CSE 108 Practice: N-Dimensional Vector Class in C++

Objective: In this practice, you will implement a class to represent an n-dimensional vector in C++.

Implement the `Vector` Class

Class Requirements:

- **Data members**:
    - o `int n`: stores the dimension of the vector.
    - o `double* elements`: dynamically allocated array of size n to store the vector elements.

Class Functionality:

1. **Constructor**:
    - o A constructor that takes an integer `n` and an array to initialize the vector. It dynamically allocates memory using `malloc` and copies the array items into the **elements**.
2. **Destructor**:
    - o A destructor that frees the dynamically allocated memory for **elements** using `free`.
3. **Function Overloading**:
    - o `add`:
        1. `add(Vector& v)`: Addition of another vector (same dimension).
        2. `add(double value)`: Addition of a scalar value to all elements of the vector.
    - o `subtract()`:
        1. `subtract(Vector& v)`: Subtraction of another vector (same dimension).
        2. `subtract(double value)`: Subtraction of a scalar value from all elements of the vector.
    - o `product()`:
        1. `product(Vector& v)`: Dot product with another vector (same dimension).
        2. `product(double value)`: Scalar multiplication with all elements of the vector.
4. **Other Member Functions**:
    - o `void set_element(int index, double value)`: Sets the value of the element at the specified index.
    - o `double get_element(int index)`: Returns the value of the element at the specified index.
    - o `void display()`: Displays the vector in a user-friendly format.

N.B. **You need to pass the objects by reference to the add, subtract and product functions. If you pass by value, the copy constructor will be called and the dynamically allocated double* will be copied, leading to double free errors (once when the parameter goes out of scope, and another when the original object is freed).**

Sample Main function:

```
int main() {
    double arr1[] = {1.2, 2.3, 3.4, 4.5};
    Vector v1(4, arr1);  // 4-dimensional vector initialized with arr1

    cout << "Vector v1: ";
    v1.display();  // Output: [1.2, 2.3, 3.4, 4.5]

    double arr2[] = {5.0, 6.0, 7.0, 8.0};
    Vector v2(4, arr2);  // Another 4-dimensional vector initialized with arr2

    cout << "Vector v2: ";
    v2.display();  // Output: [5.0, 6.0, 7.0, 8.0]

    v1.add(v2);
    cout << "After adding v2 to v1: ";
    v1.display();  // Output: [6.2, 8.3, 10.4, 12.5]

    v1.add(2.0);
    cout << "After adding scalar 2.0 to v1: ";
    v1.display();  // Output: [8.2, 10.3, 12.4, 14.5]

    v1.subtract(v2);
    cout << "After subtracting v2 from v1: ";
    v1.display();  // Output: [3.2, 4.3, 5.4, 6.5]

    v1.subtract(1.5);
    cout << "After subtracting scalar 1.5 from v1: ";
    v1.display();  // Output: [1.7, 2.8, 3.9, 5.0]

    double dotProduct = v1.product(v2);
    cout << "Dot product of v1 and v2: " << dotProduct << endl;  // Output: 92.6

    v1.product(3.0);
    cout << "After multiplying v1 with scalar 3.0: ";
    v1.display();  // Output: [5.1, 8.4, 11.7, 15.0]
    return 0;
}
```