Algorithmic Adventures: The Virtual Bistro – Inheritance and Polymorphism with the Dish Class

---

Project Overview

This semester, you will continue your culinary journey by expanding the virtual bistro simulation you began in Project 1. Building on the foundation of the `Dish` class, you will create subclasses that represent specific types of dishes: `Appetizer`, `MainCourse`, and `Dessert`. Each subclass will introduce new attributes and behaviors, demonstrating your understanding of inheritance and polymorphism in Object-Oriented Programming (OOP).

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 interact with these dishes, allowing them to explore different flavors, cooking methods, and culinary styles.

The goal of this project is to strengthen your OOP skills by implementing subclasses of the `Dish` class. You will also learn how to use inheritance to create specialized classes that extend the functionality of a base class.

Here is the link to accept this project on Github Classroom: https://classroom.github.com/a/i0w-kOor

---

Part 1: 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: Additional Resources

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

- Code Beauty on inheritance programming
- Caleb Curry on base and subclass inheritance
- Geeks for Geeks on Enumerated Types
- Educative on Structs

Part 3: The Subclasses of Dish

You will implement the `Appetizer`, `MainCourse`, and `Dessert` classes, each of which inherits from the `Dish` class.

Key Concepts

- The `Appetizer`, `MainCourse`, and `Dessert` classes will extend the `Dish` class by introducing new attributes and functions. This is an exercise in inheritance, where the base class (`Dish`) is extended by specialized subclasses.

Important Notes (READ THEM)

- All function arguments should be references in this project, unless specified otherwise.
- You should follow all aforementioned programming conventions and best practices.
- DO NOT USE NAMESPACE STD ... You are responsible for any points you lose by doing so.
- If the return type of a function or the type of a parameter isn't specified, you're expected to be able to deduce what it should be based on the private member variables of the class and what the function is supposed to accomplish.
- In this project, you are not being asked to create display functions for the subclasses.

The Appetizer Class

The `Appetizer` class is a subclass of `Dish` that represents an appetizer dish.

The Appetizer class must define the following types INSIDE the class definition:

- **Serving Style (enum)**: Describes how the appetizer is served.
  - Enum: `ServingStyle { PLATED, FAMILY_STYLE, BUFFET }`

The Appetizer class must have the following private member variables:

- `ServingStyle serving_style_`: The serving style of the appetizer.
- `int spiciness_level_`: An integer representing the spiciness level of the appetizer on a scale of 1 to 10.
- `bool vegetarian_`: A flag indicating if the appetizer is vegetarian.

The Appetizer class must have the following public member functions:

Constructors:

- **Default Constructor:**

```
/**
 * Default constructor.
```

```
    * Initializes all private members with default values.
    */
```

- **Parameterized Constructor:**

```
   /**
    * Parameterized constructor.
    * @param name The name of the appetizer.
    * @param ingredients The ingredients used in the appetizer.
    * @param prep_time The preparation time in minutes.
    * @param price The price of the appetizer.
    * @param cuisine_type The cuisine type of the appetizer.
    * @param serving_style The serving style of the appetizer.
    * @param spiciness_level The spiciness level of the appetizer.
    * @param vegetarian Flag indicating if the appetizer is vegetarian.
    */
```

Accessors and Mutators:

- **setServingStyle:**

```
/**
 * Sets the serving style of the appetizer.
 * @param serving_style The new serving style.
 * @post Sets the private member `serving_style_` to the value of the
parameter.
 */
setServingStyle
```

- **getServingStyle:**

```
/**
 * @return The serving style of the appetizer (as an enum).
 */
getServingStyle
```

- **setSpicinessLevel:**

```
/**
 * Sets the spiciness level of the appetizer.
 * @param spiciness_level An integer representing the spiciness level of
the appetizer.
```

```
 * @post Sets the private member `spiciness_level_` to the value of the
parameter.
 */
setSpicinessLevel
```

- **getSpicinessLevel:**

```
/**
 * @return The spiciness level of the appetizer.
 */
getSpicinessLevel
```

- **setVegetarian:**

```
/**
 * Sets the vegetarian flag of the appetizer.
 * @param vegetarian A boolean indicating if the appetizer is
vegetarian.
 * @post Sets the private member `vegetarian_` to the value of the
parameter.
 */
setVegetarian
```

- **isVegetarian:**

```
/**
 * @return True if the appetizer is vegetarian, false otherwise.
 */
isVegetarian
```

The MainCourse Class

The `MainCourse` class is a subclass of `Dish` that represents a main course dish.

The MainCourse class must define the following types INSIDE the class definition:

- **Cooking Method (enum)**: Describes the method used to cook the main course.
  - Enum: `CookingMethod { GRILLED, BAKED, FRIED, STEAMED, RAW }`
- **SideDish (struct)**: Represents a side dish associated with the main course.

    - Struct: `SideDish` with members `std::string name` and `Category category`.
- **Category (enum)**: Describes the category of the side dish.
    - Enum: `Category { GRAIN, PASTA, LEGUME, BREAD, SALAD, SOUP, STARCHES, VEGETABLE }`

The MainCourse class must have the following private member variables:

- `CookingMethod cooking_method_`: The cooking method used for the main course.
- `std::string protein_type_`: A string representing the type of protein used in the main course.
- `std::vector<SideDish> side_dishes_`: A vector containing SideDish structs representing the side dishes served with the main course.
- `bool gluten_free_`: A flag indicating if the main course is gluten-free.

The MainCourse class must have the following public member functions:

Constructors:

- **Default Constructor:**

```
  /**
   * Default constructor.
   * Initializes all private members with default values.
   */
```

- **Parameterized Constructor:**

```
  /**
   * Parameterized constructor.
   * @param name The name of the main course.
   * @param ingredients A vector of the ingredients used in the main
course.
   * @param prep_time The preparation time in minutes.
   * @param price The price of the main course.
   * @param cuisine_type The cuisine type of the main course.
   * @param cooking_method The cooking method used for the main course.
   * @param protein_type The type of protein used in the main course.
   * @param side_dishes A vector of the side dishes served with the main
course.
   * @param gluten_free Boolean flag indicating if the main course is
gluten-free.
   */
```

Accessors and Mutators:

- **setCookingMethod:**

```
/**
 * Sets the cooking method of the main course.
 * @param cooking_method The new cooking method.
 * @post Sets the private member `cooking_method_` to the value of the
parameter.
 */
setCookingMethod
```

- **getCookingMethod:**

```
/**
 * @return The cooking method of the main course (as an enum).
 */
getCookingMethod
```

- **setProteinType:**

```
/**
 * Sets the type of protein in the main course.
 * @param protein_type A string representing the type of protein.
 * @post Sets the private member `protein_type_` to the value of the
parameter.
 */
setProteinType
```

- **getProteinType:**

```
/**
 * @return The type of protein in the main course.
 */
getProteinType
```

- **addSideDish:**

```
/**
 * Adds a side dish to the main course.
 * @param side_dish A SideDish struct containing the name and category
of the side dish.
```

```
 * @post Adds the side dish to the `side_dishes_` vector.
 */
addSideDish
```

- **getSideDishes:**

```
/**
 * @return A vector of SideDish structs representing the side dishes
served with the main course.
 */
getSideDishes
```

- **setGlutenFree:**

```
/**
 * Sets the gluten-free flag of the main course.
 * @param gluten_free A boolean indicating if the main course is gluten-
free.
 * @post Sets the private member `gluten_free_` to the value of the
parameter.
 */
setGlutenFree
```

- **isGlutenFree:**

```
/**
 * @return True if the main course is gluten-free, false otherwise.
 */
isGlutenFree
```

The Dessert Class

The `Dessert` class is a subclass of `Dish` that represents a dessert dish.

The Dessert class must define the following types INSIDE the class definition:

- **Flavor Profile (enum)**: Describes the dominant flavor of the dessert.
  - Enum: `FlavorProfile { SWEET, BITTER, SOUR, SALTY, UMAMI }`

The Dessert class must have the following private member variables:

- `FlavorProfile flavor_profile_` : The flavor profile of the dessert.
- `int sweetness_level_` : An integer representing the sweetness level of the dessert on a scale of 1 to 10.
- `bool contains_nuts_` : A flag indicating if the dessert contains nuts.

The Dessert class must have the following public member functions:

Constructors:

- **Default Constructor:**

```
  /**
   * Default constructor.
   * Initializes all private members with default values.
   */
```

- **Parameterized Constructor:**

```
/**
   * Parameterized constructor.
   * @param name The name of the dessert.
   * @param ingredients The ingredients used in the dessert.
   * @param prep_time The preparation time in minutes.
   * @param price The price of the dessert.
   * @param cuisine_type The cuisine type of the dessert.
   * @param flavor_profile The flavor profile of the dessert.
   * @param sweetness_level The sweetness level of the dessert.
   * @param contains_nuts Flag indicating if the dessert contains nuts.
   */
```

Accessors and Mutators:

- **setFlavorProfile:**

```
/**
 * Sets the flavor profile of the dessert.
 * @param flavor_profile The new flavor profile.
 * @post Sets the private member `flavor_profile_` to the value of the
parameter.
 */
setFlavorProfile
```

- **getFlavorProfile:**

```
/**
 * @return The flavor profile of the dessert (as an enum).
 */
getFlavorProfile
```

- **setSweetnessLevel:**

```
/**
 * Sets the sweetness level of the dessert.
 * @param sweetness_level An integer representing the sweetness level of
the dessert.
 * @post Sets the private member `sweetness_level_` to the value of the
parameter.
 */
setSweetnessLevel
```

- **getSweetnessLevel:**

```
/**
 * @return The sweetness level of the dessert.
 */
getSweetnessLevel
```

- **setContainsNuts:**

```
/**
 * Sets the contains_nuts flag of the dessert.
 * @param contains_nuts A boolean indicating if the dessert contains
nuts.
 * @post Sets the private member `contains_nuts_` to the value of the
parameter.
 */
setContainsNuts
```

- **containsNuts:**

```
/**
 * @return True if the dessert contains nuts, false otherwise.
```

```
 */
containsNuts
```

Part 4: 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.

Important Note: Some of the functions are returning enums, which means that **you can either return enums in the output or convert those enums into the corresponding strings.** For Project 2, the test.cpp grading will not be as strict as it was in Project 1. This is also the last project in which we will require the submission of your test file.

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

1. **Instantiate an appetizer with the default constructor:**

   - Set its spiciness level to 7 using the appropriate setter function.
   - Set its serving style to FAMILY_STYLE using the appropriate setter function.
   - Set the vegetarian flag to true using the appropriate setter function.
   - Print out the appetizer's information using the accessor functions.

   Expected Output:

   ```
   Dish Name: UNKNOWN
   Ingredients:
   Preparation Time: 0 minutes
   Price: $0.00
   Cuisine Type: OTHER
   Spiciness Level: 7
   Serving Style: FAMILY_STYLE
   Vegetarian: True
   ```

2. **Instantiate a main course with the parameterized constructor:**

   - Name: "Grilled Chicken"
   - Ingredients: ["Chicken", "Olive Oil", "Garlic", "Rosemary"]

- Preparation Time: 30
- Price: 18.99
- Cuisine Type: AMERICAN
- Cooking Method: GRILLED
- Protein Type: "Chicken"
- Add side dishes: "Mashed Potatoes" with category "Starches" and "Green Beans" with category "Vegetable"
- Set the gluten-free flag to true using the appropriate setter function.
- Print out the main course's information using the accessor functions.

Expected Output:

```
Dish Name: Grilled Chicken
Ingredients: Chicken, Olive Oil, Garlic, Rosemary
Preparation Time: 30 minutes
Price: $18.99
Cuisine Type: AMERICAN
Cooking Method: GRILLED
Protein Type: Chicken
Side Dishes: Mashed Potatoes (Starches), Green Beans (Vegetable)
Gluten-Free: True
```

3. **Instantiate a dessert with the parameterized constructor:**

- Name: "Chocolate Cake"
- Ingredients: ["Flour", "Sugar", "Cocoa Powder", "Eggs"]
- Preparation Time: 45
- Price: 7.99
- Cuisine Type: FRENCH
- Flavor Profile: SWEET
- Sweetness Level: 9
- Set the contains_nuts flag to false using the appropriate setter function.
- Print out the dessert's information using the accessor functions.

Expected Output:

```
Dish Name: Chocolate Cake
Ingredients: Flour, Sugar, Cocoa Powder, Eggs
Preparation Time: 45 minutes
```

```
Price: $7.99
Cuisine Type: FRENCH
Flavor Profile: SWEET
Sweetness Level: 9
Contains Nuts: False
```

Part 5: Submission

You will submit your solution to Gradescope via GitHub Classroom. You do not need to submit Dish.cpp or Dish.hpp from the starter code. Do not modify those files. The autograder will grade the following files:

- `Appetizer.hpp`
- `Appetizer.cpp`
- `MainCourse.hpp`
- `MainCourse.cpp`
- `Dessert.hpp`
- `Dessert.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 9/30**.
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.**

THIS PROJECT IS ESPECIALLY LONG. If you understand Project 1 and paid attention to the lectures regarding inheritance, it will not be difficult, but it will be *a lot of coding*. ***START EARLY!***

Authors: Michael Russo, Georgina Woo, Prof. Wole

Credit to Prof. Ligorio