C++ Basics Practice Questions
September 19, 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.

Question 1: Smart Garage

A **Smart Garage** manages a fleet of **Car** objects.

Each car is identified by a unique **VIN**, has a *make*, *model*, *price* and a **service-history** consisting of several **ServiceRecord** objects.

In addition, a car may have a **dynamic array of “damage-codes”** (integers that represent parts that were repaired or replaced – e.g., **101 = front-bumper**, **205 = brake-pad**).

The garage software must:

- Create cars in several ways (default, parameterised, copy).
- Keep all internal data **encapsulated** – no direct external access.
- Provide **read-only** (const) access to information that must not be modified.
- Support **shallow copy** of the service-history (the vector of records) while performing a **deep copy** of the damage-code array.
- Maintain a **static counter** of how many **Car** objects exist at any moment.
- Allow **heap allocation** of an array of **Car** objects.
- Offer a **global utility** that can compute the average price of a collection of cars.

Requirements:

1. Class ServiceRecord

Data members (private)	Type	Purpose
date_	std::string	Service date – format YYYY-MM-DD.
mileage_	int	Vehicle mileage at the time of service (must be ≥ 0).
description_	std::string	Short textual description of the service performed.

Member functions (public)
Constructors
ServiceRecord(); – default constructor (creates an “empty” record).
ServiceRecord(const std::string& date, int mileage, const std::string& description); – full initialiser (validates mileage).
Copy-control
ServiceRecord(const ServiceRecord&);
ServiceRecord& operator=(const ServiceRecord&);
Getters – all const
const std::string& getDate() const;
int getMileage() const;
const std::string& getDescription() const;
Utility
void print() const; – prints the record in a human-readable format.

2. Class Car

Data members (private)	**Type**	**Purpose**
vin_	std::string	Vehicle Identification Number – unique key.
make_	std::string	Manufacturer name.
model_	std::string	Model name.
price_	double	Current market price (non-negative).
serviceHistory_	ServiceRecord*	Dynamic array that stores the service records.
serviceCount_	size_t	Number of valid service records stored.
serviceCap_	size_t	Allocated capacity of the serviceHistory_ array.
damageCodes_	int*	Dynamic array that stores “damage codes”.
damageCount_	size_t	Number of valid damage-code entries.
damageCap_	size_t	Allocated capacity of damageCodes_.
static size_t totalCars_;	static	Counter of live Car objects.

Member functions (public)
Constructors
Car(); – default constructor (empty strings, price = 0, no dynamic arrays). Increments **totalCars_**.
Car(const std::string& vin, const std::string& make, const std::string& model, double price); – parameterised constructor (validates **price**). Increments **totalCars_**.
Car(const Car& other); – **copy constructor**. Performs a **deep copy** of both the **serviceHistory_** and **damageCodes_** arrays, copying the stored elements. Increments **totalCars_**.
Destructor
~Car(); – releases both dynamic arrays (**serviceHistory_** and **damageCodes_**) and decrements **totalCars_**.
Assignment operator
Car& operator=(Car other); – copy-and-swap implementation (strong exception safety).
Friend swap

friend void swap(Car& lhs, Car& rhs) noexcept; – swaps all non-static members (including the raw pointers and their size/capacity fields).
Static accessor
static size_t getTotalCars(); – returns the current value of **totalCars_**.
Setters (validation where required)
void setVIN(const std::string& vin);
void setMake(const std::string& make);
void setModel(const std::string& model);
void setPrice(double price); – throws **std::invalid_argument** if **price < 0**.
Getters – all **const**
const std::string& getVIN() const;
const std::string& getMake() const;
const std::string& getModel() const;
double getPrice() const;
size_t getServiceCount() const;
size_t getDamageCount() const;
const ServiceRecord* getServiceHistory() const; – returns a pointer to the internal service-record array (read-only).
const int* getDamageCodes() const; – returns a pointer to the internal damage-code array (read-only).
Business-logic functions
void addService(const ServiceRecord& rec); – appends a record to **serviceHistory_**; automatically expands the array when capacity is exhausted.
void addDamageCode(int code); – appends a damage code to **damageCodes_**; automatically expands the array when needed.
void printInfo() const; – prints VIN, make, model, price, number of service records, and number of damage codes in a readable format.
(private) Helper functions
void reserveService(size_t newCap); – grows **serviceHistory_** while preserving existing records.
void reserveDamage(size_t newCap); – grows **damageCodes_** while preserving existing codes.

3. Global functions

Function	Signature	Description
Average price	double averagePrice(const Car* arr, size_t n);	Returns the arithmetic mean of the price of all Car objects in the array arr (size n). Uses only the public getPrice() accessor.
Compare by price (ascending)	bool compareByPriceAsc(const Car& a, const Car& b);	Returns true if a.getPrice() < b.getPrice(). Useful with std::sort.
Compare by price (descending)	bool compareByPriceDesc(const Car& a, const Car& b);	Returns true if a.getPrice() > b.getPrice().
Equality based on VIN	bool areCarsEqual(const Car& a, const Car& b);	Returns true when the VINs are identical (a.getVIN() == b.getVIN()).
Maximum-price car in an array	const Car* maxPriceCar(const Car* arr, size_t n);	Returns a pointer to the Car with the highest price; nullptr if n == 0.
Find car by VIN	const Car* findCarByVIN(const Car* arr, size_t n, const std::string& vin);	Linear search; returns a pointer to the first car whose VIN matches vin, or nullptr if not found.
Count cars containing a specific damage code	size_t countCarsWithDamage(const Car* arr, size_t n, int code);	Returns how many cars in the array have code present in their damage-code array.
Swap two Car objects (non-member helper)	void swapCars(Car& a, Car& b);	Calls the class-friend swap(a, b). Demonstrates swapping without making swap a member.

4. Main-function tasks

#	Action	Expected Result
1	Print the initial static-counter value (**Car::getTotalCars()**).	Counter = **0** (no **Car** objects exist yet).
2	Default-construct a **Car** object (**Car cDefault;**).	**totalCars** increments to **1**; all members are empty/zero; **cDefault.printInfo()** s hows empty data.
3	Parameter-construct a **Car** with VIN, make, model and price.	**totalCars** increments to **2**; object holds the supplied values; **printInfo()** displays them.
4	Add three damage codes to the car from step 3 using **addDamageCode**.	**damageCount** becomes **3** and the internal array holds the three codes; **printInfo()** shows them.
5	Create two **ServiceRecord** objects and add them to the car from step 3 via **addService**.	**serviceCount** becomes **2**; the service-history array stores both records; **printInfo()** lists the records (date, mileage, description).
6	**Copy-construct** a new **Car** from the object created in step 3.	**totalCars** increments to **3**; the new object has identical data.
6-a	Display the damage-code arrays of the original and the copy, then modify a code in the copy.	The original array remains unchanged → confirms **deep copy** of damage codes.
6-b	Query **getServiceCount()** on both original and copy.	Both report **2** services → copy has the same service history (shallow copy of the simple records).
7	Default-construct another **Car** and assign the object from step 3 to it (**cAssign = cParam**).	**totalCars** increments to **4**; **cAssign** now mirrors **cParam**.
7-a	Change a damage code in **cAssign** and verify the original's code is unchanged.	Confirms deep copy of damage-code array on assignment.
8	Allocate a dynamic array of **N = 3 Car** objects (**new Car[N]**).	**totalCars** increments by **3** (now should be previous total + 3).
8-a	Initialise each element of the heap array via assignment with distinct VIN, make, model and price.	Each element holds its own data; no overlap between objects.

8-b	Use a helper lambda (or function) that receives a **Car&**, an **int*** of damage codes, and a **ServiceRecord*** array, then calls **addDamageCode** and **addService** for each element.	Each heap object receives its own set of damage codes and service records; the service records make use of the **ServiceRecord** class.
8-c	Loop over the heap array and call **printInfo()** for each car.	Output shows the three cars with their respective VINs, prices, damage codes and service histories (proving independent storage).
9	Call the global function **averagePrice** with the heap array and **N**.	Returns the arithmetic mean of the three car prices (≈ 34 332.83 for the sample data) and prints it.
10	Call the additional global utilities: **maxPriceCar,** **findCarByVIN** (search for a known VIN), **countCarsWithDamage** (for a specific code), **areCarsEqual** (compare two heap cars)	VIN of the most expensive car is printed. The searched car is found and its price displayed. Correct count of cars containing the given damage code is shown. Equality check returns **false** because VINs differ.
11	Delete the heap array (**delete[] garage**).	**totalCars** decreases by **3** (returns to the count after step 7).
12	Attempt to modify a **const Car&** by calling a non-const mutator.	Compilation error – demonstrates const-correctness.
13	Print the final static-counter value before exiting **main**.	Value reflects the number of still-alive **Car** objects (should be **0** after **main** ends; you'll see the count before leaving the function).

5. **Service-Record sample Data**

Variable	**Date (YYYY-MM-DD)**	**Mileage**	**Description**
srOilChange	2022-03-15	15 000	Oil change
srBrakeReplace	2023-01-10	25 000	Brake pads replacement
srTireRotate	2023-06-20	30 000	Tire rotation
srInspection	2024-02-05	35 000	Annual safety inspection

6. Car-Object samples

Variable	VIN	Make	Model	Price (USD)
cHonda	1HGCM	Honda	Accord	19 999.99
cAcura	JH4KA	Acura	TLX	27 999.49
cTesla	5YJ3E	Tesla	Model 3	39 999.00
cBMW	WBA4	BMW	3 Series	34 999.99
cFord	1FA6P	Ford	Mustang	31 200.00

7. Damage-code arrays (int values)

Array name	Contents (damage-code IDs)	Intended car
damageHonda	{101, 205, 307}	cHonda
damageAcura	{102, 208}	cAcura
damageTesla	{110, 220, 330, 440}	cTesla
damageBMW	{150, 250}	cBMW
damageFord	{175, 285, 395, 505, 615}	cFord