Run executables without arguments
Executables can be run from the prompt by absolute path or by name.
"$ sTest" or "$ /home/dan/sTest"
In the case of calling an executable by name, if the executable is not found in the working directory, the program will attempt to find the program in one of the PATH directories. If this is not successful, the prompt will return a "file not found" error. Calling an executable with an invalid absolute path will also result in a "file not found" error. Executables are run using the execv() funciton and all executables are run in at least a separate forked process.

Run executables with arguments
Executables can be run with or without arguments. Executables with arguments can simply be run by typing the arguments after the executable name.
$ sTest arg1 arg2 … argn
Using the execv() function instead of execl() we were allowed to pass in an array of arguments to each executable.

***set* for HOME and PATH work properly**
In the program we set the HOME and path by using the setenv("NAME", value, overwrite); function. This creates an environment variable if one doesn't already exist. If one does exist and overwrite is 0, it won't change the variable. However, in the program I set overwrite to 1 in the calls so the environment variable will be overwritten by whatever value is inputted. These values can be retrieved by calling getenv("NAME"). In the mail while loop we check to see if the input string starts with $ set, and if it does we call our set() function. This function then detects which environment variable to set and sets it to the new value using setenv("NAME",value,1).

exit and quit work properly
The input strings $ exit ,$ quit,and $ q all cause the program to call terminate(); Which sets *bool running* to false, and this will cause it to exit the while loop. When we leave the while loop it calls exit(0) which exits the program.

***cd* (with and without arguments) and *pwd* works properly**
$ cd [arg] appends the specified directory to the current working path:
meh:~/home/dan$ cd Downloads will change the working directory to: /home/dan/Downloads
$ cd with no arguments will set the working directory to the HOME environment variable.
$ pwd will print the current working directory
meh:~/home/dan$ pwd
/home/dan <--printed working directory

PATH works properly. Give error messages when the executable is not found
PATH is originally set to the pre-existing PATH. When we call set in our programs input it then calls the set() function. This sets the environment variable using setenv(). In our code we parse the PATH, append the executable that we want to run. Check if it exists. If it exists, we return that string as our argument[0] to run using execv. If it doesn't exist, we check if it is in the next directory within the path. If it is not in the working directory or any of the path directories then we return an "does not exist" error.

Child process inheritance (inherit environment variable)
$PATH and $HOME are set as environment variables. By default, forked processes (child processes) inherit a duplicate of the parent's environment. We can access any of the environment variables that the child inherited from the parent using getenv("NAME").

Allow background/foreground execution (&)
The flag (&) to run a process in the foreground or background is detected in the get_command function. This sets a flag in the command_t struct (Bool execBg) to to true to execute the process in the background and to false to execute in the foreground. Note: "&" should be treated like an argument meaning that there should be a whitespace character between the "&" and the last argument.Additionally, if a command is run in the background, the process ID is printed for the user and the user is returned to the quash prompt. When an executable is run in the foreground, the parent process waits for the child executable to return before returning to the prompt. However when executables are run in the background, the parent process spawns the child forks, stores the child process ID into a linked list structure, and then immediately returns to the prompt. When background child processes return, the SIGCHILD signal is caught by a handler function which reaps the child and the process ID is expunged from the linked list structure.

Printing/reporting of background processes, (including the jobs command)
When a background process is begun, the parent stores the child process id into a linked list structure. When a background process exits, the SIGCHLD signal is caught by a handler and the process id is printed for the user with a message indicating the successful reaping of the child. Additionally, upon successful reaping, the child's process ID is removed from the linked list structure.
The "$ jobs" command is used to read the linked list structure and will print a list of all currently running background processes to the terminal with their assigned job IDs and the name of their executable.

Allow file redirection (> and <)
Input(<): Inputting from a file reads a file by opening the specified file using fopen(). I then take the first word of the input as argument[0], read in a line from the file using fgets(), append the input from the file as arguments to the given command, fork a child process off of the child process we are currently inside, and execute the command. The parent child process waits for its child to return then calls the next command with arguments from the file. This repeats for all the lines in the file. These are executed sequentially, but since we are already inside a child process, these sequential commands can still be called in the background of the main parent.

Output(>): Writing output from executables to a file is handled by reading a file name from the cmd structure, setting up a filestream, and utilizing dup2() to redirect the executables output to the filestream. This action creates a new file if none

exist and appends the executable data to the end of the file as opposed to overwriting the destination file. Additionally, like any other execution, this execution is performed in a forked process.

Allow pipes ($ ls | wc | more)
The “|” operator can be used to performing piping operations between separate functions.
“$ find | head | sort” returns the head of the find operation and then sorts them alphabetically.One or multiple pipes can be used in this fashion.
All occurrences of piping is handled in a special executable which begins by running the command through an unholy piped commands parser. In the case of the following command “$ find | head | sort”. find, head, and sort would be separated into their own command structs and placed into an array to be returned to the pipe engine.After the commands are separated into the command array structure, they enter the pipe engine which spawns and links pipes dynamically determined by the size of the command array

kill command (kill SIGNUM JOBID)
The kill command in quash allows you to specify the job number of which child process you want to kill it.
Inside the program it looks to see if the inputted command is $ kill and calls the killChild() function in the program. The killChild() function calls the search_by_job_id(int job_id) that is part of our linked list. This function returns the PID if the job exists in the list and 0 if it doesn’t. In killChild() 0 is received from search_by_job_id() then there is an error message saying that the job was not found. Otherwise it calls kill(PID, SIGNUM) which sends the signal to the child process specified by the PID.

combination of pipe, redirection, and background execution ($ wc < in.txt > out.txt &)
To perform the combination executions, we took care to design our code with modularity in mind. The command_t structure was expanded to contain all the necessary data for these operations and was passed between them like a token. The pipe handler, background execution, and io redirection functionalities were written as discrete functions which could all be called by higher level functions to determine the path of execution determined by the original command line input. Parsers determine which operations need to be performed and call the functions and execute commands in the appropriate order. Combination functions like this are treated as a single job with a single overarching jobid for all subroutines that may occur.

debugging
We debugged by inputting the required commands and seeing if they ran as expected. We used printf() to see variables that were otherwise not printed to the terminal output. These debugging printf() calls were removed from the code later once we verified all the commands were working properly and setting variables to expected values.
We also used a small function that was able to take in an arbitrary number of arguments that slept for a few seconds before printing to the terminal how many arguments it had. This was used for testing background applications and the command “$find | head | sort” to test and debug our piping functions. To test the input from file operations (<) we wrote a small file with commands to execute through quash for the “./quash < input.txt” command. For “command < input.txt” another small file was written that contained several lines of arguments to run the command with.