Class: CSE 130
Professor: Faisal Nawab
Fall 2020

Design Document: dog program

Contributors:
Adriel Naranjo - adjnaran
Abbas Engineer - afengine

Objective: write a program in C (can't use C FILE lib funcs.) to make cat, a command that will copy data from each of the files specified on the command line to standard output. This includes binary and must support the case where ‘dog -’ which just allows the user type in the GUI and once enter is hit, the output is the same thing as was written.

Initial Assumptions

1) Probably going to need to use system calls: open, read, write, closels
2) Definitely going to need a buffer

Algorithm:
Checks if there are any arguments in cmd and makes sure argv[1] isn't a ‘-’
get the number of files entered after dog to STDOUT iteratively.
go through each file
if file doesn't exist then throw an error and continue
if the file does exist
int n to make sure there are file contents
and buffer of course to read into and write out of
read and write using the system calls.
close that file

Pseudocode:

```
Create a buffer

Case: dog a_file.txt b_file.bin
        If argc > 1:
                For each file:
                        open file READONLY
                        Write byte read into buffer:
                Print buffer stdout.

case: dog -
        As long as the null terminator is NOT seen || s  trchr(argv[1] , '-') != NULL:
                Print characters out to STDOUT
```

Test Cases:

1. ./dog -   => renders standard input to standard output
2. ./dog   => renders standard input to standard output
3. ./dog file1 - file2 file3   => renders files then standard input to output in order
4. ./dog file1 file2 file3   => renders files to standard output in order
5. ./dog file1.bin file2 file3   => renders both .txt and .bin to standard output in order

What skills were needed?

1. Read and write from contents of a single file
2. Loop through all files
3. Copy into buffer and print using std output
4. Learn makefiles again
5. Work around not using File * library

Assignment Question:

Q: How does the code for handling a file differ from that for handling standard input?
A: dog command required different resources depending on what was entered.

When there was nothing entered after dog or a '-' after dog (the primitive case), then anything typed into stdin needed to be "pointed out" using stdout so the code was simple.. Write all the data from stdin to stdout while there is any.

When files/records are entered after the dog command (argv[] string elements), to maintain data fidelity, the contents of each file had to be written into a buffer then be read out from the buffer and printed to stdout. This happens with all files that exist in the current working directory and were entered after the dog command. This was nearly as simple as the base case but required the code written to handle cases where a file may not exist in the pwd.

Q: 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. (This topic will be covered at the end of week 1 or early week 2, but you can start earlier and answer this question by reading section 1.3.)
Review:
Modularity: replace an inferior module with an improved one, thus allowing incremental improvement of a system without completely rebuilding it.
Abstraction: the ability of any module to treat all the others entirely on the basis of their external specifications, without need for knowledge about what goes on inside. This additional requirement on modularity is the definition of abstraction. Abstraction can also be defined as the separation of interface from internals, of specification from implementation and can include

techniques like *fitting* where one would gauge *tolerances* to of components to maximize or minimize a metric of performance.
Layering: way to reduce module interconnections is to employ a particular method of module organization.
Hierarchical: also reduces interconnections among modules but in a different, specialized way. Assemble a small group of subsystems to produce a larger subsystem. This process continues until the final system has been constructed from a small number of relatively large subsystems. The result is a tree-like structure.

A: By process of elimination we have narrowed down to one being, Modularity. This conclusion was made because modularity fits the context of this assignment the best. By the definition above, modularity has to do with incrementally building and improving a system or process. In the dog program it was required that different resources needed to be allocated by the system in order for the program to work as if 'cat'. Creating one function for handling the base case and another function to handle other cases allowed our program to be dynamic and display characteristics of incrementing.