Tina's Burger Bistro

This project is meant to simulate a fast food bistro. It is also meant to demonstrate what a fairly complete EE205 project should look like at the end.

Note...

...if you want to look at this file in a nicer format, look into "How to render Markdown to PDF on Linux". You'll probably end up using `pandoc` in order to do this.

Also, if you can get and use `xdg-open`, it can be useful for opening `.pdf` files from the terminal.

Overall Design Architecture

The bistro's design is formed by the following classes:

- Customer
- Cashier
- Cook
- Kitchen
- Storeroom (`std::map<std::string, std::map<Ingredient, std::size_t>>`)
- SupplyRunner
- Order
- Dish (`enum class : int`)
- Ingredient (`std::string`)

Customers are put onto a std::queue. They are handled by a Cashier, who gives them a unique ID (starting from 0, counting up by 1 each time), takes their order, and charges them money. If they give invalid orders or cannot pay, they are expelled from the restaurant. After the **Cashier** takes a Customer's order, the Customer's ID is sent along with the ordered items in an Order object and put onto an Order stack (`std::stack<Order>`). Then, the Cook is meant to interface with the stack.

The **Cook** takes Orders off of the stack, and then looks it up in the RecipeBook (a map from `std::string` dish names to `std::map<Ingredient, std::size_t>`). Then, it asks the SupplyRunner to get the required ingredients, and uses the Kitchen to prepare the dish with the returned ingredients. After preparing the dish, the Cook will send it along with the correct customer ID as a `std::pair<std::size_t, Dish>` to the output `std::queue<std::pair<std::size_t, dish>`. (Dish, by the way, is simply an `enum class : int`).

The **SupplyRunner** is meant to be configured with a Storeroom, which holds ingredients and how many of that type of ingredient it currently has. When asked for a number of ingredients, the SupplyRunner will do its best and check whether the Storeroom has enough ingredients to supply. If it doens't, the SupplyRunner will throw a `const char*` exception – otherwise, it will provide the correct ingredients as a `std::vector<Ingredient>`. Note that it is not required to return the ingredients in any specified order – just that the correct ingredients are returned in the vector in the correct overall amounts.

Testing

In this project, we use Catch2 as our testing framework – a single-header library that is put into our `./dep/` directory. Thus, we do not need a linking phase to link the testing library in.

Our test cases live in the `./tst/` directory, with the test case suites prefixed by `test-`. We also have 3 examples files we created that are prefixed by `example-`.

Building Tina's Burger Bistro

Building the project is handled by our `Makefile` in the root directory of our project. One need only run `make` to build every executable, even the testing ones!

However, you can also run specialized rules to create the test case suites for only a certain class, such as `make runner` to create `./bin/test-SupplyRunner.out`. Inspect the Makefile manually if you wish for more detail.

Notable Concepts Used

Doxygen is a documentation tool that can be used to turn specially formatted comments into full-blown documentation for a project (one I like to use is HTML format so I can open it as a web page). It has support for C++. Every single header file in `./include/` is commented in the style required for Doxygen. Note that this is not required to build the project, but it is a nice addon if you wish to generate more readable documentaiton than just comments in your source files and have it unified in one place. It is a fairly common tool that you may see in the future on some larger projects.

Dependency injection is a concept where instead of you directly including objects in a class that are default constructed, instead, you take any objects you rely on in the constructor and save references, copies, or moved versions of them. The

reason to do this is to allow yourself to use "mock" versions of dependency objects if the finished versions of other classes you are working on are not done yet. This is a relatively common OOP design pattern seen in production. Another related term is "inversion of control."

Unit testing is a form of program testing where you break a program into its part and test them individually in order to assume that the composition of small, verified parts created a safe, verified overall program. In OOP, the units of testing are usually classes – therefore, in this project, we provide the 3 test case suites that you need to pass in order to pass this lab.