Jessica Pan
jeypan@ucsc.edu
Faisal Nawab
Fall 2019 – Asgn0

WRITEUP

The testing I performed on my program was both unit and whole-system testing. As I built my program, I'd test every small part that I would have finished programing, making sure that part of the program ran properly. Basically, after each part of my program was written, I'd test if for bugs. I first programmed for the first command *./dog*, where the program would print what I typed back into the standard output. After testing it and making sure it produced the proper output, I moved on to program for the next command *./dog -*. I then proceeded to test it for bugs and a proper output after writing that part of the program. I repeated this process for the dog program, doing tests and debugging for every command possible used for cat. Below I've listed the different tests I performed for each command after I've finished programming it:

1. *./dog*
   - Tests: I ran the command and checked if the program printed the same text I wrote to STDIN back into STDOUT.
2. *./dog --*
   - Tests:
     a) I ran the command and checked if it did the same as the previous command.
     b) I then ran *./dog --* to check for the same functionality as *./dog –*
     c) Next, I tested for *./dog --f* to see if it would produce an error to STDERR. In this case it would and should produce *dog: --f: No such file or directory*.
3. *./dog [filename]*
   - Tests:
     a) I created random text files and ran them with the dog program to check if *dog* would spit out what's in the file into STDOUT. I then ran multiple text files at the same time with *dog* to have the program spit out what's in all the files to STDOUT at the same time.
     b) I then created a simple pdf file and ran it with *dog* to check if the program was able to write the pdf file to STDOUT. I also found a sample binary file online and did the same thing with it as the pdf file. [I did this to any other files that had different file extensions than .txt].
     c) Next, I checked if the program would produce an error message to STDERR when I ran *dog* with a directory. The error message would look like: *dog: <dir>: Is a directory*.

   d) The next test was that I ran *dog* with a nonexistent file to expect an error message to STDERR saying: *dog: <nonexistent_filename>: No such file or directory*.

4. *./dog [standard input] [filename]*
   - Tests: The tests I performed before this fourth bullet point has covered majority of the unit testing. In this fourth bullet, I'm performed the whole-system testing. Below are different commands I used to test and perform the whole-system testing: (let *'f', 'g' , and 'h'* represent a file containing contents, and '-' as the standard input)
     a) *./dog f - g*
     b) *./dog -- f g <nonexistent_filename>*
     c) *./dog f f - h - -*
     d) *./dog <nonexistent_filename> - <nonexistent_filename> - h*
     e) *./dog - - g h h <dir> --f f --*
     f) *./dog - -- -- <dir> <dir> <nonexistent_filename>*
     g) *./dog <dir> f g h <nonexistent_filename>*

To answer the question for this assignment: How does the code for handling a file differs from that of handling standard input? What concept is this an example of?

The code that handles standard input simply checks for the arguments of the program when it's ran. If there are no arguments, or arguments of '-' and/or '--' then the program reads from the standard input and prints to the standard output. However, this process differs when the code handles a file.

First, we check if the filename given actually exists and if it's not a directory. We then open the file, given that we're able to open it. Once we're able to open it, we can read what's written in the file, given that the function we use to read the file doesn't give us an error. After reading the file, the program prints out what was read back into the standard output. In any case, for whichever file we weren't able to run with the code, the program would produce an error message to the standard error.

This program handles different scenarios that could happen at the same time, separately. It breaks down the amount of work that needs to be done into smaller pieces, separating different parts of the work load to different part of the program. This reduces the number of interactions with the program, which reduces the number of bugs. It also allows for interchangeable modules, making it easier to replace certain modules in the program. I believe this concept is an example of modularity.