Spring 2020

Data Structures and Algorithms

Restaurant Management

Project Requirements

Objectives

By the end of this project, the student should be able to:
- Write a **complete object-oriented C++ program** with **Templates** that performs a non-trivial task.
- Use data structures to solve a real-life problem.
- Understand unstructured, natural language problem description and derive an appropriate design.
- Intuitively modularize a design into independent components and divide these components among team members.

Introduction

Finally, the data structures TAs have decided to give up their career and start a new less tiring business, a Restaurant. Yes, it is time to farewell bits and bytes and start working with stacks of wheat packs, queues of customers' orders and loops of Swiss roll. The TAs, however, believe that they can use computer simulation to assess and enhance their service. Currently, they have '*M'* **cooks** and they want to determine the *service criteria* -- that is, the criteria based on which an order should be serviced (i.e. prepared) earlier or later than others, to maximize average customer satisfaction. Because the TAs are currently busy running their new business, they ask you to help them, using your programming skills and knowledge of data structures, to write a simulation program that **simulates** the **Restaurant kitchen system** and **calculates** some statistics that measure average customer satisfaction.

Project Phase	%
Phase 1 (Week 9)	**35%**
Phase 2 (Week 14)	**65%**

NOTES:

- Number of students per team = **4 students from the same tutorial (time and location).**

- Each member must be responsible for writing some project modules (e.g. some classes or some functions) and must answer some questions showing that he/she understands both the program logic and the implementation details. The work load between team members must be almost equal.

- **The project code must be totally yours. The penalty of cheating (taking or giving) any part of the project from any other source is not ONLY taking ZERO in the project grade but also taking MINUS FIVE (-5) from other class work grades. So it is better to deliver an incomplete project other than cheating it. It is totally your responsibility to keep your code private.**

- **Late Delivery is not allowed**

Orders & Cooks

The restaurant has a number of available cooks that it should prepare the incoming orders.

Orders:
The following information is available for each order:

- **Arrival Time Stamp:** When the order was made.
- **Order Type:** There are 3 types of orders: VIP, Vegan orders and Normal orders.
    - **VIP orders** must be served first before vegan and normal orders.
    - **Vegan orders** are the orders that needs to be prepared by specialized cooks using all plant-based ingredients.
    - **Normal orders** are the orders that neither VIP nor vegan.
- **Order Size:** the number of dishes for this order (in dishes).
- **Order Price:** the total order price the customer should pay.

Cooks:
At startup, the system loads (from a file) information about the available **cooks**. For each cook, the system will load the following information:

- **Cook specialization:** There are 3 types: VIP cooks, vegan cooks and Normal cooks.
    - **VIP cooks** are cooks with higher cooking abilities so they are best for doing high profile orders.
    - **Vegan cooks** are cooks that tolerates making all vegetable food.
    - **Normal cooks** are the cooks that neither VIP nor vegan.
- **Break Duration:** Each cook takes a break after serving **n** consecutive orders.

Note:
The **cook speed** (the number of dishes it can prepare in one timestep) is the same for all cooks of the same type.

Orders Service Criteria

To determine the next order to serve (if a cook is available), the following **Service Criteria** should be applied for all the arrived un-serviced orders **at each time step:**

1) First, Serve **VIP orders** by ANY available type of cooks, but there is a priority based on the cook's type over the others to prepare VIP orders: First use VIP Cooks THEN Normal Cooks THEN Vegan Cooks. This means that we don't use Normal cook unless all VIP cooks are busy, and don't use Vegan cooks unless all other types are busy.

2) Second, Serve **Vegan orders** using available V*egan cooks **ONLY**.* If all vegan cooks are busy, wait until one is free.

3) Third, Serve **Normal orders** using any type of cooks *EXCEPT vegan cooks*, but First use the available Normal cooks THEN VIP cooks (if all Normal are busy).

4) If an order cannot be served at the current time step, it should wait for the next time step to be checked if it can be serviced or not. If not, it should wait again.

   Notes: If the orders of a specific type cannot be serviced in the current time step, try to service the other types (e.g. if Vegan orders cannot be serviced in the current time step, this does NOT mean not to service the Normal orders)

That is how we prioritize the service between different order types, but how will we prioritize the service between the orders of **the same type**?

- **For Vegan and Normal orders,** orders that arrive first should be serviced first.
- **For VIP orders,** there is a priority equation for deciding which of the available VIP orders to serve first. VIP orders with higher priority must be serviced first.

☞ You should develop a reasonable **weighted** priority equation depending on at least the following factors: *Order Arrival Time*, *Order Money*, *and Order Size*.

There are some additional services the restaurant offers to its customers:

- **For Normal orders ONLY**,
  - ❑ The customer can **promote** it to become VIP order by paying more money
  - ❑ The customer can **cancel** the order as long as it is not assigned to a cook yet
  - ❑ If a **NORMAL** order waits more than **N** timesteps from its arrival time without being assigned to a cook, it will be **automatically promoted** to be a VIP order. (**N** is read from the input file)

Simulation Approach & Assumptions

You will use incremental time steps. You will divide the time into discrete time steps of 1 unit time each and simulate the changes in the system at each *timestep*.

Some Definitions

- **Arrival Time (AT):**
  The timestep at which the order is issued by the customer.

- **Waiting Order:**
  The order that has arrived (i.e. Order **AT** < current timestep but not served yet).
  At **each time step**, you should choose the order**s** to prepare from the waiting orders.

- **In-service Order:**

  The order that are being prepared (assigned to a cook) but not finished yet.

- **Serviced Order:**
  The order that is finished and serviced to the customer.

- **Waiting Time (WT):**
  The time interval from the arrival of an order until it is assigned to a cook.

- **Service Time (ST):**
  The time interval that a cook needs to prepare an order (the time spent in preparing the dishes).

- **Finish Time (FT):**
  The timestep at which the order is successfully serviced to the customer.
  $$FT = AT + WT + ST$$

Assumptions
- If the cook is available at timestep T, he/she can prepare a new order starting from that timestep.
- More than one order can arrive at the same timestep. Also, more than one order can be assigned to different cooks at the same timestep as long as there are available cooks to prepare them.
- A cook can only be preparing one order at a time.
- A cook cannot be assigned an order during his break.

Input/Output File Formats

Your program should receive all information to be simulated from an input file and produces an output file that contains some information and statistics about the simulation. This section describes the format of both files and gives a sample for each.

The Input File Format

- The 1st line contains three integers: **N   G   V**
  Each integer represents the number of cooks of different types.
    - ❑ **N** for normal, **G** for vegan, and **V** for VIP cooks.
- First 2nd line contains three integers: **SN   SG   SV**
    - ❑ **SN:** the speed of normal cooks
    - ❑ **SG:** the speed of vegan cooks
    - ❑ **SV:** the speed of VIP cooks.
- The 3rd line contains four integers: **BO   BN   BG   BV**
    - ❑ **BO**: the number of orders a cook must prepare before taking a break
    - ❑ **BN**: the break duration (in timesteps) for normal cooks
    - ❑ **BG**: the break duration for vegan ones
    - ❑ **BV**: the break duration for VIP cooks.
- Then a line with only one integer **AutoP** that represent the number of timesteps after which an order is automatically promoted to VIP.
- The next line contains one integer *M* that represents the number of **events**
- Then the input file contains *M* lines (one line for **each event**). An event can be:
    - ❑ Arrival of a new order. Denoted by letter **R**, or
    - ❑ Cancellation of an existing order. Denoted by letter **X**, or
    - ❑ Promotion of an order to be a VIP order. Denoted by letter **P**.

NOTE: The lines of all events are **sorted by the event time** (**TS)** in **ascending** order.

Events Lines

- **Arrival event line** have the following info
    - ❑ **R**(letter R in the beginning of its line) means an arrival event
    - ❑ **TYP** is the order type (*N: normal, G: vegan, V: VIP*).
    - ❑ **TS** is the event timestep. (order arrival time)
    - ❑ **ID** is a unique sequence number that identifies each order.
    - ❑ **SIZE** is the number of dishes of the order
    - ❑ **MONY** is the total order money.
- **Cancellation event line** have the following info
    - ❑ **X**(Letter X) means an order cancellation event
    - ❑ **TS** is the event timestep.
    - ❑ **ID** is the id of the order to be canceled. This ID must be of a Normal order.

- **Promotion event line** have the following info
  - ❑ **P**(Letter P) means an order promotion event occurring
  - ❑ **TS** is the event timestep.
  - ❑ **ID** is the id of the order to be promoted to be VIP. It must be of a Normal order.
  - ❑ **ExMony** if the extra money the customer paid for promotion.

Sample Input File

5	3	1		➔ no. of cooks for each type		
2	3	6		➔ cooks speeds for each type		
3 4		3	2	➔ Breaks info line		
25				➔ Auto promotion limit		
8				➔ no. of events in this file		
R	**N**	**7**	**1**	**15**	**110**	➔ Arrival event example
R	**N**	**9**	**2**	**7**	**56**	
R	**V**	**9**	**3**	**21**	**300**	
R	**G**	**12**	**4**	**53**	**42**	
X	**15**	**1**				➔ Cancellation event example
R	**N**	**19**	**5**	**17**	**95**	
P	**19**	**2**	**62**			➔ Promotion event example
R	**G**	**25**	**6**	**33**	**127**	

The Output File Format

The output file you are required to produce should contain **M** output line of the format
FT   ID   AT   WT   ST
which means that the order identified by sequence number **ID** has arrived at time **AT.**
It then waited for a <u>period</u> **WT** to be served**.** It has then taken **ST** ticks to be prepared
and finished at timestep **FT**.

The output lines **must be sorted** by **FT** in ascending order. If more than one order is
finished at the same timestep, **they should be ordered by ST**.

Then the following statistics should be shown at the end of the file
1- Total number of orders and total number of each order type
2- Total number of cooks and total number of each type
3- Average waiting, and average service time
4- Percentage of automatically-promoted orders (relative to total normal orders)

Sample Output File

The following numbers are just for clarification and are not produced by actual calculations.

FT	ID	AT	WT	ST
18	1	7	5	6
44	10	24	2	18
49	4	12	20	17

.... and so on ....
Orders: 124 [Norm:100, Veg:15, VIP:9]
cooks: 9   [Norm:5, Veg:3, VIP:1]
Avg Wait = 12.3, Avg Serv = 25.65
Auto-promoted: 7

Program Interface

The program can run in one of three modes: **interactive, step-by-step** or **silent mode**. When the program runs, it should ask the user to select the program mode.

Interactive mode: allows user to monitor the restaurant operation.
At each time step, program should provide output **similar** to that in the figure. In this mode, program pauses for a user mouse click to display the output of the next timestep.
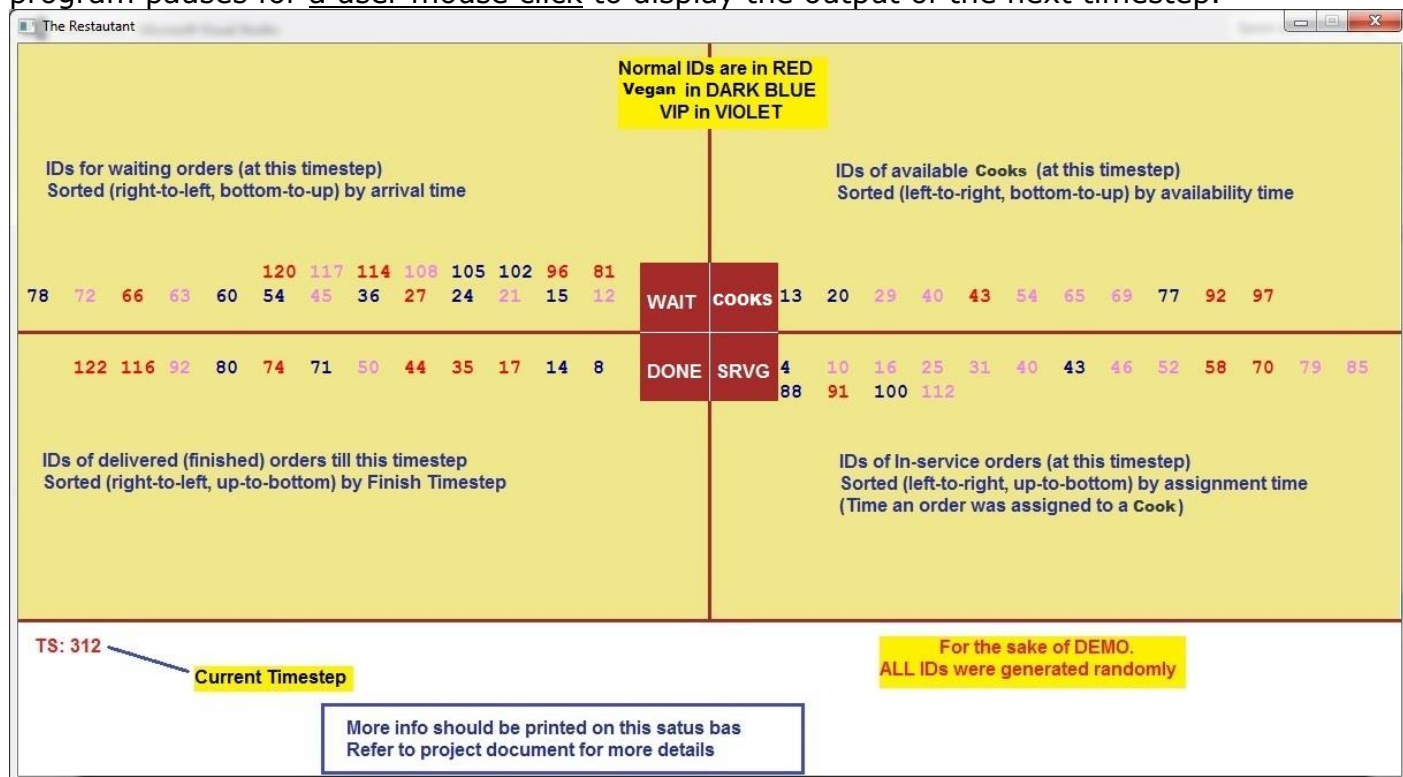
At the bottom of the screen, the following information should be printed:
- Simulation Timestep Number
- Number of waiting orders of each order type
- Number of available cook of each type
- Type & ID for **ALL** cook**s** and order**s** that were assigned in the **last** timestep only.
  e.g. **N6(V3)** ➔ normal cook#6 is assigned VIP order#3 to prepare
- Total number of orders served so far of each order type

Step-by-step mode is identical to interactive mode except that each time step, the program waits for one second (not for mouse click) then resumes automatically.

☞ You are provided a code library (set of functions) for drawing the above interface. (See - Appendix A)
In **silent mode**, the program produces only an output file (See the "File Formats" section). It does not draw anything graphically.
No matter what mode of operation your program is running, **the output file** should be produced.

Project Phases

You are given a partially implemented code that you should extend in phase 1 and phase 2. It is implemented using classes. You are required to write **object-oriented** code with **Templates** for data structure classes. The graphical user interface GUI for the project is almost all implemented and given to you.

 Before explaining the requirement of each phase, All the following are NOT allowed to be used in your project:
- You are not allowed to use **C++ STL** or any external resource that implements the data structures you use. ***This is a data structures course where you should build data structures yourself from scratch.***

- You need to get instructor's approval before making any **custom (new)** data structure.
    Note: No approval is needed to use the known data structures.

- ***Do NOT allocate the same Order more than once***. Allocate it once and make whatever data structures you chose points to it (pointers). Then, when another list needs an access to the same order, DON'T create a new copy of the same order. Just **share** it by making the new list point to it or **move** it from current list to the new one. *SHARE, MOVE, DON'T COPY...*

- You are not allowed to use **global variables** in your implemented part of the project.
- You need to get instructor approval before using **friendships.**

Phase 1

In this phase you should finish implementing ALL **data structures** needed for BOTH phases without implementing the logic related to servicing the order**.** The required parts to be **finalized** and delivered at this phase are:
1- **Full data members** of Order, cook, and Restaurant Classes.
2- **Full "Template" implementation for ALL data structures DS** that you will use to represent **the lists of orders,events and cooks**. All data structure needed for both project phases must be finished in this phase.

   Important: Keep in mind that you are **NOT** selecting the DS that would **work in phase1 only**.
   You must choose the DS that would work efficiently for both phase1 & phase2.

   When choosing the DS think about the following:
   a. How will you store **waiting orders**? Do you need a separate list for each type?
   b. What about the **cooks lists**?
   c. Do you need to store **finished orders**? When should you delete them?
   d. **Which list type** is much suitable to represent each list? You must take into account the **complexity of the main operations** needed for each list (e.g.

insert, delete, retrieve, shift, sort …etc.). For example, If the most frequent operation in a list is deleting from the middle, use a data structure that has low complexity for this operation.

You need to justify your choice for each DS and why you separated or joined lists. Selecting the appropriate DS for each list is the core target of phase 1 and the project as a whole.

==Note: you need to read *"File Format"* section to see how the input data and output data are sorted in each file as it may affect the selection of the data structures.==

3- **<u>File loading function</u>.** The function that reads input file to:
   a. Load cooks data and populate cooks lists(s).
   b. Load events data and populate the events list

4- **<u>Simple Simulator function for Phase 1</u>**. The main purpose of this function is to test your data structures and how to move orders and cooks between lists. This function should:
- Perform any needed initializations
- Call file loading function
- At *each timestep* do the following:
   a. Get the events that should be executed at current timestep
      ▪ For arrival event, generate an order and add it to the appropriate waiting orders list.
      ▪ For cancellation event, delete the corresponding **normal** order (**if found**)
      ▪ Ignore promotion events
   b. **Pick one order** from each order type and move it to In-service list(s).
      Note 1: the order you choose to delete from each type must be the first order that should be assigned to an available cook in phase 2.
      Note 2: *NO actual cooks check_availability/assignment is required in Phase 1.*
   c. Each **5 timesteps**, move an order of each type from In-service list(s) to finished list(s)
   d. Update the interface as follows:
      i. Add all orders/cooks to be drawn to the GUI::DrawingList
      ii. Update the Interface to display IDs for all what is in GUI::DrawingList
      iii. Number of waiting orders of each order type
      iv. Number of available cooks of each type
- The simulation function stops when there are no more events nor active orders in the system

Notes about phase 1:
- No output files should be produced at this phase.
- In this phase, you can go to the next timestep by mouse click
- No order service or assigning cooks will be done in this phase. However, all the lists of the project should be implemented in that phase.
- Make sure you read **Project Evaluation** and **Individuals Evaluation** section mentioned below.

Phase 1 Deliverables:
Each team is required to submit the following:
- A text file named *ID.txt* containing team members' names, IDs, and emails.
- **Phase 1 full code** [Do not include executable files].
- **Three Sample input files** (test cases).
- **Phase 1 document** with 1 or more pages describing:
    - ❑ The DS you have chosen for each list
    - ❑ Your justification of all your choices with the complexity of the most frequent or major operation for each list.
- Compress all the above in one file named after your team number (e.g. Phase1_team3.zip)

Phase 2

In this phase, you should extend code of phase 1 to build the full application and produce the final output file. Your application should support the different operation modes described in "Program Interface" section.

Phase 2 Deliverables:
- Each team is required to submit the following:
- A text file named *ID.txt* containing team members' names, IDs, and emails.
- **Final Project Code** [Do not include executable files].
- **Six Comprehensive sample input files (test cases) and their output files.**
- **Work Load Document**. Don't forget to **Print** it and bring it with you in the discussion day.
- Compress all the above in one file named after your team number (e.g. Phase2_team3.zip)

Project Evaluation

These are the main points that will be graded in the project:
- **Data Structure & Algorithm**: After all, this is what the course is about. You should be able to provide a concise description and a justification for: (1) the data structure(s) and algorithm(s) you used to solve the problem, (2) the **complexity** of the chosen algorithms, and (3) the logic of the program flow.

- **Programming Concepts:**
  - **Modularity:** A **modular** code does not mix several program features within the same unit (module). For example, the code module that does the core of the simulation process should be separate from the code that reads the input file which, in turn is separate from the code that implements the data structure.
  - **Maintainability:** A maintainable code is the one whose modules are easily modified or extended without a severe effect on other modules.

- **Interface modes**: Your program should support the three modes described in the document.

- **Successful Compilation:** Your program must compile successfully with zero errors. Delivering the project with any compilation errors will make you lose a large percentage of your grade.

- **Test Cases**: You should prepare comprehensive different test cases (at least 6). Your program should be able to simulate different scenarios not just trivial ones.

- **Coding style**: How elegant and **<u>consistent</u>** is your coding style (indentation, naming convention ...etc.)? How useful and sufficient are your comments? This will be graded.

Individuals Evaluation

Each member must be responsible for writing some project modules (e.g. some classes or some functions) and must answer some questions showing that he/she understands both the program logic and the implementation details. <u>The work load between team members must be almost equal</u>.

✓ The grade of **each student** will be divided as follows:
- **[70%]** of the student grade is on his individual work (the project parts he is responsible for).
- **[25%]** of the student grade is on integrating his work with the work of ALL students who <u>Finished</u> or nearly finished their project part.
- **[5%]** of the student grade is on cooperation with other team members and helping them if they have any problems implementing their parts (helping does NOT mean implementing their parts).

- ✓ If one or more members didn't make their work, the other members will NOT be affected **as long as** the students who finished their part integrated all their parts together AND their part can be tested to see the output.
- ✓ If a student couldn't finish his part, try to still integrate it with the other parts to at least take the integration grade.

You should **inform the TAs** before the deadline **with a sufficient time (some weeks before)** if any problems happened between team members to be able to take grading action appropriately.

Bonus Criteria (maximum 10% of project grade)

- **[3%] cook speed/break time:** cooks of the same type may have different speeds and different break times. The cooks of a specific type must be sorted by their speed. Higher speed cooks of a type have the higher priority to be assigned to orders than lower speed cooks of the same type.
- **[3%] cooks Health Emergency Problems:** cooks can get injured (with knives or heat) during cooking. If this happened, they should wait for a rest period to have the necessary medication. If the system needs this cook before its rest period is over, this will cause the cook's speed to be decreased to its half until his break.
- **[4%] More order/cook types:** Think about **two** more order/cook types other than those given in the document. The load of the logic of the two orders must be acceptable (**Needs instructor approval**).

Appendix A
Guidelines on the Provided Framework

The main classes of the game are Restaurant, Order, Cook, Event, and GUI (Graphical User Interface).

Event classes:

There are three types of events; <u>Arrival, Cancel, and Promote events</u>. You are given a base class called "***Event***" that stores event time and related order ID. This is an abstract class with a pure virtual function "**Execute**". The logic of Execute function depends on the Event type.

For each type of the three events, there should be a class derived from ***Event*** class. Each derived class should store data related to its operation. Also each of them should override function **Execute** as follows:

1- ArrivalEvent::Execute ➔ should create a new order and add it to the <u>appropriate</u> list
2- CancelEvent::Execute ➔ should cancel the requested order if found
3- PrompteEvent::Execute ➔ should move a normal order to the VIP list and update order's data.

Class Restaurant has a queue of ***Event*** pointers to store all events loaded from the file at system startup. At each timestep, the code loops on the events queue to dequeue and execute all events that should take place at current timestep.

GUI class

It is the class responsible for ALL drawings and ALL inputs. It contains the input and output functions that you should use to show status of the orders at each timestep.

Main members of class GUI:
1. ***DrawingList****:* an array of **pointers to DrawingItem**
    - Each **DrawingItem** contains: ID, color, region/quarter of the item to be drawn.
    - The items to be drawn include Order and Cooks.
    - This array should point to ALL items to be drawn at current timestep.
    - At the end of each timestep, it should be reset to be reloaded again with items of the next tiemstep.

2. Function ***AddtoDrawingList(Order *pO)***
    - Creates a DrawingItem and extracts its info (Order ID, color, region) from pO.
    - Adds the item to the DrawingList

3. Function ***AddtoDrawingList(Cook *pC)***
    - Creates a DrawingItem and extracts its info (Cook ID, color) from pC.
    - Adds the item to the DrawingList

4. Function ***ResetDrawingList( )***
    - Resets the drawing list (at the end of each time step)

5. Function ***DrawAllItems( )***
    -   Responsible for drawing ALL items (order/cooks) of array *DrawingList*
    -   For each Item in the array, it calls ***DrawSingleItem*** function to draw it.

6. Function ***DrawSingleItem(DrawingItem* p, int RegionCount)***
    -   Draws the passed item (p) by printing its ID on the screen.
    -   The screen is divided into four quarters to print IDs of waiting orders, free cooks, in-service orders and finished orders. (See Program Interface section above).
    -   Each item **type** is printed in a different color
    -   RegionCount parameter is used to place the order in its correct relative position with respect to other orders in the same region.

7. Function ***UpdateInterface( )***
    -   It first clears the interface then re-draws everything again
    -   Called every timestep to display current system status.

8. Function ***PrintMsg(string msg)***
    -   Prints ***msg*** on the status bar
    -   Use this function to print information about each region at every timestep
    -   **You should update this function to print multiple lines at the bottom of the interface window.**

9. Function ***string GetString( )***
    -   Reads a string from the GUI window
    -   **Use this function to read input file name**

[Important Note]:
    The drawing list (DrawingList) is just used to draw items. It has nothing to do with the data structure you will be using to store orders/cooks in the system.
    You can pick whatever data structure that is suitable for order/cook manipulation.
    For each order/cook to be drawn on the screen, just prepare a pointer to it then call **AddtoDrawingList** to add it then call **UpdateInterface** to show them all

    Finally, a demo code (***Demo_Main.cpp***) is given just to test the above functions and show you how they can be used. It has nothing to do with phase 1 or phase 2. You should write your main function of each phase yourself.