Algorithmic Adventures: The Virtual Bistro – The Kitchen

It is now time to define the simulation setting. In this project, you will implement the `Kitchen` class, where you will manage your `Dish` objects. You will create functionality that allows you to add and remove `Dish` objects from the `Kitchen`, and you will also be able to query the `Kitchen` for information about the `Dish` objects it contains. You will also be able to release dishes from the kitchen based on their preparation time and cuisine type.

Here is the link to accept the Github Classroom Assignment:
https://classroom.github.com/a/HA2wkiZd

This project consists of two parts:

1. You will modify the `Dish` class.
2. You will implement the `Kitchen`, as a subclass of `ArrayBag`, to hold `Dish` objects.

You will NOT modify `ArrayBag`. This class will remain as distributed.

Some additional resources:

- Abstract Data Types:
  - Geeks for Geeks
  - Neso Academy
- Template Classes:
  - CPP Manual
  - Geeks for Geeks
  - Tutorials Point
- Operator Overloading:
  - CPP Manual

- Geeks for Geeks
- Programiz

Implementation

Work through the tasks sequentially (implement and test). Only move on to a task when you are positive that the previous one has been completed correctly. Remember that the names of classes and methods must exactly match those in this specification (FUNCTION NAMES, PARAMETER TYPES, RETURNS, PRE AND POST CONDITIONS MUST MATCH EXACTLY).

Remember, you must thoroughly document your code! All of the documentation requirements from the previous projects still apply!

Task 1: Modify the Dish class

Define and implement the following additional public member functions:

```
/**
  @param : A const reference to the right-hand side of the `==` operator.
  @return : Returns true if the right-hand side dish is "equal", false
otherwise.
            Two dishes are equal if they have the same name, same cuisine
type, same preparation time,
            and the same price.
*/
operator==
```

Explanation: By this definition, only the aforementioned subset of the dish's attributes must be equal for two dishes to be deemed "equal."

Example: In order for `dish1 == dish2`, we only need:

- The same name
- The same cuisine type
- The same preparation time
- The same price

```
/**
  @param : A const reference to the right-hand side of the `!=` operator.
  @return : Returns true if the right-hand side dish is NOT "equal" (`!=`),
false otherwise.
            Two dishes are NOT equal if any of their name, cuisine type,
preparation time, or price are not equal.
*/
operator!=
```

Explanation: By this definition, one or more of the aforementioned subset of the dish's attributes must only be different for two dishes to be deemed "NOT equal."

Task 2: Implement the `Kitchen` class as a subclass of `ArrayBag`

The `Kitchen` class is a subclass of `ArrayBag` that stores `Dish` objects.

Data Types

The `Kitchen` class must have the following private member variables:

- An integer sum of the preparation times of all the dishes currently in the kitchen. (total*prep_time*)
- An integer count of all of the elaborate dishes in the kitchen. An elaborate dish is one that has 5 or more ingredients, and a prep time of an hour or more. (count*elaborate*)

The `Kitchen` class must have the following public member functions:

- **Constructor**

  ```
  /**
    * Default constructor.
    * Default-initializes all private members.
    */
  ```

Unique Methods:

```
/**
  * @param : A reference to a `Dish` being added to the kitchen.
  * @post : If the given `Dish` is not already in the kitchen, adds the
`Dish` to the kitchen and updates the preparation time sum and vegetarian
dish count if the dish is vegetarian.
  * @return : Returns true if a `Dish` was successfully added to the
kitchen, false otherwise.
           Hint: Use the above definition of equality to help determine if a
`Dish` is already in the kitchen.
*/
newOrder
```

```
/**
  * @param : A reference to a `Dish` leaving the kitchen.
  * @return : Returns true if a dish was successfully removed from the
kitchen (i.e.,
items_), false otherwise.
  * @post : Removes the dish from the kitchen and updates the preparation
time sum.
           If the `Dish` is vegetarian, it also updates the vegetarian
count.
*/
serveDish
```

```
/**
  * @return : The integer sum of preparation times for all the dishes
currently in the kitchen.
*/
getPrepTimeSum
```

```
/**
  * @return : The average preparation time (int) of all the dishes in the
kitchen. The lowest possible average prep time should be 0.
  * @post : Computes the average preparation time (double) of the kitchen
rounded to the NEAREST integer.
*/
calculateAvgPrepTime
```

```
/**
  * @return : The integer count of the elaborate dishes in the kitchen.
```

```
*/
elaborateDishCount
```

```
/**
  * @return : The percentage (double) of all the elaborate dishes in the
kitchen. The lowest possible percentage should be 0%.
  * @post : Computes the percentage of elaborate dishes in the kitchen
rounded up to 2 decimal places.
*/
calculateElaboratePercentage
```

```
/**
  * @param : A reference to a string representing a cuisine type with a
value in
             ["ITALIAN", "MEXICAN", "CHINESE", "INDIAN", "AMERICAN",
"FRENCH", "OTHER"].
  * @return : An integer tally of the number of dishes in the kitchen of the
given cuisine type.
             If the argument string does not match one of the expected
cuisine types, the tally is zero.
                 NOTE: No pre-processing of the input string necessary, only
uppercase input will match.
*/
tallyCuisineTypes
```

```
/**
  * @param : A reference to an integer representing the preparation time
threshold of the dishes to be removed from
             the kitchen, with a default value of 0.
  * @post : Removes all dishes from the kitchen whose preparation time is
less than the given time.
           If no time is given, removes all dishes from the kitchen. Ignore
negative input.
  * @return : The number of dishes removed from the kitchen.
*/
releaseDishesBelowPrepTime
```

```
/**
  * @param : A reference to a string representing a cuisine type with a
value in
             ["ITALIAN", "MEXICAN", "CHINESE", "INDIAN", "AMERICAN",
"FRENCH", "OTHER"],
             or a default value of "ALL" if no cuisine type is given.
  * @post : Removes all dishes from the kitchen whose cuisine type matches
the given type.
           If no cuisine type is given, removes all dishes from the kitchen.
  * @return : The number of dishes removed from the kitchen.
           NOTE: No pre-processing of the input string necessary, only
uppercase input will match.
           If the input string does not match one of the expected cuisine
types, do not remove any dishes.
*/
releaseDishesOfCuisineType
```

```
/**
  * @post : Outputs a report of the dishes currently in the kitchen in the
form:
             "ITALIAN: {x}\nMEXICAN: {x}\nCHINESE: {x}\nINDIAN:
{x}\nAMERICAN: {x}\nFRENCH: {x}\nOTHER: {x}\n\n
              AVERAGE PREP TIME: {x}\nVEGETARIAN: {x}%\n"
             Note that the average preparation time should be rounded to the
NEAREST integer, and the
             percentage of vegetarian dishes in the kitchen should be
rounded to 2 decimal places.

             Example output:
             ITALIAN: 2
             MEXICAN: 3
             CHINESE: 2
             INDIAN: 1
             AMERICAN: 1
             FRENCH: 2
             OTHER: 2

             AVERAGE PREP TIME: 62
             VEGETARIAN: 53.85%
*/
kitchenReport
```

Testing

Although you will no longer submit your test file, you must continue to thoroughly and methodically test your code.

Start by stubbing all expected functions. Have all function declarations in the `.hpp` file and stubs for all functions in the `.cpp` file. When submitted as such, your program will compile, although you will fail all tests, since you have not implemented any functions yet. If your program compiles, you will have at least established that all functions have the correct name, parameters, and return type.

What is a stub?
A stub is a dummy implementation that always returns a single value for testing (or has an empty body, if void). Don’t forget to go back and implement the stub!!! If you put the word STUB in a comment, some editors will make it more visible.

Now you can start implementing and testing your project, **ONE FUNCTION AT A TIME**!

Write a `main()` function to test your implementation. Choose the order in which you implement your methods so that you can test incrementally: i.e., implement constructors, then accessor functions, then mutator functions.

For each class, test each function you implement with all edge cases before you move on to implement the next function. This includes all constructors.

TEST THE FUNCTIONS LOCALLY BEFORE SUBMITTING TO GRADESCOPE! If it does not work on your machine, it *definitely* won't work on the autograder.

Makefile Compilation Instructions

How to compile with your Makefile:
In the terminal, in the same directory as your `Makefile` and your source files, use the following command:

```
make rebuild
```

This assumes you did not rename the `Makefile` and that it is the only one in the current directory.

Please review the Makefile guide in #links-and-resources on discord! If you're having problems compiling, please make sure you're following the correct coding conventions for the classes we're using in this project.

Grading Rubric

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

Important Notes

You must start working on the projects as soon as they are assigned to detect any problems with submitting your code and to address them with us well before the deadline so that we have time to get back to you before the deadline.

There will be no negotiation about project grades after the submission deadline.

Submission

We will grade the following files:

- `Dish.hpp`
- `Dish.cpp`
- `Kitchen.hpp`
- `Kitchen.cpp`

Although Gradescope allows multiple submissions, it is not a platform for testing and/or debugging, and it should not be used for that. You MUST test and debug your program locally. To help you not rely too much on Gradescope for testing, we will only allow **5 submissions per day**.

Before submitting to Gradescope, you MUST ensure that your program compiles using the provided `Makefile` and runs correctly on the Linux machines in the labs at Hunter (see detailed instructions on how to upload, compile, and run your files in the “Programming Guidelines” document). This is your baseline: if it runs correctly there, it will run correctly on Gradescope. If it does not, you will have the necessary feedback (compiler error messages, debugger, or program output) to guide you in debugging, which you don’t have through Gradescope. “But it ran on my machine!” is not a valid argument for a submission that does not compile. Once you have done all the above, submit it to Gradescope.

Due Date:

This project is due on 10/15 at 5:30PM EST.

Please note that this project is due on Tuesday, on which classes follow a Monday schedule, because Monday 10/14 is a federal holiday.

No late submissions will be accepted.

Help

Help is available via drop-in tutoring in Lab 1001B (see Blackboard or Discord for schedule). You will be able to get help if you start early and go to the lab early. We only have 2 UTAs in the lab, and the days leading up to the due date will be crowded, so you will not be able to get much help then.

Authors: Michael Russo, Georgina Woo, Prof. Wole

Credit to Prof. Ligorio