C++ Basics Practice Questions
September 17, 2025

General Guidelines:

1. Read the question thoroughly. Understand the scenario, requirements, and expected functionality before you begin coding.
2. Use the exact variable names, function names, and class names provided in the question.
   This ensures consistency and helps in automated or manual evaluation.
3. Follow all technical requirements as specified.
4. Follow MISRA C++ naming conventions.
   - Use descriptive and consistent variable names (e.g., componentIdentifier, not id)
   - Avoid abbreviations and reserved keywords
   - Maintain proper formatting and indentation
5. Ensure your code is modular, readable, and well-commented.
   - Break down logic into functions
   - Add comments to explain key parts of your code
6. Validate all user inputs.
   - Check for valid ranges, non-empty strings, and correct enum selections
7. Test your program with the sample data provided.
   - Ensure all functionalities work as expected
8. Submit only the required .cpp and .h files.
   - Make sure it compiles and runs without errors
9. Do not modify the structure of the question.
   - Stick to the design and flow as described
10. Assume suitable data whenever required.

Question 1: ADAS Pedestrian Detection

Scenario:

An autonomous vehicle uses an ADAS (Advanced Driver Assistance System) module to detect pedestrians around it. The system identifies multiple pedestrians and determines which one is closest to the vehicle to prioritize safety measures such as emergency braking or alerting the driver.

You are tasked with implementing a simplified version of this detection logic.'

Requirements:

1. Define a class Pedestrian with the following members:
    - uint32 pedestrian_id — a unique identifier for each pedestrian.
    - float distance_from_vehicle — the current distance of the pedestrian from the vehicle.
2. Implement the following in the class:
    - A constructor to initialize both members.
    - void updateDistance(float newDistance) — updates the pedestrian's distance.
    - float getDistance() const — returns the current distance.
3. Memory Allocation:
    - Dynamically allocate an array of 3 Pedestrian objects on the heap using the following sample data:
        - Pedestrian 1: ID = 101, Distance = 12.4 meters
        - Pedestrian 2: ID = 102, Distance = 8.7 meters
        - Pedestrian 3: ID = 103, Distance = 15.1 meters
4. Closest Pedestrian Detection:
    - Implement a function void findClosestPedestrian(Pedestrian* array, uint32 size, const Pedestrian*& closest) that assigns the reference pointer closest to the pedestrian with the smallest distance_from_vehicle. Assume no ties.

5. Global Functions:
    - void printPedestrianInfo(const Pedestrian& ped) — prints pedestrian ID and distance.
    - void printClosestPedestrian(const Pedestrian* closest) — prints the closest pedestrian's info if the pointer is not null.
6. Main Task:
    - Create the pedestrian objects dynamically.
    - Use the detection function to find the closest pedestrian.
    - Print all pedestrian details using printPedestrianInfo.
    - Highlight the closest pedestrian using printClosestPedestrian.
    - Properly deallocate the array to avoid memory leaks.

Question 2: ADAS Adaptive Cruise Control

Scenario:
An autonomous vehicle equipped with Adaptive Cruise Control (ACC) uses radar and vision sensors to track surrounding vehicles. The system continuously monitors multiple vehicles ahead and identifies the **lead vehicle** — the one closest to the ego vehicle — to adjust speed and maintain a safe following distance.

To improve diagnostics and system awareness, the ACC module also tracks:

- The **minimum distance ever recorded** among all tracked vehicles.
- The **total number of tracked vehicles created** during runtime.

These shared values are maintained using static members.

Requirements:

1. **Define a class TrackedVehicle** with the following members:
    - uint32 vehicle_id — a unique identifier for each tracked vehicle.
    - float speed — the current speed of the vehicle.
    - float distance — the distance from the ego vehicle.
    - static float min_recorded_distance — tracks the smallest distance ever recorded across all vehicles.
    - static int vehicle_count — tracks the total number of TrackedVehicle objects created.
2. **Implement the following in the class:**
    - A constructor to initialize all members and Update min_recorded_distance if the current vehicle's distance is smaller.
    - Increment vehicle_count each time a new object is created.
    - void display() const — prints the vehicle's ID, speed, and distance.
    - const TrackedVehicle* compareDistance(const TrackedVehicle& other) const — returns a pointer to the vehicle with the smaller distance. If distances are equal, return this explicitly.
    - static float getMinRecordedDistance() — returns the smallest distance recorded across all vehicles.

- static int getVehicleCount() — returns the total number of tracked vehicles created.

3. **Memory Allocation:**
   - Dynamically allocate an array of 4 TrackedVehicle objects on the heap using the following sample data:
     - Vehicle 1: ID = 501, Speed = 80 km/h, Distance = 60 m
     - Vehicle 2: ID = 502, Speed = 78 km/h, Distance = 45 m
     - Vehicle 3: ID = 503, Speed = 85 km/h, Distance = 100 m
     - Vehicle 4: ID = 504, Speed = 76 km/h, Distance = 40 m
4. **Lead Vehicle Detection:**
   - Implement a function void findLeadVehicle(TrackedVehicle* arr, uint32 size, const TrackedVehicle*& lead) that assigns the reference pointer lead to the vehicle with the smallest distance.
5. **Global Functions:**
   - void printTrackedVehicle(const TrackedVehicle& vehicle) — prints the vehicle's details.
   - void printLeadVehicle(const TrackedVehicle* lead) — prints the lead vehicle's details or "None found" if the pointer is null.
6. **Main Task:**
   - Dynamically allocate an array of 4 TrackedVehicle objects on the heap using the sample data.
   - Use the detection function to find the lead vehicle.
   - Print all vehicle details using printTrackedVehicle.
   - Highlight the lead vehicle using printLeadVehicle.
   - Print the minimum recorded distance using getMinRecordedDistance().
   - Print the total number of tracked vehicles created using getVehicleCount().
   - Properly deallocate the heap memory to avoid leaks.

Question 3: ADAS Forward Collision Warning

Scenario:

An autonomous vehicle uses Forward Collision Warning (FCW) as part of its ADAS (Advanced Driver Assistance System) suite. This system continuously monitors objects ahead — such as vehicles, barriers, or pedestrians — and evaluates the risk of collision based on their relative speed and distance from the ego vehicle.

Your task is to simulate the risk assessment logic for multiple detected objects.

Requirements:

1. **Define a class** DetectedObject with the following members:
    - uint32 object_id — a unique identifier for each object.
    - float rel_speed — relative speed with respect to the ego vehicle.
    - float distance — distance from the ego vehicle.
2. **Implement the following in the class:**
    - A constructor to initialize all members.
    - void display() const — prints the object's ID, relative speed, and distance.
    - bool isHigherRisk(const DetectedObject& other) const — returns true only if the calling object has both:
        - smaller distance **and**
        - higher rel_speed than other. Use the this pointer explicitly.
    - void updateValuesByValue(DetectedObject obj) — modifies the **copy** of the object by increasing rel_speed by 2 and decreasing distance by 5.
    - void updateValuesByReference(DetectedObject& obj) — modifies the **original** object by increasing rel_speed by 2 and decreasing distance by 5.
3. **Memory Allocation:**
    - Dynamically allocate an array of 3–4 DetectedObject instances on the heap using the following sample data:
        - Object 1: ID = 701, RelSpeed = 15, Distance = 55

    - Object 2: ID = 702, RelSpeed = 20, Distance = 35
    - Object 3: ID = 703, RelSpeed = 10, Distance = 30
4. **Risk Assessment:**
    - Implement a function void findHighestRiskObject(DetectedObject* arr, uint32 size, const DetectedObject*& riskObj) that assigns the reference pointer riskObj to the object with the highest risk (based on isHigherRisk). Return the first match in case of a tie.
5. **Global Functions:**
    - void printDetectedObject(const DetectedObject& obj) — prints object details.
    - void printHighestRiskObject(const DetectedObject* obj) — prints the highest risk object or a warning if nullptr.
    - void updateObjectValuesByValueGlobal(DetectedObject obj) — calls updateValuesByValue inside.
    - void updateObjectValuesByReferenceGlobal(DetectedObject& obj) — calls updateValuesByReference inside.
6. **Main Task:**
    - Create the objects dynamically.
    - Call both update functions and print before/after states using global print functions.
    - Find and print the highest risk object.
    - Properly deallocate heap memory.

Question 4: ADAS Sensor Temperature Monitoring

Scenario

An autonomous vehicle's ADAS (Advanced Driver Assistance System) continuously monitors the health of its onboard sensors — including **LIDAR**, **RADAR**, and **CAMERA** — to ensure optimal performance. Each sensor reports temperature readings at regular intervals, and the system must track these readings to detect overheating or anomalies.

To enhance safety, the system also tracks the **highest temperature ever recorded across all sensors**, regardless of type. This shared value is maintained using a static member.

Requirements

1. **Define an enumeration SensorType with the following values:**
   - LIDAR
   - RADAR
   - CAMERA
2. **Define a class SensorArray with the following members:**
   - int sensor_id — unique identifier for the sensor.
   - SensorType type — type of the sensor.
   - double* temperature_readings — dynamically allocated array of temperature readings.
   - int num_readings — number of readings stored.
   - static double global_max_temperature — tracks the highest temperature across all sensors.
3. **Implement the following in the class:**
   - A **constructor** that:
     - Allocates memory for the temperature array.
     - Initializes it with provided data.
     - Updates global_max_temperature if any reading exceeds the current value.
   - A **destructor** that properly frees the allocated memory.

- double getMaxTemperature() const — returns the highest temperature reading for that sensor.
- void printSensorInfo() const — prints sensor ID, sensor type (as string), and maximum temperature.
- static double getGlobalMaxTemperature() — returns the highest temperature recorded across all sensors.

4. **Global Functions**
   - const char* sensorTypeToString(SensorType t) — returns a string representation of the sensor type.
   - void printSensor(const SensorArray& s) — prints sensor information using the class method.
   - void printAllSensors(const SensorArray* arr, int size) — prints information for all sensors in the array.
5. **Sample Data**

Create 3 sensors with the following data:

Sensor ID	Type	Temperature Readings	Num Readings
801	LIDAR	[35.5, 36.1, 34.9]	3
802	RADAR	[39.0, 38.7, 39.3]	3
803	CAMERA	[30.2, 31.0, 30.5]	3

6. **Main Task**
   - Dynamically allocate an array of SensorArray objects on the heap using the sample data.
   - Print each sensor's info using printSensor.
   - Print the **global maximum temperature** using the static function getGlobalMaxTemperature().
   - Free all allocated memory properly to avoid leaks.

Question 5: Smart Parcel Weight Analyzer

Scenario:
A logistics company is developing a smart parcel analyzer to automate weight classification of packages. The system must support different types of input formats — grams, kilograms, and pounds — and classify parcels as Light, Medium, or Heavy based on their weight.

You are tasked with implementing the core logic using function overloading.

Requirements

1. **Define a class ParcelAnalyzer with:**
    - No data members required.
2. **Implement three overloaded functions named classifyWeight:**
    - string classifyWeight(int grams); // Accepts weight in grams
    - string classifyWeight(float kilograms); // Accepts weight in kilograms
    - string classifyWeight(double pounds); // Accepts weight in pounds

3. **Classification Rules:**

Unit	Light	Medium	Heavy
Grams	< 500 g	500 g to 2000 g	> 2000 g
Kilograms	< 0.5 kg	0.5 kg to 2.0 kg	> 2.0 kg
Pounds	< 1.1 lb	1.1 lb to 4.4 lb	> 4.4 lb

4. **Global Function**:
    - void printClassification(const string& label); // Prints the classification result
5. **Main Task**
    - Create an object of ParcelAnalyzer.
    - Call each overloaded function with sample inputs.
    - Print the classification results using printClassification

6. Sample Inputs and Expected Outputs

Function Call	Input Value	Expected Output
classifyWeight(450)	450 grams	Light
classifyWeight(1500)	1500 grams	Medium
classifyWeight(2500)	2500 grams	Heavy
classifyWeight(0.3f)	0.3 kg	Light
classifyWeight(1.5f)	1.5 kg	Medium
classifyWeight(3.0f)	3.0 kg	Heavy
classifyWeight(0.9)	0.9 lb	Light
classifyWeight(2.5)	2.5 lb	Medium
classifyWeight(5.0)	5.0 lb	Heavy