Reflection:

The project that I have done is to develop a simulation of an assembly line with any number of stations. And each workstation holds a set of stock items. The line manager is used to move the customer orders along the line filling the orders at each station and making sure customer orders are being filled. Each customer order contains a list of items that need to be filled. And at each station, a deque of customer orders are being placed on the station to receive service. Order can either be completed or incomplete due to inventory. The project has included most of the topics that I have learned throughout the semester. Milestone instructions have provided clear and extremely helpful implementation orders, logic, and knowledge that helped me complete the project. The following is a brief explanation of the concepts that I have used and also the difficulties that I have faced while completing this project.

Milestone 1 – Utilities and Station Classes:

The first milestone includes two modules, Utilities, and Station. Utilities is a helper module that can parse string data from input files that can be used to set up and configure the assembly line. And Station is a module that is used to manage a single station on the assembly line, and each station handles a specific item for filling customer orders.

The Utilities module includes the method to extract a token from a string that begins at a certain position to its delimiter, and the next position can be updated which specifies the start position of the next extracted token. But if the delimiter cannot be found until the last character of the string has reached, then from the starting position next\_pos until the last character of the string will be extracted. I have used member functions inside of the string class such as find\_first\_of() to find character in a string, substr() to generate substring, and length() to return the length of the string. Also, I have used member constants npos to identify the end of the string. Also, Utilities contains the class variable m\_delimiter that is being shared by all the instances of Utilities objects. The private class member m\_delimiter can be set or returned from the class functions that are being implemented. The length of the token being extracted is being used for display purposes, and it can also be set or returned with the member functions inside of the Utilities class.

The Station module managed a single station that handled a specific item. This class contains all the station information which includes the item information that is being handled, such as the name of the item, description of the station, serial number, and the number of the items in the station (in stock). All of that information is being extracted from the file Station1 and Station2, each line from the file represents an item being handled in a station, and the file is read line by line and the object being created with the Station constructor. Each object contains inventory information about how many items we have available. With the help of Utilities class, we can extract the information from the string which is the argument of the one-argument constructor. And this string is the line being extracted from the file. I have used the functionalities inside of the Utilities class such as extractToken to extract each Station member variable from the string. The m\_widthField is being updated as the maximum length of the item name being changed each time, which is being used to align the output for item name across all the records being retrieved from the same file. Also, the number of items can be updated and retrieved as well.

Milestone 2 – CustomerOrder Class

CustomerOrder module contains all the functionalities for processing the customer orders as they are being moved from one station to another based on the item contained. For example, if a CustomerOrder needs one Dresser, if it is in the inventory, and move to "basket" / "track" and move the CustomerOrder to the next station, and so on. We can imagine as fill the item of the CustomerOrder as take the item from the Station and move to the "basket" / "track". Once all the items of the order have been filled, then the order is ready/completed for the customer. CustomerOrder class contains the information about the customers and the list of items in the order, also this class encapsulated another Item class that contains all the Item information. Just like the Station module, after the records are loaded from the file CustomerOrders.txt line by line, each CustomerOrder object will be created with its one argument constructor. And with the help of Utilities class like I have talked about in the Station module, information like customer name and order name can be extracted from the string. But the class also contains a dynamically allocated array of pointers of type Item\*, each element of the array points to a dynamically allocated object of type Item. So, when we try to extract the Item information from the file string, they need to be allocated dynamically, and each element of the array is pointing to an Item. For resizing, I have used std::copy to copy every item from the old array of Item\* to the new array Item\* (which is the member variable). The count for the items inside of the CustomerOrder can be incremented each time when a new item is created. Like in the Station module, the class variable m\_widthField is also implemented to be used to display the item name in the CustomerOrder with an updated specific field width being set at run time. Since the class contains a resource, that is why I have implied with Rule of Five. The CustomerOrder object cannot be copied and it can only perform move operation, as each Customer order is unique. As m\_lstItem contains a dynamically allocated array of pointers, also each pointer is pointing to a dynamically allocated object of type Item. That is why I have used iterator to make sure each item resource is being pointed to by the array of pointers being taken care of first, then the dynamically allocated array of pointers, to be able to move one CustomerOrder to another one, this is used to processing customer orders as they move from station to station along the assembly line; and it is also being implemented inside of the destructor to take care of the memory management. We also have a method inside of the CustomerOrder class to check if all the items in the customer orders have been filled with the isFilled() function, I have used the std::all\_of algorithm to check each m\_isFilled property of each element inside m\_lstItem. And if want to check all the items that match the item name in the customer orders have been filled, I have used the isItemFilled member function and std::all\_of algorithm again to check if the item exists in the order or all items have been filled. Also, the function fillItem is to fill item information if the item handled by the station match with one item in the customer order. Basically, this function is checking whether if the item exists in the inventory, to determine if the information can be filled or not. I have implemented all functions with mostly algorithms instead of for loop, to review the knowledge that I have learned in the second half of the semester.

However, right now the stations are not connected. Once a station receives a customer order but we do not know where or which station should the order go next. CustomerOrder contains items that needed to be handled through each station. That is why the stations need to be connected.

Milestone 3 – Workstation and LineManager module

To deal with the situations that the stations need to be connected in order to fill a CustomerOrder, the file AssemblyLine.txt is included to introduce us to how the stations are connected with each other. Inside it contains the link from one station to another, which is the pointer to the next Workstation. And it will be connected as a link until reaching the last station, and the last station will point to null. The stations are then being connected with a start and end as a list. linkStations() method can reorder the elements of the list, by using getNextStation() to get the address of the next Workstation, we are able to know how the Workstations are being connected. Inside of the Workstation module, I have implemented three collections that are global variables that will store customer orders, pending as the orders to be placed onto the assembly line at the first station; completed as the orders have been removed from the last station, and have been filled; incomplete as the orders have been removed from the last station and could not be filled. When we load from the file CustomerOrders, first it will add to the pending collection, it will be added to the collection as line by line from the CistomerOrder.txt file. Inside of the run() of LineManager class, first, we need to check if the pending collection contains customer orders, if there is, we are taking the item out and moving to the first station, and we also need to maintain a pointer to the first station. In order to add a customer order to a station, we can use the += operator that is being implemented in the Workstation class to add an order to a Workstation. After the first order is added to the assembly line, we can remove the CustomerOrder from the pending container. At each station, I have used algorithm std::for\_each to do fill() operation for each element inside of the vector activeLine with type Workstation\*. If the fill() operation does not succeed, no operation will happen at the station. After fill() operation, we would need to move the CustomerOrder to the next station, if the order requires no more service at the station, we are moving it to the next station, but if there is no next station in the assembly line, then the order is moved to the completed queue. Also, if the order cannot be filled which means that the item in the station does not have enough inventory, then we move the order to the next station, but if there is no next station in the assembly line, then the order will be moved into the incomplete queue. That is why I have used for\_each again to iterate over the activeLine, and check each Workstation\* element with attemptToMoveOrder() function to attempt to move the order to the front of the queue to the next station in the assembly line. Afterward, by checking the total of the amount of CustomerOrder in the completed collection and the amount of CustomerOrder in incomplete collection with the m\_cntCustomer (amount of all customer orders), we can find out if all the orders have been filled. Each CustomerOrder that is inside of the Pending order will be taken out one by one, and perform all the actions above.