Lab 6 - Data Structures and Systems

Goals

- In this lab, you will experience:
    - Dynamically-resizable arrays (vector)
    - Associative arrays (with ordered keys) (map)
    - Queues (queue)
    - Stacks (stack)
    - Creating classes according to specification
    - Working as a team to complete a multi-object system while working independently in parallel

Foreword

- **This lab simulates finishing a part of a final project.** It is expected that you get to know the codebase and *pretend that you are working on a half-finished final project*. Every single one of the classes involved with this "mini-project" is documented with informative comments in Doxygen style, and there are test cases written using the Catch2 testing framework to gauge whether you are implementing the class correctly, so please **dive deep** in order to succeed in this lab.
    - **Read the README.md.**
- **This is the last significant programming lab before you transition over to your final project, so we will be introducing additional elements related to testing.** You can find two examples files in the **`learning/`** directory that explain **assert()** and **Catch2**, respectively.
- You can run **make** in order to test all programs, or you can run the following commands
    - **`make cashier`**
    - **`make cook`**
    - **`make runner`**
    - **`make excashier`**
    - **`make excook`**
    - **`make exrunner`**
        - to create the appropriate example or test case suite program.
    - **`make clean`** can also be used to clean up the object files in **src/** and the executables in **bin/**

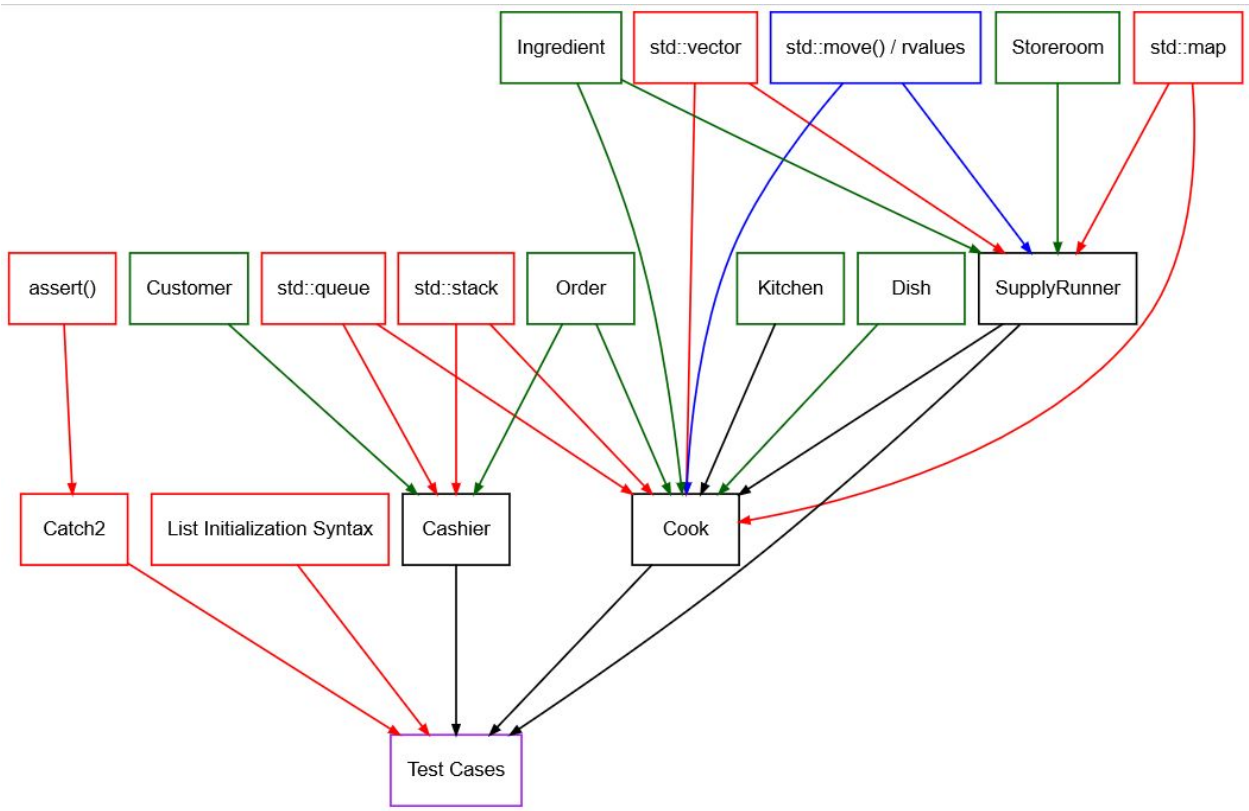

Figure 1 - The concept map for this lab. Red: concepts to learn. Blue: past concepts. Green: other classes in the project already completed. Black: classes to implement the .cpp for.

Task: Finish the Project

Your team is completing your EE205 final project, "Tina's Burger Bistro," a system simulator that mirrors a fast food restaurant. You have heard that the rest of your team did a majority of the work before mysteriously vanishing due to "medical reasons" (but you, of course, know better). Naturally, you are furious that the rest of your team decided to ignore your final project, but you know that the A is valuable.

After reading the README.md Markdown file, you are confident that you can finish the project by simply filling in the three missing classes' implementations (Cashier, Cook, and SupplyRunner) and pass their three corresponding Catch2 test case suites: (test-Cashier.cpp, test-Cook.cpp, test-SupplyRunner). You also notice that the three example programs in the tst/ directory should also work correctly once you are finished.

- Your task is to finish the three classes mentioned above by implementing them, and have them pass their test case suites.
- Don't worry! A lot of the documentation and knowledge required to do the project is spread out amongst:

    - The project README.md file
    - The Doxygen-formatted class comments
    - The **learning/** directory that contains some information on **assert()** and **Catch2**
    - The **Catch2** test cases and their givens, whens, and thens comments
    - The design tips as a part of this lab manual's supplementary material.
- ***Note: Cook depends on SupplyRunner, so you need to finish SupplyRunner before you can test Cook.***

Submission

- **Comment your code! If your code does not work correctly, you may be able to get *partial credit* based on your thinking that we can infer from the comments!**
- In a file called **team.txt**, write your group number on the first line. On subsequent lines, write the last, then first names of your team members separated by a single comma, each on its own line. If your name contains whitespace (e.g. the single space in the last name, "da Silva"), you may keep the whitespace.
- Place your source code files into a directory called **project**, using the following directory structure:
    - `project/`
        - `Makefile`
        - `src/`
            - `<source files>`
            - `<no .o files>`
        - `include/`
            - `<header files>`
        - `dep/`
            - `catch.hpp`
        - `docs/` `<create this>`
            - `team.txt`
        - `bin/`
            - `NO EXECUTABLES SHOULD BE HERE UPON SUBMISSION!`
        - `tst/`
            - `<example files>`
            - `<Catch2 test case files>`
- Archive and compress your **project** directory using the command:
    - `tar czf name_of_archive_file.tar.gz project`
    - The name of the archive file should be in the following format:
        - `ee205-lab<lab-number>-group<group-number>.tar.gz`
- Submit the *tar.gz* file under your group's entry on Laulima for the assignment.
    - For your safety, test uncompressing and unpacking your archive in a clean folder, then compile your source files. Ensure that the executable will run correctly.

Supplementary Material

Design Tips: Cashier

- One of the focal points of implementing Cashier is calculating the cost of the order. This involves:
    - parsing a string into pairs of numbers and dishes to order
        - You may want to use **split()** from **restaurant-utility.hpp** to do this.
    - verifying that the dishes ordered are valid
    - calculating the cost of the order
        - The helper function **calculate-order-cost()** from **restaurant-utility.hpp** can be a great help. If you can get the order items into the format of a vector of strings, you can use this function to calculate the cost immediately, and, if it detects an item not on the menu, it will throw an exception.
- Another major feature of Cashier is the ability to **expel()** Customer objects that do not have enough money to pay. **expel()** is a member function on Customer.
    - You can do this by getting the amount of money that a Customer has after you calculate the total cost of an Order.
- **Note: if the customer queue is empty. do not do anything.**
- Remember to pop customers off the front of the queue once you are done taking their order or expelling them!

Design Tips: SupplyRunner

- SupplyRunner is used by Cook, but SupplyRunner also uses a Storeroom. You'll find that Storeroom is just a "typedef" or alias for another typename -- a map that maps dish names (as std::string) to amounts of ingredients (formatted as another map from ingredient names to number of ingredients).
    - You'll want to brush up on **std::map**. Maps are a common data structure found in the software engineering world by many names: dictionaries, hash maps, and associative arrays. In short, they act as arrays whose index type can be something other than a number.
- You may want to search "how to iterate or go through a std::map." You'll probably want to introduce yourself to iterators.

Design Tips: Cook

- Cook is the hardest class to implement of the three assigned ones because it also uses SupplyRunner. Therefore, if your SupplyRunner is buggy and doesn't pass the unit tests, then you'll have to go back and fix it until it works. So don't do this one first.
- As a Cook, when preparing a dish, you'll have to take an Order off the stack, and then for each ordered dish, lookup the required ingredients to cook it, ask the SupplyRunner to get you the ingredients, and then send it to the Kitchen.

    - **Kitchen.cpp** has a nice function that shows what each set of ingredients turns into when you give a **std::map** to prepare a Dish with. You'll probably want a way to lookup with a string describing the Dish name to what Ingredients (std::string) you need to get from the SupplyRunner.
        - One way to do this is to initialize a compile-time **std::map** by using **list initialization syntax.** This link contains an example on how to use it with various classes at the bottom, and includes an example for std::map.
- Remember to keep the assigned Customer ID associated with the order when putting the pairs of the Customer ID with the Finished dish! Several dishes can be put onto the queue even though it may be from the same order, so be wary of this.
- **Note: if the order stack is empty. do not do anything.**
- Remember to pop orders off the stack!