version is printed
- `script6`, continuation of `script5`
  - Analogous behavior to `script5`, but it now needs to work any .h file, not just `test1.h`. The name of the file is given as a command-line argument.
- `script7`, the "final product". This combines aspects of `script2` and `script6`:
  - Creates a directory called '`new`' (deleting existing one, if any), copying all files from the '`orig`' directory to '`new`'
  - loops through each file in the '`orig`' directory:
    - All non-.h files are copied unchanged to the '`new`' directory
    - All .h files that follow the header-guard pattern are copied unchanged to the '`new`' directory
    - All .h files that do not follow the header-guard pattern are copied, with guard added, to the '`new`' directory.

More details are below, in the descriptions of the various steps.

Preliminaries

Follow the procedure you used in the previous labs to log into the Ubuntu Linux VM.

Create a directory called *lab11*. Copy the starter files for lab11 into your *lab11* directory.

Part 1: Copy all files from one directory to another

(Note: the subtasks in this lab are just suggestions to help you divide the task into reasonable parts. The only thing that will be checked are the checkpoints.)

Create `script1` so that it performs the following.

Subtask 1a. If there is already a directory named '`new`', delete it. No error message should be printed, even if the directory does not exist. (Note: `rm` has a "recursive" option.) Create a new, empty directory named '`new`'.

Subtask 1b. In addition to the above, print the names of the files in the '`orig`' directory, one per line, as in.

```
file found: file1.c
file found: file2.h
 . . .
```

This step will help you confirm that you're going through each file in the directory.

Subtask 1c. Remove the printing done in the above subtask. Instead, copy each file from the '`orig`' directory to the '`new`' directory.
(hint: To get each file in the directory, `($ls orig))`. Refer to lab6 if needed for a reminder on how to write if statements and loops.)

Checkpoint 1 [20 points]: Show your lab instructor/assistant your script and its execution.

Part 2: Copy files from one directory to another, with modifications to .h files

Copy your `script1` file to `script2`. Modify `script2` so that it performs the following.

Subtask 2a. Same behavior as `script1`, but as you copy each file, report whether or not it is a .h file. (hint: You can perform a comparison on a file's name using == "*<filename>*". You can also use the * before the filename to denote any character.)

Subtask 2b. Rather than reporting about files being .h files, the contents of the .h files in the '`new`' directory should be uppercased versions of the originals. All other files should be exact copies.
(hint: Consider using the tr utility)

Checkpoint 2 [20 points]: Show your lab instructor/assistant your script and its execution.

Part 3: Extract the first and last "significant" lines of a file

Create `script3` so that it moves into the `orig` directory and then performs the following.

Subtask 3a. For the file `test1.h`. Print the first line in the file that contains something other than a //-style comment or whitespace. If the file contains no such lines, print nothing. You can test with test0.h which does not contain any relevant lines (so you should get no output), and test1.h which does (so you should print the first relevant line).
(hint: Consider using egrep)

(Note, for use in future checkpoints, it will be helpful to store the contents of this line in a variable.)

Subtask 3. For the file `test1.h`. Print the last line in the file that contains something other than a //-style comment or whitespace. If the file contains no such lines, print nothing. You can test with test0.h

which does not contain any relevant lines (so you should get no output), and test1.h which does (so you should print the last relevant line).

(Note, for use in future checkpoints, it will be helpful to store the contents of this line in a variable.)

Checkpoint 3 [20 points]: Show your lab instructor/assistant your script and its execution.

Part 4: Report whether a .h file needs to have a header guard added

In Part 4, you can get the subtasks checked off along the way or you can get them checked off at the end. Either way, you can get partial credit for Part 4 depending on how far you get, even if you don't complete all five steps.

Copy your `script3` file to `script4`. Modify script4 so that it performs the following.

Subtask 4a (10 points). Instead of hard-coding the name of the input file (like test1.h above), take the input file from the command line.

Subtask 4b (10 points). Comment out the printed lines from `script3`. Instead, report to the console whether the first line (as detected in Part 3) starts with '`#ifndef`', possibly preceded by whitespace. You can test your code with test1.h, which does not start with '`#ifndef`', and test2.h which does start with '`#ifndef`'.

Subtask 4c (10 points). Report to the console whether the last line (as detected in Part 3) starts with '`#endif`', possibly preceded by whitespace. You can test your code with test1.h, which does not end with '`#endif`', and test3.h which does.

Subtask 4d (10 points). Comment out the reporting of the above three subtasks. Instead, use the results of the above subtasks to report whether a given file needs to have a header guard added. If a header guard is needed, specify in a single line of output whether the #ifndef, or #endif, or both are needed. You can test your code with test1.h, which needs the starting and ending header guard, test2.h which needs just the ending guard, and test3, which doesn't need either.

Checkpoint 4 [40 points]: Show your lab instructor/assistant your script and its execution.

Part 5: Print a (possibly modified) copy of a .h file to the console

Copy your `script4` file to `script5`. Modify `script5` so that it performs the following.

Print an exact copy of the given file to the console, except that if the file needs to be wrapped with a header guard, the printed contents are preceded/followed by the header guard text. For now, if you have to add the #ifndef, you can use the filename following the #ifndef. You will correct that in the next part.

Checkpoint 5 [5 points]: Show your lab instructor/assistant your script and its execution.

Extra Credit

Part 6: Print a (possibly modified) copy of a .h file to the console

Copy your `script5` file to `script6`. Modify `script6` so that it performs the following.

Check that the symbol being tested/defined corresponds to the name of the actual file. For example, if the file name were `bellyFlop.h`, then the corresponding symbol would be `BELLYFLOP_H`. If the symbol is correct, no changes are needed. If the symbol is incorrect, insert the correct symbol.

Checkpoint 6 [5 points]: Show your lab instructor/assistant your script and its execution.

Part 7: Perform the (possibly modified) copying for every file in the original directory

Here you will be combining your `script2` and `script6` work.

Copy your `script2` file to `script7`. Append the contents of `script6` to `script7`; move the `script7` contents into the body of the `script2` loop. Perform whatever fix-up needs to be done (renaming of variables, commenting out print lines) in order to get the script to work.

The effect of the script should be that:

- The directory '`new`', if present, is deleted.
- A directory named '`new`' is created.
- All files from '`orig`' are copied to '`new`', where any .h files that need to have guards added are so modified in '`new`' directory.

Checkpoint 7 [5 points]: Show your lab instructor/assistant your script and its execution. Perform a "make" in the new directory; ensure that the program compiles and runs.

Part 8: Finish up

Close any editor windows that are still active. Save your work (follow previous labs for info on how to do this). Close all terminals (type exit). Close the terminal window you used to log into the Linux machine by typing exit. Log out of the Virtual Desktop. Close the VMware client. Finally, log out of the PC. You are finished.