Here's your **Package Delivery System** code reformatted in a structured and organized manner, similar to how we did with the Library Management System:

System Requirements

- **Package Management**: The system should allow adding, managing, and tracking packages with different priorities.
- **Delivery Management**: It should process packages based on their priority (Express vs. Standard) and manage their statuses (In Transit, Delivered, Returned, Delayed).
- **Route Management**: It should manage routes between locations and find the shortest path for deliveries using Dijkstra's algorithm.
- **Tracking System**: It should allow real-time tracking of packages and display their current location.

Solution Overview

The Package Delivery System consists of several classes:

1. **Package**: Represents a package with properties such as type, weight, priority, and deadline.
2. **Delivery**: Manages the delivery process, including package status and processing of packages based on priority.
3. **RouteManager**: Manages routes between locations and implements Dijkstra's algorithm for finding the shortest path.
4. **TrackingSystem**: Tracks packages in real-time, managing their locations and any optional regional information.

Classes and Functions

1. **Package Class**:
    - **Properties**: `packageType`, `weight`, `priority`, `deadline`.
    - **Methods**:
        - `generateLabel()`: Generates a label for the package.
        - `getType()`: Returns the type of the package.
        - `getPriority()`: Returns the priority of the package.
        - `getDeadline()`: Returns the deadline of the package.

2. **Delivery Class**:
    - **Properties**: `packageStatus`, `packageQueue`.
    - **Methods**:
        - `addPackage(Package*, PackageStatus)`: Adds a package to the delivery queue.
        - `updateStatus(string, PackageStatus)`: Updates the status of a package.
        - `getStatus(string)`: Retrieves the status of a package.
        - `processNextPackage()`: Processes the next package based on priority.
        - `returnPackage(string)`: Marks a package as returned.

3. **RouteManager Class**:
    - **Properties**: `graph`.

- **Methods**:
  - `addRoute(string, string, int)`: Adds a route between two locations.
  - `findShortestPath(string, string)`: Finds the shortest path using Dijkstra's algorithm.
  - `transportPackage(const vector<string>&, Delivery&, TrackingSystem&, string)`: Simulates transportation of a package along a path.

4. **TrackingSystem Class**:

- **Properties**: `packageData`.
- **Methods**:
  - `trackPackage(string, string, string)`: Tracks a package's location and optional region.
  - `updateTracking(string, string, string)`: Updates the tracking information of a package.
  - `getTrackingInfo(string)`: Retrieves the current tracking information of a package.

Complete Code

```
#include <iostream>
#include <string>
#include <map>
#include <queue>
#include <vector>
#include <algorithm>
#include <thread>
#include <chrono>

using namespace std;

// Enum to represent package priority types
enum PackagePriority { STANDARD, EXPRESS };

// Enum to represent package status
enum PackageStatus { IN_TRANSIT, DELIVERED, RETURNED, DELAYED };

// Base class for packages
class PackageBase {
public:
    virtual ~PackageBase() {}
    virtual string generateLabel() = 0;  // Pure virtual function for generating
labels
};

// Concrete class representing a package
class Package : public PackageBase {
private:
    string packageType;  // Standard or Express
    float weight;
    PackagePriority priority;  // Stores the priority of the package
    int deadline;  // Time left until the deadline (lower value = more urgent)
```

```
public:
    Package(string type, float w, int d) {
        packageType = type;
        weight = w;
        deadline = d;
        priority = (packageType == "Express") ? EXPRESS : STANDARD;  // Set
priority based on package type
    }

    string generateLabel() override {
        return "Label-" + packageType + "-" + to_string(weight);
    }

    string getType() {
        return packageType;
    }

    PackagePriority getPriority() {
        return priority;
    }

    int getDeadline() const {
        return deadline;
    }
};

// Custom comparator for the priority queue
class ComparePriority {
public:
    bool operator()(Package* p1, Package* p2) {
        // First compare by priority, then by deadline
        if (p1->getPriority() == p2->getPriority()) {
            return p1->getDeadline() > p2->getDeadline();  // Shorter deadline
gets higher priority
        }
        return p1->getPriority() > p2->getPriority();  // Higher priority (Express
> Standard)
    }
};

// Class to manage the delivery process with prioritization for express deliveries
class Delivery {
private:
    map<string, string> packageStatus;  // Tracks packageID and its status
    priority_queue<Package*, vector<Package*>, ComparePriority> packageQueue;  //
Priority queue for packages

    // Helper function to convert PackageStatus to string
    string getStatusString(PackageStatus status) {
        switch (status) {
            case IN_TRANSIT: return "In Transit";
            case DELIVERED: return "Delivered";
            case RETURNED: return "Returned";
            case DELAYED: return "Delayed";
```

```
            default: return "Unknown"; // Handle any unforeseen status
        }
    }

public:
    void addPackage(Package* package, PackageStatus status) {
        string packageID = package->generateLabel();
        packageStatus[packageID] = getStatusString(status);  // Use helper
function to set status
        packageQueue.push(package);  // Add package to the priority queue based on
its type (Express > Standard)
    }

    void updateStatus(string packageID, PackageStatus newStatus) {
        if (packageStatus.find(packageID) != packageStatus.end()) {
            packageStatus[packageID] = getStatusString(newStatus);  // Use helper
function to update status
        }
    }

    string getStatus(string packageID) {
        return packageStatus[packageID];  // Get current status of the package
    }

    string processNextPackage() {
        if (!packageQueue.empty()) {
            Package* nextPackage = packageQueue.top();
            packageQueue.pop();  // Process the highest priority package
            return nextPackage->generateLabel();
        }
        return "No Packages in Queue";
    }

    void returnPackage(string packageID) {
        if (packageStatus.find(packageID) != packageStatus.end()) {
            packageStatus[packageID] = getStatusString(RETURNED);  // Mark the
package as returned
        }
    }
};

// Class to manage routes for package delivery
class RouteManager {
private:
    map<string, map<string, int>> graph;  // Graph to represent routes between
locations

public:
    // Add a route with distance between two locations
    void addRoute(string from, string to, int distance) {
        graph[from][to] = distance;
        graph[to][from] = distance;  // Assuming bidirectional routes
    }
```

```
    // Dijkstra's algorithm to find the shortest path
    vector<string> findShortestPath(string start, string end) {
        map<string, string> previous;  // Store previous locations for path
reconstruction
        map<string, int> distances;  // Store distances from start
        priority_queue<pair<int, string>, vector<pair<int, string>>,
greater<pair<int, string>>> queue;  // Min-heap for Dijkstra's

        // Initialize distances to infinity and set the start node's distance to 0
        for (const auto& pair : graph) {
            distances[pair.first] = INT_MAX;
        }
        distances[start] = 0;
        queue.push({0, start});

        while (!queue.empty()) {
            auto [currentDistance, currentLocation] = queue.top();
            queue.pop();

            if (currentLocation == end) break;  // Exit if we've reached the
destination

            for (const auto& neighbor : graph[currentLocation]) {
                int distance = neighbor.second;
                int newDistance = currentDistance + distance;

                // If the new distance is shorter, update the distance and add to
the queue
                if (newDistance < distances[neighbor.first]) {
                    distances[neighbor.first] = newDistance;
                    previous[neighbor.first] = currentLocation;
                    queue.push({newDistance, neighbor.first});
                }
            }
        }

        // Reconstruct the shortest path
        vector<string> path;
        for (string at = end; at != ""; at = previous[at]) {
            path.push_back(at);  // Add node to path
        }
        reverse(path.begin(), path.end());  // Reverse to get the correct order
        return path;
    }

    // Simulates the transportation of a package along the given path
    void transportPackage(const vector<string>& path, Delivery& delivery,
TrackingSystem& trackingSystem, string packageID) {
        for (const string& location : path) {
            // Simulate the transportation delay (e.g., 1 second per location)
            cout << "Transporting package " << packageID << " to " << location <<
endl;
            std::this_thread::sleep_for(std::chrono::seconds(1));  // Simulate
time delay
```

```
            // Update the status as in transit at each location
            delivery.updateStatus(packageID, IN_TRANSIT);

            // Update tracking information
            trackingSystem.updateTracking(packageID, location);
        }

        // Check if the package has reached the

 final destination
        delivery.updateStatus(packageID, DELIVERED);
        cout << "Package " << packageID << " has been delivered!" << endl;
    }
};

// Class to manage tracking information of packages
class TrackingSystem {
private:
    map<string, string> packageData;  // Store package ID and location

public:
    void trackPackage(string packageID, string location, string region) {
        packageData[packageID] = "Package is currently at " + location + " in " +
region;
    }

    void updateTracking(string packageID, string location) {
        packageData[packageID] = "Package is currently at " + location;  // Update
package location
    }

    string getTrackingInfo(string packageID) {
        return packageData[packageID];  // Get tracking information for the
package
    }
};

// Main function to demonstrate the functionality
int main() {
    Delivery delivery;
    RouteManager routeManager;
    TrackingSystem trackingSystem;

    // Adding routes between locations
    routeManager.addRoute("Warehouse", "LocationA", 10);
    routeManager.addRoute("LocationA", "LocationB", 15);
    routeManager.addRoute("LocationB", "LocationC", 20);
    routeManager.addRoute("Warehouse", "LocationC", 25);

    // Creating packages
    Package package1("Express", 5.0, 3);  // Package with Express type
    Package package2("Standard", 10.0, 10);  // Package with Standard type
```

```
    // Adding packages to delivery system
    delivery.addPackage(&package1, IN_TRANSIT);
    delivery.addPackage(&package2, IN_TRANSIT);

    // Finding shortest path and transporting the package
    vector<string> shortestPath = routeManager.findShortestPath("Warehouse",
"LocationB");
    delivery.transportPackage(shortestPath, delivery, trackingSystem,
package1.generateLabel());

    // Track package information
    cout << trackingSystem.getTrackingInfo(package1.generateLabel()) << endl;  //
Display tracking info for package 1

    return 0;
}
```

Code Explanation

- **Package Class**: Represents the package entity with properties like type, weight, priority, and deadline. It generates a unique label for identification.
- **Delivery Class**: Manages the delivery process, including adding packages to a priority queue based on their type, updating their status, and processing the next package for delivery.
- **RouteManager Class**: Manages delivery routes and implements Dijkstra's algorithm to find the shortest path between locations. It also simulates package transportation.
- **TrackingSystem Class**: Tracks package locations and provides updates throughout the delivery process.

Possible Enhancements

1. **User Interface**: Consider adding a user interface for better interaction with the system.
2. **Database Integration**: Implement a database for persistent storage of packages, routes, and tracking information.
3. **Advanced Features**: Add features like notifications for package status changes and delivery estimates.
4. **Error Handling**: Introduce error handling mechanisms for robustness.

This structure can help clarify the design and functionality of your system, making it easier to modify or expand in the future. Let me know if you have any specific changes or enhancements in mind!