To effectively approach any Low-Level Design (LLD) question during interviews, consider using this generic structure:

1. **Problem Understanding**

- **Clarify Requirements:** Ask questions to clarify requirements, constraints, and expected behaviors.
- **Define Scope:** Determine what features are essential and what can be excluded for simplicity.

2. **Identify Key Components**

- **Classes and Interfaces:** Identify main entities (classes) and their relationships. Define interfaces for abstraction.
- **Attributes and Methods:** Specify essential attributes and methods for each class. Use encapsulation principles.

3. **Data Structures**

- **Choose Appropriate Data Structures:** Identify suitable data structures based on expected operations (e.g., maps for quick lookups, queues for processing orders).
- **Complexity Analysis:** Explain time and space complexities for chosen structures.

4. **High-Level Design**

- **System Architecture:** Create a high-level diagram outlining classes and their relationships.
- **Interactions:** Detail how components interact (e.g., sequence diagrams).

5. **Handling Edge Cases**

- **Identify Potential Edge Cases:** Consider scenarios that may cause issues (e.g., null values, data integrity).
- **Propose Solutions:** Suggest how to handle these cases in design.

6. **Scalability and Performance**

- **Scaling Strategy:** Discuss how the system can scale (e.g., horizontal scaling, load balancing).
- **Performance Optimization:** Mention techniques for optimizing performance (e.g., caching, indexing).

7. **Code Implementation**

- **Skeleton Code:** Write basic implementation for key classes and methods. Ensure readability and maintainability.
- **Use of Design Patterns:** Integrate relevant design patterns (e.g., Factory, Singleton) to enhance design.

8. **Testing and Validation**

- **Testing Strategies:** Outline unit testing and integration testing strategies.
- **Validation Techniques:** Discuss how to validate input and output for correctness.

9. **Conclusion**

- **Summarize Design:** Recap the design decisions made, focusing on flexibility, maintainability, and performance.

Generic C++ Code Structure

Here’s a generic C++ skeleton code that you can adapt for various LLD problems:

```cpp
#include <iostream>
#include <string>
#include <map>
#include <queue>
#include <vector>

using namespace std;

// Base class for entities
class EntityBase {
public:
    virtual ~EntityBase() {}
    virtual void performAction() = 0; // Pure virtual function
};

// Concrete class for a specific entity
class ConcreteEntity : public EntityBase {
private:
    string name;
    int id;

public:
    ConcreteEntity(string n, int i) : name(n), id(i) {}

    void performAction() override {
        cout << "Action performed by " << name << endl;
    }

    string getName() { return name; }
};

// Main application logic class
class SystemManager {
private:
    map<int, ConcreteEntity> entities; // Collection of entities
    queue<int> processingQueue; // Queue for processing

public:
    void addEntity(int id, string name) {
        entities[id] = ConcreteEntity(name, id);
    }

    void processEntities() {
        while (!processingQueue.empty()) {
            int id = processingQueue.front();
            processingQueue.pop();
            entities[id].performAction();
        }
    }

    void enqueueEntity(int id) {
        processingQueue.push(id); // Enqueue for processing
    }
};

// Entry point
int main() {
    SystemManager manager;

    // Adding entities
    manager.addEntity(1, "Entity1");
    manager.addEntity(2, "Entity2");

    // Enqueue entities for processing
    manager.enqueueEntity(1);
    manager.enqueueEntity(2);

    // Process the entities
    manager.processEntities();

    return 0;
}
```

Code Explanation:

- `EntityBase`: Abstract class defining a common interface for entities.
- `ConcreteEntity`: Represents a specific entity with its own attributes and behaviors.
- `SystemManager`: Manages the collection of entities and processing logic, including adding and processing entities.
- `main()`: Demonstrates adding entities and processing them.

By using this structure, you can adapt your approach to various LLD scenarios effectively.