CS61C Spring 2018 Project 1: philphix

TA In Charge: Nikhil Athreya

Due Thursday, February 1, 2018 at 23:59:59

Goals

The objective of this project is to serve as an introduction to the C language. This project will cover a lot of different features of the C language, including the C file input/output library, memory allocation, string manipulation, and string coercion to void pointers and vice versa. Although there is conceptually a lot to learn to complete this project, the actual code you need to write is relatively short.

Background

Philphix is a very simple and silly replacement tool that accepts a single command line argument, the name of a replacement set to use. This replacement set consists of pairs of "words", each pair on its own line. The first word only consists of alphanumeric characters, but the second word can include any non-whitespace printable character.

For each word (sequence of letters and numbers unbroken by any other character) in the input it processes it to see if it should be replaced. It first checks if that word is in the replacement set, then checks that word with all but the first character converted to lower case, and finally checks if the word converted completely to lower case. If there is no match the word is printed to standard output unchanged; if there is a match, it prints the highest priority match.

Your assignment is to complete the implementation of `philphix`. To do this, you will need to implement 4 functions in philphix.c: `stringHash(void *s)`, `stringEquals(void *s1, void *s2)`, `readDictionary(char *filename)`, and `processInput()`.

Just as a reminder, this project is to be completed **individually**.

Setup

To get starter code for this project, we will be using git.

1. First, you need to create a private repo with the format proj1-xxx, where xxx is your login. The process for this is very similar to what was done in lab. Please make sure this repo is **NOT** your work repo however. Furthermore, please set this to **PRIVATE**. Just to make sure, set this to **PRIVATE**. If you do not set this to **PRIVATE**, horrible things will happen to you and you will be severely punished. So before continuing, make sure your repo is **PRIVATE**. Not setting your repo to private, even if by mistake, will be seen as an intention to cheat.
2. From your access management settings, give 'cs61c-staff' admin access to your repo. You also must do this from the very beginning to avoid any penalties placed upon your grade.

3. Did you set your repo to **PRIVATE** and give the 'cs61c-staff' admin access? If yes, enter into the directory of your class account that you would like to hold your proj1-xxx repository. Once in this directory, run the following:

```
git clone https://mybitbucketusername@bitbucket.org/mybitbucketusername/proj1-xxx.git
cd proj1-xxx
git remote add proj1-starter https://github.com/61c-teach/sp18-proj1-starter.git
git fetch proj1-starter
git merge proj1-starter/master -m "merge proj1 skeleton code"
```

Once you complete these steps, you will have the proj1 directory inside of your repository, which contains the files you need to begin working.

As you develop your code, you should make commits in your git repo and push them as you see fit. Be sure to project.

Overview

After you have a copy of the starter code, make sure you have the following files:

- `hashtable.c` - Code for a generic hashtable.
- `hashtable.h` - Header file for hashtable.c. For more information on header files and how they are used, check out this link. You should not need to modify either hashtable file, though take a look at them and see how they work.
- `philphix.c` - Contains various functions for our spellchecker. You will need to implement 4 functions in philphix.c: `stringHash(void *s)`, `stringEquals(void *s1, void *s2)`, `readDictionary(char *filename)`, and `processInput()`.
- `philphix.h` - Header file defining functions in philphix.c. You may modify philphix.h if you wish to declare additional helper functions which you implement in philphix.c.
- `Makefile` - Makefile for compiling philphix, as well as for running tests. For more information on Makefiles and how they are used, check out this link.
- `diff.sh` - A bash script to use for additional testing. This will only work on Linux/Mac machines. First run `chmod +x diff.sh` and then run `./diff.sh` to see how to use the script. Use this script for additional testing once your code works on the given testing, replacement, and reference files.
- `replace.txt` - Sample set of replacement rules.
- `test.txt` - Sample input file.
- `reference.txt` - Sample output file.

Spend some time looking over these files and figuring out what is in each file and what they are used for.

Instructions

As noted above, your task is to implement the 4 functions in philphix.c: `stringHash(void *s)`, `stringEquals(void *s1, void *s2)`, `readDictionary(char *filename)`, and `processInput()`. More information about the purpose of each of these functions can be found inside

philphix.c.

You can type the following in your project 1 directory to compile and test your program against a sample set of inputs:

```
$ make test
```

You can, however, safely output all sorts of debugging information to stderr, as this will be ignored by our scripts and by the test routine provided in the Makefile.

While completing this project will require getting familiar with many parts of the C language (C file input/output library, do memory allocation, manipulate strings, and coerce strings to void pointers and vice versa just to name a few), your final solution should not be all that long.

Start early! You likely will have to learn and experiment a lot with C to complete this project, so make sure you budget enough time.

For grading purposes, we will be splitting the project into two parts:

1. When you first are writing your solution to the project, you can assume that that both the replacement set and the input won't contain words longer than **60** characters. Sucessfully doing this will be worth **70%** of your grade on the project.
2. For the remainding **30%** of your grade, we will test your code against both dictionaries and inputs that contain words of an unbounded length. We suggest trying to get a solution with a bounded character count before attempting this part.

Submissions

There are **two** steps required to submit proj1. Failure to perform both steps will result in loss of credit:

1. First, you must submit using the standard unix submit program on the instructional servers. This assumes that you followed the earlier instructions and did all of your work inside of your `git` repository. To submit, follow these instructions after logging into your cs61c-xxx class account:

   ```
   cd proj1-xxx
   submit proj1
   ```

   Once you type `submit proj1`, follow the prompts generated by the submission system. It will tell you when your submission has been successful and you can confirm this by looking at the output of `glookup -t`. Remember, you only need to submit two files: `philphix.c` and `philphix.h`.
2. Additionally, you must submit proj1 to your bitbucket repository. To do so, follow these instructions after logging into your cs61c-xxx class account:

   ```
   cd proj1-xxx
   git add -u # should add all modified files in proj1 directory
   git commit -m "Project 1 submission"
   git tag -f "proj1-sub" # The tag MUST be "proj1-sub". Failure to do so will result in loss of credit.
   git push origin master --tags # Note the "--tags" at the end. This pushes tags to bitbucket
   ```

Grading

The grading for this project will be done almost entirely by automated scripts. Your output must exactly match the specified format, which makes correctness the primary goal of this project. Again, you are to do this work individually.

One particular aspect of C is *nondeterminism*: something that may run perfectly fine on one computer doesn't run the same way on another due to memory bugs or other issues. We will be grading exclusively on the *hive* cluster, so you must make sure your code works correctly on those systems. And although we will attempt to help you run your code on your own computer rather than the hive, please understand that running on your own computer may result in problems we are not equipped to debug.

One significant aspect that you should focus on is testing. You can create some python scripts to produce larger sets of input. Other good tests for robustness include using large binaries on standard input (with the output redirected to `/dev/null`).