C++ Basics Practice Questions
October 8, 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.
11. If you are not aware of few functionalities then those implementations can be ignored.
12. Provide Appropriate File Name:
    YourName_MonthDate_Task1_main.cpp
    YourName_MonthDate_Task1_className.cpp
    YourName_MonthDate_Task1_className.h

    Ex
    John_Oct1_Task2_main.cpp
    John_Oct1_Task2_myClass.cpp
    John_Oct1_Task2_myClass.h

Task 1: Vehicle Diagnostic Logger

You are tasked with building a **Vehicle Diagnostic Logger** for an automotive service center. Each vehicle undergoes a diagnostic scan that produces a **report** containing multiple **sensor readings**. Each reading includes the sensor name, its measured value, and a severity level.
The severity level must be represented using an **enumeration**, and the report must be modeled using **composition**, where the report object contains multiple sensor reading objects.

Requirements
Design and implement the following in C++:

1. **Enum Class**:
   - Define an enum class named SeverityLevel with the following values:
     - Low, Medium, High, Critical
2. **Class: SensorReading**
   - Attributes:
     - sensorName (string)
     - value (double)
     - severity (SeverityLevel)
   - Behaviors:
     - Constructor to initialize all attributes
     - A method to display the reading
     - A method to retrieve the severity level
3. **Class: DiagnosticReport**
   - Composition:
     - Contains a collection (e.g., array) of SensorReading objects
   - Behaviors:
     - Method to add a new sensor reading
     - Method to display all readings
     - Method to count how many readings are of a given severity or higher

---

Sample Data

Use the following sensor readings to populate your report:

Sensor Name	**Value**	**Severity**
EngineTemp	105	High
OilPressure	20	Medium
BrakeFluidLevel	5	Critical
BatteryVoltage	12.5	Low

Tasks to Perform in main()

Write a main() function that performs the following:

1. Create a DiagnosticReport object.
2. Add the above sample readings to the report.
3. Display all sensor readings.
4. Count and display how many readings have severity High or Critical.

Task 2: Extension: Advanced Diagnostic Report Features

New Functionalities to Implement

1. **Enum-to-String Conversion**
    - Add a utility function to convert SeverityLevel enum values to readable strings (e.g., "High").
    - This will improve display clarity and support sorting/filtering.
2. **Severity-Based Filtering**
    - Add a method in DiagnosticReport:
    - SensorReading* filterBySeverity(SeverityLevel level, int& count) const;
    - Returns all readings that match the given severity exactly.
3. **Sorting by Severity**
    - Add a method to sort readings in descending order of severity.
    - void sortBySeverityDescending();
4. **Severity Distribution Map**
    - Add a method to compute how many readings fall under each severity level.
    - Display a map like {Low: 1, Medium: 2, High: 3, Critical: 1}.
5. **Sensor Lookup**
    - Add a method to retrieve a reading by sensor name:
    - SensorReading* findReadingByName(const std::string& name);
    - Return pointer to the reading if found, else nullptr.

Additional Tasks for main()

- Display the severity distribution map.
- Sort and display readings by severity.
- Filter and display only Critical readings.
- Search for a specific sensor (e.g., "BrakeFluidLevel") and display its details.

Task 3: Student Performance Analyzer

You are building a **Student Performance Analyzer** for a university's internal grading system. Each student has a set of **SubjectScores**, and each score has a **GradeLevel** represented using an enum. The system must analyze performance trends and support exporting filtered data.

Requirements

Design and implement the following in C++:

1. **Enum Class: GradeLevel**
   - Define an enum class with values: Fail, Pass, Merit, Distinction
2. **Class: SubjectScore**
   - Attributes:
     - subjectName (string)
     - score (int)
     - grade (GradeLevel)
   - Behaviors:
     - Constructor to initialize all attributes
     - Method to display the score
     - Method to retrieve the grade
3. **Class: Student**
   - Composition:
     - Contains a raw array of SubjectScore objects (use dynamic allocation)
   - Behaviors:
     - Method to add scores (assume fixed max size, e.g., 10)
     - Method to compute average score
     - Method to count how many subjects are Distinction
     - Method to print all Merit and Distinction scores

Sample Data

Subject	Score	Grade
Math	92	Distinction
Physics	78	Merit
Chemistry	65	Pass
History	45	Fail
English	88	Distinction

Tasks to Perform in main()

1. Create a Student object.
2. Add the above sample scores.
3. Display all scores.
4. Show average score.
5. Count and display how many subjects are Distinction.
6. Print top scores (Merit and Distinction)

Task 4: Extension: Global Function Enhancements

Global Function 1: Grade Summary Across Students

void summarizeGrades(Student* students[], int studentCount);

- **Purpose**: Aggregates grade distribution across multiple students.
- **Behavior**:
    - Iterates through all students and their scores.
    - Computes total counts of each GradeLevel across all subjects.
    - Displays a summary like:
    - Fail: 3
    - Pass: 5
    - Merit: 4
    - Distinction: 6
- **Edge Case**: Handles nullptr entries in the student array defensively.

Global Function 2: Top Scorer by Subject

const SubjectScore* findTopScorer(Student* students[], int studentCount, const std::string& subject);

- **Purpose**: Finds the highest score for a given subject across all students.
- **Behavior**:
    - Searches for the subject in each student's scores.
    - Returns pointer to the SubjectScore with the highest score.
    - If subject not found, returns nullptr.
- **Edge Case**: Handles ties, missing subjects, and empty student arrays.

Global Function 3: Export All Distinctions

void exportAllDistinctions(Student* students[], int studentCount, const std::string& filename);

- **Purpose**: Print all Distinction scores across students
- **Behavior**:
    - Iterates through all students and scores.

    - Display subject name, score, and student index or name (if available).

Suggested main() Tasks

- Create an array of Student* with 2–3 students.
- Populate each with sample scores.
- Call summarizeGrades() to display grade distribution.
- Call findTopScorer() for "Math" and display result.
- Call exportAllDistinctions()