Lab Exercise 3: Constructors & Shallow vs Deep Copy

🧩 Problem 1: Implement a Safe Dynamic Array Class

Difficulty: Medium
Tags: Constructors, Deep Copy, Memory Management

You are given a partially implemented class DynamicArray. Complete it so that:

- It stores an array of integers using int*
- Supports initialization with a size (all elements = 0)
- Implements a **deep copy** in the copy constructor
- Has a destructor

Your task: Implement the missing parts.

cpp

```
class DynamicArray {

private:

    int* data;

    int size;


public:

    // TODO: Default constructor (size = 0)

    // TODO: Parameterized constructor (int n)

    // TODO: Copy constructor (deep copy!)

    // TODO: Destructor

    int& operator[](int index); // return data[index]

    int getSize() const { return size; }

};
```

Example:

cpp

```
DynamicArray a(3);
a[0] = 10;
DynamicArray b = a; // must be deep copy
b[0] = 20;
cout << a[0]; // prints 10
```

✅ **Expected behavior**: No memory leaks, no shared pointers.

🧩 Problem 2: Detect Shallow Copy Bug

Difficulty: Easy
Tags: Shallow Copy, Debugging

The following code crashes or produces undefined behavior. **Explain why**, and **fix it** by implementing a proper copy constructor.

cpp

```
class Text {
    char* buffer;
public:
    Text(const char* s) {
        buffer = new char[strlen(s) + 1];
        strcpy(buffer, s);
    }
    void print() { cout << buffer << endl; }
    ~Text() { delete[] buffer; }
};

int main() {
    Text t1("Hello");
    Text t2 = t1; // Problem here!
    t1.print();

    t2.print();

}
```

Your answer should include:

1. Explanation of the bug (1–2 sentences)
2. Fixed version of the class with deep copy

💡 **Hint**: What happens when both destructors run?

🧩 Problem 3: Clone a Linked List with Random Pointer (Simplified)

Difficulty: Hard
Tags: Deep Copy, Custom Copy Constructor

You are given a simple node class:

cpp

```
class Node {

public:

    int val;

    Node* next;

    Node(int x) : val(x), next(nullptr) {}

};
```

Implement a **LinkedList** class that:

- Has a head pointer (Node* head)
- Constructor from a vector of values
- **Copy constructor that performs a deep copy** (new nodes, same sequence)
- Destructor that deletes all nodes

Note: Do **not** use STL containers internally.

Example:

cpp

```
LinkedList list1({1, 2, 3});
LinkedList list2 = list1; // deep copy
list2.head->val = 99;
cout << list1.head->val; // must print 1
```

✅ **Goal**: Two independent linked lists.

🧩 Problem 4: Rule of Three Checker

Difficulty: Medium
Tags: Rule of Three, Constructors, Destructors

Which of the following classes **violate the Rule of Three** (i.e., need custom copy constructor/destructor/assignment but don’t have them)?

For each, answer **Yes** or **No**, and justify in one line.

cpp

```
// A
class A {
    int x;
    double y;
public:
    A(int a, double b) : x(a), y(b) {}
};

// B
class B {
    string name;
public:
    B(string n) : name(n) {}
};


// C
class C {
    FILE* fp;
public:
    C(const char* filename) { fp = fopen(filename, "r"); }
    ~C() { if (fp) fclose(fp); }
};


// D
class D {
    int* arr;
public:
    D(int n) { arr = new int[n]; }
    ~D() { delete[] arr; }
};
```

🧩 Problem 5: String Pool vs Unique Strings

Difficulty: Medium
Tags: Deep Copy, Optimization, Constructors

Design a class UniqueString that **always makes a deep copy** of input C-strings, even if two strings have identical content.

Also, design a second class SharedString that **uses shallow copy** (for efficiency) — but only if you can guarantee safety (assume read-only usage).

Tasks:

1. Implement UniqueString with deep copy (safe for mutation)

2. Implement SharedString with shallow copy (unsafe if modified, but efficient)
3. Write a test showing how modifying a UniqueString doesn’t affect its copy, but modifying a SharedString **would** (if allowed)

⚠️ **Note**: For SharedString, do **not** allow modification — make data const char* and provide only print().

Example test:

cpp

```
UniqueString u1("test");
UniqueString u2 = u1;
// If we had a set() method, u2.set("new") shouldn't affect u1

SharedString s1("hello");
SharedString s2 = s1; // shares pointer → saves memory
s1.print(); // "hello"
s2.print(); // "hello"
```

💡 **Learning goal**: Trade-off between safety (deep) and performance (shallow).