Coding Problems

1 FizzBuzz Problem

Problem Statement:

Write a function in C++ that prints the numbers from 1 to `n`. But for multiples of three, print `"Fizz"` instead of the number, and for the multiples of five, print `"Buzz"`. For numbers which are multiples of both three and five, print `"FizzBuzz"`.

Function Signature:

```cpp
void fizzbuzz(int n);
```

Example Test Case:

The following `main()` function runs a test with `n = 15` and should produce output matching the expected result described below.

```cpp
#include <iostream>
using namespace std;

void fizzbuzz(int n) {
    // TODO: Implement this function
}

int main() {
    fizzbuzz(15);
    return 0;
}
```

Expected Output:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
```

```
13
14
FizzBuzz
```

2 Working with Vectors: Student Scores Analyzer

Problem Statement:

Write a C++ program to read and analyze a list of student test scores. Your task is to implement a function named `analyzeScores()` that performs the following steps using a `std::vector<int>`:

1. Read a list of integer scores from the user until `-1` is entered (which should not be included in the list).
2. Display the total number of scores entered.
3. Compute and print the average score (as a `double`, rounded automatically by standard output).
4. Remove all scores from the vector that are strictly less than the average.
5. Print the updated list of scores, one per line.

Function Signature:

```cpp
void analyzeScores();
```

Test Case:

Use the following `main()` function to test your implementation.

```cpp
#include <iostream>
using namespace std;

int main() {
    analyzeScores();
    return 0;
}
```

Example Input (typed by user):

```
80
90
70
60
100
-1
```

Expected Output:

```
Total scores entered: 5
Average score: 80
Scores above or equal to average:
80
90
100
```

3 Class Design: BankAccount

Problem Statement:

Design and implement a C++ class named `BankAccount` to model a simple bank account. The class should meet the following requirements:

- The class must have two private data members:
  - `name` (a `string`): the name of the account holder
  - `balance` (a `double`): the account balance
- Provide a constructor that takes the account holder's name and an initial balance as parameters.
- Provide getter methods for both the name and the balance.
- Provide a setter method to update the name.
- Provide two public methods:
  - `deposit(double amount)`: adds the amount to the balance
  - `withdraw(double amount)`: subtracts the amount from the balance, but only if sufficient funds are available

Function Signatures:

```cpp
class BankAccount {
    // Implement this class
};
```

Test Case:

Use the following `main()` function to test your implementation.

```cpp
#include <iostream>
using namespace std;

int main() {
    BankAccount acc("Alice", 1000.0);

    cout << "Name: " << acc.getName() << endl;
    cout << "Initial Balance: " << acc.getBalance() << endl;

    acc.deposit(500.0);
    cout << "After deposit: " << acc.getBalance() << endl;
    acc.withdraw(200.0);
    cout << "After withdrawal: " << acc.getBalance() << endl;

    acc.withdraw(2000.0);  // Should not allow, balance unchanged
    cout << "After failed withdrawal: " << acc.getBalance() << endl
    ;

    acc.setName("Bob");
    cout << "Updated Name: " << acc.getName() << endl;

    return 0;
}
```

Expected Output:

```
Name: Alice
Initial Balance: 1000
After deposit: 1500
After withdrawal: 1300
After failed withdrawal: 1300
Updated Name: Bob
```

4 Dynamic Memory and Rule of Three

Write the implementation for the following C++ class that models a Game Inventory system.

```
  #ifndef INVENTORY_H
  #define INVENTORY_H

  #include <string>

  class Item {
  public:
    Item(const std::string& name, int quantity);
    std::string getName() const;
    int getQuantity() const;

  private:
    std::string name;
    int quantity;
  };

  class Inventory {
  public:
    Inventory();
    Inventory(const Inventory& other);
    Inventory& operator=(const Inventory& other);
    ~Inventory();

    void addItem(const std::string& name, int quantity);
    void removeItem(const std::string& name);
    void printInventory() const;

  private:
    Item** items;   // dynamic array of pointers to Items
    int size;       // current number of items
    int capacity;   // allocated capacity

    void resize();
    void deepCopy(const Inventory& other);
    void freeMemory();
  };

  #endif // INVENTORY_H

// main.cpp
// Test driver for Inventory/Item exercise.
// Assumes printInventory() output format described above.

// Expected output is at the bottom of this file in a block comment.

#include <iostream>
#include "Inventory.h"

int main() {
    // [1] Default construct: should be empty
    Inventory inv;
    std::cout << "[1] Newly created inventory\n";
    inv.printInventory();  // Inventory: (empty)

    // [2] Add three distinct items
    inv.addItem("Apples", 10);
    inv.addItem("Bananas", 5);
    inv.addItem("Carrots", 12);

    std::cout << "\n[2] After adding 3 items\n";
    inv.printInventory();
    // Inventory:
    // - Apples (10)
    // - Bananas (5)
    // - Carrots (12)

    // [3] Remove one existing item
    inv.removeItem("Bananas");

    std::cout << "\n[3] After removing Bananas\n";
    inv.printInventory();
    // Inventory:
    // - Apples (10)
    // - Carrots (12)

    // [4] Test copy constructor (deep copy)
    Inventory copy = inv;
    std::cout << "\n[4] Copy-constructed inventory (should match [3])\n";
    copy.printInventory();
    // Inventory:
    // - Apples (10)
    // - Carrots (12)
    // Mutate original; copy should remain unchanged if deep copy
    is correct
    inv.removeItem("Apples");

    std::cout << "\n[5] Original after removing Apples\n";
    inv.printInventory();
    // Inventory:
    // - Carrots (12)

    std::cout << "\n[6] Copy remains unchanged\n";
    copy.printInventory();
    // Inventory:
    // - Apples (10)
    // - Carrots (12)

    // [5] Test copy assignment (deep copy)
    Inventory assigned;
    assigned = inv;  // copy assignment from current ’inv’

    std::cout << "\n[7] Assigned-from-original inventory (should
    match [5])\n";
    assigned.printInventory();
    // Inventory:
    // - Carrots (12)

    // Final destructor checks happen on scope exit (inv, copy,
    assigned)

    return 0;
}

/*
===========================
EXPECTED PROGRAM OUTPUT
===========================

[1] Newly created inventory
Inventory: (empty)

[2] After adding 3 items
Inventory:
- Apples (10)
- Bananas (5)
- Carrots (12)

[3] After removing Bananas
Inventory:
- Apples (10)
- Carrots (12)

[4] Copy-constructed inventory (should match [3])
Inventory:
- Apples (10)
- Carrots (12)

[5] Original after removing Apples
Inventory:
```

```
- Carrots (12)

[6] Copy remains unchanged
Inventory:
- Apples (10)
- Carrots (12)

[7] Assigned-from-original inventory (should match [5])
Inventory:
- Carrots (12)

*/
```

5 Merging Two Singly Linked Lists

You are given a basic implementation of a singly linked list storing `int` values. Your task is to write a member function `mergeWith` that merges the current list with another sorted linked list, resulting in a single sorted linked list. Both lists are sorted in non-decreasing order.

Requirements

- Do not create new nodes.
- Reuse existing nodes and rearrange their `next` pointers.
- After merging, the current list (`*this`) should point to the merged result.

Given Code

```cpp
#include <iostream>

struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

class LinkedList {
private:
    Node* head;

public:
    LinkedList() : head(nullptr) {}

    void insertAtEnd(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next)
            temp = temp->next;
        temp->next = newNode;
    }

    void print() const {
        Node* temp = head;
        while (temp) {
            std::cout << temp->data << " -> ";
            temp = temp->next;
        }
        std::cout << "NULL\n";
    }

    // TODO: Implement this
    void mergeWith(LinkedList& other);
};
```

Your Task

Complete the implementation of the `mergeWith` function that merges two sorted lists.

Expected Behavior (Test)

```
int main() {
    LinkedList list1;
    list1.insertAtEnd(1);
    list1.insertAtEnd(3);
    list1.insertAtEnd(5);

    LinkedList list2;
    list2.insertAtEnd(2);
    list2.insertAtEnd(4);
    list2.insertAtEnd(6);

    list1.mergeWith(list2);
    list1.print(); // Expected: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> NULL

    return 0;
}
```