P7: Recursive File Expansion

Due: Friday Dec. 3 by 11:59PM

What is File Expansion?

In many computer languages (be they for programming, text formatting or some other application) you can specify in a file `A` to *include* the contents of some other file `B`. Furthermore, this included file `B` may itself contain directives to include yet other files and so on. The process of "file expansion" takes as input a file such as `A` and produces its "expanded version." (By now, some of you are thinking "recursion"!).

In C and C++, this feature is implemented with the `#include` directive. Suppose we have a file called `root.txt` which looks like this:

```
<some-text-before-include.>

#include "myFile.txt"

<some-text-after-include.>
```

The process of *file expansion* starting from the file `root.txt` produces a *new* file which has all the text before the `#include` directive followed by the (expanded) contents of the file `myFile.txt` followed by the remainder of the file `root.txt` of the file `myFile.txt`.

This allows us to stitch together multiple existing files to create another file. In addition, if we want some text repeated in the final output, we just include that file multiple times rather than cutting and pasting (further, if we want to edit that text, we only have to do it in one place).

Why do we say the expansion process is **recursive**? Because the included files (like `myFile.txt` above may itself have `#include` directives in it and so on!

In this project we will use this same directive, but for arbitrary files (not just C or C++ source files). The files on the next page illustrate how this process works.

```
            Introduction

This is a great book by two authors.
The first part is written by Larry.
The second part is written by Jane.
Enjoy.

#include "larry.txt"
#include "jane.txt"

                The End
```

book.txt

```
    Chapter 1:  Larry's Vision

I begin this chapter with a proverb.

#include "proverb.txt"

Blah blah blah.

I think that says it all.
```

larry.txt

```
   "Do not remove a fly from your
   friend's forehead with a hatchet."

              -- Chinese Proverb
```

proverb.txt

```
    Chapter 2:  Jane's Vision

Consider the following proverb:

#include "proverb.txt"

I live by this proverb except when
the friend in question is Larry!

blah blah blah
```

jane.txt

Given the preceding files, expansion of the file `book.txt` would produce the following output file:

```
            Introduction

This is a great book by two authors.
The first part is written by Larry.
The second part is written by Jane.
Enjoy.

    Chapter 1:  Larry's Vision

I begin this chapter with a proverb.


     "Do not remove a fly from your
    friend's forehead with a hatchet."

                -- Chinese Proverb


Blah blah blah.

I think that says it all.


    Chapter 2:  Jane's Vision

Consider the following proverb:


     "Do not remove a fly from your
    friend's forehead with a hatchet."

                -- Chinese Proverb


I live by this proverb except when
the friend in question is Larry!

blah blah blah


              The End
```

Again, notice that the process is recursive since included files may themselves include other files (e.g., `book.txt` includes `larry.txt` which includes `proverb.txt`.)

Remember that any recursive procedure must terminate – it can't just keep calling itself for ever. Similarly, the result of a file expansion must be a *finite* output file.

Is it possible to create a group of files for which there is no well-defined finite output? Yes! For example, if a file includes itself.

Key part of project: If such a situation occurs (infinite expansion) a program doing the

expansion should report the error and terminate.

(If your program has already produced some output before detecting this situation and then prints the error message when the issue is detected, that is fine!)

Note however that a file may be included multiple times and still produce finite output (e.g., `proverb.txt` is included twice).

Your Program

You are to write a C program called `expand` which takes a single command line argument: the name of the root input from which expansion begins.

Your program with then expand the input file and print the resulting expansion to `stdout`.

In the event of an error, the program prints an appropriate message to `stderr` and terminates. It does not attempt to recover from the error.

Some suggestions, details, notes, assumptions and requirements:

- (requirement / hint) Cycle detection / infinite expansion detection.

  In the event that the files result in infinite expansion, your program reports the error and terminates.

  Thus, when your program is processing an `#include` statement, it somehow needs to be able to answer the question *"will expanding this file at this point create a cycle and therefore infinite expansion?"*

  This can be achieved by maintaining a data structure containing the names of files in a particular state (I'm being intentionally vague here...). Being able to query this data structure (*"is file X in there?"*); add filenames to it (*"add filename Y"*) and remove filenames from it (*"remove filename Z"*).

  (You should be thinking "Abstract Data Type" I hope).

  You'll need to figure out exactly how to manage this data structure and what it means for a filename to be in it. So consider the above not a "how-to", but a set of hints.

  You will partially be graded on your design which in this case means how well encapsulated this data structure will be an evaluation criteria.

- (detail) You should know by now how to open and read files using `fopen`, `fclose` and operations on the `FILE` type.

  Be sure to close files once you are done reading them.

- (assumption) You may assume that each line in the input files has no more than 256 characters (an arbitrarily chosen "big number"). This lets you use `fgets` to safely read the input line by line with no risk of overflow.

  For the ambitious, you can figure out how to determine if an input file exceeds 256 after you call `fgets` and report the error and terminate (not required, and no extra credit!).

  A sensible overall approach is to first read a line into a buffer and then process the buffer to decide what to do.

- (detail) A valid `#include` directive must start at the *first position* of a line. If the string "`#include`" appears anywhere else, it is treated as just part of the text in the file (e.g., maybe the author intended it to be there).

- (detail) We only allow quoted filenames. In other words, `#include` statements like `#include <stdio.h>` are considered errors for us.

- (detail) If a line begins with an include statement, but there is no quoted file name after it, this is an error and you should report it and terminate.
- (detail) **There must be whitespace between `#include` and the quoted filename** (this may be any amount of white space so long as the filename is on the same line).
- (detail) Similarly, if an include statement specifies a non-existent file, you should report the error and terminate.
- (detail) If an otherwise correct include statement has extraneous text after it *on the same line*, simply ignore the extra text (this makes your life easier).
- (note) the C preprocessor `cpp` can be run on most UNIX systems and will produce almost the same results as your program will if you run it with the `-P` flag. (The only difference seems to be that blank lines in the input do not seem to be repoduced in the output of `cpp`). For more info, type `man cpp` or google for some documentation.

Remember to observe good programming practices:

- No globals
- Good variable names
- If a function starts to get long, break it up into subtasks.
- For each function write a header comment specifying the meaning of each parameter and the semantics of the method – i.e., concisely and unambiguously what it does, returns, etc.
- Encapsulate the behavior of certain data structures where it makes sense. (See bullet item on cycle-detection above).

Testing

Spend some time coming up with interesting test cases. Do not just run on the toy example in this handout!

You have also been provided several additional test cases in the folder `test-cases` (including the example from this handout).

You are free to share test cases with your classmates.

Program Submission

Submissions will be through Gradescope You will submit the following in an archive file:

- All source files you have written – .c files and .h files.
- (optional) a `readme` file if there is anything you want the grader to know about your code.
- a `makefile` allowing us to simply type `make expand` to compile your program.