Final Project Overview

You are building an application in C++ that models an assembly line. What is an assembly line? It is basically how products are built in a factory. For example, a toy model car has different components. The wheel, the body etc. In a factory making these cars, there is a line of ‘processes’ that go into making the final product, the car. These are called ‘workstations’. One workstation is going to build part of the car, maybe the wheels. The 2nd workstation maybe builds the exterior of the car. The 3rd makes the seats etc etc. These workstations together are called an assembly line.

**Milestone 1**

In Milestone 1 you only build Utilities and Station modules.

Utilities Module

Utilities Module – basically a utility module that provides helper functions or classes that do some processing. In this project we have extractToken function for example.

The main purpose of this utility module is string parsing. You will work with files that have information about the assembly line. Or maybe information about your factory in some form. Maybe information about the workstations or information about customer orders too.

The utilities module basically allows you to parse the strings from the file. Your utilities module extracts parts of that string so that you can create objects of your different modules that you can create later on. You get the string as a parameter to extractToken, and you extract a single token from that string, that represents maybe an object that you need to create such as a station or workstation or customer order object.

Station Module

Station module represents 1 station of a factory that is dedicated to make 1 specific item of a product (ex car wheels). So you are building those 2 modules in milestone 1.

Use header guards, and all the best practices, rule of 5, deallocating memory etc.

**Milestone 1**

**Utilities and Station classes**

Milestone 1 implements and tests the Utilities and Station modules.

* The Utilities module supports the parsing of input files, which contain information used to setup and configure the assembly line.
* The Station module manages information about a station on the assembly line, which holds a specific item and fills customer orders.

The specifications of these two modules are defined below.

**Utilities Module**

Parsing string data from input files into tokens is performed uniformly for all objects within the simulation system. The Utilities type provides the basic functionality required for all objects in the system.

The Utilities class has the following structure:

***Instance Variable***

* m\_widthField -- specifies the length of the token extracted; used for display purposes; default value is 1.

***Class Variable***

* m\_delimiter -- separates the tokens in any given std::string object. All Utilities objects in the system **share the same delimiter**.

***Member Functions***

* void setFieldWidth(size\_t newWidth) -- sets the field width of the current object to the value of parameter newWidth
* 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 a token from string str referred to by the first parameter.

This function:

* + uses the delimiter to extract the next token from str starting at position next\_pos.
    - If successful, return a copy of the extracted token found (without spaces at the beginning/end), update next\_pos with the position of the next token, and set more to true (false otherwise).
  + reports an exception if a delimiter is found at next\_pos.
  + updates the current object's m\_widthField data member if its current value is less than the size of the token extracted.

**Note:** in this application, str represents a single line that has been read from an input file.

***Class Functions***

* static void setDelimiter(char newDelimiter) -- sets the delimiter for this class to the character received
* static char getDelimiter() -- returns the delimiter for this class.

**Station Module**

A Station object manages a single station on the assembly line. Each station handles a specific item for filling customer orders.

The Station class has the following structure:

***Instance Variables***

* the id of the station (integer)
* the name of the item handled by the station (string)
* the description of the station (string)
* the next serial number to be assigned to an item at this station (non-negative integer)
* the number of items currently in stock (non-negative integer)

***Class Variables***

* m\_widthField -- the maximum number of characters required to print to the screen the *item name* 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 that instance, and id\_generator is incremented. Initial value is 0.

***Public Functions***

* custom 1-argument constructor
  + upon instantiation, a 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.
  + this constructor uses a Utilities object (defined locally) to extract each token from the record and populates the Station object accordingly.
  + this constructor assumes that the string contains 4 fields separated by the delimiter, in the following order:
    - name of the item
    - starting serial number
    - quantity in stock
    - description
  + the token delimiter is a single character, specified by the client and previously stored into the Utilities class of objects.
  + this constructor extracts *name*, *starting serial number*, and *quantity* from the string first
  + before extracting *description*, 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 the records retrieved from the file.
* const std::string& getItemName() const - returns the name of the current Station object
* size\_t getNextSerialNumber() – returns the next serial number to be used on the assembly line and increments m\_serialNumber
* size\_t getQuantity() const – returns the remaining quantity of items in the Station object
* void updateQuantity() – subtracts 1 from the available quantity; should not drop below 0.
* void display(std::ostream& os, bool full) const -- inserts information about the current object into stream os.
  + if the second parameter is false, this function inserts only the ID, name, and serial number in the format: ID | NAME | SERIAL |
  + if the second parameter if true, this function inserts the information in the following format: ID | NAME | SERIAL | QUANTITY | DESCRIPTION
  + the ID field uses 3 characters, the NAME field uses m\_widthField characters, the QUANTITY field uses 4 characters, the SERIAL field uses 6 characters; the DESCRIPTION has no formatting options (see the sample output for other formatting options)
  + this function terminates the printed message with an endline

**Tester Module**

The tester module and the input files are included in 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!**

Some notes:

A static function is **a member function of a class that can be called even when an object of the class is not initialized**. A static function cannot access any variable of its class except for static variables. The 'this' pointer points to the object that invokes the function.

**Milestone #2**

Milestone 2 implements and tests the CustomerOrder module.

The CustomerOrder module contains all the functionality for processing customer orders as they move from Station to Station along the assembly line. The Station where a given order currently rests fills a request for one item of that station, if there is any such request.

**CustomerOrder Module**

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

The CustomerOrder class has the following structure:

***Item Definition***

struct Item

{

std::string m\_itemName;

size\_t m\_serialNumber{0};

bool m\_isFilled{false};

Item(const std::string& src) : m\_itemName(src) {};

};

***Instance Variables***

* 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)
* size\_t m\_cntItem – a count of the number of items in the customer's order
* Item\*\* m\_lstItem – a dynamically allocated array of pointers. Each element of the array points to a dynamically allocated object of type Item (see below). **This is the resource** that your class must manage.

***Class Variable***

* static size\_t m\_widthField – the maximum width of a field, used for display purposes

***Member Functions***

* default constructor
* a custom 1-argument constructor that takes a reference to an unmodifiable string. This constructor uses a local Utilities object to extract the tokens from the string and populate the current instance. The fields in the string are (separated by a delimiter):
  + Customer Name
  + Order Name
  + the list of items making up the order (at least one item)

After finishing extraction, this constructor updates CustomerOrder::m\_widthField if the current value is smaller than the value stored in Utilities::m\_widthField.

* a CustomerOrder object should not allow copy operations. The copy constructor should throw an exception if called and the copy operator= should be deleted.
* a move constructor. This constructor should "promise" that it doesn't throw exceptions. Use the noexcept keyword in the declaration and the definition.
* a move assignment operator. This operator should "promise" that it doesn't throw exceptions. Use the noexcept keyword in the declaration and the definition.
* 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 true if all items specified by itemName have been filled. If the item doesn't exist in the order, this query returns true.
* void fillItem(Station& station, std::ostream& os) – this modifier fills **one** item in the current order that the Station specified in the first parameter handles.
  + if the order doesn't contain the item handled, this function does nothing
  + if the order contains items handled, and the Station's inventory contains at least one item, this function fills the order with one single item. It subtracts 1 from the inventory and updates Item::m\_serialNumber and Item::m\_isFilled. It also prints the message Filled NAME, PRODUCT [ITEM\_NAME].
  + if the order contains items handled but unfilled, and the inventory is empty, this function prints the message Unable to fill NAME, PRODUCT [ITEM\_NAME].
  + all messages printed are terminated by an endline
* void display(std::ostream& os) const – this query 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 TO BE FILLED
  + you must use IO manipulators to format this output.

**Tester Module**

The tester module and input files are included in 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 2:

You will create a new class called customer order. A customer order says for example (I want a car that is red, and has 20 inch wheels). Each of those items will have a specific station dedicated to them. The item list will determine how the customer order goes through the stations.

For the constructor, you need to allocate memory for the lssItem array before you can add items to the array.

fillItem – fills one item

Milestone 3

**Milestone 3**

Milestone 3 implements and tests the Workstation and LineManager modules.

The LineManager module first configures the assembly line and then moves CustomerOrders along it (from start to finish). The LineManager object configures the Workstation objects identified by the user, and moves orders along the line one step at a time. A Workstation is a Station that the LineManager has activated on the user's request. At each step, every Workstation fills one item in a Customer Order, if possible. The manager moves orders from station to station. Once an order has reached the end of the line, it is either complete or incomplete. An order is incomplete if one or more stations had an insufficient number of items in stock to cover that order's requests.

**Workstation Module**

The Workstation module consists of three double-ended queues of CustomerOrder and the Workstation class. The queues (global variables) hold the orders at either end of the assembly line:

* g\_pending holds the orders to be placed onto the assembly line at the first station.
* g\_completed holds the orders that have been removed from the last station and have been completely filled.
* g\_incomplete holds the orders that have been removed from the last station and could not be filled completely.

Each queue is accessible outside this module's translation unit.

The Workstation class defines the structure of an active station on the assembly line and contains all the functionality for filling customer orders with station items. Each Workstation is-a-kind-of Station. A Workstation object manages order processing for a single Item on the assembly line. Since a Workstation object represents a single location on the assembly line for filling customer orders with items, the object cannot be copied or moved. Make sure that this capability is deleted in your definition of the Workstation class.

The Workstation class includes the following additional information:

***Instance Variables***

* m\_orders – is a double-ended-queue with CustomerOrders entering the back and exiting the front. These are orders that have been placed on this station to receive service (or already received service).
* m\_pNextStation – a pointer to the next Workstation on the assembly line.

***Member Functions***

* a custom 1-argument constructor -- receives a reference to an unmodifiable reference to std::string and passes it to the Station base class.
* void fill(std::ostream& os) – this modifier fills the order at the front of the queue if there are CustomerOrders in the queue; otherwise, does nothing.
* bool attemptToMoveOrder() – attempts to move the order order at the front of the queue to the next station in the assembly line:
  + if the order requires no more service at this station or cannot be filled (not enough inventory), move it to the next station; otherwise do nothing
    - if there is no next station in the assembly line, then the order is moved into g\_completed or g\_incomplete queue
  + if an order has been moved, return true; false otherwise.
* void setNextStation(Workstation\* station) – this modifier stores the address of the referenced Workstation object in the pointer to the m\_pNextStation. Parameter defaults to nullptr.
* Workstation\* getNextStation() const – this query returns the address of next Workstation
* void display(std::ostream& os) const – this query inserts the name of the Item for which the current object is responsible into stream os following the format: ITEM\_NAME --> NEXT\_ITEM\_NAME
  + if the current object is the last Workstation in the assembly line this query inserts: ITEM\_NAME --> End of Line.
  + in either case, the message is terminated with \n
* Workstation& operator+=(CustomerOrder&& newOrder) – moves the CustomerOrder referenced in parameter newOrder to the back of the queue.

**LineManager Module**

The LineManager class manages an assembly line of active stations and contains the following information:

***Instance Variables***

* std::vector<Workstation\*> m\_activeLine – the collection of workstations for the current assembly line.
* size\_t m\_cntCustomerOrder – the total number of CustomerOrder objects
* Workstation\* m\_firstStation - points to the first active station on the current line

***Member Functions***

* LineManager(const std::string& file, const std::vector<Workstation\*>& stations) - this constructor receives the name of the file that identifies the active stations on the assembly line (example: AssemblyLine.txt) and the collection of workstations available for configuring the assembly line.

The file contains the linkage between workstation pairs. The format of each record in the file is WORKSTATION|NEXT\_WORKSTATION. The records themselves are not in any particular order.

This function stores the workstations in the order received from the file in the m\_activeLine instance variable. It loads the contents of the file, stores the address of the next workstation in each element of the collection, identifies the first station in the assembly line and stores its address in the m\_firstStation attribute. This function also updates the attribute that holds the total number of orders in the g\_pending queue. If something goes wrong, this constructor reports an error.

**Note**: to receive full marks, use STL algorithms throughout this function, except for iterating through the file records (one while loop); marks will be deducted if you use any of for, while or do-while loops except for iterating through the file records.

* void reorderStations() - this modifier reorders the workstations present in the instance variable m\_activeLine (loaded by the constructor) and stores the reordered collection in the same instance variable. The elements in the reordered collection start with the first station, proceeds to the next, and so forth until the end of the line.
* bool run(std::ostream& os) – this modifier performs **one** iteration of operations on all of the workstations in the current assembly line by doing the following:
  + keeps track of the current iteration number (use a local variable)
  + inserts into stream os the iteration number (how many times this function has been called by the client) in the format Line Manager Iteration: COUNT<endl>
  + moves the order at the front of the g\_pending queue to the m\_firstStation and remove it from the queue. This function moves only one order to the line on a single iteration.
  + for each station on the line, executes one fill operation
  + for each station on the line, attempts to move an order down the line
  + return true if all customer orders have been filled or cannot be filled, otherwise returns false.
* void display(std::ostream& os) const -- this query displays all active stations on the assembly line in their current order

**Tester Module**

The tester module and input files are included in 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!**

Milestone 3 Explanation

Workstation inherits from the station module

You are creating 2 modules : Workstation and LineManager

The LineManager constructor is going to be creating some workstation objects

Workstation inherits from Station

The Workstation module consists of three double-ended queues of CustomerOrder and the Workstation class. The queues (global variables) hold the orders at either end of the assembly line:

* g\_pending holds the orders to be placed onto the assembly line at the first station.
* g\_completed holds the orders that have been removed from the last station and have been completely filled.
* g\_incomplete holds the orders that have been removed from the last station and could not be filled completely.

3 deques of CustomerOrder

Basically, g\_pending is initially where all the CustomerOrders are. Then, when they go through the assembly line, they become either complete (g\_completed) or incomplete (g\_incomplete). This explains the 3 deques.

March 29th lecture:

3 deques above from the workstation class explained. They are global deques. (extern)

The LineManager class manages an assembly line of active stations and contains the following information:

You can think of the LineManager as a configuration of the workstations. It sets up the workstations and connects them together, so each workstation knows which is next.

How do we code that?

LineManager(const std::string& file, const std::vector<Workstation\*>& stations) - this constructor receives the name of the file that identifies the active stations on the assembly line (example: AssemblyLine.txt) and the collection of workstations available for configuring the assembly line.

The assembly line has active workstations. The active workstations you get from another vector of many workstations.

Ex: vector has 9 workstations. IT doesn’t mean all the workstations are active in the assembly line. This constructor uses the file to see which workstations from the vector are going to go to the assembly line as active.

The workstations in the vector do not have the connections to the next vector yet.

Basically, the LineManager constructor uses the file to determine the order of workstations in the assembly line (that is the LineManager object).