Assignment 0 Design Document

Introduction

Objective:

The purpose of this assignment is to replicate the functionality of the most basic version of the unix cat command. By most basic, this means the bare bone functions of cat; it is unnecessary to implement flags. The program will be called dog.

Data Design and Structures

Parameters:

Most parameters passed into functions consist of types ssize_t and uint8_t.

Functions:

main() contains the majority of code using other functions like open(2), read(2), write(2), close(1) to handle both files and standard input.

Function read_and_write() is made specifically to only handle standard input to reduce code redundancy within main().

warn() is used for error checking.

Functionality

Run dog in unix command line. Expected output should follow unix cat exactly except in reverse order. Run dog with text file(s) or binary file(s) and it will output the contents of the file(s) to standard output. Able to handle multiple files inputted in any order, but must reproduce the same errors as unix cat printed to standard error and continue execution on remaining files. Run dog with no arguments or a dash (-) and it will copy standard input to standard output until EOF signal is received.

Constraints

The program, dog, must be written in C with restrictions on C libraries that contain FILE * functions. Most I/O functions are prohibited from being used. It must also include fixed-size buffers that may not allocate more than 32 KiB of memory. A general coding guideline must be followed as well, which can be observed here: https://d1b10bmlvqabco.cloudfront.net/attach/k8ala0scdtafx/k035i6v78svoz/k8q6n8nrcxsh/CodingGuidelinesAndStandards.pdf

Implementation

main():

1. Initialize buffer size
2. check if # of arguments = 1 then call function read_and_write() to handle this case
3. For loop to access each file given on command line starting from last argument

4. check if argument = “-” then call function read_and_write() to handle this case and decrement loop counter
5. if loop counter = 0 then dog execution complete else continue
6. open file and check if valid return value for error handling
7. while true read the opened file and check if valid return value for error handling
8. if error occurs break out of loop and close opened file
9. else if read finished reading file break of out loop and close opened file
10. write to file to standard output and check if valid return value for error handling

read_and_write():

1. while true read in standard input
2. check for errors if read does have valid return value then break out of loop
3. if read finished reading then break out of loop
4. write to standard input to standard output and check if valid return value for error handling

Testing

Testing for dog was first done under unit-testing which later transitioned to system-testing. While developing the read_and_write() function, it was tested on the unix command line by running dog only for the cases where read_and_write() were called (when there were no arguments given, or a “-” was found). The expected output would be standard input printing to standard output until an EOF signal was received. Function read_and_write() was tested enough times to make sure that it performed as expected within the cases it was made for by running tests like ./dog - and ./dog . Next, the testing moved towards the program as a whole. Here, tests became a bit more complex as it had more cases to account for such as running dog with a “-”/no arguments as well as both large and small text/binary files together. The testing expected that text/binary files would still be outputted and the “-”/no argument cases would still print standard input to standard output until exited all within the same execution of dog. Some test cases would be:

```
./dog test.txt -
./dog - test.txt
./dog dog
./dog test.txt - test2.txt
```

There are plenty more combinations that were used to cover all cases but these are just a few examples. In order to ensure that dog was replicating cat outputs exactly, unix commands diff, cmp, and hexdump were used to compare dog outputs with cat outputs. After running dog and cat on the same files, the output of each execution was saved to different text files with unix piping/redirection. From there diff/cmp/hexdump were called to check the differences between the files. Lastly, the unix input redirection was also used to test files. If dog printed the same output as cat after comparing the redirection output of the files as well as the redirection input of the files then the test was labelled successful.