Question 1: Smart Diagnostics – Fault Simulation Engine

Introduction

In modern automotive systems, fault simulation is critical for validating embedded diagnostics. This challenge tasks you with building a modular fault simulation engine using C++ that models sensors, actuators, and fault injection logic—all managed via pointers and references.

Scenario

You're part of a diagnostics team designing a simulation engine for an STM32-based automotive ECU. The system must simulate sensor faults, actuator delays, and log anomalies. Faults are injected dynamically, thresholds are updated via reference, and all memory is managed manually.

Class Specifications

Sensor (Abstract Base Class)

- Attributes:
  int id, float value, float threshold, std::string status
- Methods:
  virtual void simulateFault() = 0
  bool isFaulty() → returns true if value > threshold

TemperatureSensor, PressureSensor (Derived)

- Implement simulateFault() to inject realistic fault values

Actuator

- Attributes:
  int id, std::string state, int responseTime
- Method:
  void triggerAction()

FaultInjector

- Attributes:
  std::vector<Sensor*> sensors, std::vector<Actuator*> actuators

- Methods:
  void injectSensorFault(Sensor* s)
  void injectActuatorDelay(Actuator& a)
  void injectAllFaults()
  void reportStatus()

SimulationEngine

- Accepts ConfigManager& and FaultLogger*
- Dynamically allocates sensors and actuators
- Runs fault cycles and logs results

ConfigManager

- Method:
  void updateThreshold(Sensor& s, float newThreshold)

FaultLogger

- Method:
  void log(Sensor& s) → logs fault to file

Global Functions

```
void logFault(Sensor* s);                    // Logs fault via pointer
void analyzeSystem(FaultInjector* fi);       // System-wide fault analysis
```

Sample Data

```
TemperatureSensor* t1 = new TemperatureSensor(101, 95.0, 90.0);
PressureSensor* p1 = new PressureSensor(102, 45.0, 50.0);
Actuator* a1 = new Actuator(201, "Idle", 120);
```

Tasks

- Implement all classes with proper memory management
- Use pointers for dynamic allocation and traversal
- Use references for configuration and logging

- Simulate a test run with 3 sensors and 2 actuators
- Log faults and generate a system report

Concepts Reinforced

- Polymorphism, inheritance
- Pointer-based object management
- Reference-based updates and logging
- Global function integration
- Manual memory cleanup

Question 2: Sensor Grid – Dynamic Reference Binding

Introduction

In embedded systems, dynamic binding to critical components is essential. This challenge simulates a sensor grid where a controller binds to the highest-priority sensor using references, while managing all sensors via pointers.

Scenario

You're developing a sensor grid for an autonomous vehicle. The controller must always reference the most critical sensor (highest priority). Sensor health is monitored via reference, and global functions help rebind and visualize the grid.

Class Specifications

Sensor

- Attributes:
  int id, float value, int priority, std::string status
- Method:
  void updateValue(float v)

Controller

- Attributes:
  Sensor* sensorList[10]
  Sensor& criticalSensor
- Methods:
  void bindCriticalSensor()
  void updateCriticalSensor() → uses global function
  void printStatus()

SensorDiagnostics

- Method:
  std::string checkHealth(Sensor& s)

Global Functions

```
void rebindCriticalSensor(Sensor*& ref, Sensor* list[], int size);
void printSensorMap(Sensor* list[], int size);
```

Sample Data

```
Sensor* s1 = new Sensor(1, 75.0, 3);
Sensor* s2 = new Sensor(2, 60.0, 5);
Sensor* s3 = new Sensor(3, 90.0, 2);
Sensor* sensorList[3] = {s1, s2, s3};
```

Tasks

- Dynamically allocate 10 sensors with varying priorities
- Use reference to bind to highest priority sensor
- Modify sensor via reference and observe changes
- Use global functions to rebind and print sensor map
- Clean up memory safely

Concepts Reinforced

- Reference-to-pointer rebinding
- Pointer arrays and dynamic access
- Global function manipulation of references
- Reference-based health checks
- Manual memory management

Question 3: Command Queue – Linked Execution Engine

Introduction

Command queues are vital in embedded actuator systems. This challenge involves building a linked list of commands using pointers and executing them via reference-to-pointer traversal.

Scenario

You're designing a command queue for a robotic actuator. Each command is stored in a node and executed sequentially. The queue is traversed using reference-to-pointer logic, and global functions help manage execution and display.

Class Specifications

Command

- Attributes:
  std::string name, int duration

CommandNode

- Attributes:
  Command* cmd, CommandNode* next

CommandQueue

- Attributes:
  CommandNode* head
- Methods:
  void enqueue(Command* c)
  void executeNext(CommandNode*& current)
  void executeAll() → uses global function
  void clear()

CommandExecutor

- Method:
  void run(Command& c)

Global Functions

void advanceQueue(CommandNode*& current);

void printQueue(CommandNode* head);

Sample Data

Command* c1 = new Command("Start Motor", 100);

Command* c2 = new Command("Open Valve", 50);

Command* c3 = new Command("Stop Motor", 80);

Tasks

- Dynamically create and enqueue 5 commands
- Traverse queue using reference to pointer
- Execute commands via reference
- Use global functions to advance and print queue
- Implement destructor logic to avoid leaks

Concepts Reinforced

- Reference-to-pointer traversal
- Linked list manipulation via pointers
- Global function-based queue control
- Reference-based command execution
- Manual memory cleanup
