**Part A**

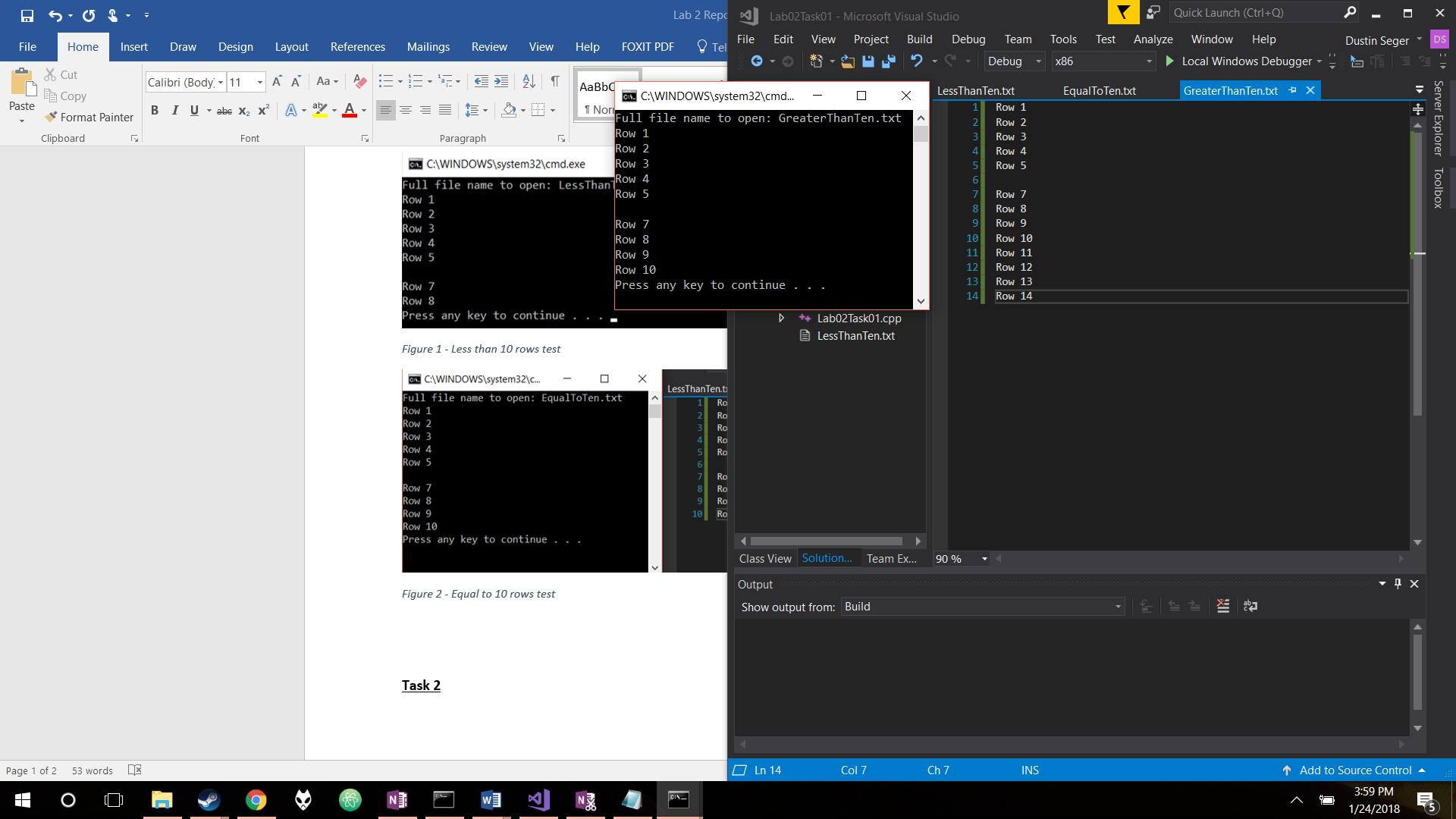


Figure 1 – Task 1 output with file that has greater than 10 rows

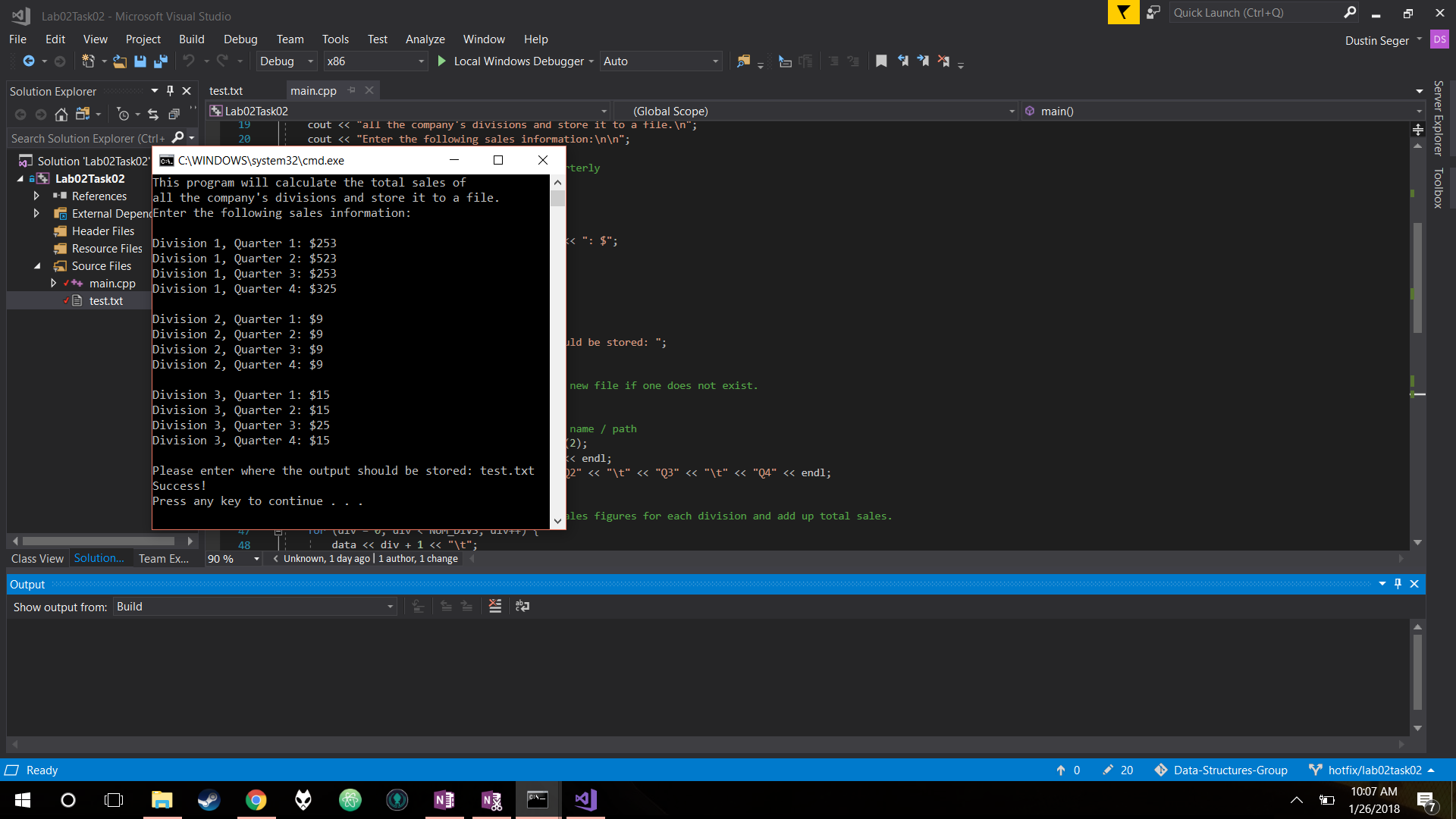
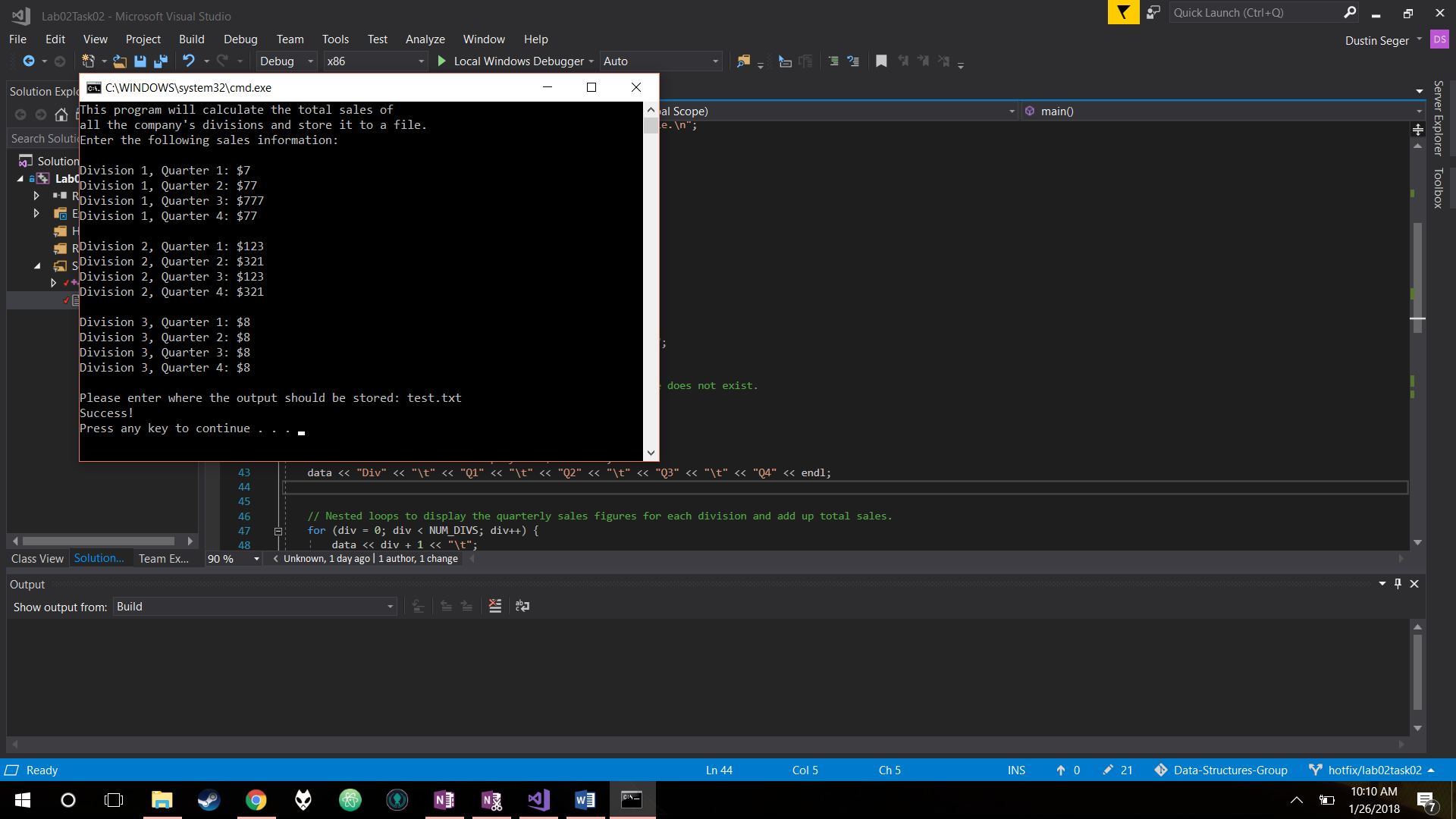
 

Figure 2 – Task 2 writing to new file output Figure 3 – Task 2 appending to existing file  
 Resulting output file included in lab submission.

In this assignment, the concepts of classes and file streams were explored. Classes are a way of storing and organizing data in order to follow proper Object-Oriented Programming. This was explored in Task 3 by modifying a code with a struct to use a class stored in separate files instead. File streams allow for data to be read from and written to text files. This was explored in each task through modifications to the code to read specified data from or send it to files. Reading from files can be seen in Figure 1 which displays the contents of the text file and the resulting code output from Task 1. Writing to files can be seen in Figures 2 and 3 which show the output of Task 2. Figure 2 shows the sending of data to a file that does not exist, and Figure 3 shows the output of appending to the same file we just created. The resulting file consults both the new file and existing file tests and is included in our lab submission.

Classes and file streams are especially important in this course and a career in CS. The concept of classes is fundamental to CS as a whole. Classes allow for data to be reusable and clear to its intent. By using classes, a developer can create custom objects that directly cater to their needs and abstract out what the classes do for other users and fellow developers. This is important for readability with coworkers or classmates that may also be working on the same project. It also allows for control over the custom object and what the user should be able to modify, thus securing and verifying your data if need be. File streams are useful primarily for reading and documenting large amounts of data directly to a file that others can see and read. This could be useful in a workplace or classroom setting when dealing with data that would be tedious to enter in manually. File streams can also be useful if used for testing purposes, such as putting some test data into a file that is read in the code run rather than manually inputting the data each time you run a test. This saves time and increases efficiency, especially in large projects that require a long time to develop.

**Part B**

Each task required the use of access flags within stream variables to open files, and choosing the correct flags was crucial to each task’s completion. For Task 1, the program only needed to read lines from a text file, so the access flag “ios::in” was used. This was chosen because it does exactly is needed by simply reading data from a file or failing out if the file does not exist. This failure was then accessed later to gracefully handle the error. In Task 2, appending to a file was required. Naturally, “ios::out” was initially considered for use, but this deletes existing data since it truncates as in “ios::trunc” by default. Thus, “ios::app” was used since it appends to files and creates new ones if not found, which was also required in this task.Our final Task 3 code does not use any access flags because ifstream was used, which defaults to “ios::in” behavior. Since this flag fits our purposes as it did in Task 1, a change was not needed.

**Part C**

Task 3 required implantation of a class structure and a human readable input file. The class, Product, was designed such that all member functions were public, and all member variables were private. The reasoning behind this was that there has to be diverse functionality available to all, but we did not want the user to have the ability to change variables on their own. By keeping the variables private, there is greater data integrity, and we can prevent invalid data.

As for the human readable file, this was designed such that it is very straightforward for the user to read, but also very easy to use for the programmer. By labeling the ID, Units, and Price of each product followed by a blank line between products, it is easy to see where one product begins and another ends. This format was made easy to use from a programming perspective by adding colons after the names of the members and before the numbers. This created an easy delimiter to split up strings and store the data into an object.

**Part D**

The testing of part 3 was a little more complex than expected. After seeing the requirements, a flag was created during the initial code writing to detect when the input file had less products than expected. The code was also initially set up such that the loop that reads the file only runs for the product count. These methods very nearly made the program work perfectly on the first time. However, the preemptive fix for having less products than expected was a bit shortsighted. When the code ran in these cases, it was displaying the default Product instance of all 0’s. This default made it easily identifiable in the code where the problem was. What this experience taught me is that it’s not necessarily important to make perfect code on the first try. If you plan around the requirements, you can fulfill most of them with no issue, or at least make resulting bugs easy to identify. Setting default constructors to something easily identifiable or using some descriptive print statements can also make the debugging process much easier.

All files should be compiled with default windows settings for the gcc compiler.