Algorithmic Adventures: The Virtual Bistro – Managing Kitchen Stations with Linked Lists

Project Overview

Welcome back to your culinary adventure! In this project, you will enhance your virtual bistro simulation by implementing a kitchen station management system. Building upon your previous work with dishes, you will now create a `KitchenStation` class and a `StationManager` class that inherits from a singly linked list template. This will allow you to manage various cooking stations efficiently.

As the head chef and manager, you need to coordinate multiple kitchen stations, ensuring that each station can handle specific dishes and that shared ingredients are managed properly. This project will test your understanding of linked lists, inheritance, templates, dynamic memory allocation, and advanced data structures in C++.

Note: The singly linked list template class `LinkedList<T>` is already implemented and uses nodes. You will inherit from this class to create your `StationManager`.

Here is the link to accept this project on GitHub Classroom:
https://classroom.github.com/a/O2esO2Jm

Documentation Requirements

As with all previous projects, documentation is crucial. You will receive 15% of the points for proper documentation. Ensure that you:

1. **File-level Documentation**: Include a comment at the top of each file with your name, date, and a brief description of the code implemented in that file.
2. **Function-level Documentation**: Before each function declaration and implementation, include a comment that describes:
   - **@pre**: Preconditions for the function.
   - **@param**: Description of each parameter.
   - **@return**: Description of the return value.
   - **@post**: Postconditions after the function executes.
3. **Inline Comments**: Add comments within your functions to explain complex logic or important steps.

Remember: All files and all functions must be commented, including both `.hpp` and `.cpp` files.

Additional Resources

If you need to brush up on or learn new concepts, the following resources are recommended:

- **Linked Lists in C++**:
  - Linked Lists (GeeksforGeeks)
  - Linked Lists in C++ (Programiz)
- **Templates and Inheritance**:
  - Templates in C++ (cplusplus.com)
- **Dynamic Memory Allocation**:
  - Pointers and Dynamic Memory (cplusplus.com)

Implementing the Kitchen Station Management System with Linked Lists

You will implement the `KitchenStation` class and a `StationManager` class that inherits from a singly linked list template to manage multiple kitchen stations.

Key Concepts

- **Templates**: Using the singly linked list template class.
- **Inheritance**: `StationManager` will inherit from the singly linked list template.
- **Linked Lists**: Manage kitchen stations in a list that allows traversal in one direction.
- **Dynamic Memory Allocation**: Manage `Dish` and `Ingredient` objects dynamically.

Task 1: Implement the Ingredient Struct

Add the following `Ingredient` struct to Dish.hpp, but outside the Dish class.

```
/**
 * Struct representing an ingredient.
 */
struct Ingredient {
    std::string name;
    int quantity;            // Quantity in stock
    int required_quantity;   // Quantity required for a dish
    double price;           // Price per unit

    // Constructors
    Ingredient();
    Ingredient(const std::string& name, int quantity, int required_quantity,
double price);
};
```

Task 2: Modify the Dish Class

Start with the `Dish` class provided that's based on previous projects. Make the following necessary modifications.

- **Member Variables**:
  - `std::vector<Ingredient> ingredients_;` // List of required ingredients
- **Member Functions**:
  - Modify the accessors and mutators for ingredients, as well as the Dish constructors.

Task 3: Implement the KitchenStation Class

The `KitchenStation` class represents a cooking station in the kitchen.

Data Members:

- `station_name_`: `std::string` representing the station's name (e.g., "Grill Station", "Sauce Station").
- `dishes_`: `std::vector<Dish*>` storing pointers to dishes that the station can prepare.
- `ingredients_stock_`: `std::vector<Ingredient>` representing the ingredients available at the station.

Member Functions:

- **Constructors / Destructor**:
  - **Default Constructor**:

    ```
    /**
     * Default Constructor
     * @post: Initializes an empty kitchen station with default values.
     */
    ```

  - **Parameterized Constructor**:

```
/**
 * Parameterized Constructor
 * @param station_name A string representing the station's name.
 * @post: Initializes a kitchen station with the given name.
 */
```

- **Destructor**:

```
/**
 * Destructor
 * @post: Deallocates all dynamically allocated dishes in the
station.
 */
```

- **Member Functions**:
  - **getName()**

```
/**
 * Retrieves the name of the kitchen station.
 * @return: The name of the station.
 */
```

  - **setName()**

```
/**
 * Sets the name of the kitchen station.
 * @param name A string representing the new station name.
 * @post: Updates the station's name.
 */
```

- **getDishes()**

```
/**
 * Retrieves the list of dishes assigned to the kitchen station.
 * @return A vector of pointers to Dish objects assigned to the station.
 */
```

- **getIngredientsStock()**

```
/**
 * Retrieves the ingredient stock available at the kitchen station.
 * @return A vector of Ingredient objects representing the station's
ingredient stock.
 */
```

- **assignDishToStation()**

```
/**
 * Assigns a dish to the station.
 * @param dish A pointer to a Dish object.
 * @post: Adds the dish to the station's list of dishes if not
already present.
 * @return: True if the dish was added successfully; false
otherwise.
 */
```

- **replenishStationIngredients()**

```
/**
 * Replenishes the station's ingredient stock.
 * @param ingredient An Ingredient object.
 * @post: Adds the ingredient to the station's stock or updates the
quantity if it already exists.
 */
```

- **canCompleteOrder()**

```
/**
 * Checks if the station can complete an order for a specific dish.
 * @param dish_name A string representing the name of the dish.
 * @return: True if the station has the dish assigned and all
required ingredients are in stock; false otherwise.
 */
```

- **prepareDish()**

```
/**
 * Prepares a dish if possible.
 * @param dish_name A string representing the name of the dish.
 * @post: If the dish can be prepared, reduce the quantities of the
used ingredients accordingly. If the stock ingredient is depleted to
0, remove the ingredient from the Kitchen Station.
 * @return: True if the dish was prepared successfully; false
```

```
otherwise.
 */
```

Task 4: Implement the StationManager Class

The `StationManager` class will inherit from the `LinkedList<KitchenStation*>` template.

Member Functions:

- **Constructor / Destructor**:
  - **Default Constructor**:

    ```
    /**
     * Default Constructor
     * @post: Initializes an empty station manager.
     */
    ```

  - **Destructor**:

    ```
    /**
     * Destructor
     * @post: Deallocates all kitchen stations and clears the list.
     */
    ```

- **Member Functions**:
  - **addStation()**

    ```
    /**
     * Adds a new station to the station manager.
     * @param station A pointer to a KitchenStation object.
     * @post: Inserts the station into the linked list.
     * @return: True if the station was successfully added; false
    otherwise.
     */
    ```

  - **removeStation()**

    ```
    /**
     * Removes a station from the station manager by name.
    ```

```
 * @param station_name A string representing the station's name.
 * @post: Removes the station from the list and deallocates it.
 * @return: True if the station was found and removed; false
otherwise.
 */
```

- **findStation()**

```
/**
 * Finds a station in the station manager by name.
 * @param station_name A string representing the station's name.
 * @return: A pointer to the KitchenStation if found; nullptr
otherwise.
 */
```

- **moveStationToFront()**

```
/**
 * Moves a specified station to the front of the station manager
list.
 * @param station_name A string representing the station's name.
 * @post: The station is moved to the front of the list if it
exists.
 * @return: True if the station was found and moved; false
otherwise.
 */
```

- **mergeStations()**

```
/**
 * Merges the dishes and ingredients of two specified stations.
 * @param station_name1 The name of the first station.
 * @param station_name2 The name of the second station.
 * @post: The second station is removed from the list, and its
contents are added to the first station.
 * @return: True if both stations were found and merged; false
otherwise.
 */
```

- **assignDishToStation()**

```
/**
 * Assigns a dish to a specific station.
 * @param station_name A string representing the station's name.
 * @param dish A pointer to a Dish object.
 * @post: Assigns the dish to the specified station.
 * @return: True if the station was found and the dish was assigned;
false otherwise.
 */
```

- **replenishIngredientAtStation()**

```
/**
 * Replenishes an ingredient at a specific station.
 * @param station_name A string representing the station's name.
 * @param ingredient An Ingredient object.
 * @post: Replenishes the ingredient stock at the specified station.
 * @return: True if the station was found and the ingredient was
replenished; false otherwise.
 */
```

- **canCompleteOrder()**

```
/**
 * Checks if any station in the station manager can complete an
order for a specific dish.
 * @param dish_name A string representing the name of the dish.
 * @return: True if any station can complete the order; false
otherwise.
 */
```

- **prepareDishAtStation()**

```
/**
 * Prepares a dish at a specific station if possible.
 * @param station_name A string representing the station's name.
 * @param dish_name A string representing the name of the dish.
 * @post: If the dish can be prepared, reduces the quantities of the
used ingredients at the station.
 * @return: True if the dish was prepared successfully; false
otherwise.
 */
```

Testing

Testing your code thoroughly is essential to ensure functionality and robustness. While no sample data files are provided, it's recommended that you create various `Ingredient`, `Dish`, and `KitchenStation` objects yourself to simulate different scenarios. Start with simple cases and gradually increase complexity as you confirm the functionality of each part.

Testing Steps:

1. **Initialize Ingredients and Dishes**:
    - Manually create several `Ingredient` objects with varying quantities and required amounts.
    - Create multiple `Dish` objects and assign them specific ingredients from your ingredient set. This will allow you to test `Dish` functionality and ensure the `Dish` objects correctly reference the ingredients needed.
2. **Create Kitchen Stations**:
    - Instantiate several `KitchenStation` objects (e.g., "Grill Station," "Pasta Station").
    - Use `assignDishToStation()` to add dishes to each kitchen station. Verify that each station can correctly manage its list of assigned dishes.
3. **Manage Ingredients**:
    - Use `replenishStationIngredients()` to add ingredients to each station's inventory.
    - Experiment with adding both new ingredients and updating quantities for existing ones. Verify that the ingredient stock is correctly updated.
4. **Test Order Completion**:
    - Test `canCompleteOrder()` to see if a station can fulfill an order for a given dish based on available ingredients.
    - Attempt to use `prepareDish()` to make dishes at each station, confirming that ingredient quantities decrease as expected when a dish is successfully prepared.
5. **Edge Cases**:
    - Test situations where ingredients are insufficient to complete a dish and ensure that `canCompleteOrder()` and `prepareDish()` handle these cases correctly.
    - Try preparing a dish at a station that doesn't have it assigned and confirm the function's response.
    - Remove a station and check that the list updates appropriately, ensuring no memory leaks or dangling pointers.
6. **Verify Information**:
    - Verify that the list contains the correct station names and assigned dishes.
    - Test this after adding, removing, and modifying stations to ensure consistent output.

Testing Tips:

- Start by stubbing functions and progressively implement functionality.
- Test each class and function individually before integrating.
- Use assertions or output statements to verify expected outcomes.

Submission

You will submit your solution to Gradescope via GitHub Classroom. The autograder will grade the following files:

- `Dish.hpp`
- `Dish.cpp`
- `KitchenStation.hpp`
- `KitchenStation.cpp`
- `StationManager.hpp`
- `StationManager.cpp`

Note: The `LinkedList<T>` template class is already implemented and should not be modified.

Submission Instructions:

- Ensure that your code compiles and runs correctly on the Linux machines in the labs at Hunter College.
- Use the provided Makefile to compile your code.
- Push the code you want to submit to the GitHub repository created by GitHub Classroom.
- Submit your assignment on Gradescope by linking it to your GitHub repository.

Grading Rubric

- **Correctness**: 80% (distributed across unit testing of your submission)
- **Documentation**: 15%
- **Style and Design**: 5% (proper naming, modularity, and organization)

Due Date

This project is **due on November 14th, 2024 at 5:30 PM EST**.
No late submissions will be accepted.

Important Notes

- **Start Early**: Begin working on the project as soon as it is assigned to identify any issues and seek help if needed.
- **No Extensions**: There will be no extensions and no negotiation about project grades after the submission deadline.
- **Help Resources**: Help is available via drop-in tutoring in Lab 1001B (see Blackboard or Discord for schedule). Starting early will allow you to get the help you need.

Authors: Michael Russo, Georgina Woo, Prof. Wole

Credit to Prof. Ligorio