Design Document

Assignment 0: DOG

Garrett Webb
CruzID: gswebb
10/18/2020

Goals:

The goal of this program, aside from learning the lab format and setting up the environment properly, is to mimic the "cat" command in Linux.

Assumptions:

When designing this program, we assume a few things, including:

- Basic functionality should be the same as "cat"
- No flag support.
  i.e. "cat -x y", where "x" is some modifier to program functionality and "y" is a list of program arguments is not supported.
- Error messages should be the same
  i.e. "cat y" if "y" does not exist should return the same error as running "./dog y"
- There is a fixed 32KiB buffer, and cannot be allocated more memory.

Design:

In designing this program, the general concept is to have nested loops. One outer loop to iterate through the files as presented in the program arguments, e.g. file1, file2, file3. The inner loop to iterate through the individual file, reading its contents. Of course there is logic and error checking to make sure that these are done in the correct order and on valid files. We also need to echo user input if there are no files chosen, or copy user input to the buffer if a "-" is used as a filename, so that it can be piped into another file using bash commands.

There are many cases we check for before attempting to open and read a file. Before we even enter the outer loop, we check how many arguments are given. If none, we echo user input. If there are arguments given, we begin with the first one, and work towards the last. We first check if the argument is a "-" and if it is, we echo user input again. If we do not have a "-", we attempt to open the file. If it is a file that does not exist, or it is a directory, or the file is inaccessible by the program, give the appropriate warning and move to the next argument. Once all of these cases are out of the way, we write the file to output using the inner loop.

This will be visualized in the pseudocode section that follows.

```
function main(argc, argv){
    if(argc == 1){
        //if there are no arguments passed in, we echo output.
        while(no read/write failures){
            readcheck = read(buffer)
            writecheck = write(buffer)
            if( writecheck != readcheck ) {
                perror( "dog" );
            }
        }
    }
    else{
        while( fileCounter <= fileNumMax ) {
            if( strcmp( "-", argv[ fileCounter ] ) == 0 ) {
                //first check if the argument is a “-“ that means we echo input
                while(no read/write failures){
                    readcheck = read(buffer)
                    writecheck = write(buffer)
                    if( writecheck != readcheck ) {
                           perror("dog")
                    }
                 }
            }
            else{
                //it is an actual file argument, open it
                file = open( argv[ fileCounter ])

                if( file does not exist ) {
                    give warning message
                    fileCounter++;
                    continue;
                }
                if( file is directory ) {
                    give warning message
                    fileCounter++;
                    continue;
                }
                if ( user has no permissions to open ) {
                    give warning message
                    continue;
                }
                else{
                    //file exists, is not directory, and is accessible
                    //now we read it to the buffer
                    filenum = 1;
                    while ( read/write has no error ){
                        readcheck = read( file into buffer )
                        writecheck = write( buffer )
```

```
                        if( writecheck != readcheck ) {
                            perror( "dog" );
                        }
                    }
                }
                //gg go next
                close( file );
            }
            fileCounter++;
        }
    }
}
```

Process:

While designing the program, I figured it would be easiest to replicate the user input echo functionality of the program first, so I began with only that implemented and tested until I had it right. I then added functionality to print a singular file, and tested that until it worked as required. I kept incrementally implementing functionality and error-handling and testing edge cases until the program worked as the specification required.

Testing:

To test the program, I initially used manually put together text files and manually ran them, comparing to the output of the cat command with the diff command. This took a lot of typing, so I created a simple bash script that does the following:

- make clean
- make
- run ./dog with many different files, some empty, some large, some directories, some nonexistent, some inaccessible, some binary, some hyphen, some text, some with strange ASCII or Unicode characters, and pipe the output into another file.
- Run cat with the same files in reverse order and pipe the output into another text file.
- Run diff on the two output files, outputting the difference between them to another file.
- Make clean
- Remove the intermediary output files

This script gave me the ability to quickly test my code and all of its edge cases with a single command.

Question:

How does the code for handling a file differ from that for handling standard input? Describe how this complexity of handling two types of inputs can be solved by one of the four ways of managing complexity described in Section 1.3.

Answer:

When reading standard input, we infinitely loop to continuously read what the user inputs, until they press ctrl+c to quit. This differs from reading from a file because when we read a file, the read() function returns a value 0 or less if some sort of error happened in the read, i.e. you reached the end of the file. We simply loop until the returned value is no longer greater than 0. This is an example of reusable code, since the code is virtually the same, but using it in different scenarios with different inputs can change the functionality of it. This complexity of handling two different types of input can be handled using modularity. A good decision would be to turn the section of code that reads from the file and writes to the buffer into function that can be called with different parameters for different scenarios. In my program, I chose to simply re-use the code instead of making it into a modular function and calling that function when it was needed. This is because of the small size of this program; I chose to reduce the overhead and have a couple lines of duplicated code.