**Project Overview**

The purpose of this term project is to put your C++ Object Oriented skills to practice by developing a simulated Assembly Line.

The project simulates an assembly line that fills customer orders from inventory. Each customer order consists of a list of items that need to be filled. The line consists of a set of stations. Each station holds an inventory of items for filling customer orders and a queue of orders to be filled. Each station fills the next order in the queue if that order requests its item and that item is still in stock. A line manager moves the customer orders from station to station until all orders have been processed. Any station that has used all of the items in stock cannot fill any more orders. Orders become incomplete due to a lack of inventory at one or more stations. At the end of all processing, the line manager lists the completed orders and the orders that are incomplete.

The project is broken down into 3 milestones to help guide you through the process of implementing, debugging and execution of a more complex application than a regular lab/workshop.

**Project Deadlines**

The deadlines for the project's milestones are on Sunday at midnight (see the schedule below).

| **Milestone** | **Date** |
| --- | --- |
| #1 | Nov 14 |
| #2 | Nov 21 |
| #3 | Nov 28 |

All files that you submit (\*.h and \*.cpp) should contain the following comment at the top (with your information filled):

// Name:  
// Seneca Student ID:  
// Seneca email:  
// Date of completion:  
//  
// I confirm that I am the only author of this file  
//   and the content was created entirely by me.

**Milestone #3** can have an extension until December 08 2020, with a penalty of 10% for each day that the submission is late. After December 08 2020, the submission process is considered closed.

* submissions received on Nov 29 2020 can receive **max 90%**;
* submissions received on Nov 30 2020 can receive **max 80%**;
* submissions received on Dec 01 2020 can receive **max 70%**;
* submissions received on Dec 02 2020 can receive **max 60%**;
* submissions received on Dec 03 2020 can receive **max 50%**;
* submissions received on Dec 04 2020 can receive **max 40%**;
* submissions received on Dec 05 2020 can receive **max 30%**;
* submissions received on Dec 06 2020 can receive **max 20%**;
* submissions received on Dec 07 2020 can receive **max 10%**;
* submissions received on Dec 08 2020 can receive **max 0%**;

**The project is mandatory!** Students that don't submit a complete working project, cannot pass this course. A project is considered complete if all three milestones have been submitted by the deadline and the implementation follows the requirements.

**Mil**esto**ne 1**

## Utilities and Station classes

This milestone will implement and test the Utilities and Station classes.

* Utilities class is an object used to support the parsing of input files to setup and configure the assembly line simulation.
* Station class encapsulates the information about a given station on the assembly line that fills customer orders.

The specifications of the classes are defined below.

## Utilities Class

Parsing string data from input files is performed the same way for all objects within the system. The Utilities type provides the basic functionality required for all objects in the assembly line system.

Each Utilities object holds the following information:

* m\_widthField -- specifying the length of the token extracted; used for display purpose later; default value is 1.
* m\_delimiter -- separates the tokens from a given std::string. All Utilities objects **share the same delimiter**.

The member functions of the Utilities class include:

* void setFieldWidth(size\_t newWidth) -- sets the field width of the current object to the value of the parameter
* size\_t getFieldWidth() const -- returns the field width of the current object
* std::string extractToken(const std::string& str, size\_t& next\_pos, bool& more) -- extracts tokens from the first parameter
  + This function uses the delimiter to extract the next token from str starting at next\_pos.
  + If successful, it returns the extracted token found and sets more to true; false otherwise.
  + This function reports an exception if there are two delimiters with no token between them.
  + This function updates the current object's m\_widthField data member if its current value is less than the size of the token extracted.
  + **Note:** str represents a single line read from an input file
* static void setDelimiter(char newDelimiter) -- sets the delimiter for this class to the character received
* static char getDelimiter() -- returns the delimiter character.

## Station Class

An Station object manages a single station on the assembly line. Each station handles a single item in the customer order.

Each Station holds the following information:

* the id of a station (integer)
* the name of the item handled that the station (array of characters)
* the description of the station (array of characters)
* the next serial number to be assigned to an item at this station (integer)
* the number of current items left in stock (integer)

***Static Data***

* m\_widthField -- the maximum characters required to print to screen the *item name*, *serial number* and *quantity* for any object of type Station. Initial value is 0.
* id\_generator -- a variable used to generate IDs for new instances of type Station. Every time a new instance is created, the current value of the id\_generator is stored in the instance, and id\_generator is incremented. Initial value is 0.

***Public Functions***

* custom constructor
  + upon instantiation, an Station object receives a reference to an unmodifiable std::string. This string contains a single record (one line) that has been retrieved from the input file specified by the user.
  + the constructor uses an Utilities object (created local in the function) to extract each token from the record and populates the Station object accordingly.
  + **Note:** You can assume that a record from the file contains 4 fields separated by a delimiter, in the following order:
    - name of the item
    - starting serial number
    - quantity in stock
    - description
  + the delimiter of the tokens is a single character, specified by the client and stored into the Utilities object.
  + once the constructor has extracted *name*, *starting serial number*, and *quantity* from the record, it updates Station::m\_widthField to the maximum value of Station::m\_widthField and Utilities::m\_widthField.
  + **Note:** the display(...) member function uses this field width to align the output across all of the records retrieved from the file.
* const std::string& getItemName() const - returns the name of the current Station object
* unsigned int getNextSerialNumber() – returns the next serial number to be used on the assembly line and increments m\_serialNumber
* unsigned int getQuantity() const – returns the remaining quantity of the current Station object
* void updateQuantity() – subtracts 1 from the available quantity; should not go below 0.
* void display(std::ostream& os, bool full) const -- inserts the content of the current object into first parameter.
  + if the second parameter is false, this function inserts only the ID, name and serial number in the format: [ID] Item: NAME [SERIAL]
  + if the second parameter if true, this function uses the following format: [ID] Item NAME [SERIAL] Quantity: QTY Description: DESCRIPTION
  + ID field will use 3 characters, the NAME and QTY fields will use m\_widthField characters, serial number field will use 6 characters; the DESCRIPTION has no formatting options
  + this function will terminate the printed message with an endline

**Tester Module**

The tester module and input files have been added to the repository. Do not modify any of them.

**Sample Output**

Look in the file ms1\_output.txt for the command line necessary to start the application and the expected output.

**Submission**

Before the due date, upload to your matrix account the following files:

* Utilities.h
* Utilities.cpp
* Station.h
* Station.cpp

From a command prompt, execute the following command:

~profname.proflastname/submit 345\_ms1

and follow the instructions.

**A successful submission does not guarantee full credit!**

**Milestone #2**

This milestone will implement and test the CustomerOrder class.

The CustomerOrder module contains all the functionality for handling customer orders as they move along the assembly line. As the line manager moves an order along the assembly line, the station where that order currently rests fills a request for an item of that station, if there is any such request. Once the order has reached the end of the line, the order is either completed or incomplete. One cause of incompleteness is the lack of sufficient items in stock at a station.

**CustomerOrder class**

A CustomerOrder object manages a single order on the assembly line and contains the following information:

* std::string m\_name – the name of the customer (e.g., John, Sara, etc)
* std::string m\_product – the name of the product being assembled (e.g., Desktop, Laptop, etc)
* unsigned int m\_cntItem – a count of the number of items for the customer's order
* Item\*\* m\_lstItem – a dynamically allocated array of pointers. Each element of the array is a dynamically allocated object of type Item (see below). **This is the resource** that your class must manage.
* static size\_t m\_widthField – the maximum width of a field, used for display purposes

The member functions of the CustomerOrder class include:

* default constructor
* a custom constructor that takes as a parameter a reference to a string containing a single record from the input file. This constructor uses an Utilities object to extract the tokens and populate the current instance. After the extraction is finished, will update CustomerOrder::m\_widthField if the value stored there is smaller than the value stored in Utilities::m\_widthField.
  + fields in the record are (separated by a delimiter):
    - Customer Name
    - Order Name
    - the list of items making up the order (at least one item)
* a CustomerOrder object should not allow copy operations. The copy constructor should throw an exception if called and the copy operator= should be deleted.
* move constructor. This constructor should "promise" that it doesn't throw exceptions. Use the noexcept keyword in the prototype.
* move assignment operator. This operator should "promise" that it doesn't throw exceptions. Use the noexcept keyword in the prototype.
* a destructor
* bool isOrderFilled() const – returns true if all the items in the order have been filled; false otherwise
* bool isItemFilled(const std::string& itemName) const – returns the ItemInfo::m\_fillState of the item specified as a parameter. If the item doesn't exist in the order, return true.
* void fillItem(Station& station, std::ostream& os) – fills the item in the current order that corresponds to the station passed into the function.
  + if the order doesn't contain the item handled at the station, this function does nothing
  + if the order contains items handled at the station, and the inventory contains at least one element, then this function subtracts 1 from the inventory and updates Item::m\_serialNumber and Item::m\_isFilled. Also it prints the message Filled NAME, PRODUCT [ITEM\_NAME].
  + if the order contains items handled at the station, and the inventory is empty, then this function prints the message Unable to fill NAME, PRODUCT [ITEM\_NAME].
  + all messages printed should be terminated by an endline
* void display(std::ostream& os) const – displays the state of the current object in the format (see the sample output for details)

CUSTOMER\_NAME - PRODUCT  
[SERIAL] ITEM\_NAME - STATUS  
[SERIAL] ITEM\_NAME - STATUS  
...

* + SERIAL - a field of width 6
  + ITEM\_NAME - a field of size m\_widthField
  + STATUS is either FILLED or MISSING
  + you will have to use IO manipulators for the output.

Also, add to the header CustomerOrder.h, the following structure:

struct Item  
{  
std::string m\_itemName;  
unsigned int m\_serialNumber = 0;  
bool m\_isFilled = false;  
​  
Item(const std::string& src) : m\_itemName(src) {};  
};

**Tester Module**

The tester module and input files have been added to the repository. Do not modify any of them.

**Sample Output**

Look in the file ms2\_output.txt for the command line necessary to start the application and the expected output.

**Submission**

Before the due date, upload to your matrix account the following files:

* Utilities.h
* Utilities.cpp
* Station.h
* Station.cpp
* CustomerOrder.h
* CustomerOrder.cpp

From a command prompt, execute the following command:

~profname.proflastname/submit 345\_ms2

and follow the instructions.

**A successful submission does not guarantee full credit!**

# Milestone 3

This milestone will implement and test the Workstation and LineManager classes.

The Workstation class is a simulation of a station on the assembly line and contains all the functionality for filling customer orders with items. Each Workstation is-a Station (use inheritance) and is responsible for a particular Item.

The LineManager class is responsible for the execution and movement of CustomerOrders along the assembly line (from start to finish). The line manager moves orders along the assembly line one step at a time. At each step, each station fills one order. The manager moves orders that are ready from station to station. Once an order has reached the end of the line, it is either completed or is incomplete. An order can be incomplete due to insufficient items in stock to cover its requests.

**Workstation Class**

A Workstation object manages a single Item on the assembly line.

Workstation inherits from class Station and contains the following additional information:

* m\_orders – is a double ended queue with new CustomerOrders coming in one side and exiting out the other once filled.
* m\_pNextStation – a pointer to the next station on the assembly line

The member functions of the Workstation class include:

* a custom constructor -- upon instantiation, a Workstation object receives a reference to an unmodifiable std::string. This string contains a single record (one line) that has been retrieved from the input file specified by the user to be used for Station instantiation.
  + this constructor will also set the m\_pNextStation to a safe state.
* A Workstation object represents a single location on the assembly line for filling items into customer orders. Therefore, a Workstation object cannot be copied or moved. You must make sure this capability has been deleted from your Workstation definition.
* void runProcess(std::ostream&) – runs a single cycle of the assembly line for the current station.
  + If there are CustomerOrders in the queue, this function fills the order in the front at the current Workstation.
* bool moveOrder() – if the order at the front of the queue doesn't require service at the current station, move it to the next station on the assembly line and return true. Otherwise, do nothing and return false. If the queue is empty, return false.
* void setNextStation(Station& station) – stores the provided Station object's reference into the m\_pNextStation pointer.
* const Workstation\* getNextStation() const – return the next station on the assembly line
* bool getIfCompleted(CustomerOrder& order) – if the order at the front of the queue is completed, this function removes from the queue, places it in the parameter and returns true; otherwise returns false. If the CustomerOrder queue is empty, returns false.
* void display(std::ostream&) – writes the name of the Item this Workstation is responsible for into the parameter: ITEM\_NAME --> NEXT\_ITEM\_NAME
  + if m\_pNextWorkstation does not exist it writes: ITEM\_NAME --> END OF LINE.
  + the messages are terminated with an endline
* Workstation& operator+=(CustomerOrder&&) – moves the parameter onto the back of the CustomerOrder queue.

**LineManager class**

A LineManager object manages the entire assembly line and contains the following information:

* std::vector<Workstation\*> AssemblyLine – A container containing all the references of the Workstation objects on the assembly line
* std::deque<CustomerOrder> ToBeFilled – A queue of customer orders to be filled
* std::deque<CustomerOrder> Completed – A queue of customer orders completed
* unsigned int m\_cntCustomerOrder – The number of CustomerOrder objects the assembly line started with.

The member functions of the LineManager class include:

* custom constructor
  + with the following arguments:
    - a reference to an unmodifiable std::string. This string contains the filename specified by the user to be used for linking the assembly line objects (example: AssemblyLine.txt)
    - a reference to a std::vector<Workstation\*> that contains the addresses of all the Workstation objects created for the assembly line
    - a reference to a std::vector<CustomerOrder> that contains all the CustomerOrder objects to be filled
  + the constructor will:
    - read the records from file and setup all the m\_pNextStation references in the Workstation objects, linking each Workstation object together to form the assembly line
    - Move all the CustomerOrder objects onto the back of the ToBeFilled queue
    - Copy all the Workstation objects into the AssemblyLine container
* bool run(std::ostream& os) – this function performs **one** cycle of operations on the assembly line by doing the following:
  + Insert into os the iteration number (how many times this function has been called by the client; use only local variables to count) in the format Line Manager Iteration: COUNT<endl>
  + If there are any CustomerOrder objects in the ToBeFilled queue, move the one at the front of the queue onto the starting point of the AssemblyLine (you have to identify which station is the starting point of your assembly line). Only one order can be moved on the assembly line on each call to this function.
  + Loop through all stations on the assembly line and run one cycle of the station's process
  + Loop through all stations on the assembly line and move the CustomerOrder objects down the line.
    - **Hint:** completed orders should be moved into the Completed queue.
  + return true if all customer orders have been filled, otherwise returns false.
* void displayCompletedOrders(std::ostream& os) const -- displays all the orders that were completed
* void displayStations() const -- display all stations on the assembly line in the order they were received from the client
* void displayStationsSorted() const -- display all stations on the assembly line in the order they are linked.

**Tester Module**

The tester module and input files have been added to the repository. Do not modify any of them.

**Sample Output**

Look in the file ms3\_output.txt for the command line necessary to start the application and the expected output.

**Submission**

Create a **text** file named reflect.txt. Add any comments you wish to make.

Before the due date, upload to your matrix account the following files:

* Utilities.h
* Utilities.cpp
* Station.h
* Station.cpp
* CustomerOrder.h
* CustomerOrder.cpp
* Workstation.h
* Workstation.cpp
* LineManager.h
* LineManager.cpp
* reflect.txt

From a command prompt, execute the following command:

~profname.proflastname/submit 345\_ms3

and follow the instructions.

**A successful submission does not guarantee full credit!**