Project 6: Enhancing the Virtual Bistro – Dish Queues

Project Overview

Welcome back to your culinary journey! In this project, you will enhance your virtual bistro simulation by modifying the **StationManager** class. Building upon your previous work with dishes, kitchen stations, and dietary accommodations, you will introduce a **queue of dishes** within the StationManager to manage the order in which dishes are prepared. Additionally, you will implement functions to process all dishes in the queue, displaying detailed results, including any station replenishments along the way.

As the head chef and manager, you need to efficiently handle the preparation of dishes, ensuring that each dish is queued, adjusted for dietary accommodations, and processed in the correct order. This project will test your understanding of **queues**, **inheritance**, **virtual functions**, **dynamic memory allocation**, and advanced data structures in C++.

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

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:

- **Queues**:
  - Queues in C++ (GeeksforGeeks)
  - Queues in C++ (cplusplus.com)
- **Templates**:
  - Templates in C++ (cplusplus.com)
- **Dynamic Memory Allocation**:
  - Pointers and Dynamic Memory (cplusplus.com)

Implementing the Dish Queue in the StationManager with Dietary Accommodations

You will modify the `StationManager` class to include a dish queue and implement functions to manage it. Additionally, you will handle dietary accommodations by adjusting dishes as they are added to the queue and handle out of stock ingredients by using a backup ingredients vector and new functions alongside previously implemented ones.

Key Concepts

- **Queues**: Using `std::queue` to manage dishes in a First-In, First-Out (FIFO) order, ensuring dishes are prepared in the sequence they were added.
- **Dynamic Memory Allocation**: Allocating and deallocating memory for `Dish` objects dynamically to handle dish preparation and memory management efficiently.
- **Virtual Functions**: Employing virtual functions in base and derived classes to handle specific behaviors, such as dietary accommodations, ensuring extensibility and flexibility.

Task 1: Modify the `StationManager` Class to Include a Dish Queue & Include Accessors & Mutators

Add the following private member variable to `StationManager`:

- **dish_queue_**: `std::queue<Dish*>` storing pointers to dynamically allocated `Dish` objects that need to be prepared.
- **backup_ingredients_**: `std::vector<Ingredient>` representing the backup stock of ingredients that can be used to replenish station ingredients when needed.

Add the following accessors & mutators:

1. **getDishQueue**

```
/**
 * Retrieves the current dish preparation queue.
 * @return A copy of the queue containing pointers to Dish objects.
 * @post: The dish preparation queue is returned unchanged.
 */
```

2. **getBackupIngredients**

```
/**
 * Retrieves the list of backup ingredients.
 * @return A vector containing Ingredient objects representing backup
supplies.
 * @post: The list of backup ingredients is returned unchanged.
 */
```

3. **setDishQueue**

```
/**
 * Sets the current dish preparation queue.
 * @param dish_queue A queue containing pointers to Dish objects.
 * @pre: The dish_queue contains valid pointers to dynamically allocated
Dish objects.
 * @post: The dish preparation queue is replaced with the provided
queue.
 */
```

Task 2: Implement the `addDishToQueue` Functions

Implement two overloaded `addDishToQueue` functions in `StationManager`:

1. **addDishToQueue**

```
/**
 * Adds a dish to the preparation queue without dietary accommodations.
 * @param dish A pointer to a dynamically allocated Dish object.
 * @pre: The dish pointer is not null.
 * @post: The dish is added to the end of the queue.
 */
```

2. **addDishToQueue**

```
/**
 * Adds a dish to the preparation queue with dietary accommodations.
 * @param dish A pointer to a dynamically allocated Dish object.
 * @param request A DietaryRequest object specifying dietary
accommodations.
 * @pre: The dish pointer is not null.
 * @post: The dish is adjusted for dietary accommodations and added to
the end of the queue.
 */
```

Task 3: Implement the `prepareNextDish` Function

Implement the `prepareNextDish` function in `StationManager`:

```
/**
 * Prepares the next dish in the queue if possible.
 * @pre: The dish queue is not empty.
 * @post: The dish is processed and removed from the queue.
 * If the dish cannot be prepared, it stays in the queue
 * @return: True if the dish was prepared successfully; false otherwise.
 */
```

Task 4: Implement the `displayDishQueue` Function

Implement the `displayDishQueue` function in `StationManager`:

```
/**
 * Displays all dishes in the preparation queue.
```

```
 * @pre: None.
 * @post: Outputs the names of the dishes in the queue in order (each name
is on its own line).
 */
```

Task 5: Implement the `clearDishQueue` Function

Implement the `clearDishQueue` function in `StationManager`:

```
/**
 * Clears all dishes from the preparation queue.
 * @pre: None.
 * @post: The dish queue is emptied and all allocated memory is freed.
 */
```

Task 6: Implement the `replenishStationIngredientFromBackup` Function

Implement the `replenishStationIngredientFromBackup` function in `StationManager`:

```
/**
 * Replenishes a specific ingredient at a given station from the backup
ingredients stock by a specified quantity.
 * @param station_name A string representing the name of the station.
 * @param ingredient_name A string representing the name of the ingredient
to replenish.
 * @param quantity An integer representing the amount to replenish.
 * @pre None.
 * @post If the ingredient is found in the backup ingredients stock and has
sufficient quantity, it is added to the station's ingredient stock by the
specified amount, and the function returns true.
 *       The quantity of the ingredient in the backup stock is decreased by
the specified amount.
 *       If the ingredient in backup stock is depleted (quantity becomes
zero), it is removed from the backup stock.
 *       If the ingredient does not have sufficient quantity in backup
stock, or the ingredient or station is not found, returns false.
 * @return True if the ingredient was replenished from backup; false
```

```
otherwise.
 */
```

Task 7: Implement the `addBackupIngredients` Function

Implement the `addBackupIngredients` function in `StationManager`:

```
/**
 * Sets the backup ingredients stock with the provided list of ingredients.
 * @param ingredients A vector of Ingredient objects to set as the backup
stock.
 * @pre None.
 * @post The backup_ingredients_ vector is replaced with the provided
ingredients.
 * @return True if the ingredients were added; false otherwise.
 */
```

Task 8: Implement the `addBackupIngredient` Function

Implement the `addBackupIngredient` function in `StationManager`:

```
/**
 * Adds a single ingredient to the backup ingredients stock.
 * @param ingredient An Ingredient object to add to the backup stock.
 * @pre None.
 * @post If the ingredient already exists in the backup stock, its quantity
is increased by the ingredient's quantity.
 *       If the ingredient does not exist, it is added to the backup stock.
 * @return True if the ingredient was added; false otherwise.
 */
```

Task 9: Implement the `clearBackupIngredients` Function

Implement the `clearBackupIngredients` function in `StationManager`:

```
/**
 * Empties the backup ingredients vector
 * @post The backup_ingredients_ private member variable is empty.
 */
```

Task 10: Implement the `processAllDishes` Function with Detailed Output

Implement the `processAllDishes` function in `StationManager`:

```
/**
 * Processes all dishes in the queue and displays detailed results.
 * @pre: None.
 * @post: All dishes are processed, and detailed information is displayed
(as per the format in the specifications), including station replenishments
and preparation results.
 * If a dish cannot be prepared even after replenishing ingredients, it
stays in the queue in its original order...
* i.e. if multiple dishes cannot be prepared, they will remain in the queue
in the same order
 */
```

Example Output Format:

When processing each dish, the output should follow this format. Note the different cases and how you should account for them. Each of the different sections represent the function being called on a different StationManager object. Please consider the different scenarios that could lead to different outputs at various points when the program is running. The other functions called are indicated clearly, so for functions with return types, that return might be relevant.

```
PREPARING DISH: Spaghetti Bolognese
Pasta Station attempting to prepare Spaghetti Bolognese...
Pasta Station: Insufficient ingredients. Replenishing ingredients...
Pasta Station: Ingredients replenished.
Pasta Station: Successfully prepared Spaghetti Bolognese.


All dishes have been processed.
```

```
PREPARING DISH: Vegan Salad
Salad Station attempting to prepare Vegan Salad...
Salad Station: Successfully prepared Vegan Salad.


All dishes have been processed.
```

```
PREPARING DISH: Seafood Paella
Seafood Station attempting to prepare Seafood Paella...
Seafood Station: Insufficient ingredients. Replenishing ingredients...
Seafood Station: Unable to replenish ingredients. Failed to prepare Seafood
Paella.
Seafood Paella was not prepared.


All dishes have been processed.
```

```
PREPARING DISH: Grilled Chicken
Grill Station attempting to prepare Grilled Chicken...
Grill Station: Unable to replenish ingredients. Failed to prepare Grilled
Chicken.
Oven Station attempting to prepare Grilled Chicken...
Oven Station: Successfully prepared Grilled Chicken.

PREPARING DISH: Beef Wellington
Grill Station attempting to prepare Beef Wellington...
Grill Station: Dish not available. Moving to next station...
Oven Station attempting to prepare Beef Wellington...
Oven Station: Dish not available. Moving to next station...
Pasta Station attempting to prepare Beef Wellington...
Pasta Station: Dish not available. Moving to next station...
Salad Station attempting to prepare Beef Wellington...
Salad Station: Dish not available. Moving to next station...
Beef Wellington was not prepared.


All dishes have been processed.
```

```
PREPARING DISH: Garden Salad
Grill Station attempting to prepare Garden Salad...
Grill Station: Dish not available. Moving to next station...
Pasta Station attempting to prepare Garden Salad...
Pasta Station: Dish not available. Moving to next station...
Salad Station attempting to prepare Garden Salad...
Salad Station: Insufficient ingredients. Replenishing ingredients...
Salad Station: Ingredients replenished.
Salad Station: Unable to prepare Garden Salad.
Garden Salad was not prepared.


All dishes have been processed.
```

Submission

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

- `StationManager.hpp`
- `StationManager.cpp`

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.**
- You are expected to thoroughly test your code in fashion similar to the previous projects for submitting the project or seeking help from TAs. This project does not have a testing guide, as part of the requirement is for you to test yourselves.
- 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 December 2nd, 2024 at 11:00 PM EST**.

You can receive an extra 5/80 points on the project by submitting early and scoring at least 20 points on an early submission. That early submission deadline will be November 22nd at 11:00 PM EST.

No late submissions will be accepted.

Please note that due to the previous deadline extensions, this due date falls after several days during which the college is closed, which means that the TAs will not be available.

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