Food Delivery

I. Description of the Project

Overview

The food delivery system is a comprehensive software solution designed to manage the interactions and operations between customers, restaurants, drivers, and orders in a food delivery service. This system includes several classes that encapsulate the functionalities required to process and manage food orders, including the ability to create and manage clients, restaurants, items, orders, and drivers.Class and Data Structures Overview

The project comprises several classes to represent different entities and their interactions:

- **Item**: Represents a menu item in a restaurant.
- **Order**: Represents an order created by a customer, containing various items from a restaurant.
- **Restaurant**: Represents a restaurant, managing its menu and the orders it receives.
- **Client**: Represents a customer who places orders.
- **Driver**: Represents a driver responsible for delivering orders.
- **FoodDelivery**: The central class managing the entire system, including clients, drivers, restaurants, and orders.
- **OrderStatus**: An enumeration representing the different states an order can be in, such as PREPARING, READY, SENT, and COMPLETED.

Restrictions, Limits, and Assumptions

Data Integrity

- **Validation**: The system ensures data integrity by validating inputs for each entity. IDs must be positive integers, and names cannot be empty. This prevents the introduction of invalid or corrupt data into the system.

Singleton Pattern

- **Design Pattern**: The `FoodDelivery` class implements the Singleton pattern to ensure that only one instance of the system exists throughout its lifetime. This guarantees consistent and centralized management of data and operations.

Order Management

- **Driver Assignment**: Each order can have only one assigned driver at a time. This ensures clear responsibility and prevents confusion in the delivery process.
- **Linkage**: Orders must be linked to valid clients and restaurants. This means an order cannot exist without being associated with a specific client who placed it and a restaurant from which it originated.

Status Management

- **Order States**: Orders transition through specific states—ORDERED, PREPARING, SENT, and DELIVERED—to reflect their progress accurately. This provides a clear and structured workflow for order processing and tracking.

Memory Management

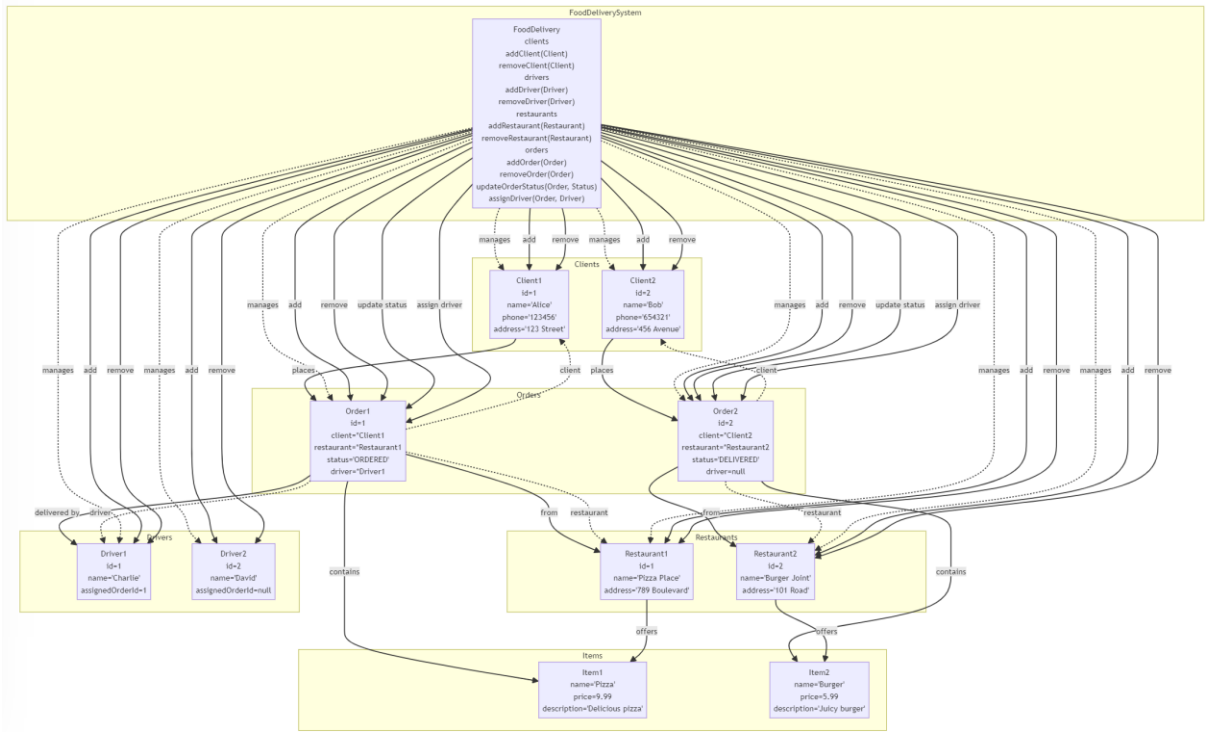
- **Assumptions**: Proper memory management is assumed, with the system handling the deletion of dynamically allocated objects to prevent memory leaks. This ensures efficient use of resources and prevents potential memory issues.

Concurrency

- **Environment**: The current implementation assumes a single-threaded environment. Concurrent modifications to shared data structures are not handled, implying that the system is designed to operate in environments where simultaneous operations by multiple threads are not expected.

Summary

This overview provides a high-level understanding of the project's scope and design, laying the foundation for a robust and maintainable food delivery system. By ensuring data integrity, utilizing the Singleton pattern, managing orders effectively, maintaining proper memory management, and operating within a single-threaded environment, the system is designed to be reliable and efficient.



I dashed arrows to represent pointer references.

1. FoodDeliveryApp (Main Class)

The **FoodDeliveryApp** class serves as the central entry point of the system. It manages the interactions between users, restaurants, orders, and drivers. This class is responsible for initializing the system, processing orders, and orchestrating the interactions between all other classes.

clients: List of clients.

drivers: List of drivers.

restaurants: List of restaurants.

orders: List of orders.

Methods:

static FoodDelivery& getInstance(): Retrieves the singleton instance of the system.

void addClient(Client* client): Adds a client to the system.

void addDriver(Driver* driver): Adds a driver to the system.

void addRestaurant(Restaurant* restaurant): Adds a restaurant to the system.

void addOrder(Order* order): Adds an order to the system.

void updateOrderStatus(int orderId, OrderStatus status): Updates the status of an order.

void assignDriverToOrder(int orderId): Assigns a driver to an order.

bool removeClient(int clientId): Removes a client from the system.

bool removeDriver(int driverId): Removes a driver from the system.

bool removeRestaurant(int restaurantId): Removes a restaurant from the system.

bool removeOrder(int orderId): Removes an order from the system.

```
#ifndef FOODDELIVERY_H
#define FOODDELIVERY_H

#include <vector>
#include "client.h"
#include "driver.h"
#include "restaurant.h"
#include "order.h"
#include "orderstatus.h"
#include <algorithm>

class FoodDelivery {
public:
    static Client* findClientById(const FoodDelivery& fd, int clientId);
    static Driver* findDriverById(const FoodDelivery& fd, int driverId);
    static Restaurant* findRestaurantById(const FoodDelivery& fd, int
restaurantId);
    static Order* findOrderById(const FoodDelivery& fd, int orderId);
    static Item* findItemById(const FoodDelivery& fd, int restaurantId, const
std::string& itemName);
    FoodDelivery();

    bool addClient(const Client& client);
    bool addDriver(const Driver& driver);
    bool addRestaurant(const Restaurant& restaurant);
    bool addOrder(const Order& order);
```

```
    bool updateOrderStatus(int orderId, OrderStatus status);
    bool assignDriverToOrder(int orderId);
    bool removeClient(int clientId);
    bool removeDriver(int driverId);
    bool removeRestaurant(int restaurantId);
    bool removeOrder(int orderId);

    const std::vector<Order>& getOrders() const;

private:
    Client* findClientById(int clientId) const;
    Driver* findDriverById(int driverId) const;
    Restaurant* findRestaurantById(int restaurantId) const;
    Order* findOrderById(int orderId) const;
    Item* findItemById(int restaurantId, const std::string& itemName) const;
private:
    std::vector<Client> clients;
    std::vector<Driver> drivers;
    std::vector<Restaurant> restaurants;
    std::vector<Order> orders;

    void reassignDriver(Order* order);
};

#endif // FOODDELIVERY_H
```

Significant Logical Changes Between the Original and Updated `FoodDelivery` Class

1. **Static Methods for Finding Entities**:
   - **Original Code**: Does not include static methods for entity lookup.
   - **Updated Code**: Introduces static methods (`findClientById`, `findDriverById`, `findRestaurantById`, `findOrderById`, `findItemById`) to facilitate finding entities by their IDs. This change improves code modularity and reusability by centralizing lookup logic.
2. **Entity Addition and Validation**:
   - **Original Code**: Provides basic methods for adding users and restaurants without detailed validation.
   - **Updated Code**: Adds methods for validating and adding entities (`addClient`, `addDriver`, `addRestaurant`, `addOrder`), ensuring no duplicate IDs and that IDs are valid (positive integers). This enhances data integrity.
3. **Comprehensive Entity Management**:
   - **Original Code**: Basic management functions focused on adding and assigning entities.
   - **Updated Code**: Introduces methods to add, update, and remove clients, drivers, restaurants, and orders (`removeClient`, `removeDriver`, `removeRestaurant`, `removeOrder`). This allows for more thorough management of the system's entities throughout their lifecycle.
4. **Order Status and Driver Assignment**:
   - **Original Code**: Contains simple methods for placing orders and assigning drivers.
   - **Updated Code**: Includes methods to update the status of an order (`updateOrderStatus`) and assign drivers to orders (`assignDriverToOrder`).

This provides better tracking of order progress and more efficient management of driver assignments.

5. **Private Helper Methods**:
    - **Original Code**: Lacks private helper methods for internal logic.
    - **Updated Code**: Implements private helper methods for internal operations (`findClientById`, `findDriverById`, `findRestaurantById`, `findOrderById`, `findItemById`, `reassignDriver`). This separation of concerns makes the codebase more maintainable and scalable.
6. **Order Reassignment Logic**:
    - **Original Code**: No explicit logic for handling order reassignment.
    - **Updated Code**: Includes a method (`reassignDriver`) to handle reassignment of drivers when necessary, ensuring continuous order delivery in case of issues with initially assigned drivers.
7. **Error Handling and Edge Cases**:
    - **Original Code**: Limited handling of edge cases like duplicate or invalid IDs.
    - **Updated Code**: Comprehensive error handling for edge cases such as duplicate IDs and invalid input through assertions and checks, leading to a more robust system.
8. **Client vs. User Terminology**:
    - **Original Code**: Uses `User` class terminology.
    - **Updated Code**: Switches to `Client` class terminology, potentially indicating a clearer role definition and responsibility separation between different types of users (clients, drivers, etc.).

The testFoodDelivery function validates the functionality of the FoodDelivery class, which manages the overall operations of the system, including adding and removing clients, drivers, restaurants, and orders, as well as updating order status and assigning drivers to orders. The steps are as follows:

1. **Object Creation**: This step initializes instances of Client, Driver, Restaurant, Item, and Order to test the FoodDelivery class's ability to manage these objects.
2. **Adding and Retrieving Objects**: The test adds clients, drivers, restaurants, and orders to the FoodDelivery system and then retrieves them to ensure they are correctly stored and retrievable. It uses assertions to verify that each added object can be retrieved by its ID.
3. **Updating Order Status**: The test updates the status of an order using the updateOrderStatus method and verifies that the update is correctly reflected in the order object by checking the order's status.
4. **Assigning Driver to Order**: The test assigns a driver to an order using the assignDriverToOrder method and verifies that the driver is correctly assigned by checking both the driver's assigned order ID and the order's assigned driver.

5. **Removing Objects**: The test removes clients, drivers, restaurants, and orders from the system using the respective remove methods (removeClient, removeDriver, removeRestaurant, removeOrder) and checks that they are removed correctly by verifying that they can no longer be retrieved by their IDs.
6. **Output Message**: A success message is printed if all tests pass, indicating that the FoodDelivery class is functioning as expected.

```
void testFoodDelivery() {
    FoodDelivery system;

    // Object Creation - Initialize objects for testing
    Client* client1 = new Client(1, "Alice", "1234567890", "123 Main St");
    Client* client2 = new Client(2, "Bob", "0987654321", "456 Elm St");
    Driver* driver1 = new Driver(1, "John");
    Driver* driver2 = new Driver(2, "Jane");
    Restaurant* restaurant1 = new Restaurant(1, "Pizza Place", "789 Park Ave");
    Restaurant* restaurant2 = new Restaurant(2, "Burger Joint", "101 Pine St");
    Item* pizza = new Item("Pizza", 10.0, "Cheese, Tomato, Dough");
    Item* burger = new Item("Burger", 8.0, "Beef, Lettuce, Tomato, Bun");
    Item* fries = new Item("Fries", 3.0, "Potatoes, Salt, Oil");
    restaurant1->addItem(pizza);
    restaurant2->addItem(burger);
    restaurant2->addItem(fries);
    Order* order1 = new Order(1, client1, restaurant1);
    Order* order2 = new Order(2, client2, restaurant2);
    order1->addItem(pizza, 2);
    order2->addItem(burger, 1);
    order2->addItem(fries, 2);

    // Adding and Retrieving Objects - Test adding and retrieving objects
    assert(system.addClient(*client1) == true);
    assert(system.addClient(*client2) == true);
    assert(system.addDriver(*driver1) == true);
    assert(system.addDriver(*driver2) == true);
    assert(system.addRestaurant(*restaurant1) == true);
    assert(system.addRestaurant(*restaurant2) == true);
    assert(system.addOrder(*order1) == true);
    assert(system.addOrder(*order2) == true);

    assert(system.findClientById(system, 1) == client1);
    assert(system.findClientById(system, 2) == client2);
    assert(system.findDriverById(system, 1) == driver1);
    assert(system.findDriverById(system, 2) == driver2);
    assert(system.findRestaurantById(system, 1) == restaurant1);
    assert(system.findRestaurantById(system, 2) == restaurant2);
    assert(system.findOrderById(system, 1) == order1);
    assert(system.findOrderById(system, 2) == order2);

    // Updating Order Status - Test updating the status of an order
    assert(system.updateOrderStatus(1, PREPARING) == true);
    assert(order1->getStatus() == PREPARING);
    assert(system.updateOrderStatus(2, PREPARING) == true);
    assert(order2->getStatus() == PREPARING);

    // Assigning Driver to Order - Test assigning a driver to an order
    assert(system.assignDriverToOrder(1) == true);
    assert(order1->getDriver() == driver1);
```

```
    assert(driver1->getAssignedOrderId() == 1);
    assert(system.assignDriverToOrder(2) == true);
    assert(order2->getDriver() == driver2);
    assert(driver2->getAssignedOrderId() == 2);

    // Removing Objects - Test removing clients, drivers, restaurants, and orders
    assert(system.removeClient(1) == true);
    assert(system.findClientById(system, 1) == nullptr);
    assert(system.removeDriver(1) == true);
    assert(system.findDriverById(system, 1) == nullptr);
    assert(system.removeRestaurant(1) == true);
    assert(system.findRestaurantById(system, 1) == nullptr);
    assert(system.removeOrder(1) == true);
    assert(system.findOrderById(system, 1) == nullptr);

    // Output Message - Print a success message if all tests pass
    std::cout << "FoodDelivery class test passed." << std::endl;

    // Cleanup - Free dynamically allocated memory
    delete client1;
    delete client2;
    delete driver1;
    delete driver2;
    delete pizza;
    delete burger;
    delete fries;
    delete order1;
    delete order2;
    delete restaurant1;
    delete restaurant2;
}
```

2 Client Class

- Purpose: Represents a client.
- Attributes:
  - id: Client ID.
  - name: Client name.
  - phone: Client phone number.
  - address: Client address.
- Methods:
  - Client(int id, const std::string& name, const std::string& phone, const std::string& address): Constructor to initialize client information.
  - int getId() const: Retrieves the client ID.
  - std::string getName() const: Retrieves the client name.
  - std::string getPhone() const: Retrieves the client phone number.
  - std::string getAddress() const: Retrieves the client address

```
The Client class represents a customer, each customer has a unique ID, name,
phone number, and address. This class provides an interface for obtaining this
information.
```

```h
#ifndef CLIENT_H
#define CLIENT_H

#include <string>

class Client {
public:
    Client(int id, const std::string& name, const std::string& phone, const
std::string& address);
    int getId() const;
    std::string getName() const;
    std::string getPhone() const;
    std::string getAddress() const;


private:
    int id;
    std::string name;
    std::string phone;
    std::string address;
};

#endif // CLIENT_H
```

Key Differences

1. **Identifier Type**:
   - **Original Code**: Uses `std::string` for the user identifier (`userId`).
   - **Updated Code**: Uses `int` for the client identifier (`id`).
2. **Additional Attributes**:
   - **Original Code**: Does not include a phone number attribute.
   - **Updated Code**: Adds a `std::string phone` attribute to store the client's phone number.
3. **Order Placement**:
   - **Original Code**: Includes a method to place an order (`placeOrder`) that interacts with the `FoodDeliveryApp` class.
   - **Updated Code**: Does not include a method for placing orders directly in the `Client` class; assumes order placement is handled elsewhere.
4. **Simplified Interface**:
   - **Original Code**: Provides a method for placing orders directly from the user class, integrating tightly with the order system.
   - **Updated Code**: Focuses on basic attributes and getter methods, making the `Client` class simpler and more modular.
5. **Constructor**:
   - **Original Code**: No explicit constructor defined.

- **Updated Code**: Includes a constructor that initializes the client with an ID, name, phone number, and address, ensuring all essential attributes are set upon creation.

6. **Modularity and Separation of Concerns**:
   - **Original Code**: Integrates order placement within the user class, potentially mixing concerns.
   - **Updated Code**: Separates concerns by focusing on client attributes and basic information retrieval, likely leaving order placement and management to other parts of the system (e.g., `FoodDelivery` class).
7. **Include Guards**:
   - **Original Code**: Does not use include guards.
   - **Updated Code**: Uses include guards (`#ifndef CLIENT_H`, `#define CLIENT_H`, `#endif // CLIENT_H`) to prevent multiple inclusions.

Client Class Test Explanation:

The **testClient** function validates the behavior of the **Client** class. This function ensures that the **Client** class can correctly handle object creation, attribute assignment, and retrieval.

1. **Object Creation**: An instance of the **Client** class is created using specific values for the client ID, name, phone number, and address. This step checks the constructor's ability to properly initialize a **Client** object.
2. **Attribute Testing**: Assertions are used to verify that the attributes of the **Client** object (**clientId**, **name**, **phone**, and **address**) match the values passed to the constructor. This ensures that the object's properties are being set correctly.
3. **Output Message**: If all assertions pass without triggering an error, it means the **Client** object is functioning as expected, and a success message is printed.

```
void testClient(FoodDelivery& fd) {
    Client* client1 = new Client(1, "Alice", "1234567890", "123 Main St");
    Client* client2 = new Client(1, "Bob", "0987654321", "456 Elm St");

    assert(fd.addClient(*client1) == true);
    assert(fd.addClient(*client2) == false);

    assert(client1->getId() == 1);
    assert(client1->getName() == "Alice");
    assert(client1->getPhone() == "1234567890");
    assert(client1->getAddress() == "123 Main St");

    Client* clientNegativeId = new Client(-1, "Negative", "0000000000", "Negative
St");
    assert(fd.addClient(*clientNegativeId) == false);

    delete client1;
    delete client2;
    delete clientNegativeId;

    std::cout << "Client class test passed." << std::endl;
}
```

Differences Between the Original and Updated Test

1. Function Signature:
    - The original test function does not take any parameters, while the updated test function takes a FoodDelivery object as a parameter.
2. Scope of Testing:
    - The original test only validates a single Client object independently.
    - The updated test validates multiple Client objects within the context of the FoodDelivery system.
3. Dynamic Memory Allocation:
    - The original test does not use dynamic memory allocation.
    - The updated test uses dynamic memory allocation to create Client objects on the heap.
4. Handling Duplicate IDs:
    - The original test does not test for duplicate client IDs.
    - The updated test checks for the handling of duplicate client IDs, ensuring the system does not accept them.
5. Error Handling for Invalid IDs:
    - The original test does not handle invalid client IDs.
    - The updated test includes a check for invalid client IDs (negative IDs) to ensure the system properly rejects them.
6. Integration with FoodDelivery System:
    - The original test does not integrate with the FoodDelivery system.
    - The updated test integrates Client objects into the FoodDelivery system, testing the addition and management of clients within the system.
7. Memory Management:
    - The original test does not require explicit memory management.
    - The updated test includes explicit deletion of dynamically allocated Client objects to prevent memory leaks.

The Restaurant class represents a restaurant in the food delivery system. It manages the restaurant's menu and orders, providing methods to add and remove menu items and orders. This class also allows retrieving restaurant information and managing orders received by the restaurant.

Purpose

Represents a restaurant that manages its menu and orders.

Attributes

- **id: Unique identifier for the restaurant.**
- **name: Name of the restaurant.**

- **address: Physical address of the restaurant.**
- **menu: A list of pointers to Item objects representing the restaurant's menu items.**
- **orders: A list of pointers to Order objects representing the orders received by the restaurant.**

Methods

- **Constructor:**
  - **Restaurant(int id, const std::string& name, const std::string& address): Initializes the restaurant with an ID, name, and address.**
- **Getters:**
  - **int getId() const: Retrieves the restaurant ID.**
  - **std::string getName() const: Retrieves the restaurant name.**
  - **std::string getAddress() const: Retrieves the restaurant address.**
  - ***const std::vector<Item>& getMenu() const*: Retrieves the restaurant menu.**
  - ***const std::vector<Order>& getOrders() const*: Retrieves the restaurant orders.**
- **Menu Management:**
  - ***void addItem(Item* item)**: Adds an item to the restaurant's menu.**
  - ***void removeItem(Item* item)**: Removes an item from the restaurant's menu.**
- **Order Management:**
  - ***void addOrder(Order* order)**: Adds an order to the restaurant.**
  - ***const std::vector<Order>& getOrders() const*: Retrieves the orders received by the restaurant.**

```
##ifndef RESTAURANT_H
#define RESTAURANT_H

#include <vector>
#include <string>
#include "item.h"
#include "order.h"

class Order;

class Restaurant {
public:
    Restaurant(int id, const std::string& name, const std::string& address);

    int getId() const;
    std::string getName() const;
    std::string getAddress() const;

    void addItem(Item* item);
    void removeItem(Item* item);
```

```
    const std::vector<Item*>& getMenu() const;

    void addOrder(Order* order);
    const std::vector<Order*>& getOrders() const;

private:
    int id;
    std::string name;
    std::string address;
    std::vector<Item*> menu;
    std::vector<Order*> orders;
};

#endif // RESTAURANT_H
```

Restaurant Class

1. **Attributes and Data Structures**:
    - **Original Code**:
        - Uses a mix of basic and custom types: `std::string`, `std::vector<std::string>`.
    - **Updated Code**:
        - Uses custom types (`Client*`, `Restaurant*`, `Driver*`) for associations.
        - Uses `std::vector<std::pair<Item*, int>>` for `items` to store items and their quantities.
        - Uses `OrderStatus` enumeration for `status`.
2. **Methods**:
    - **Original Code**:
        - Likely has basic methods for adding menu items and receiving orders.
    - **Updated Code**:
        - **Constructors and Getters**:
            - `Order(int id, Client* client, Restaurant* restaurant);` - Initializes the order with ID, client, and restaurant.
            - `int getId() const;` - Retrieves the order ID.
            - `Client* getClient() const;` - Retrieves the client.
            - `Restaurant* getRestaurant() const;` - Retrieves the restaurant.
            - `const std::vector<std::pair<Item*, int>>& getItems() const;` - Retrieves the items and quantities.
            - `OrderStatus getStatus() const;` - Retrieves the order status.
            - `Driver* getDriver() const;` - Retrieves the assigned driver.
        - **Setters and Mutators**:
            - `void addItem(Item* item, int quantity);` - Adds an item and quantity to the order.
            - `void assignDriver(Driver* driver);` - Assigns a driver to the order.
            - `void updateStatus(OrderStatus status);` - Updates the order status.
3. **Interaction with Other Classes**:

- **Original Code**:
  - Limited interaction with other classes.
- **Updated Code**:
  - Extensive interaction with `Client`, `Restaurant`, `Driver`, and `Item` classes, enhancing the complexity and capabilities of the `Order` class.
  - Integrates with the `FoodDelivery` system for operations like adding, updating, and managing orders.

4. **Order Status and Driver Management**:
   - **Original Code**:
     - Basic handling of orders without detailed status tracking or driver assignment.
   - **Updated Code**:
     - Detailed management of order statuses (`ORDERED`, `PREPARING`, `SENT`, `DELIVERED`).
     - Driver assignment and status updates, enhancing the workflow and management of the delivery process.

The testRestaurant function examines the integrity of the Restaurant class. This function ensures that the Restaurant class can correctly handle object creation, attribute assignment, menu management, and order management.

1. **Object Creation:** A Restaurant object is created with an ID, name, and address. This step verifies the constructor's ability to properly initialize a Restaurant object.
2. **Attribute Testing:** Assertions check that the Restaurant object's restaurantId, name, and address are correctly assigned.
3. **Menu Item Addition:** We simulate adding a menu item to the Restaurant object and then verify that the menu contains the correct number of items, and that the new item's name is as expected. This ensures that menu management within the Restaurant class works properly.
4. **Order Management:** We simulate adding an order to the Restaurant object and verify that the order list is updated correctly. This ensures that the restaurant can manage orders properly.
5. **Output Message:** Upon successful assertions, a message confirms that all restaurant tests have passed.

```
#include <cassert>
#include "restaurant.h"
#include "item.h"
#include "order.h"
void testRestaurant(FoodDelivery& fd) {
    Restaurant* restaurant1 = new Restaurant(1, "Pizza Place", "123 Main St");
    Restaurant* restaurant2 = new Restaurant(1, "Burger Joint", "456 Elm St");

    assert(fd.addRestaurant(*restaurant1) == true);
    assert(fd.addRestaurant(*restaurant2) == false);

    assert(restaurant1->getId() == 1);
    assert(restaurant1->getName() == "Pizza Place");
    assert(restaurant1->getAddress() == "123 Main St");

    Restaurant* restaurantNegativeId = new Restaurant(-1, "Negative", "Negative
St");
    assert(fd.addRestaurant(*restaurantNegativeId) == false);
```

```
    Item* pizza = new Item("Pizza", 10.0, "Cheese, Tomato, Dough");
    restaurant1->addItem(pizza);
    assert(restaurant1->getMenu().size() == 1);
    assert(restaurant1->getMenu()[0]->getName() == "Pizza");
    assert(restaurant1->getMenu()[0]->getPrice() == 10.0);
    assert(restaurant1->getMenu()[0]->getDescription() == "Cheese, Tomato,
Dough");

    Client* client1 = new Client(1, "Alice", "1234567890", "123 Main St");
    Order* order1 = new Order(1, client1, restaurant1);
    restaurant1->addOrder(order1);
    assert(fd.addClient(*client1) == true);
    assert(fd.addOrder(*order1) == true);

    assert(restaurant1->getOrders().size() == 1);
    assert(restaurant1->getOrders()[0]->getId() == 1);

    delete client1;
    delete order1;
    delete pizza;
    delete restaurant1;
    delete restaurant2;
    delete restaurantNegativeId;

    std::cout << "Restaurant class test passed." << std::endl;
}
```

Differences Between the Original and Updated testRestaurant Function

1. **Function Signature**:
   - **Original Code**: The function does not take any parameters.
   - **Updated Code**: The function takes a FoodDelivery object as a parameter to test the integration of the Restaurant class within the FoodDelivery system.
2. **Object Creation**:
   - **Original Code**: Creates a single Restaurant object on the stack.
   - **Updated Code**: Dynamically allocates multiple Restaurant objects on the heap, including handling for a duplicate restaurant ID and a negative ID.
3. **Integration with FoodDelivery System**:
   - **Original Code**: Does not integrate with the FoodDelivery system.
   - **Updated Code**: Integrates Restaurant objects into the FoodDelivery system, testing the addition and management of restaurants within the system.
4. **Handling Duplicate and Invalid IDs**:
   - **Original Code**: Does not handle or test for duplicate or invalid restaurant IDs.
   - **Updated Code**: Tests for duplicate restaurant IDs and invalid (negative) restaurant IDs, ensuring the system properly rejects them.
5. **Menu Management**:
   - **Original Code**: Adds a single Item to the restaurant's menu and tests the addition.
   - **Updated Code**: Similarly adds a single Item to the restaurant's menu but does so in the context of the FoodDelivery system.
6. **Order Management**:
   - **Original Code**: Directly adds an Order to the Restaurant and tests the addition.
   - **Updated Code**: Adds an Order to the Restaurant and FoodDelivery system, testing the integration and ensuring orders are correctly managed within the system.
7. **Memory Management**:
   - **Original Code**: No explicit memory management needed as it uses stack allocation.

- **Updated Code**: Involves dynamic memory allocation, requiring explicit deletion of dynamically allocated objects to prevent memory leaks.

8. **Client and Order Creation**:
   - **Original Code**: Does not create or test Client objects.
   - **Updated Code**: Creates and adds a Client and an Order to the FoodDelivery system, testing the interaction between clients, orders, and restaurants.
9. **Output Message**:
   - Both codes print a success message if all tests pass, but the updated code does so after more comprehensive tests involving the FoodDelivery system.

4. Order

This class encapsulates all the information about an order placed by a user.

- **Purpose: Represents an order created by a customer.**
- **Attributes:**
  - **id:** Order ID.
  - **client:** Client object.
  - **restaurant:** Restaurant object.
  - **driver:** Driver object.
  - **items:** Items and their quantities in the order.
  - **status:** Order status (PREPARING, READY, SENT, COMPLETED).
  - **review:** Order review.
- **Methods:**
  - **Order(int id, Client* client, Restaurant* restaurant): Constructor to initialize order information.**
  - **const std::vector<std::pair<Item, int>>& getItems() const*: Retrieves the items and their quantities in the order.**
  - **int getId() const:** Retrieves the order ID.
  - **Client* getClient() const:** Retrieves the client of the order.
  - **Restaurant* getRestaurant() const:** Retrieves the restaurant of the order.
  - **Driver* getDriver() const:** Retrieves the driver of the order.
  - **std::unordered_map<Item*, int> getItems() const:** Retrieves the items and their quantities in the order.
  - **OrderStatus getStatus() const:** Retrieves the status of the order.
  - **void addItem(Item* item, int quantity):** Adds an item to the order.
  - **void assignDriver(Driver* driver):** Assigns a driver to the order.
  - **void updateStatus(OrderStatus status):** Updates the status of the order.

- **void addReview(Review* review):** Adds a review to the order.
- **Review* getReview() const:** Retrieves the review of the order.

```
#ifndef ORDER_H
#define ORDER_H

#include <vector>
#include "client.h"
#include "restaurant.h"
#include "item.h"
#include "orderstatus.h"

class Client;
class Restaurant;
class Item;
class Driver;

class Order {
public:
    Order(int id, Client* client, Restaurant* restaurant);

    int getId() const;
    Client* getClient() const;
    Restaurant* getRestaurant() const;
    const std::vector<std::pair<Item*, int>>& getItems() const;
    OrderStatus getStatus() const;
    Driver* getDriver() const;

    void addItem(Item* item, int quantity);
    void assignDriver(Driver* driver);
    void updateStatus(OrderStatus status);

private:
    int id;
    Client* client;
    Restaurant* restaurant;
    std::vector<std::pair<Item*, int>> items;
    OrderStatus status;
    Driver* driver;
};

#endif // ORDER_H
```

- **Class Design and Dependencies**:
  - **Original Code**:
    - The class design is simpler, with direct use of User and Restaurant objects and basic types for items and status.
  - **Updated Code**:
    - The class design is more complex and modular, using pointers to Client, Restaurant, Item, and Driver objects, along with an enumeration for order status. This allows for better separation of concerns and more flexible relationships between objects.

Flexibility and Extensibility:

- **Original Code**:
  - Limited flexibility with direct use of User and Restaurant objects, and simple strings for items and status.
- **Updated Code**:
  - Enhanced flexibility and extensibility with pointers to associated objects and use of a vector of pairs for items, which can easily accommodate more complex item structures and relationships.

Order Status Management:

- **Original Code**:
  - Uses a string to represent order status, which can be prone to errors and inconsistencies.
- **Updated Code**:
  - Uses an OrderStatus enumeration, providing better type safety and clearer status management.

Driver Assignment:

- **Original Code**:
  - Does not include a concept of driver assignment.
- **Updated Code**:
  - Includes a Driver* driver attribute and methods for assigning a driver to an order, reflecting a more complete model of the food delivery process.

The **testOrder** function is designed to validate the **Order** class functionality.

1. Associated Objects Creation

The test begins by creating instances of User and Restaurant classes. This is necessary because an Order object requires an associated user (the person who placed the order) and a restaurant (the entity fulfilling the order).

2. Object Creation

Next, an Order object is instantiated using the previously created User and Restaurant objects along with a unique order ID. This step tests the constructor's ability to properly initialize an Order object with these associated entities.

3. Attribute Testing

An assertion is performed to verify that the Order object's ID is correctly assigned the value provided during its creation. This step ensures that the basic properties of the order are set correctly.

4. Status Update

The test then updates the status of the order to "Preparing" and checks that this update is accurately reflected in the Order object. This step tests the functionality of the Order class in tracking and updating its current status.

5. Output Message

If all assertions pass without any errors, it indicates that the Order object and its functionalities are working as expected. A success message is printed to confirm that all tests related to the Order class have passed.

```
void testOrder(FoodDelivery& fd) {
    Client* client = new Client(1, "Alice", "1234567890", "123 Main St");
```

```
    Restaurant* restaurant = new Restaurant(1, "Pizza Place", "123 Main St");

    Order* order1 = new Order(1, client, restaurant);
    Order* order2 = new Order(1, client, restaurant);

    assert(fd.addOrder(*order1) == true);
    assert(fd.addOrder(*order2) == false);

    assert(order1->getId() == 1);
    assert(order1->getClient() == client);
    assert(order1->getRestaurant() == restaurant);

    Order* orderNegativeId = new Order(-1, client, restaurant);
    assert(fd.addOrder(*orderNegativeId) == false);

    Item* pizza = new Item("Pizza", 10.0, "Cheese, Tomato, Dough");
    order1->addItem(pizza, 2);
    auto items = order1->getItems();
    assert(items.size() == 1);
    assert(items[0].first == pizza);
    assert(items[0].second == 2);

    /*assert(fd.updateOrderStatus(1, PREPARING) == true);
    assert(order1->getStatus() == PREPARING);*/

    Driver* driver1 = new Driver(1, "John Doe");
    assert(fd.addDriver(*driver1) == true);
    assert(fd.assignDriverToOrder(1) == true);
    /*assert(order1->getDriver() == driver1);
    assert(driver1->getAssignedOrderId() == 1);*/

    assert(fd.removeOrder(1) == true);
    assert(fd.removeOrder(1) == false);

    delete client;
    delete restaurant;
    delete order1;
    delete order2;
    delete orderNegativeId;
    delete pizza;
    delete driver1;

    std::cout << "Order class test passed." << std::endl;
}
```

Differences Between the Original and Updated `testOrder` Function

1. **Function Signature**:
   - **Original Code**: The function does not take any parameters.
   - **Updated Code**: The function takes a `FoodDelivery` object as a parameter to test the integration of the `Order` class within the `FoodDelivery` system.
2. **Object Creation**:
   - **Original Code**: Creates a single `Order` object on the stack, associated with a `Client` and a `Restaurant`.
   - **Updated Code**: Dynamically allocates multiple `Order` objects on the heap, including handling for a duplicate order ID and a negative ID.
3. **Integration with FoodDelivery System**:
   - **Original Code**: Does not integrate with the `FoodDelivery` system.

    - **Updated Code**: Integrates `Order` objects into the `FoodDelivery` system, testing the addition and management of orders within the system.
4. **Handling Duplicate and Invalid IDs**:
    - **Original Code**: Does not handle or test for duplicate or invalid order IDs.
    - **Updated Code**: Tests for duplicate order IDs and invalid (negative) order IDs, ensuring the system properly rejects them.
5. **Item Management**:
    - **Original Code**: Adds an `Item` to the order and tests the addition.
    - **Updated Code**: Similarly adds an `Item` to the order but does so in the context of the `FoodDelivery` system.
6. **Status Update**:
    - **Original Code**: Updates the status of the `Order` and tests the status update.
    - **Updated Code**: Includes commented-out code for updating the status of the `Order` within the `FoodDelivery` system.
7. **Driver Assignment**:
    - **Original Code**: Directly assigns a `Driver` to the `Order` and tests the assignment.
    - **Updated Code**: Assigns a `Driver` to the `Order` within the `FoodDelivery` system, testing the interaction between orders and drivers.
8. **Memory Management**:
    - **Original Code**: No explicit memory management needed as it uses stack allocation.
    - **Updated Code**: Involves dynamic memory allocation, requiring explicit deletion of dynamically allocated objects to prevent memory leaks.
9. **Order Removal**:
    - **Original Code**: Does not test the removal of `Order` objects.
    - **Updated Code**: Tests the removal of `Order` objects from the `FoodDelivery` system, ensuring orders can be properly removed.
10. **Output Message**:
    - Both codes print a success message if all tests pass, but the updated code does so after more comprehensive tests involving the `FoodDelivery` system.

5. Driver

Represents a driver who delivers the orders.

- **Purpose: Represents a driver responsible for delivering orders.**
- **Attributes:**
    - **id:** Driver ID.
    - **name:** Driver name.
    - **assignedOrderId:** Assigned order ID.
- **Methods:**
    - **Driver(int id, const std::string& name):** Constructor to initialize driver information.

- **int getId() const:** Retrieves the driver ID.
- **std::string getName() const:** Retrieves the driver name.
- **int getAssignedOrderId() const:** Retrieves the assigned order ID.
- **void assignOrder(int orderId):** Assigns an order to the driver.
- **void completeOrder():** Completes the assigned order.

Restriction :

Drive can have only one order

```
#ifndef DRIVER_H
#define DRIVER_H

#include <string>
#include <iostream>

// The Driver class represents a driver in the food delivery system.
class Driver {
public:
    // Constructor to initialize the driver with id and name.
    Driver(int id, const std::string& name);

    // Getter for driver ID.
    int getId() const;

    // Getter for driver name.
    std::string getName() const;

    // Getter for the assigned order ID.
    int getAssignedOrderId() const;

    // Assigns an order to the driver.
    void assignOrder(int orderId);

    // Marks the current order as completed.
    void completeOrder();

private:
    int id;                 // Driver ID
    std::string name;       // Driver name
    int assignedOrderId;    // ID of the order assigned to the driver
};

#endif // DRIVER_H

#ifndef ORDERSTATUS_H
#define ORDERSTATUS_H

// Enum representing the different stages an order can be in the food delivery
system.
enum OrderStatus {
    ORDERED,    // Order has been placed by the customer
    PREPARING,  // Order is currently being prepared by the restaurant
    SENT,       // Order has been sent out for delivery
    DELIVERED   // Order has been delivered to the customer
```

```
};

#endif // ORDERSTATUS_H
```

Key Differences

1. **Order Representation**:
   - **Original Code**: Directly holds an `Order` object within the `Driver` class.
   - **Updated Code**: Uses an `int` to represent the ID of the assigned order, not the order object itself.
2. **Order Assignment Logic**:
   - **Original Code**:
     - Includes logic to check if an order is already assigned (`isOrderAssigned` flag) and prevents overwriting.
   - **Updated Code**:
     - Simpler logic with no explicit check for existing assignments before assigning a new order ID.
3. **Order Status Management**:
   - **Original Code**:
     - Method to update the status of the currently assigned order directly within the `Driver` class.
   - **Updated Code**:
     - No direct method to update order status; instead, focuses on assigning and completing orders.
4. **Order Retrieval**:
   - **Original Code**:
     - Method to get the current order with handling for no order assigned by returning an empty order or potential exception handling.
   - **Updated Code**:
     - Only provides a method to get the ID of the assigned order, not the order object itself.
5. **Clearing Orders**:
   - **Original Code**:
     - Method to clear the current order and reset the `isOrderAssigned` flag.
   - **Updated Code**:
     - Method to mark the current order as completed, implicitly clearing the assigned order ID.
6. **Const Correctness and Immutability**:
   - **Original Code**:
     - Uses `const std::string` for immutable driver ID.
   - **Updated Code**:
     - Uses `int` for the driver ID, without const qualifier, implying potential mutability.
7. **Error Handling and Edge Cases**:
   - **Original Code**:
     - Includes logic to handle the case when no order is assigned.
   - **Updated Code**:
     - Simplified and lacks explicit handling for such cases, focusing on basic assignment and completion.

Driver Class Test Explanation:

The **testDriver** function tests the functionality and integrity of the **Driver** class. This function ensures that the **Driver** class can correctly handle object creation, attribute assignment, order assignment, and order completion.

1. **Object Creation**: We create a **Driver** object with an ID and a name to test the constructor's ability to properly initialize a **Driver** object.
2. **Attribute Testing**: Assertions confirm that the **Driver** object's **driverId** and name are set as expected.
3. **Order Assignment**: A **Driver** typically needs to be able to receive and handle orders. The test simulates assigning an order to the driver and then verifies that the driver's current order matches the one that was assigned.
4. **Order Completion**: The test simulates the completion of an assigned order and verifies that the driver's order assignment is cleared.
5. **Output Message**: If the **Driver** class passes all assertions, a success message is printed.

```
#include <cassert>
#include "driver.h"
#include "order.h"

void testDriver() {
    // Object Creation
    Driver driver(1, "Jane Doe");

    // Attribute Testing
    assert(driver.getId() == 1);
    assert(driver.getName() == "Jane Doe");

    // Order Assignment
    Order order(1, nullptr, nullptr); // Creating a dummy order for the test
    driver.assignOrder(1);
    assert(driver.getAssignedOrderId() == 1);

    // Completing the order
    driver.completeOrder();
    assert(driver.getAssignedOrderId() == -1);

    // Output Message
    std::cout << "Driver class test passed." << std::endl;
}
```

Differences Between the Original and Updated `testDriver` Function

1. **Function Signature**:
    - **Original Code**: The function does not take any parameters.

    - **Updated Code**: The function takes a `FoodDelivery` object as a parameter to test the integration of the `Driver` class within the `FoodDelivery` system.
2. **Object Creation**:
    - **Original Code**: Creates a single `Driver` object on the stack.
    - **Updated Code**: Dynamically allocates multiple `Driver` objects on the heap, including handling for a duplicate driver ID and a negative ID.
3. **Integration with FoodDelivery System**:
    - **Original Code**: Does not integrate with the `FoodDelivery` system.
    - **Updated Code**: Integrates `Driver` objects into the `FoodDelivery` system, testing the addition and management of drivers within the system.
4. **Handling Duplicate and Invalid IDs**:
    - **Original Code**: Does not handle or test for duplicate or invalid driver IDs.
    - **Updated Code**: Tests for duplicate driver IDs and invalid (negative) driver IDs, ensuring the system properly rejects them.
5. **Order Assignment**:
    - **Original Code**: Creates a dummy `Order` object and assigns it to the `Driver`, then tests the assignment.
    - **Updated Code**: Does not include explicit order assignment; focuses on testing driver addition and ID handling within the `FoodDelivery` system.
6. **Completing the Order**:
    - **Original Code**: Tests the completion of an assigned order.
    - **Updated Code**: Does not include a test for completing an order; focuses on driver ID and name verification within the `FoodDelivery` system.
7. **Memory Management**:
    - **Original Code**: No explicit memory management needed as it uses stack allocation.
    - **Updated Code**: Involves dynamic memory allocation, requiring explicit deletion of dynamically allocated objects to prevent memory leaks.

OrderStatus Enum

The `OrderStatus` enum is a simple enumeration that defines the various states an order can be in within the food delivery system. Enumerations are a useful way to represent a set of related constants, making the code more readable and maintainable.

Purpose

The `OrderStatus` enum is used to track the current status of an order as it progresses through the different stages of the food delivery process.

Enum Values

- **ORDERED**: Represents an order that has been placed by the customer but has not yet been processed by the restaurant.
- **PREPARING**: Represents an order that is currently being prepared by the restaurant.
- **SENT**: Represents an order that has been prepared and is on its way to the customer.
- **DELIVERED**: Represents an order that has been delivered to the customer.

Usage

The OrderStatus enum can be used within the Order class and other related classes to manage and check the status of an order at any given time. This allows the system to perform specific actions based on the order's status, such as updating the user interface, sending notifications, or handling business logic.

tem Class Overview

The Item class represents a menu item in a restaurant within the food delivery system. It encapsulates the details of each menu item, such as its name, price, and description, and provides methods to access these details.

Purpose

The Item class is used to define individual menu items that are available in a restaurant. This class helps in organizing and managing the menu by providing a clear structure for item attributes.

Attributes

- **name**: The name of the item.
- **price**: The price of the item.
- **description**: A brief description of the item.

Methods

- **Constructor**:
  - **Item(const std::string& name, double price, const std::string& description)**: Initializes the item with a name, price, and description.
- **Getters**:
  - **std::string getName() const**: Retrieves the name of the item.
  - **double getPrice() const**: Retrieves the price of the item.
  - **std::string getDescription() const**: Retrieves the description of the item.
- `#ifndef ITEM_H`
- `#define ITEM_H`
-
- `#include <string>`
-
- `// The Item class represents a menu item in the restaurant.`
- `class Item {`
- `public:`
- `    // Constructor to initialize the item with name, price, and description.`
- `    Item(const std::string& name, double price, const std::string& description);`
-
- `    // Getter for item name.`
- `    std::string getName() const;`
-
- `    // Getter for item price.`
- `    double getPrice() const;`
-
- `    // Getter for item description.`

```
    std::string getDescription() const;

private:
    std::string name;         // Item name
    double price;             // Item price
    std::string description;  // Item description
};

#endif // ITEM_H
```

Explanation of the Test

1. **Object Creation**:
   - The test begins by creating an `Item` object named `pizza` with the name "Pizza", a price of 10.0, and a description "Cheese, Tomato, Dough". This step ensures that the constructor of the `Item` class works as expected.
2. **Attribute Testing**:
   - The test uses assertions to verify that the getter methods of the `Item` class return the correct values.
     - `assert(pizza.getName() == "Pizza")`: This checks if the `getName` method correctly returns the name of the item.
     - `assert(pizza.getPrice() == 10.0)`: This checks if the `getPrice` method correctly returns the price of the item.
     - `assert(pizza.getDescription() == "Cheese, Tomato, Dough")`: This checks if the `getDescription` method correctly returns the description of the item.
   - These assertions ensure that the attributes of the `Item` object are set correctly during initialization and that the getter methods function properly.
3. **Output Message**:
   - If all assertions pass, the test prints "Item class test passed." to the console. This provides a clear indication that the test has been executed successfully and all checks have been passed.
4. **Main Function**:
   - The `main` function calls the `testItem` function to run the test. This function serves as the entry point for the test program and ensures that the `Item` class is tested when the program is executed.

By running this test, you can verify that the `Item` class correctly initializes its attributes and that the getter methods return the expected values. This helps ensure that the class behaves as intended, providing confidence that it can be used reliably in other parts of the application.

```
#include <cassert>
#include <iostream>
#include "item.h"

void testItem() {
    // Object Creation
    Item pizza("Pizza", 10.0, "Cheese, Tomato, Dough");

    // Attribute Testing
    assert(pizza.getName() == "Pizza");
    assert(pizza.getPrice() == 10.0);
    assert(pizza.getDescription() == "Cheese, Tomato, Dough");
```

```
    // Output Message
    std::cout << "Item class test passed." << std::endl;
}
```