For this project, we (**Noah Sauber** & **Tyler Robinson**) only wrote one source file, "**shell.cpp**," that handled the entire implementation of the problem. Compiling this file with **make** will place an executable called "**shell**" in the current directory. Entering the command '**make run**' will both compile **and** run the program. We practiced iterative development throughout the entire process, and this write-up will be organized in a manner that attempts to explain the implementation chronologically.

The first goal was to parse input from the user. We initially made sure that the command didn't exceed 80 characters. If so, the program continues asking for user input. We then used a stringstream in order to split the input by spaces, and this gave us the tokens. After this, we made sure that these tokens contained valid characters by checking each character in the tokens against those stored in a valid character bank. If all characters in each token are valid, the tokens are split by the pipe operator and placed into a vector of vectors of strings which is meant to represent the token groups present in the user-inputted command. After this, the tokens are examined in order to determine if the entered command is a valid one. This is done by considering the "illegal" cases presented in section 2.1.3, and throwing errors if these cases are seen in the entered command. After this is all done, it is time for the program to start attempting to interpret the command.

At first, we wanted to make sure that a single, non-piped command with no I/O redirection could properly be ran. For this reason, we started off by only interpreting the first token group that was inputted, fully intending on building upon this code later. The main logic of the command interpreter is a recursive runCommand function that ended up taking four parameters: the token groups vector, the array of file descriptors used for piping, a 'pipe counter' that would be used keep track of what file descriptors would be overwritten, and a constant value used to for loop indexing. Each call to this function will take a token group off the end of the token groups vector leading to a base case in which this resulting vector has size zero by the time the function is being called again. For the case with only one command, this function is only called once. The first thing to take care of was to create a char* array that contained the all the tokens of the token group (the entire command) plus a closing null argument. After the call to fork, this array is passed to the execvp() function in the child branch of the pid-checking if statement. The parent then waits for this process to finish executing, and thus, resumes once the command is completed. This was all that needed to be done for one command.

Next, we wanted to keep testing only one command, but this time, the program should be able to handle I/O redirection. The initial checking for I/O redirection occurred in the runCommand function. The first thing we noticed when implementing this was that the size of the command arguments array would need to be altered in order to not include the '<' or '>' characters. Once this was handled by a simple checking of the tokens and resizing if necessary, the input file or output file was opened before the call to fork() in preparation of I/O redirection by the child process. Then, the child would replace the stdout or stdin with the new file descriptors generated for the output of input file respectively. After this, execvp() is called, and the parent takes care of closing the output or input file descriptors after its call to wait.

This concluded all the requirements for the shell apart from the ability to handle piping. Up until now, the recursive runCommand function was only called once for each command. For piping, this number will increase to be the number of token groups specified by the original user inputted command. As mentioned before, the last token group is taken from the end of the token groups vector and this is command is what is interpreted first. The first thing that needs to be handled for pipes is to determine where in the command we are currently looking. This is handled by an if statement based on the pipe counter in order to determine which file descriptor needs to be overwritten. The pipe counter will always be equal to 2 plus the read pipe end to be overwritten and 1 minus the write pipe end to be overwritten. Therefore, if the pipe counter is zero, we know that we only need to overwrite the write end, and if the pipe counter is equal to its maximum value, we only need to overwrite the read end. Otherwise, we are somewhere in the middle and we should overwrite both the read and write ends corresponding to what value the pipe counter is. After all these calls to dup2 in order to overwrite the file descriptors, the runCommand is recursively called, decreasing the pipe counter by 2 and passing it to the function in order to keep track of the appropriate file descriptors to overwrite. The function execvp() is then called for each of the piped commands. After this, the parent closes all the file descriptors after waiting for each command to execute.