Here's the complete solution for the Order Management System, including a simplified constructor for the `Order` class, a destructor, and the structured explanation and code as previously outlined:

System Requirements

- **Order Management**: The system should allow adding, processing, and returning orders with varying priorities.
- **Inventory Management**: It should manage product stock efficiently.
- **Payment Processing**: The system should handle payment for orders and update their status.
- **Order Tracking**: It should provide a way to view the current status of all orders.

Solution Overview

The Order Management System consists of several classes:

1. **Order**: Represents an individual order with properties such as order ID, customer name, status, amount, and priority.
2. **Inventory**: Manages product stock and provides functionalities to add, reduce, and check stock.
3. **Payment**: Handles the payment process and updates the order status.
4. **OrderManagementSystem**: Manages the overall operations, including adding orders, shipping them based on priority, processing returns, and displaying order statuses.
5. **CompareOrder**: A comparator for ordering based on order priority and ID.

Classes and Functions

1. **Order Class**:
    - **Properties**: `orderId`, `customerName`, `status`, `amount`, `priority`.
    - **Methods**:
        - `generateLabel()` - Generates a label for the order.
        - Destructor for cleanup.

2. **Inventory Class**:
    - **Methods**:
        - `addProduct(name, quantity)` - Adds a product to the inventory.
        - `reduceStock(name, quantity)` - Reduces the stock for a product if available.
        - `restock(name, quantity)` - Restocks a product.
        - `getStock(name)` - Retrieves the current stock for a product.

3. **Payment Class**:
    - **Methods**: `processPayment(order)` - Processes the payment for an order.

4. **OrderManagementSystem Class**:
    - **Properties**: `orders`, `inventory`, `orderQueue`.
    - **Methods**:
        - `addOrder(customerName, amount, priority)` - Adds a new order.
        - `shipOrders()` - Ships orders based on priority.

- `returnOrder(orderId)` - Processes returns for specific orders.
- `showOrders()` - Displays the list of orders.

5. **CompareOrder Class**:
    - **Methods**:
        - `operator()` - Compares orders based on priority and order ID.

Complete Code

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <string>
#include <algorithm> // For std::sort
#include <queue>     // For priority queue

using namespace std;

// Enum for Order Status
enum OrderStatus { PENDING, SHIPPED, DELIVERED, RETURNED };

// Enum for Order Priority
enum OrderPriority { LOW, MEDIUM, HIGH };

// Order Class to represent an order
class Order {
public:
    int orderId;
    string customerName;
    OrderStatus status;
    double amount;
    OrderPriority priority;

    // Constructor
    Order(int id, string name, double amt, OrderPriority pri)
        : orderId(id), customerName(name), status(PENDING), amount(amt),
priority(pri) {}

    // Destructor
    ~Order() {
        cout << "Order " << orderId << " destroyed." << endl;
    }

    string generateLabel() const {
        return "Order-" + to_string(orderId) + "-" + customerName;
    }
};

// Comparator for orders based on priority and status
class CompareOrder {
public:
```

```
    bool operator()(const Order* a, const Order* b) {
        if (a->priority == b->priority) {
            return a->orderId > b->orderId; // Lower order ID gets priority if
same priority
        }
        return a->priority < b->priority; // Higher priority first
    }
};

// Inventory Class to manage stock
class Inventory {
    unordered_map<string, int> stock; // Maps product name to stock quantity

public:
    void addProduct(string name, int quantity) {
        stock[name] += quantity;
    }

    bool reduceStock(string name, int quantity) {
        if (stock[name] >= quantity) {
            stock[name] -= quantity;
            return true;
        }
        return false;
    }

    void restock(string name, int quantity) {
        stock[name] += quantity; // Increase stock for returns
    }

    int getStock(string name) const {
        return stock.at(name); // Get available stock
    }
};

// Payment Class to handle payment processing
class Payment {
public:
    void processPayment(Order& order) {
        // Simulate payment processing
        order.status = SHIPPED; // Update order status after payment
        cout << "Payment processed for " << order.generateLabel() << endl;
    }
};

// Order Management System Class
class OrderManagementSystem {
    vector<Order> orders; // Vector to store orders
    Inventory inventory; // Inventory object
    priority_queue<Order*, vector<Order*>, CompareOrder> orderQueue; // Priority
queue for orders

public:
    void addOrder(string customerName, double amount, OrderPriority priority) {
```

```
        int orderId = orders.size() + 1; // Simple order ID assignment
        orders.emplace_back(orderId, customerName, amount, priority);
        orderQueue.push(&orders.back()); // Add the order to the priority queue
    }

    void shipOrders() {
        // Process and ship each order in priority order
        while (!orderQueue.empty()) {
            Order* nextOrder = orderQueue.top();
            orderQueue.pop(); // Remove from queue

            if (nextOrder->status == PENDING) {
                Payment payment;
                payment.processPayment(*nextOrder);
                cout << "Order " << nextOrder->orderId << " shipped." << endl;
            }
        }
    }

    void returnOrder(int orderId) {
        for (auto& order : orders) {
            if (order.orderId == orderId) {
                order.status = RETURNED; // Update order status to returned
                inventory.restock("ProductName", 1); // Simulating restock for
returned item
                cout << "Order " << orderId << " has been returned." << endl;
                return;
            }
        }
        cout << "Order ID not found for return." << endl;
    }

    void showOrders() const {
        for (const auto& order : orders) {
            cout << "Order ID: " << order.orderId
                 << ", Customer: " << order.customerName
                 << ", Status: "
                 << (order.status == PENDING ? "Pending" :
                     order.status == SHIPPED ? "Shipped" :
                     order.status == DELIVERED ? "Delivered" : "Returned")
                 << ", Amount: $" << order.amount
                 << ", Priority: "
                 << (order.priority == HIGH ? "High" :
                     order.priority == MEDIUM ? "Medium" : "Low")
                 << endl;
        }
    }
};

int main() {
    OrderManagementSystem oms;

    // Adding orders with different priorities
    oms.addOrder("Alice", 100.50, HIGH);
```

```
    oms.addOrder("Bob", 200.75, MEDIUM);
    oms.addOrder("Charlie", 150.25, LOW);
    oms.showOrders();

    // Shipping all orders with priority handling
    oms.shipOrders();
    cout << "\nUpdated Order List:\n";
    oms.showOrders(); // Show updated order status

    // Processing return of an order
    oms.returnOrder(1);
    cout << "\nFinal Order List:\n";
    oms.showOrders(); // Show final order status

    return 0;
}
```

Explanation of the Code

- **Order Class**: Represents an order with attributes such as ID, customer name, amount, status, and priority. It has a constructor for easy instantiation and a destructor for cleanup, outputting a message when an order is destroyed. The `generateLabel()` method creates a unique label for each order.
- **CompareOrder Class**: A comparator for orders based on their priority and order ID, ensuring that higher priority orders are processed first.
- **Inventory Class**: Manages the stock of products, allowing for adding products, reducing stock upon order confirmation, and restocking upon returns.
- **Payment Class**: Handles the payment process for orders and updates their status accordingly.
- **OrderManagementSystem Class**: Central class managing orders and inventory. It allows adding new orders, processing shipments based on priority, handling returns, and displaying the current status of all orders.

This solution implements an efficient order management system with prioritization, enabling better handling of customer orders while maintaining a clear overview of inventory and order statuses. The inclusion of a destructor ensures proper resource management, especially if the class were to manage more complex resources in future enhancements.