# **Hướng dẫn Triển khai Project "Car Dashboard"**

## 

## **Giới Thiệu Project**

## Project "Car Dashboard" là một hệ thống mô phỏng giao diện điều khiển cho xe ô tô, tập trung vào các chức năng quản lý vận tốc, điều chỉnh chế độ lái, giám sát tiêu hao năng lượng, và đảm bảo an toàn. Hệ thống sử dụng dữ liệu từ file CSV để cập nhật thông tin thời gian thực, giúp người dùng theo dõi tình trạng xe và thực hiện các điều chỉnh cần thiết.

## **Mục tiêu của Project**

## Hiển thị Thông Số Vận Hành: Cung cấp thông tin về tốc độ xe, mức pin, điều hòa và quãng đường còn lại.

## Quản lý Chế Độ Lái: Cho phép lựa chọn chế độ lái SPORT hoặc ECO, mỗi chế độ sẽ có đặc điểm vận hành riêng.

## Tính Toán Tiêu Hao Năng Lượng: Dự đoán mức tiêu hao năng lượng dựa trên vận tốc, điều hòa, và mức gió.

## Đảm Bảo An Toàn: Quản lý các yếu tố an toàn, như ngăn việc đạp ga và phanh cùng lúc.

## **Kỹ Thuật Sử Dụng trong Project**

## Design Patterns: Project sử dụng các mẫu thiết kế như Singleton, Observer, và Command để tổ chức mã nguồn hiệu quả.

## Đa Luồng (Multithreading): Các thread riêng biệt xử lý dữ liệu từ file CSV và đầu vào từ người dùng.

## Bảo vệ Dữ Liệu (Mutex): Sử dụng cơ chế khóa (mutex) để đảm bảo dữ liệu không bị xung đột khi nhiều thread truy cập.

## **Cấu trúc Thư mục**

| Car\_Dashboard\_Project/ ├── App/ │ ├── Inc/ │ │ ├── DashboardController.hpp │ │ ├── DisplayManager.hpp │ │ ├── SpeedCalculator.hpp │ │ ├── BatteryManager.hpp │ │ ├── DriveModeManager.hpp │ │ └── SafetyManager.hpp │ └── Src/ │ ├── Main.cpp │ ├── DashboardController.cpp │ ├── DisplayManager.cpp │ ├── SpeedCalculator.cpp │ ├── BatteryManager.cpp │ ├── DriveModeManager.cpp │ └── SafetyManager.cpp ├── bin/ └── Data/  └── Database.csv |
| --- |

## 

## **Chi Tiết Các Thành Phần Chính**

## 

### **1. Main.cpp**

# Tài liệu Hướng dẫn Triển khai File Main.hpp và Main.cpp trong Car Dashboard Project

## **Giới thiệu**

Trong project "Car Dashboard", Main.hpp và **Main.cpp** đóng vai trò là điểm khởi đầu và trung tâm điều phối cho toàn bộ hệ thống. File main.cpp chứa hàm **main()**, nơi các thành phần chính được khởi tạo và các luồng xử lý dữ liệu (threads) được thiết lập để đảm bảo các tác vụ diễn ra đồng thời mà không ảnh hưởng đến nhau.

Mục tiêu của **Main.cpp**:

1. Khởi tạo các thành phần chính của hệ thống như **DashboardController**, **DisplayManager**, **SpeedCalculator**, **BatteryManager**, **DriveModeManager**, và **SafetyManager**.
2. Thiết lập các threads để đọc dữ liệu từ file Database.csv, xử lý đầu vào từ bàn phím, và cập nhật giao diện.
3. Điều phối vòng lặp chính để đảm bảo hệ thống luôn cập nhật dữ liệu mới và hiển thị thông tin kịp thời.

#### **1. Khởi tạo các Đối tượng Chính**

Trong hàm **main()**, các đối tượng chính của hệ thống được khởi tạo lần lượt. Mỗi đối tượng chịu trách nhiệm cho một chức năng quan trọng trong hệ thống:

* **DashboardController**:
  + Vai trò: Đây là đối tượng trung tâm quản lý dữ liệu của hệ thống. Nó nhận dữ liệu từ file CSV và phân phối dữ liệu cho các thành phần khác thông qua cơ chế Observer.
  + Cách sử dụng: DashboardController cần được khởi tạo đầu tiên và đăng ký DisplayManager làm observer để đảm bảo mọi thay đổi từ DashboardController sẽ tự động cập nhật lên giao diện.
* **DisplayManager**:
  + Vai trò: Chịu trách nhiệm hiển thị thông tin trên giao diện. DisplayManager sẽ nhận thông báo từ DashboardController để cập nhật các thông số như tốc độ, mức pin, và chế độ lái.
  + Cách sử dụng: DisplayManager sẽ lắng nghe thay đổi từ DashboardController và hiển thị dữ liệu mới nhất.
* **SpeedCalculator**:
  + Vai trò: Tính toán vận tốc của xe dựa trên các yếu tố đầu vào như ga, phanh và chế độ lái.
  + Cách sử dụng: SpeedCalculator sẽ được sử dụng trong các thread xử lý đầu vào từ người dùng và điều phối các thay đổi vận tốc khi có thay đổi từ DriveModeManager.
* **BatteryManager**:
  + Vai trò: Tính toán và quản lý mức tiêu hao pin, dựa trên các thông số như vận tốc, điều hòa và mức gió.
  + Cách sử dụng: BatteryManager được gọi trong vòng lặp chính để cập nhật mức pin và ước tính quãng đường còn lại có thể di chuyển.
* **DriveModeManager**:
  + Vai trò: Quản lý các chế độ lái của xe, như SPORT và ECO, và điều chỉnh các tham số vận hành dựa trên chế độ hiện tại.
  + Cách sử dụng: DriveModeManager có thể được kích hoạt khi người dùng thay đổi chế độ lái, và điều chỉnh giới hạn tốc độ, công suất để phù hợp với chế độ.
* **SafetyManager**:
  + Vai trò: Đảm bảo các yếu tố an toàn khi điều khiển xe, ví dụ như ngăn cản việc đạp ga và phanh cùng lúc.
  + Cách sử dụng: SafetyManager sẽ được kích hoạt trong thread xử lý đầu vào từ người dùng để đảm bảo không có hành động gây nguy hiểm.

#### **2. Thiết lập các Thread**

Project sử dụng các thread để xử lý các tác vụ đồng thời. Cụ thể, ba thread chính sẽ được tạo trong hàm main() để đảm bảo các tác vụ độc lập không bị gián đoạn.

* **Thread 1:** Đọc dữ liệu từ Database.csv
  + Vai trò: Đọc dữ liệu từ file Database.csv định kỳ và cập nhật vào DashboardController.
  + Cách thực hiện:
    - Mở file Database.csv để đọc dữ liệu.
    - Lấy các thông số vận hành từ file, như tốc độ, mức pin, trạng thái điều hòa, và cập nhật vào DashboardController.
    - Sau mỗi lần đọc xong, thread sẽ nghỉ một khoảng thời gian (ví dụ: 1 giây) để giảm tải cho CPU và đảm bảo dữ liệu mới nhất được đọc kịp thời.
  + Lưu ý: Sử dụng khóa (mutex) để đảm bảo không có thread nào khác can thiệp vào dữ liệu khi đang đọc.
* **Thread 2**: Xử lý đầu vào từ bàn phím
  + Vai trò: Xử lý các lệnh điều khiển từ người dùng, như thay đổi chế độ lái, bật/tắt điều hòa, nhấn ga/phanh.
  + Cách thực hiện:
    - Lắng nghe các phím điều khiển từ người dùng.
    - Kiểm tra tính hợp lệ của lệnh (ví dụ: nếu người dùng nhấn ga và phanh cùng lúc, thread sẽ kích hoạt SafetyManager để ngăn hành động).
    - Gửi các lệnh đến các thành phần liên quan như SpeedCalculator và DriveModeManager.
  + Lưu ý: Thread này hoạt động liên tục để xử lý các thao tác người dùng trong thời gian thực.
* **Thread chính**: Điều phối vòng lặp chính để cập nhật giao diện
  + Vai trò: Điều phối các hoạt động trong hệ thống, liên tục cập nhật giao diện và điều chỉnh các thành phần liên quan.
  + Cách thực hiện:
    - Lấy thông tin từ DashboardController và gọi phương thức updateDisplay() của DisplayManager để hiển thị các thông số mới nhất lên giao diện.
    - Kiểm tra mức pin và cảnh báo khi pin gần hết bằng cách gọi BatteryManager.
    - Đồng bộ các thành phần với chế độ lái bằng cách gọi DriveModeManager.
  + Lưu ý: Thread chính cần được duy trì để đảm bảo giao diện luôn hiển thị thông tin cập nhật nhất.

#### **3. Vòng lặp chính**

Vòng lặp chính là trung tâm điều phối trong main.cpp, giúp các thành phần hoạt động đồng bộ với nhau. Dưới đây là các bước trong vòng lặp chính:

1. **Cập nhật dữ liệu từ DashboardController**
   * Lấy các thông số từ DashboardController sau khi đã được cập nhật từ file CSV bởi Thread 1.
   * Dữ liệu này bao gồm tốc độ xe, mức pin, chế độ lái, và các thông tin khác.
2. **Gọi updateDisplay() từ DisplayManager**
   * Cập nhật giao diện với các thông số mới nhất từ DashboardController.
   * DisplayManager sẽ hiển thị các giá trị như tốc độ, mức pin, trạng thái điều hòa và chế độ lái.
3. **Kiểm tra và điều chỉnh trạng thái hệ thống**
   * Kiểm tra mức pin: Gọi BatteryManager để cập nhật mức pin còn lại và dự đoán quãng đường có thể đi được. Nếu pin gần hết, hiển thị cảnh báo trên giao diện.
   * Điều chỉnh chế độ lái: Gọi DriveModeManager để giới hạn tốc độ và công suất theo chế độ lái hiện tại (SPORT hoặc ECO).
   * Kiểm tra an toàn: Gọi SafetyManager để kiểm tra các điều kiện an toàn, như ngăn cản đạp ga và phanh cùng lúc.
4. **Đồng bộ hóa các thành phần**
   * Đồng bộ SpeedCalculator, BatteryManager, và DriveModeManager với chế độ lái hiện tại và tốc độ hiện tại. Điều này giúp các thành phần hoạt động nhịp nhàng và thống nhất theo chế độ lái và tốc độ.
5. **Giám sát và bảo trì trạng thái hệ thống**
   * Liên tục kiểm tra và giám sát các trạng thái của hệ thống, đảm bảo không có xung đột hoặc lỗi trong quá trình hoạt động.
   * Nếu phát hiện lỗi (ví dụ, lỗi từ SafetyManager khi phát hiện người dùng đạp ga và phanh cùng lúc), vòng lặp chính sẽ xử lý lỗi và cập nhật lại trạng thái để đảm bảo an toàn.

### **2. DashboardController**

**DashboardController** là thành phần trung tâm trong project "Car Dashboard". Nó chịu trách nhiệm quản lý và điều phối dữ liệu từ các thành phần khác, đồng thời thông báo cho các thành phần liên quan khi có thay đổi dữ liệu. Với việc sử dụng Observer Pattern, DashboardController có thể dễ dàng thông báo cho các thành phần hiển thị hoặc xử lý khác mỗi khi có cập nhật dữ liệu mới từ file CSV hoặc các nguồn khác.

Mục tiêu của DashboardController:

1. Cập nhật và lưu trữ dữ liệu trung tâm như vận tốc, mức pin, chế độ lái, và các thông số quan trọng khác.
2. Thông báo thay đổi tới các thành phần liên quan như DisplayManager để cập nhật giao diện kịp thời.
3. Dễ dàng mở rộng thông qua cơ chế Observer, cho phép thêm các thành phần khác vào hệ thống mà không ảnh hưởng đến mã hiện tại.

#### **1. Vai trò của DashboardController**

* Quản lý dữ liệu trung tâm: DashboardController lưu trữ và quản lý các thông số hệ thống như vận tốc xe, mức pin, chế độ lái, và các thông tin cần thiết khác.
* Điều phối dữ liệu: Đối tượng này không chỉ là nơi lưu trữ dữ liệu mà còn chịu trách nhiệm điều phối và phân phối dữ liệu tới các thành phần khác.
* Observer Pattern: DashboardController đóng vai trò là "subject" trong Observer Pattern, giữ danh sách các "observer" và thông báo cho chúng mỗi khi có thay đổi dữ liệu.

#### **2. Chức năng Chính**

DashboardController cung cấp các chức năng sau:

### **updateData()**

* Mô tả: updateData() là phương thức chính để cập nhật dữ liệu vào DashboardController. Phương thức này sẽ nhận dữ liệu mới từ file CSV hoặc các nguồn dữ liệu khác, sau đó cập nhật vào các thông số của hệ thống như vận tốc, chế độ lái, mức pin, trạng thái điều hòa, v.v.
* Cách hoạt động:
  + Nhận dữ liệu mới từ file CSV: Mở file CSV (ví dụ: Database.csv) để đọc các thông số hiện tại.
  + Cập nhật các biến hệ thống: Các thông số mới sẽ được cập nhật vào các thuộc tính tương ứng trong DashboardController.
    - Vận tốc (speed): Giá trị vận tốc hiện tại của xe.
    - Chế độ lái (driveMode): Chế độ lái của xe, có thể là SPORT hoặc ECO.
    - Mức pin (batteryLevel): Mức năng lượng còn lại của pin.
    - Quãng đường dự đoán (remainingRange): Khoảng cách có thể di chuyển dựa trên mức pin hiện tại.
  + Thông báo cho các observer: Sau khi cập nhật xong dữ liệu, gọi notifyObservers() để các observer cập nhật dữ liệu mới.
* Ví dụ cập nhật dữ liệu:
  + Nếu vận tốc thay đổi từ 60 km/h lên 80 km/h, DashboardController sẽ cập nhật speed lên 80 và thông báo cho các observer như DisplayManager để hiển thị vận tốc mới này trên giao diện.

### **Observer Pattern**

Observer Pattern được sử dụng để giúp DashboardController dễ dàng thông báo cho các thành phần khác mỗi khi có thay đổi dữ liệu. Các thành phần khác (như DisplayManager) sẽ đăng ký làm "observer" của DashboardController và được thông báo khi dữ liệu thay đổi.

* Cấu trúc Observer Pattern trong DashboardController:
  + Subject: DashboardController đóng vai trò là chủ thể (subject), giữ danh sách các observer đã đăng ký.
  + Observer: Các thành phần như DisplayManager là observer của DashboardController. Mỗi khi có thay đổi, các observer sẽ được thông báo để cập nhật dữ liệu.
* Các phương thức chính trong Observer Pattern:
  + registerObserver():
    - Mô tả: Phương thức này cho phép các thành phần khác (như DisplayManager) đăng ký làm observer của DashboardController.
    - Cách sử dụng: Mỗi thành phần cần cập nhật dữ liệu từ DashboardController sẽ gọi registerObserver() để thêm mình vào danh sách observer.
    - Ví dụ: Khi khởi tạo DisplayManager, main.cpp sẽ gọi registerObserver() để đăng ký DisplayManager vào DashboardController.
  + notifyObservers():
    - Mô tả: Mỗi khi có dữ liệu mới, DashboardController sẽ gọi notifyObservers() để thông báo cho tất cả các observer rằng dữ liệu đã thay đổi.
    - Cách sử dụng: notifyObservers() sẽ lần lượt gọi phương thức update() của từng observer để cập nhật dữ liệu mới.
    - Ví dụ: Khi vận tốc thay đổi, notifyObservers() sẽ thông báo cho DisplayManager để cập nhật hiển thị tốc độ mới trên giao diện.

#### **3. Mô tả Các Phương thức Cụ thể**

### **registerObserver(Observer observer)\***

* Mô tả: Phương thức này cho phép một đối tượng đăng ký làm observer của DashboardController.
* Tham số:
  + observer: Con trỏ tới đối tượng cần đăng ký làm observer. Đối tượng này phải triển khai phương thức update() để xử lý thông báo khi có dữ liệu mới.
* Cách hoạt động:
  + registerObserver() sẽ thêm đối tượng observer vào danh sách observer của DashboardController.
  + Nếu đối tượng này đã có trong danh sách, registerObserver() sẽ không thêm lại để tránh dư thừa.
* Ví dụ: Trong quá trình khởi tạo, main.cpp sẽ gọi registerObserver(&displayManager) để đăng ký DisplayManager làm observer của DashboardController.

### **removeObserver(Observer observer)\***

* Mô tả: Phương thức này cho phép một đối tượng hủy đăng ký làm observer của DashboardController.
* Tham số:
  + observer: Con trỏ tới đối tượng cần hủy đăng ký.
* Cách hoạt động:
  + removeObserver() sẽ tìm và xóa observer khỏi danh sách observer.
* Ví dụ: Khi đóng ứng dụng, main.cpp có thể gọi removeObserver(&displayManager) để xóa DisplayManager khỏi danh sách observer của DashboardController.

### **notifyObservers()**

* Mô tả: Phương thức này thông báo cho tất cả các observer rằng dữ liệu trong DashboardController đã thay đổi.
* Cách hoạt động:
  + notifyObservers() sẽ gọi phương thức update() của từng observer trong danh sách.
  + Điều này giúp đảm bảo rằng tất cả các thành phần liên quan đều được cập nhật với dữ liệu mới nhất.
* Ví dụ: Khi dữ liệu vận tốc thay đổi từ file CSV, notifyObservers() sẽ gọi update() của DisplayManager để hiển thị vận tốc mới.

#### **4. Quy trình Cập nhật Dữ liệu Trong Hệ thống**

1. Đọc dữ liệu từ file CSV:
   * Dữ liệu như vận tốc, mức pin, và chế độ lái được đọc từ Database.csv bởi một thread riêng biệt trong main.cpp.
2. Cập nhật vào DashboardController thông qua updateData():
   * Sau khi đọc dữ liệu từ file CSV, thread đọc dữ liệu sẽ gọi updateData() của DashboardController để cập nhật các thông số.
3. Cập nhật các thuộc tính của DashboardController:
   * updateData() sẽ gán các giá trị mới cho các biến như speed, batteryLevel, driveMode, v.v.
4. Gọi notifyObservers() để thông báo cho các observer:
   * Sau khi cập nhật dữ liệu, notifyObservers() được gọi để thông báo cho tất cả các observer về thay đổi.
   * Ví dụ: DisplayManager sẽ cập nhật hiển thị các thông số như tốc độ, pin và chế độ lái.
5. Observer cập nhật giao diện hoặc thực hiện các thao tác cần thiết:
   * Khi notifyObservers() được gọi, mỗi observer sẽ tự động cập nhật thông tin dựa trên dữ liệu mới từ DashboardController.

Ví dụ HPP:

| #ifndef DASHBOARDCONTROLLER\_HPP #define DASHBOARDCONTROLLER\_HPP  #include <vector> #include <string> #include <unordered\_map> #include <memory>  // Forward declarations for the observer and other components class Observer; // Abstract base class for observers  class DashboardController { public:  // Constructor and Destructor  DashboardController();  ~DashboardController();   // Data update method  void updateData(const std::unordered\_map<std::string, std::string>& newData);   // Observer management methods  void registerObserver(Observer\* observer);  void removeObserver(Observer\* observer);  void notifyObservers() const;   // Getters for various system parameters  int getSpeed() const;  std::string getDriveMode() const;  int getBatteryLevel() const;  int getRemainingRange() const;  private:  // System parameters  int speed;  std::string driveMode;  int batteryLevel;  int remainingRange;   // List of observers  std::vector<Observer\*> observers; };  #endif // DASHBOARDCONTROLLER\_HPP |
| --- |

### **3. DisplayManager**

**DisplayManager** là thành phần quản lý việc hiển thị dữ liệu lên giao diện. Nó lắng nghe các cập nhật từ DashboardController và sử dụng các thông số mới nhất (vận tốc, mức pin, nhiệt độ điều hòa, v.v.) để cập nhật giao diện một cách chính xác. DisplayManager đóng vai trò là một observer của DashboardController, tự động cập nhật thông tin mỗi khi có thay đổi từ dữ liệu trung tâm.

## Chức năng của DisplayManager

#### **1. updateDisplay()**

* Mô tả: Phương thức updateDisplay() được gọi mỗi khi có thay đổi dữ liệu trong DashboardController. Phương thức này nhận dữ liệu từ DashboardController và cập nhật toàn bộ giao diện dựa trên thông số mới, bao gồm các thông tin như tốc độ, mức pin, và nhiệt độ điều hòa.
* Cách hoạt động:
  1. Lấy dữ liệu hiện tại từ DashboardController (như vận tốc, mức pin, và chế độ điều hòa).
  2. Cập nhật các phần tử giao diện liên quan bằng cách gọi các phương thức hiển thị cụ thể (như showSpeed(), showBatteryStatus(), và showClimateStatus()).

#### **2. Các phương thức hiển thị cụ thể**

* showSpeed():
  + Mô tả: Hiển thị vận tốc của xe lên giao diện.
  + Cách sử dụng: Được gọi từ updateDisplay() mỗi khi vận tốc thay đổi để đảm bảo giá trị hiển thị luôn là giá trị mới nhất.
* showBatteryStatus():
  + Mô tả: Hiển thị mức pin còn lại của xe.
  + Cách sử dụng: Được gọi từ updateDisplay() khi mức pin thay đổi, giúp người dùng biết được tình trạng năng lượng của xe.
* showClimateStatus():
  + Mô tả: Hiển thị trạng thái điều hòa hoặc nhiệt độ hiện tại của xe.
  + Cách sử dụng: Được gọi từ updateDisplay() để hiển thị nhiệt độ điều hòa, giúp người dùng điều chỉnh nhiệt độ phù hợp.

#### **3. Observer Pattern trong DisplayManager**

* DisplayManager là một observer của DashboardController. Khi DashboardController có thay đổi dữ liệu (qua phương thức notifyObservers()), DisplayManager sẽ được thông báo và tự động gọi updateDisplay() để hiển thị thông tin mới nhất lên giao diện.

## Ví dụ HPP:

| #ifndef DISPLAYMANAGER\_HPP #define DISPLAYMANAGER\_HPP  #include <iostream> // Để sử dụng std::cout cho việc mô phỏng hiển thị #include "Observer.hpp" // Kế thừa từ lớp Observer #include "DashboardController.hpp" // Để lấy dữ liệu từ DashboardController  // Lớp DisplayManager chịu trách nhiệm hiển thị thông tin lên giao diện class DisplayManager : public Observer { public:  // Constructor và Destructor  DisplayManager(DashboardController\* dashboardController);  ~DisplayManager();   // Phương thức cập nhật giao diện tổng quát  void updateDisplay(); // Nhận dữ liệu từ DashboardController và cập nhật giao diện   // Phương thức hiển thị cụ thể cho từng thông số  void showSpeed(int speed); // Hiển thị tốc độ  void showBatteryStatus(int batteryLevel); // Hiển thị mức pin còn lại  void showClimateStatus(int climateTemp); // Hiển thị nhiệt độ điều hòa   // Phương thức từ Observer - được gọi khi có thay đổi dữ liệu  void update() override;  private:  // Con trỏ tới DashboardController để lấy dữ liệu  DashboardController\* dashboardController; };  #endif // DISPLAYMANAGER\_HPP |
| --- |

### **4. SpeedCalculator**

### **SpeedCalculator** là thành phần chịu trách nhiệm tính toán và điều chỉnh vận tốc của xe dựa trên các yếu tố đầu vào như ga, phanh, và chế độ lái. Nó xác định vận tốc tối đa theo chế độ lái hiện tại (SPORT hoặc ECO) và điều chỉnh vận tốc để đảm bảo phù hợp với các điều kiện vận hành của xe.

## Chức năng của SpeedCalculator

#### **1. calculateSpeed()**

* Mô tả: Phương thức calculateSpeed() được gọi để tính toán vận tốc hiện tại của xe dựa trên trạng thái ga và phanh của người lái. Vận tốc sẽ tăng khi nhấn ga và giảm khi phanh, đồng thời phụ thuộc vào các giới hạn tốc độ được đặt bởi DriveModeManager.
* Cách hoạt động:
  1. Nhận đầu vào từ trạng thái ga và phanh.
  2. Tăng vận tốc nếu đang nhấn ga và giảm vận tốc nếu nhấn phanh.
  3. Đảm bảo vận tốc không vượt quá giới hạn tối đa (tùy theo chế độ lái hiện tại).

#### **2. getMaxSpeed()**

* Mô tả: Phương thức getMaxSpeed() xác định giới hạn vận tốc tối đa dựa trên chế độ lái hiện tại. Nếu xe đang ở chế độ SPORT, vận tốc tối đa sẽ cao hơn so với chế độ ECO để tăng khả năng tăng tốc và công suất.
* Cách hoạt động:
  1. Kiểm tra chế độ lái hiện tại (SPORT hoặc ECO).
  2. Trả về giới hạn vận tốc tối đa tương ứng với từng chế độ.

#### **3. adjustSpeedForDriveMode()**

* Mô tả: adjustSpeedForDriveMode() được gọi khi người lái thay đổi chế độ lái, giúp điều chỉnh vận tốc hiện tại để phù hợp với chế độ mới. Ở chế độ ECO, vận tốc sẽ được giới hạn ở mức thấp hơn để tiết kiệm năng lượng, trong khi ở chế độ SPORT, vận tốc sẽ không bị giới hạn chặt chẽ, cho phép tăng tốc nhanh hơn.
* Cách hoạt động:
  1. Kiểm tra chế độ lái hiện tại.
  2. Nếu chuyển sang ECO, giảm vận tốc về dưới mức giới hạn của chế độ này nếu vận tốc hiện tại vượt quá giới hạn.
  3. Nếu chuyển sang SPORT, cho phép vận tốc tăng nhanh hơn.

## Ví dụ HPP:

| #ifndef SPEEDCALCULATOR\_HPP #define SPEEDCALCULATOR\_HPP  #include <string>  // Lớp SpeedCalculator chịu trách nhiệm tính toán và điều chỉnh vận tốc của xe class SpeedCalculator { public:  // Constructor và Destructor  SpeedCalculator();  ~SpeedCalculator();   // Phương thức tính toán vận tốc  int calculateSpeed(bool isAccelerating, bool isBraking); // Tính toán vận tốc dựa trên ga và phanh   // Getter cho vận tốc tối đa tùy theo chế độ lái  int getMaxSpeed(const std::string& driveMode) const; // Lấy giới hạn vận tốc theo chế độ lái   // Phương thức điều chỉnh vận tốc theo chế độ lái hiện tại  void adjustSpeedForDriveMode(const std::string& driveMode); // Điều chỉnh vận tốc khi thay đổi chế độ lái   // Getter cho vận tốc hiện tại  int getCurrentSpeed() const; // Lấy giá trị vận tốc hiện tại  private:  int currentSpeed; // Vận tốc hiện tại của xe  int maxSpeedSport; // Giới hạn vận tốc tối đa cho chế độ SPORT  int maxSpeedEco; // Giới hạn vận tốc tối đa cho chế độ ECO };  #endif // SPEEDCALCULATOR\_HPP |
| --- |

### **5. BatteryManager**

**BatteryManager** chịu trách nhiệm quản lý mức tiêu hao năng lượng của pin trong quá trình vận hành xe. Lớp này tính toán mức tiêu hao pin dựa trên các yếu tố như vận tốc, điều hòa, và mức gió, đồng thời dự đoán quãng đường còn lại có thể di chuyển dựa trên mức pin hiện tại. Nó đảm bảo người lái có thể theo dõi tình trạng năng lượng của xe và có dự báo chính xác về quãng đường còn lại.

## Chức năng của BatteryManager

#### **1. calculateBatteryDrain()**

* Mô tả: Phương thức calculateBatteryDrain() tính toán mức tiêu hao pin dựa trên các yếu tố như vận tốc hiện tại, mức độ hoạt động của điều hòa và tốc độ gió.
* Cách hoạt động:
  1. Nhận đầu vào từ vận tốc xe, trạng thái điều hòa, và mức độ gió.
  2. Tính toán mức tiêu hao pin tương ứng dựa trên các yếu tố này:
     + Vận tốc càng cao thì mức tiêu hao pin càng lớn.
     + Điều hòa bật ở mức cao sẽ tiêu tốn nhiều năng lượng hơn.
     + Mức độ gió cao cũng có thể ảnh hưởng đến mức tiêu hao pin.
  3. Trả về giá trị tiêu hao pin dựa trên các yếu tố đầu vào.

#### **2. calculateRemainingRange()**

* Mô tả: Phương thức calculateRemainingRange() dự đoán quãng đường xe có thể đi được dựa trên mức pin hiện tại và mức tiêu hao pin.
* Cách hoạt động:
  1. Lấy mức pin hiện tại và mức tiêu hao pin trên mỗi km (từ calculateBatteryDrain()).
  2. Tính toán quãng đường còn lại bằng cách chia mức pin hiện tại cho mức tiêu hao pin trên mỗi km.
  3. Trả về quãng đường dự đoán xe có thể di chuyển với mức pin hiện tại.

#### **3. updateBatteryLevel()**

* Mô tả: updateBatteryLevel() cập nhật mức pin của xe dựa trên các yếu tố vận hành và giảm dần mức pin theo thời gian. Phương thức này được gọi định kỳ để duy trì tình trạng năng lượng chính xác.
* Cách hoạt động:
  1. Lấy dữ liệu về vận tốc, điều hòa, và gió từ các cảm biến hoặc từ DashboardController.
  2. Gọi calculateBatteryDrain() để xác định mức tiêu hao năng lượng hiện tại.
  3. Trừ mức tiêu hao từ pin hiện tại và cập nhật lại giá trị pin.
  4. Nếu mức pin dưới một ngưỡng nhất định, hiển thị cảnh báo để người lái biết pin sắp hết.

## Ví dụ HPP:

| #ifndef BATTERYMANAGER\_HPP #define BATTERYMANAGER\_HPP  #include <iostream>  // Lớp BatteryManager quản lý mức pin và tính toán quãng đường di chuyển dựa trên các yếu tố vận hành class BatteryManager { public:  // Constructor và Destructor  BatteryManager();  ~BatteryManager();   // Phương thức tính toán mức tiêu hao pin  double calculateBatteryDrain(int speed, int acLevel, int windLevel) const; // Tính toán tiêu hao pin   // Phương thức dự đoán quãng đường còn lại  double calculateRemainingRange() const; // Tính toán quãng đường dự đoán dựa trên mức pin hiện tại   // Phương thức cập nhật mức pin  void updateBatteryLevel(int speed, int acLevel, int windLevel); // Cập nhật mức pin dựa trên các yếu tố vận hành   // Getter cho mức pin hiện tại  double getBatteryLevel() const; // Lấy mức pin hiện tại  private:  double batteryLevel; // Mức pin hiện tại của xe  double batteryCapacity; // Dung lượng pin tối đa  double drainPerKm; // Mức tiêu hao pin trên mỗi km (cơ bản) };  #endif // BATTERYMANAGER\_HPP |
| --- |

### **6. DriveModeManager**

**DriveModeManager** chịu trách nhiệm quản lý các chế độ lái của xe, như SPORT và ECO. Mỗi chế độ lái được thiết kế để đáp ứng nhu cầu vận hành khác nhau: SPORT ưu tiên hiệu suất với công suất cao và khả năng tăng tốc mạnh, trong khi ECO tập trung vào tiết kiệm năng lượng với giới hạn tốc độ và công suất thấp hơn. DriveModeManager điều chỉnh các tham số vận hành để tối ưu hóa hiệu suất hoặc tiết kiệm năng lượng tùy thuộc vào chế độ lái hiện tại.

## Chức năng của DriveModeManager

#### **1. setDriveMode()**

* Mô tả: Phương thức setDriveMode() cho phép thiết lập chế độ lái hiện tại của xe. Người dùng có thể chọn giữa SPORT và ECO dựa trên nhu cầu vận hành.
* Cách hoạt động:
  1. Nhận tham số đầu vào là chuỗi driveMode, có thể là "SPORT" hoặc "ECO".
  2. Lưu chế độ lái được thiết lập vào biến thành viên currentDriveMode.
  3. Tự động điều chỉnh các giới hạn công suất và tốc độ dựa trên chế độ lái đã thiết lập.

#### **2. getPowerOutput()**

* Mô tả: Phương thức getPowerOutput() trả về công suất đầu ra của xe dựa trên chế độ lái hiện tại. Ở chế độ SPORT, công suất cao hơn để tối ưu hiệu suất và khả năng tăng tốc. Ở chế độ ECO, công suất thấp hơn để tiết kiệm năng lượng.
* Cách hoạt động:
  1. Kiểm tra currentDriveMode để xác định chế độ lái hiện tại.
  2. Trả về giá trị công suất tương ứng:
     + SPORT: Công suất cao hơn.
     + ECO: Công suất thấp hơn.

#### **3. limitSpeedForEcoMode()**

* Mô tả: Phương thức limitSpeedForEcoMode() giới hạn vận tốc của xe khi chế độ ECO được kích hoạt, giúp tiết kiệm năng lượng và giảm tiêu hao pin. Phương thức này sẽ đặt một giới hạn tốc độ tối đa khi xe đang chạy ở chế độ ECO.
* Cách hoạt động:
  1. Kiểm tra currentDriveMode. Nếu chế độ lái là ECO, giới hạn tốc độ sẽ được đặt dựa trên maxEcoSpeed.
  2. Nếu vận tốc hiện tại vượt quá maxEcoSpeed, điều chỉnh vận tốc xuống để phù hợp với giới hạn ECO.

## Ví dụ HPP:

| #ifndef DRIVEMODEMANAGER\_HPP #define DRIVEMODEMANAGER\_HPP  #include <string>  // Lớp DriveModeManager quản lý chế độ lái của xe, gồm các chế độ như SPORT và ECO class DriveModeManager { public:  // Enum để xác định các chế độ lái  enum class Mode { SPORT, ECO };   // Constructor và Destructor  DriveModeManager();  ~DriveModeManager();   // Phương thức thiết lập chế độ lái  void setDriveMode(Mode driveMode); // Đặt chế độ lái hiện tại (SPORT hoặc ECO)   // Phương thức lấy công suất đầu ra dựa trên chế độ lái  int getPowerOutput() const; // Trả về công suất phù hợp với chế độ lái hiện tại   // Phương thức giới hạn tốc độ khi ở chế độ ECO  int limitSpeedForEcoMode(int currentSpeed) const; // Giới hạn tốc độ nếu chế độ ECO đang kích hoạt   // Getter cho chế độ lái hiện tại  Mode getCurrentDriveMode() const; // Trả về chế độ lái hiện tại  private:  Mode currentDriveMode; // Chế độ lái hiện tại của xe  int powerOutputSport; // Công suất đầu ra cho chế độ SPORT  int powerOutputEco; // Công suất đầu ra cho chế độ ECO  int maxEcoSpeed; // Giới hạn tốc độ tối đa khi ở chế độ ECO };  #endif // DRIVEMODEMANAGER\_HPP |
| --- |

### **7. SafetyManager**

**SafetyManager** là thành phần chịu trách nhiệm đảm bảo an toàn khi điều khiển xe, đặc biệt trong các trường hợp người lái có thể thực hiện các thao tác nguy hiểm. Lớp này kiểm soát các yếu tố an toàn như ngăn việc đạp ga và phanh cùng lúc, giảm tốc độ khi phanh, và cho phép xe di chuyển tiếp khi phanh được giải phóng.

## Chức năng của SafetyManager

#### **1. checkBrakeAndAccelerator()**

* Mô tả: Phương thức checkBrakeAndAccelerator() kiểm tra xem người lái có đang nhấn cả ga và phanh cùng lúc không, vì đây là hành động nguy hiểm. Nếu phát hiện đồng thời nhấn ga và phanh, phương thức này sẽ kích hoạt cơ chế ngăn cản, ưu tiên giảm tốc độ.
* Cách hoạt động:
  1. Nhận trạng thái đầu vào từ các cảm biến hoặc các biến trạng thái cho biết người dùng có đang nhấn ga và phanh cùng lúc không.
  2. Nếu cả hai đều được kích hoạt, ưu tiên ngừng tăng tốc và chỉ cho phép giảm tốc độ để đảm bảo an toàn.

#### **2. applyBrake()**

* Mô tả: Phương thức applyBrake() giảm tốc độ của xe dần dần khi người lái nhấn phanh. Tốc độ sẽ giảm từ từ thay vì giảm đột ngột để tránh gây nguy hiểm.
* Cách hoạt động:
  1. Kiểm tra xem phanh có đang được nhấn không.
  2. Nếu phanh đang được nhấn, giảm tốc độ theo một mức nhất định cho đến khi xe dừng lại hoặc phanh được nhả.

#### **3. releaseBrake()**

* Mô tả: releaseBrake() được gọi khi người lái nhả phanh, cho phép xe tiếp tục di chuyển và tăng tốc nếu đang nhấn ga.
* Cách hoạt động:
  1. Kiểm tra trạng thái của phanh. Nếu phanh được nhả, cho phép xe trở lại trạng thái bình thường.
  2. Cập nhật trạng thái phanh để hệ thống biết phanh không còn được kích hoạt.

#### **4. isBrakeApplied()**

* Mô tả: Phương thức isBrakeApplied() trả về trạng thái của phanh, cho biết phanh có đang được nhấn hay không. Phương thức này hữu ích khi các thành phần khác cần biết trạng thái phanh để điều chỉnh các thông số vận hành.
* Cách hoạt động:
  1. Trả về giá trị của biến brakeApplied, cho biết phanh có đang được kích hoạt hay không.

## Ví dụ HPP:

| #ifndef SAFETYMANAGER\_HPP #define SAFETYMANAGER\_HPP  // Lớp SafetyManager quản lý các yếu tố an toàn khi điều khiển xe class SafetyManager { public:  // Constructor và Destructor  SafetyManager();  ~SafetyManager();   // Phương thức kiểm tra xem có đang đạp ga và phanh cùng lúc không  bool checkBrakeAndAccelerator(bool isAccelerating, bool isBraking); // Kiểm tra hành vi nguy hiểm   // Phương thức kích hoạt phanh  void applyBrake(); // Giảm tốc độ dần dần khi phanh được nhấn   // Phương thức giải phóng phanh  void releaseBrake(); // Giải phóng phanh khi nhả ra   // Kiểm tra trạng thái của phanh  bool isBrakeApplied() const; // Trả về trạng thái phanh có đang được nhấn hay không  private:  bool brakeApplied; // Biến lưu trạng thái phanh (đang được nhấn hay không)  int brakeIntensity; // Cường độ giảm tốc khi phanh (đơn vị giảm tốc mỗi lần)  int currentSpeed; // Tốc độ hiện tại của xe (được cập nhật khi phanh) };  #endif // SAFETYMANAGER\_HPP |
| --- |

# **Các Chức Năng Chính**

## **1. Quản lý Vận Tốc và Chế Độ Lái**

### **Chức năng**

Chức năng này cho phép hệ thống quản lý vận tốc của xe dựa trên chế độ lái hiện tại (SPORT hoặc ECO), đồng thời điều chỉnh vận tốc và công suất đầu ra để phù hợp với từng chế độ lái, tối ưu hiệu suất hoặc tiết kiệm năng lượng khi cần thiết.

### **Chi tiết**

**SpeedCalculator** chịu trách nhiệm tính toán vận tốc của xe dựa trên các yếu tố đầu vào như ga, phanh, và chế độ lái.

* Tính toán vận tốc:
  + Khi người dùng nhấn ga, SpeedCalculator sẽ tăng vận tốc dựa trên cường độ ga.
  + Khi phanh được nhấn, SpeedCalculator sẽ giảm vận tốc dần dần.
  + Vận tốc cuối cùng sẽ phụ thuộc vào cả trạng thái ga và phanh.
* Tương tác với SafetyManager:
  + SpeedCalculator cũng sẽ phối hợp với SafetyManager để đảm bảo rằng vận tốc không tăng khi phanh đang được nhấn.

**DriveModeManager** quản lý chế độ lái của xe, bao gồm SPORT và ECO, và điều chỉnh các tham số vận hành dựa trên chế độ hiện tại.

* Chế độ SPORT:
  + Khi ở chế độ SPORT, DriveModeManager cho phép xe hoạt động với công suất cao hơn và tốc độ tối đa lớn hơn, tạo ra khả năng tăng tốc mạnh mẽ hơn.
  + DriveModeManager sẽ nâng giới hạn công suất đầu ra và tốc độ tối đa, giúp xe đạt hiệu suất cao nhất khi cần.
* Chế độ ECO:
  + Khi ở chế độ ECO, DriveModeManager giới hạn công suất đầu ra và tốc độ tối đa để tiết kiệm năng lượng, giúp giảm thiểu tiêu hao pin.
  + DriveModeManager sẽ đặt giới hạn tốc độ thấp hơn và giảm công suất động cơ để tối ưu hóa năng lượng, phù hợp cho các hành trình dài với tiêu hao năng lượng thấp.

### Lợi ích

* Chế độ SPORT giúp xe đạt hiệu suất và tốc độ tối đa khi cần thiết, phù hợp với các điều kiện đường xá yêu cầu công suất cao.
* Chế độ ECO tối ưu hóa tiêu hao năng lượng, giúp người lái tiết kiệm pin và đi xa hơn khi cần.

## **2. Tính Toán Mức Tiêu Hao Pin và Quãng Đường Còn Lại**

### **Chức năng**

Hệ thống này giúp theo dõi và quản lý mức pin còn lại của xe, tính toán mức tiêu hao pin dựa trên các yếu tố như vận tốc, điều hòa và gió. Từ đó, dự đoán được quãng đường còn lại có thể di chuyển.

### **Chi tiết**

**BatteryManager** chịu trách nhiệm tính toán mức tiêu hao pin và dự đoán quãng đường xe có thể đi được với mức pin hiện tại.

* Tính toán mức tiêu hao pin (calculateBatteryDrain):
  + Dựa trên vận tốc: Vận tốc càng cao, mức tiêu hao pin càng lớn.
  + Điều hòa (AC): Khi điều hòa hoạt động ở mức cao, tiêu hao năng lượng càng lớn.
  + Mức gió: Mức gió cao cũng có thể tăng mức tiêu hao năng lượng.
  + BatteryManager sẽ tính toán mức tiêu hao năng lượng tương ứng và cập nhật lại mức pin của xe.
* Dự đoán quãng đường còn lại (calculateRemainingRange):
  + Sử dụng mức pin còn lại và mức tiêu hao năng lượng đã tính toán để ước lượng quãng đường còn lại.
  + Phương thức này trả về một giá trị ước tính về số km xe có thể di chuyển trước khi cần sạc pin.
* Cập nhật mức pin (updateBatteryLevel):
  + Cập nhật mức pin dựa trên các yếu tố hiện tại như vận tốc, điều hòa, và mức gió.
  + Mức pin sẽ giảm dần khi xe di chuyển và điều hòa hoạt động, giúp người dùng theo dõi tình trạng năng lượng của xe.

### **Lợi ích**

* Dự đoán quãng đường còn lại giúp người lái biết được khoảng cách tối đa mà xe có thể di chuyển với mức pin hiện tại, giúp lên kế hoạch sạc pin kịp thời và tránh tình trạng hết pin giữa chừng.
* Tối ưu hóa mức tiêu hao năng lượng cho các điều kiện lái khác nhau giúp kéo dài tuổi thọ pin và hiệu quả sử dụng năng lượng.

## **3. Đảm Bảo An Toàn Khi Điều Khiển Xe**

### **Chức năng**

Hệ thống này giúp ngăn chặn các thao tác nguy hiểm của người lái, ví dụ như đạp ga và phanh cùng lúc. Khi phanh được nhấn, hệ thống sẽ giảm tốc độ từ từ để đảm bảo an toàn.

### **Chi tiết**

**SafetyManager** quản lý các yếu tố an toàn khi điều khiển xe, đảm bảo rằng người lái không thực hiện các hành động có thể gây nguy hiểm.

* Kiểm tra đạp ga và phanh cùng lúc (checkBrakeAndAccelerator):
  + SafetyManager sẽ kiểm tra xem người dùng có đang đạp ga và phanh đồng thời không.
  + Nếu phát hiện hành vi này, hệ thống sẽ ngừng tăng tốc và chỉ cho phép xe giảm tốc độ để tránh nguy hiểm.
* Giảm tốc độ khi phanh (applyBrake):
  + Khi phanh được nhấn, SafetyManager sẽ kích hoạt applyBrake() để giảm tốc độ dần dần, tránh việc giảm tốc đột ngột gây nguy hiểm cho người lái.
  + Tốc độ sẽ giảm từ từ đến khi xe dừng hẳn hoặc khi người lái nhả phanh.
* Giải phóng phanh (releaseBrake):
  + Khi người lái nhả phanh, releaseBrake() sẽ được gọi để giải phóng phanh và cho phép xe tiếp tục di chuyển.
  + Hệ thống sẽ cho phép xe tăng tốc trở lại nếu người lái nhấn ga.
* Kiểm tra trạng thái phanh (isBrakeApplied):
  + Phương thức này trả về trạng thái hiện tại của phanh, giúp các thành phần khác trong hệ thống biết được liệu phanh có đang hoạt động hay không.

### **Lợi ích**

* SafetyManager đảm bảo rằng xe không bị tăng tốc khi đang phanh, giảm thiểu rủi ro tai nạn và đảm bảo an toàn cho người lái.
* Việc giảm tốc từ từ khi phanh giúp người lái kiểm soát xe tốt hơn, đặc biệt khi cần phanh gấp trong các tình huống khẩn cấp.