**Alarm System Project**

**Overwie**

This project provided represents a basic architecture for a control panel system that communicates with diferent sensors and devices using RF(Radio Freuency) communication simulation. The system is designed with Object Oriented Programming (OOP) principles, breaking down the tasks into separate, modular classes for easier maintenance and scalability.

**Key components**:

RFCommunication interface:

This is an abstract class that defines the common interface for communication (sending and receiving data). It allows different communication methods to to be implemented later(e.g., using RF modules like NRF24, ZigBee or LoRa).

SendData() and ReciveData() are static methods that can be called without creating an object of the class and can shared communication functionality across all instances of classes (e.g., sensors, devices) that use RF communication. The RFCommunication class implements these methods in a simplified way for simulation.

Sensor Class:

Device Class

Control Panel Class

Logger class

**Overview**

The **Alarm System** is a sophisticated security solution designed to monitor various sensors (e.g., motion detectors, door sensors) and activate an alarm when any of these sensors are triggered. The system can be armed or disarmed, and it responds to sensor events in real-time. It integrates different components such as sensors, a control panel, and an alarm system, which work together to detect potential security breaches and provide alerts.

The system uses **multithreading** to simulate real-time sensor events and asynchronous alarm activations. It also employs **mutexes** for thread safety to ensure that the system operates correctly even when multiple threads interact with the system simultaneously.

**Components**

1. **Sensors**:
   * **Motion Sensor**: Detects movement within a specified area. If movement is detected, it triggers an alarm.
   * **Door Sensor**: Monitors the opening and closing of doors. If a door is opened unexpectedly, it triggers an alarm.
   * Sensors can be easily extended to include other types (e.g., smoke sensors, window sensors) by inheriting from the base Sensor class.
2. **Control Panel**:
   * The **Control Panel** is the central unit that manages all sensors. It is responsible for:
     + Arming or disarming the system.
     + Checking whether any sensors have been triggered.
     + Activating or deactivating the alarm based on sensor triggers.
     + Resetting sensors after they have been triggered or the system is disarmed.
3. **Alarm**:
   * The **Alarm** is responsible for notifying users of a security breach. It can be activated or deactivated depending on the status of the system and triggered sensors.
   * The alarm system becomes active when one of the sensors is triggered and remains active until the system is reset or disarmed.
4. **Alarm System**:
   * The **Alarm System** is the overall manager, integrating the control panel, sensors, and alarm system.
   * It is responsible for:
     + Initializing the system by adding sensors to the control panel.
     + Arming or disarming the system.
     + Simulating sensor events, where sensors can be triggered asynchronously.
     + Resetting sensors after an alarm is triggered.

**Key Features**

1. **Multithreading**:
   * The system utilizes **multithreading** to simulate asynchronous sensor events. This allows for simultaneous triggering of multiple sensors at different times, just as it might happen in a real-world scenario.
   * The system uses **threads** to simulate time delays between sensor triggers, with the ability to handle sensor events in parallel without blocking other activities in the system.
2. **Thread Safety with Mutexes**:
   * The system uses a **mutex** (std::mutex mtx) to prevent race conditions when multiple threads access shared resources (like the control panel or the alarm). This ensures that only one thread can modify the system state at a time, preventing data inconsistencies or corruption.
3. **Arming/Disarming**:
   * The system can be **armed** or **disarmed** using the control panel. When armed, the system monitors sensors for triggers and activates the alarm if a breach is detected. When disarmed, the system does not monitor sensor states and deactivates any active alarms.
4. **Real-Time Sensor Simulation**:
   * Sensors can be triggered at different times, and the system will check for any triggered sensors in real time. Each sensor is responsible for reporting its state (whether it has been triggered), and the control panel processes these events asynchronously.
5. **Alarm Activation**:
   * Once a sensor is triggered while the system is armed, the alarm is **activated**, notifying the users of a security breach. The alarm remains active until the system is reset or disarmed.
6. **Resetting**:
   * Sensors can be reset after they have been triggered, and the system will deactivate the alarm once the reset is complete. This process allows the system to return to a normal state, ready for new sensor monitoring.

**Class Structure**

1. **Sensor Class**:
   * **Purpose**: The base class for all types of sensors (e.g., motion sensor, door sensor).
   * **Key Methods**:
     + trigger(): Marks the sensor as triggered and outputs a message.
     + reset(): Resets the sensor to a non-triggered state.
     + isSensorTriggered(): Returns whether the sensor is triggered.
2. **MotionSensor and DoorSensor Classes**:
   * **Purpose**: These classes inherit from the Sensor class and implement specific behavior for different types of sensors.
   * **Key Methods**:
     + trigger(): Overrides the base trigger() to provide specific messages for the motion or door sensors.
3. **Alarm Class**:
   * **Purpose**: Represents the alarm system. It is activated when any sensor is triggered.
   * **Key Methods**:
     + activate(): Activates the alarm, indicating a breach in security.
     + deactivate(): Deactivates the alarm, usually when the system is disarmed.
     + isAlarmActive(): Returns whether the alarm is currently active.
4. **ControlPanel Class**:
   * **Purpose**: The control panel manages sensors, arming/disarming the system, and triggering alarms when necessary.
   * **Key Methods**:
     + addSensor(Sensor\* sensor): Adds a sensor to the system.
     + armSystem(): Arms the system, enabling sensor monitoring.
     + disarmSystem(): Disarms the system and deactivates the alarm.
     + checkSensors(): Checks whether any sensor is triggered and activates the alarm if necessary.
     + resetSensors(): Resets all sensors to their non-triggered states.
5. **AlarmSystem Class**:
   * **Purpose**: The overall manager of the alarm system. It integrates the control panel, sensors, and alarm.
   * **Key Methods**:
     + initializeSystem(): Initializes the system by adding sensors to the control panel.
     + armSystem(): Arms the system via the control panel.
     + disarmSystem(): Disarms the system via the control panel.
     + simulateSensorTrigger(): Simulates asynchronous sensor triggers, where different sensors can be triggered at different times.
     + resetSystem(): Resets all sensors and deactivates the alarm.
     + checkSystemStatus(): Outputs the current status of the system (armed or disarmed) and whether the alarm is active.

**Example Workflow**

1. **System Initialization**:
   * When the system starts, sensors (e.g., motion and door sensors) are added to the control panel.
   * The system is initially disarmed, and no alarms are active.
2. **Arming the System**:
   * The system is armed via the control panel, and it starts monitoring the sensors.
3. **Sensor Trigger**:
   * One of the sensors (e.g., motion sensor) is triggered. The control panel detects the trigger and activates the alarm.
4. **Alarm Activation**:
   * The alarm is activated, notifying the user of a potential security breach.
5. **Disarming the System**:
   * The system can be disarmed by the user via the control panel, which deactivates the alarm and resets the sensors.
6. **Resetting**:
   * Once the system is disarmed, all sensors are reset, and the system returns to a non-monitoring state.

**Use Cases**

1. **Home Security**:
   * The system can be used to monitor the perimeter of a house, ensuring doors and windows are secured and motion is detected in case of an intruder.
2. **Industrial Security**:
   * The system can be used in warehouses or industrial spaces, where door sensors and motion detectors monitor for unauthorized access or unusual movements.
3. **Remote Monitoring**:
   * The system could be expanded to include remote access, allowing users to check the status or receive notifications on their mobile devices.

**Conclusion**

This alarm system provides an efficient and scalable solution for real-time security monitoring. With multithreading, the system can simulate multiple sensors working asynchronously, and using mutexes ensures that data integrity is maintained even when multiple threads interact with the system. The modular class design allows for easy expansion, adding new sensor types, alarm features, or integration with external systems.

Thread sinchronization bug

To simulate a **bug** where **two threads** try to create the Logger singleton object at the same time, we'll need to illustrate a scenario where the Logger instance is not properly synchronized. This can happen if we fail to lock the critical section when checking and creating the instance.

### Problem Scenario:

We are going to simulate the scenario where two threads simultaneously attempt to create the Logger singleton object, leading to a **race condition** where both threads might try to initialize the Logger object at the same time.

### Step 1: The Bug: Missing Synchronization in Singleton Creation

In this case, we will **remove synchronization** around the creation of the singleton Logger. Without synchronization, both threads could simultaneously check if the Logger instance is nullptr and attempt to create it. This could lead to **two Logger objects being created**, violating the Singleton pattern.

Here's how the bug could be introduced: