Algorithmic Adventures: The Virtual Bistro – The Dish Class: A Review of OOP

Project Overview

This semester, you will embark on a culinary journey developing the infrastructure for a captivating virtual bistro simulation. Set within the vibrant environment of a modern restaurant, your simulation will offer participants a unique experience where they get to manage and curate a menu filled with delectable dishes.

As the main chef and manager, you are tasked with creating a variety of dishes for the restaurant, each with its own unique set of ingredients, preparation time, and pricing. Participants of your simulation will be able to interact with various dishes, allowing them to explore different flavors, cooking methods, and culinary styles.

Each dish will have specific attributes that define its preparation process, the ingredients used, and its price. The goal of this project is to refresh your knowledge of basic Object-Oriented Programming (OOP) concepts by implementing the `Dish` class, which will be the foundation of your restaurant simulation.

This project will introduce you to GitHub Classroom so you can work with git version control. All projects in this course will be distributed via GitHub Classroom and submitted to Gradescope via GitHub. We truly hope you will start establishing best practices of version control, INCREMENTAL coding, testing, and debugging (i.e. code, test, and debug one function at a time); you will need it in the near future, so better start now!

Part 1: Getting Started with GitHub Classroom

This part is absolutely essential!!! Do not skip it and ensure you complete it before proceeding to Part 2.

- If you don't already have one, go to GitHub and create a GitHub account. You will likely use your GitHub account professionally in the future, so choose a username you will want to keep.

- Next, watch the following video to brush up on or learn the basics of Git and GitHub: Git & GitHub Crash Course
- For this project, we will use GitHub Classroom. The following video will guide you through the entire process—from accepting an assignment to submitting your solution: GitHub Classroom Tutorial
    - Although the video is about a different course, the instructions are the same (with different repo and file names). The only difference is that we will not add a distribution branch, so you can ignore the part where it says to execute the two git commands in the readme file; there are not extra instructions in the readme file on our repo).

Here is the link to accept the assignment on GitHub Classroom:
https://classroom.github.com/a/COF9_4XH

The above video will also show you how to submit to Gradescope via GitHub. Make sure to refer back to these instructions when it's time to submit.

Some additional resources if you need to brush up on basic OOP

- Code Beauty on constructors and class methods
- thenewboston on classes and objects
- McProgramming on .hpp and .cpp files

Documentation Requirements

For **ALL** projects, you will receive 15% of the points for documentation. These are the requirements:

1. **File-level Documentation**: All files must have a comment preamble with, at a minimum, your name, date, and a brief description of the code implemented in that file.
2. **Function-level Documentation**: All functions (both declarations and implementations) must be preceded by a comment preamble that includes:
    - **@pre**: Describes any preconditions. Precondition is a condition that must be true before a function is called. It specifies what the caller must ensure before calling the function.
    - **@param**: One for each parameter the function takes.
    - **@return**: Describes the return type.

- **@post**: Describes any postconditions. Postcondition is a condition that must be true after a function completes. It specifies what the function guarantees upon completion.

These together fully specify the usage of the function. You will notice that we often provide these in the project specification to describe what functionality you should implement. You will copy/paste these preambles into your code, and your code will be fully documented and easy to read and use by anyone. It is not useless work, it will help you learn how to document your code. Having said that, sometimes we must add extra guidance given the scholastic context, like giving you hints for implementation. These are not things you would normally include in your documentation of professional code. Whenever you will write additional functions not in the project specification (this will be more common in later projects), you will be expected to comment your functions in a similar way, even when the preambles are not provided by the specification.

3. **Inline Comments**: All non-trivial functions must have inline comments. Any block of code that is not self-explanatory must be preceded by a comment describing what it does (e.g., have one English sentence before each loop or conditional describing what it does).

All files and all functions must be commented. Yes both .hpp and .cpp!!! It is a lot of copy/paste, but it is not useless. If someone is reading through your code to understand what it does, they shouldn't have to consult the comments in a different file!

Part 2: The Dish Class

You will implement the `Dish` class.

Key Concepts

- Always separate interface from implementation (`Dish.hpp` and `Dish.cpp`), and you **ONLY EVER** include a class's interface (`.hpp`). This will be an implicit assumption in this course going forward. 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). This class has both accessor and mutator functions for its private data members.

The Dish Class

Every `Dish` has a name, a list of ingredients, preparation time, price, and a cuisine type.

The Dish class must define the following type INSIDE the class definition:

- An enum named `CuisineType` with values `{ITALIAN, MEXICAN, CHINESE, INDIAN, AMERICAN, FRENCH, OTHER}`

The Dish class must have the following private member variables:

- `std::string name_`: The name of the dish.
- `std::vector<std::string> ingredients_`: A list of ingredients used in the dish.
- `int prep_time_`: The preparation time in minutes.
- `double price_`: The price of the dish.
- `CuisineType cuisine_type_`: The cuisine type of the dish.

The Dish class must have the following public member functions:

Constructors:

- **Default Constructor:**

```
/**
 * Default constructor.
 * Initializes all private members with default values:
 * - name: "UNKNOWN"
 * - ingredients: Empty list
 * - prep_time: 0
 * - price: 0.0
 * - cuisine_type: OTHER
 */
```

- **Parameterized Constructor:**
  - The parameterized constructor must be able to work with only the name of the dish as a parameter, with all other parameters having default values.

```
/**
 * Parameterized constructor.
 * @param name A reference to the name of the dish.
 * @param ingredients A reference to a list of ingredients (default is
empty list).
 * @param prep_time The preparation time in minutes (default is 0).
 * @param price The price of the dish (default is 0.0).
 * @param cuisine_type The cuisine type of the dish (a CuisineType enum)
with default value OTHER.
```

```
 * @post The private members are set to the values of the corresponding
parameters.
 */
```

Accessors and Mutators:

- **setName:**

```
/**
 * Sets the name of the dish.
 * @param name A reference to the new name of the dish.
 * @post Sets the private member `name_` to the value of the parameter.
 */
setName
```

- **getName:**

```
/**
 * @return The name of the dish.
 */
getName
```

- **setIngredients:**

```
/**
 * Sets the list of ingredients.
 * @param ingredients A reference to the new list of ingredients.
 * @post Sets the private member `ingredients_` to the value of the
parameter.
 */
setIngredients
```

- **getIngredients:**

```
/**
 * @return The list of ingredients used in the dish.
 */
getIngredients
```

- **setPrepTime:**

```
/**
 * Sets the preparation time.
 * @param prep_time The new preparation time in minutes.
 * @post Sets the private member `prep_time_` to the value of the
parameter.
 */
setPrepTime
```

- **getPrepTime:**

```
/**
 * @return The preparation time in minutes.
 */
getPrepTime
```

- **setPrice:**

```
/**
 * Sets the price of the dish.
 * @param price The new price of the dish.
 * @post Sets the private member `price_` to the value of the parameter.
 */
setPrice
```

- **getPrice:**

```
/**
 * @return The price of the dish.
 */
getPrice
```

- **setCuisineType:**

```
/**
 * Sets the cuisine type of the dish.
 * @param cuisine_type The new cuisine type of the dish (a CuisineType
enum).
 * @post Sets the private member `cuisine_type_` to the value of the
parameter.
 */
setCuisineType
```

- **getCuisineType:**

```
/**
 * @return The cuisine type of the dish in string form.
 */
getCuisineType
```

- **display:**

```
/**
 * Displays the details of the dish.
 * @post Outputs the dish's details, including name, ingredients,
preparation time, price, and cuisine type, to the standard output.
 * The information must be displayed in the following format:
 *
 * Dish Name: [Name of the dish]
 * Ingredients: [Comma-separated list of ingredients]
 * Preparation Time: [Preparation time] minutes
 * Price: $[Price, formatted to two decimal places]
 * Cuisine Type: [Cuisine type]
 */
display
```

Part 3: Testing

To help you establish a good practice for testing, we will make the testing of your code part of the assignment. After the first few projects, this will simply be your regular development practice (thoroughly test every function you implement before moving to the next function), and we will no longer request that you submit a test file.

Submit a file called `test.cpp` that includes only a `main` function that does the following:

1. **Instantiate a dish with the default constructor:**
    - Set its preparation time to 30 using the appropriate setter function.
    - Set its price to 9.99 using the appropriate setter function.
    - Print out the dish's information using the display function.

    Expected Output:

```
Dish Name: UNKNOWN
Ingredients:
Preparation Time: 30 minutes
Price: $9.99
Cuisine Type: OTHER
```

2. **Instantiate a dish with the parameterized constructor:**

   - Name: "Pasta Carbonara"
   - Ingredients: ["Pasta", "Eggs", "Pancetta", "Parmesan", "Pepper"]
   - Preparation Time: 20
   - Price: 12.50
   - Cuisine Type: ITALIAN
   - Print out the dish's information using the display function.

   Expected Output:

```
Dish Name: Pasta Carbonara
Ingredients: Pasta, Eggs, Pancetta, Parmesan, Pepper
Preparation Time: 20 minutes
Price: $12.50
Cuisine Type: ITALIAN
```

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.

You must always implement and test your programs **INCREMENTALLY!!!**
What does this mean? Implement and **TEST one method at a time**.
For each class and each function within that class:

- Implement one function/method and **test it thoroughly** (write a main file with multiple test cases + edge cases if applicable).
- Only when you are certain that function works correctly and matches the specification, move on to the next.
- Implement the next function/method and test in the same fashion.

How do you do this? Write your own `main()` function to test your classes (we will provide a starting one with this first project, but you must add to it). For later projects in this course, we will not grade your test program, but you must always write one to test your classes. 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. Sometimes functions depend on one another. If you need to use a function you have not yet implemented, you can **use stubs**: a dummy implementation that always returns a single value for testing. 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.

Part 4: Submission

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

- `Dish.hpp`
- `Dish.cpp`
- `test.cpp`

Although Gradescope allows multiple submissions, it is not a platform for testing and/or debugging, and it should not be used for that purpose. You MUST test and debug your program locally. To help prevent over-reliance on Gradescope for testing, only 5 submissions per day will be allowed.

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 College. 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.

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 9/16**.
No late submissions will be accepted.

Important

You must **start working on the projects as soon as they are assigned** to detect any problems 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 extensions and no negotiation about project grades after the submission deadline.

Help

Help is available via drop-in tutoring in Lab 1001B (see Blackboard 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, **the days leading up to the due date will be crowded and you will not be able to get much help then.**

Authors: Michael Russo, Georgina Woo, Prof. Wole

Credit to Prof. Ligorio