CS 352 (Fall 16): System Programming and UNIX

Project #8

Multiple Compilation - LineSort

due at 9pm, Wed 9 Nov 2016

1 Overview

In this project, you'll be using multiple compilation for the first time. You will create two C files, one header file, and a Makefile to build the executable. We've given you a problem which doesn't really **require** that we split the problem into two pieces; however, we're splitting it so that you can get your first experience with multiple compilation.

We are going to allow more filename flexibility in this program. You must still produce an executable with exactly the name we expect - and put all of your code into a directory which has the filename we expect - but you can name the C files, and the header, whatever you would like.

Your executable must be named `lineSort`.

2 Program - Sorting Lines

In this program, you will read in a set of files, each full of text. You will sort the lines of each file, and print out the stored set from each file. You will do this one file at a time, reading the file names from the command line.

One of your C files will be responsible for I/O - reading the files, and printing them out at the end - while the other C file will provide some helper functions - one for resizing an array, and one for sorting the array.

2.1 Your Primary C File

Your first C file must include your `main()` function. It must `#include` the header file, and call functions in the "utility C file" to do most of the array manipulation. Specifically, your first C file must perform the following tasks (for each file listed on the command line):

- Allocate an array of `char` pointers - but **no longer than** 16 elements.
- Read from the file, into a series of string buffers - and save them into the array. (No processing is required, except that you may want to remove newline characters so that all of the lines are in a standard format.)
- If/when the array fills up, call `extendArray()` to make the array larger. This file **must not** resize the array by itself!
- At EOF, call `sortArray()`.
- Print out the (sorted) array at the end, and free memory.

2.2 The Header File

Your header file must include prototypes for all of the functions in your "utility file" (see below). If you use any structs or other types (you are not required to), then it must also include all of those declarations.

Your header must also use `#ifndef` and `#define` (as shown in the slides) to prevent it from being included twice - even though our project is simple enough that this isn't really a worry.

Your header must include a file header, just like a C file. You are required do document each function - but you may choose whether to put this documentation in the header file, or the C file. (Differnt programmers have different styles.)

Of course, both of your C files must `#include` this header.

2.3 The Utility C File

Your second C file - the "utility" file - must include at least the following two functions (you can add more if you'd like):

- `char **extendArray(char **oldArray, int oldLen, int newLen)`

  This function must allocate a new array using `malloc()` or `calloc()`, copy the values from the old array into the new one, and then `free()` the old array. It returns the new array (or `NULL` if there was a malloc failure).

  You are encouraged to look at the man page for `realloc()` - the standard library function which does this - **but you are not allowed to use this function in this project!**

- `void sortArray(char **array, int len)`

  This function will sort the array of strings. Strings will be compared using `strcmp()`.

  You are free to use any sorting algorithm that you want - even the infamously slow Bubble Sort. Of course, if you want to implement a more efficient sort - such as Quicksort - you are welcome to!

  You are encouraged to look at the man page for `qsort()` - the standard library function which does this - **but you are not allowed to use this function in this project!**

  NOTE: This function must not allocate or free any memory. Just move pointers around - if you find yourself trying to allocate memory, then you're not understanding how it is supposed to work.

2.4 Input Format

Your program will read from a series of text files, listed on the command line. Each file is a simple text file, with no limitations (other than that it should only contain simple ASCII characters). As with the `tokenize` program, you

should not attempt to verify that the file is text; just read the characters with `getline()` or `fgets()`.

Each line in the file can be any length, so we encourage you to read it using `getline()`. If you prefer to use `fgets()`, you will have to `malloc()` a string buffer, and copy into it - which is possible, but annoying.

Whitespace should be preserved, and treated as part of the line.

Empty lines (that is, lines with only a single newline) should be preserved, and treated as an ordinary line - just one that has zero characters on it.

The last line in the file might or might not have a newline at the end. If it does not, make sure to **add a newline** when you print it out after sorting.

2.5 Output Format

For each file, you will read the entire file into an array and sort it. When you print it out, the first line of your output should be a summary, like this:

```
The file 'foo' had 32 lines.
```

After that, print out the lines of the file, sorted, with no extra text or decoration. Remember that empty lines (lines with no characters other than the newline) and lines with only whitespace should be printed out exactly like all other lines; **do not skip or edit them.**

An empty file should have the summary line, but report that there were 0 lines.

If the file does not exist, **this is not an error.** Instead, simply report (to `stdout`) that it did not exist, with the following message:

```
The file 'foo' did not exist.
```

2.6 Command-Line Argument Format

Your program will be run as follows:

```
<progName> <file1> <file2> ...
```

If there are no arguments, then your program should not print out anything.

2.7 Error Conditions

This program has no error conditions. It should always terminate with exit status 0 - except for `malloc()` failures.

3 `gcov` and Multiple Compilation

To use `gcov` with multiple compilation, keep to these guidelines:

- Pass the `gcov` parameters to `gcc`, just like you did when there was only one file. Do this **both** when you compile the C files, and also when you link them together.

  `gcc` will generate separate `.gcno` file for each of your C files. Likewise, when you run your program, there will be a `.gcda` file for each C file.

- Run `gcov` **multiple times** - once per C file. It will generate a `.c.gcov` file for each C file.

4 Grading

As with Project 6, we will not provide a grading script, or any testcases. You must provide testcases which cover your entire code (except for `malloc()` error handling), and your code must run without `valgrind` errors. In addition, you must provide a Makefile which builds your code. See the Project 6 spec for a reference.

4.1 Example Executable

As always, we have provided an example executable for you to compare your code to. You can find it in the project directory, either online, or on Lectura.

4.2 Testcase Format Updates

This Project will use the same testcase format as Project 7.

4.3 Grading Scheme

This project has some grade items which apply to the entire project (not to individual testcases), so our grading scheme has gotten more complex:

- 10% of your grade will come from your Makefile
- 10% of your grade will come from `gcov` coverage
- Manual inspection of your header file:
  - 5% - Proper use of `#ifndef / #define`
  - 5% - Gives prototypes for all functions in the utility file
  - 5% - Included from both of the C files
  - 5% - Required comments in the header file
- 60% of your grade will come from testing your code, comparing it to the example program.

Of course, just as before, we also have a set of penalties which may be applied - these subtract from your overall score:

- -10% - Poor style, indentation, variable names
- -10% - No file header, or missing header comments for any function (other than `main()`)
- -10% - Missing checks of returns codes from standard library functions (such as, but not limited to, `malloc()`)
- -20% - Any use of `scanf()` family `%s` specifier without limiting the number of characters read.

4.4 Testcase Grading

We have decided to subdivide the score from each testcase:

- You must exactly match `stdout` to get **ANY** credit for a testcase.
- You will lose one-quarter of the credit for the testcase if the exit status or `stderr` do not match the example executable.
- You will lose one-quarter of the credit for the testcase if `valgrind` reports any errors. (This includes any memory leaks.)

As always, you lose half of your testcase points if your code does not compile cleanly (that is, without warnings using `-Wall`).

4.5 Testcase Selection

As in Project 7, we will use the testcases that you submit to perform `gcov` testing on your program - and a selection of testcases from the students (plus a few from the instructors) for checking to see if your program works **correctly.**

5 Standard Requirements

- Your C code should adhere to the coding standards for this class: `http://www.cs.arizona.edu/classes/cs352/fall16/DOCS/coding-standards.html`
- Your programs should indicate whether or not they executed without any problems via their **exit status** - that is, the value returned by `main()`. If you return 0, that means "Normal, no problems." Any other value means that an error occurred.

  In this class, we will always use the value 1 to indicate error; however, outside this class, many programs use many different exit status values - to indicate differnt types of errors.

  In `bash`, you can check the exit status of any command (including your programs) by typing `echo $?` immediately after the program runs (don't run **any** commands in-between).

6 turnin

Turn in the following directory structure using the assignment name `cs352_f16_proj08`:

```
lineSort/
    Makefile
    <at least two C files>
    <at least one header file>
    test_lineSort_*/
        <various files in various testcase directories>     (turn in several of these!)
```

7 Grade Preview

48 hours before the project is due, we will run the automated grading script on whatever code has been turned in at that time; we'll email you the result. This will give you a chance to see if there are any issues that you've overlooked so far, which will cost you points.

Of course, this grading script will not include the full set of student testcases, which we will use in the grading at the end, but it will give you a reasonable idea of what sort of score you might earn later on.

We will only do this once for this project. If you want to take advantage of this opportunity, then make sure that you have working code (which compiles!) turned in by 48 hours before the due date.