Read Me

Operating System:
Coded with C++ 11 on XCode and tested on Mac OS Sierra.

Assumptions:
Program will only be used on MacOS.
Input and output directories are entered correctly: eg /Users/Wesley/Desktop/input/
When processing commands there will not be miscellanous dashes before the input command or periods after the .csv extension.

Please ensure when entering the directory that there is a slash at the end and beginning of the arguement.

How To Run:
To execute the program on MacOS. Open the terminal. Navigate to where the program is stored is using the cd command. Enter ./output

To add any command line arguments enter ./output followed by -h or -init.

First Time Setup and Initialisation:

For every initialisation you will be required to enter your user_name. This is done so that the program can find or create a config file. The config file stores read and write path directories which you will be required to setup if it can't find a config file.

Config File:
The config file is called jumo_accountancy_config.txt and will be saved to your Desktop. I felt this would be more convenient for the tester for both inspecting and deleting purposes. Before executing the program I suggest creating a directory for the input and output of the program.

Input Directory:
The input directory should only have .csv files in the exact same format as was given.

Output Directory:
The output directory will only store the output of the program and so can have anything in it. Please note that any files with the same names as the input files in the output directory may be overwritten at runtime.

Starting the program:
The program should be executed from the command line as there is no GUI. It can accept two arguements as input.

-h or -help - prints some help information on the program
-init - reinitialises the config file.

Running the program:
When the program is running you can enter in various inputs. The most basic ones are detailed below, you can find additional ones by entering -h.

-h - prints a list of commands (different to the command line help)
* - processes all input files to the output directory
<filename>.csv - processes the selected file. Dont input the diamond brackets.
-q or -quit - exits the program.

Choice of language:
I opted to use C++ for this project as I enjoy coding in it a lot and i have been using it a lot quite frequently. At first I thought it might be better to do it in Java or C# however there would be a brief acclimatisation period for that and I wasn't sure if I would have enough time to complete the project to the scope I had set.

Plugins / 3rd Party Libraries:

Read Me

None. I believed it would be best to code everything from vanilla C++ for this as I wasn't sure what environments the tester would be running and I also believed it would be more convenient for the tester.

Scope of the Project:
This part will be quite lengthy as I will detail some of the considerations I made and will make reference to specific classes and methods.

Initial Scope:

The initial scope included at bare minimum the assignment specifications. Beyond that I tried to think of actually making use of this kind of code on a day to day basis or regular basis and the kind of requirements I would have. One of the most obvious is low maintenance and convenience, if the program is run once every two months then the code should be very simple, small and print as much as output as is reasonable to assist the user, it should also include various print functions to assist the user as much as possible. To try and keep it low maintenance I opted to create a config file and input and output directory, this would enable the user to come in and begin working where they left off last. There would also be various functions like printing the current input and output directories, listing all files in the input directory, processing all files at once or selective processing and totaling, counting and averaging. In the real implementation of the program I would have the config file saved in the Library directory however for inspecting and deleting convenience I opted to have the config file placed on the users Desktop where it could be easily viewed and managed.

To assist with the number of print functions I would be making I chose to create a singleton messaging class which would be the centre of all print functions. Anytime any function had to be printed it would be go here. I felt this would assist with future scaling and modifications as there would be a single point of reference to go to for all message CRUD needs. The singleton pattern would also enable ease of use throughout the program.

I created the ICurrency as an interface to assist with any future growth needs in the program, specifically handling different currencies and currency conversions. I opted to follow the Strategy and Facade Pattern for this as it would enable additions to be made to the currency handling capability without impacting or requiring any changes elsewhere in the program. The Facade pattern would come in through the currency conversion method, as it is currently implemented the conversion is simply one to one, however using the facade pattern you could create other methods that would retrieve the realtime currency conversion rate and apply it.

The input manager class handles all the inputs at runtime and passes them to other methods that handle the actual dirty work. The main problem with this class is in the processInput method. This method suffers from the large if else tree problem and is the least scaling friendly. In future updates of the code I would opt to change this to a map that takes in a string key and points to a function.

The argsProcess class is the shortest lived class. To save some memory this class is only initialised at startup and deleted once the config file has been setup.

The commandProcessor class handles all the dirty work of summing, counting and averaging the csv files. I wasn't entirely sure how many lines each csv file would contain nor how many files could feature in a directory. To overcome this I opted to parse the lines and files one at a time and save them to a local variable. The local variable could then be used however is needed. An area for some optimisation in this would be caching recently read files somewhere. Currently the read file and readline methods do not cache anything and whenever a new command is given by the user all the files and lines must be reread. In future versions if there was a need for additional calculations such as standard deviation, interest rate or default rate calculations a few changes could be made through implementing a few more variables or alternatively reimplement various methods through an interface and apply a pattern to allow greater flexibility.