My program is a terminal emulator written in C++ that follows the workflow described in the assignment guide as closely as possible. It reads commands in commands.txt, and executes them concurrently one by one, printing the desired outputs and creating new files if necessary. At each step, it prints information about the command to parse.txt. command.txt is read and all its contents are parsed into a vector of vector of vector of strings in a structured manner before any command execution, with a pre-defined index for each part of each command, inferred intuitively by the functions I defined. All suggestions about what functionalities should be used were followed: pipes were used to enable inter-process communication, commands were handled by one process to enable threading, file streams were used to read and process pipes properly, and dup2() was used to rewire inputs and outputs. I tried to make my code as modular as possible, with various functions for basic tasks, although the handling of commands takes place in main(). Below is a description of how my program handles the problems mentioned in the guide.

First of all, it is not true that “-” is a special character that should be handled separately. execvp() can correctly execute [option] arguments that contain a dash without a problem. However, there are indeed special characters that execvp() and the built-in shell interpret differently. These are quotation marks (namely the single and double quotation marks). Inputs that contain these marks are completely valid – in fact, they are necessary when the input contains spaces to specify that the input is actually one argument (e.g. grep “I am a danger” input1.txt or, equivalently, grep ‘I am a danger’ input1.txt). While the shell successfully executes such commands, execvp fails, presumably containing the quotation marks as part of the input rather than considering them as signs. My program handles these characters by first identifying whether the input string starts with any kind of quotation mark – since the commands are assumed to be correct, such inputs also have to end with another quotation mark, so that condition is not checked. It then escapes them by simply slicing the inputs in a way that excludes the quotation marks.

In the case of output redirectioning, my program does the following (in the child process):

1. Opening an output file with the provided name and the relevant permissions.
2. Copying the file descriptor of the output to STDOUT via dup2(), essentially directing terminal output to the file.
3. Closing the original output file.
4. Executing the command, which now posts its output to the output file rather than the terminal.

Using a pipe per command, input redirectioning is performed as follows:

1. Creating a pipe.
2. Forking.
   * In the child process:
     1. Opening the input file with the provided name and the relevant permissions.
     2. Copying the file descriptor of the input to STDIN via dup2(), essentially directing the input file to the command (dealt by the parent process).
     3. Closing the input file.
     4. Closing the read end of the pipe.
     5. Copying the file descriptor of the write end of the pipe to STDOUT via dup2(), essentially ensuring that output will be written to the pipe instead of the terminal.
     6. Closing the write end of the pipe.
   * In the parent process:
     1. Closing the write end of the pipe.
     2. Running a listener thread (with the file descriptor of the read end and a pointer to the global lock given as parameters) that reads and prints the contents of the pipe.
     3. Either adding the child process ID and the thread ID to the queues of running jobs (if the command is running in the background), or waiting for the thread and the child process to terminate (if the command is not running in the background).

For background jobs, two queues in the form of vectors are maintained: pids and tids. pids keeps track of initiated child processes, whereas tids keeps track of initiated threads. If a job is a background job, the parent process adds its child process to pids (if there is output redirectioning) or adds its child process to pids and adds the thread ran by the child process to tids (if there is either no redirectioning or just input redirectioning). It then proceeds to the execution of the next command. On the other hand, if the job is not a background job, the parent process simply waits for every process in pids and every thread in tids to end via waitpid() and pthread\_join() respectively. It then clears both pids and tids. If the command is “wait”, the procedure is the same. To make sure that all processes and threads completed execution, the program waits for all processes in pids and threads in tids to end (implicitly executing wait) right before completely terminating, so that the final output always contains the outputs of commands run in the background near the end of commands.txt. Before I implemented this, the final output was variable when the last line of commands.txt included a background job – the program sometimes terminated before it was executed and sometimes did not.

Listener threads use the function listen, which takes a struct called thread\_args as input. This struct contains a pointer to the global lock/mutex, and the file descriptor of the read end of a pipe. A variable named “flag” keeps track of whether the mutex is held or not (and is marked as true in the beginning), and while it is true, the thread tries to lock the global lock. If it can do so, it prints 10 dashes, reads the pipe, and prints the content of the pipe until it is completely consumed. Afterwards, it prints 10 dashes, marks the flag as false, unlocks the mutex, closes the pipe and returns NULL. As can be inferred from this description, there is a single global lock that all threads lock and unlock to achieve mutual exclusion. Threads are run only when there is either no redirectioning or just input redirectioning in the command (as described in the program structure under 1.2 in the guide). Printing information about the command to parse.txt predates any command execution and thus any thread.