**Practice Q1: BankAccount (encapsulation + static aggregates)**

You are to implement a small banking model.

(a) Define a class BankAccount with private data members: string owner, double balance. Provide a parameterized constructor BankAccount(string owner, double openingBalance) that sets the owner and initializes balance (must be ≥ 0; if negative, set to 0).

(b) Implement member functions:  
• void deposit(double amount) (ignore non-positive; otherwise increase balance)  
• bool withdraw(double amount) (succeeds only if 0 < amount ≤ balance; return true/false)  
• double getBalance() const (accessor)  
• void print() const → Owner: <owner>, Balance: $<balance with 2 dp>

(c) Add class-level bookkeeping: static members  
• static int s\_count = number of active accounts  
• static double s\_totalFunds = sum of balances across all active accounts  
Update these in the constructor and a destructor so they remain consistent. Also update totals inside deposit/withdraw.

(d) In main(), create three accounts with different opening balances. Perform at least five operations across the accounts (a mix of deposits/withdrawals, including one failed withdrawal). After each operation print the affected account’s status. At the end, print:  
Accounts: <s\_count>, Total funds: $<s\_totalFunds>

(e) Briefly explain in comments how encapsulation is enforced in your class (1–2 lines).

Constraints: use <string>, <iostream>, <iomanip> only; no global variables other than the required statics inside the class.

**Practice Q2: Thermostat (state update + hysteresis)**

Model a simple thermostat controller with hysteresis (no physics).

(a) Define class Thermostat with private members:  
double setpoint, double deadband, bool heaterOn. Provide a constructor Thermostat(double sp, double db) that sets setpoint and deadband (require db > 0), and initializes heaterOn to false.

(b) Implement void update(double currentTemp) that toggles heaterOn using hysteresis:  
• Turn on if currentTemp < setpoint - deadband/2  
• Turn off if currentTemp > setpoint + deadband/2  
Otherwise, leave state unchanged.

(c) Provide:  
• void setSetpoint(double sp)  
• void setDeadband(double db) (ignore/guard invalid values ≤ 0)  
• bool isHeating() const  
• void print() const → SP=<setpoint>, DB=<deadband>, Heating=<ON|OFF>

(d) In main(), construct one thermostat (setpoint=22.0, deadband=1.0). Simulate a sequence of at least 10 temperatures (e.g., from a vector<double> with values above/below the thresholds). For each reading: call update(temp), then print(). Ensure both transitions (OFF→ON and ON→OFF) occur at least once.

(e) Add a static member int s\_updates counting how many times update(...) was called. Print its value at the end of main().

Constraints: stick to console I/O; no random generation required. Use clear comments to justify the hysteresis boundaries you chose.

**Practice Q3: Inventory/SKU (collections + static totals)**

Build a tiny inventory system.

(a) Define class SKU with private members:  
string name; double price; int qty. Provide a constructor SKU(string name, double price, int qty) that clamps invalid inputs (if price < 0 → 0, if qty < 0 → 0).

(b) Implement:  
• void restock(int n) (n>0; increases qty)  
• bool sell(int n) (0<n≤qty; decrement and return true; else false)  
• void discount(double pct) where pct is in [0,90] meaning reduce price by pct% (e.g., 20 → price\*=0.8). Ignore invalid pct.  
• double value() const → returns price \* qty  
• void print() const → SKU <name>: price=$<price 2dp>, qty=<qty>, value=$<value 2dp>

(c) Add class-level aggregates:  
static int s\_totalSKUs (number of live SKU objects) and static double s\_totalInventoryValue (sum of all value() across live objects). Update these in the constructor/destructor and in every operation that changes price or qty, so that the static total always reflects current objects.

(d) In main(), create a vector<SKU> with at least 4 items. Perform at least 6 operations mixing restock, sell, and discount across different items. After each operation, print the affected item via print(). At the end, print:  
SKUs: <s\_totalSKUs>, Inventory value: $<s\_totalInventoryValue>

(e) Add a brief comment (1–2 lines) explaining how you ensured s\_totalInventoryValue stays consistent (hint: subtract old value then add new value when mutating).

Constraints: no maps/sets required; keep to <vector>, <string>, <iostream>, <iomanip>. Avoid floating-point surprises by printing with fixed/2dp.

**Encapsulation** — hide data; expose safe ops.

**Abstraction** — define what, not how (interface).

**Inheritance** — reuse/extend behavior.

**Polymorphism** — same call, different behavior.