Lab Assignment #1 [Weight: ~5% of the Course Grade]

Topic: Use List ADT in Software Design

- **For this assignment, you must work in pairs.**
- **Include the name and ID for each group member in your files.**
- **For C/C++ implementation, you must separate class header and class implementation. For Python and Java implementations, keep the class implementation in one file.**
- **Please submit your completed assignment before the dropbox closes on LEARN.**

Polynomial Representation as an ADT

Let us implement the mathematical concept of a polynomial as an ADT using sequential list ADT as a template.

For C/C++ implementation, store your class definition as **`lab1_polynomial.hpp`** and your class implementation as **`lab1_polynomial.cpp`**. Also, include **`main()`** inside the **`.cpp`** file for testing.

Step1.

Create a class called **Polynomial** that will store coefficients of a polynomial expression as integers in an array.

As internal storage, you may use the **`<vector>`** library or a dynamically-sized integer array. To initialize a dynamic array, use **`int* data = new int[size]`**. To delete the array, use **`delete [] data`**. You must not use the **`<list>`** library from the Standard Template Library (STL).

Sample starter code for the **`<vector>`** and dynamically-sized integer array options, respectively:

```
class Polynomial {
      vector<int> data;
      // size already stored inside vector
public:
      Polynomial(int A[], int size) {
            data.resize(size);
            …
      }
      ~Polynomial() {
      // data will be deleted on its own
      }
```

```
class Polynomial {
      int* data;
      int data_size;
public:
      Polynomial(int A[], int size) {
            data = new int[size];
            …
      }
      ~Polynomial() {
            delete [] data;
      }
```

The coefficients should be stored in increasing order of exponents, from smallest (e.g., **`data[0]`** location) to largest (e.g., **`data[size-1]`** location). You should store both positive and negative coefficients, but only handle non-negative exponents. Coefficients that are 0 should also be stored.

For example, the polynomial $5x^5 + 3x^4 + (-2)x^3 + 1x^0$ should be stored as an array **`{1, 0, 0, -2, 3, 5}`**.

Polynomial(int A[], int size);

Implement a class constructor that takes as input two parameters: an array of integers, **int A[]**, and the size of the array, **int size**. The inputted array contains all the polynomial coefficients. The constructor creates the matching **Polynomial** instance and populates an internal array with inputted values. See the above sample starter code for details.

Polynomial();

Implement a special class constructor that takes no inputs and generates a polynomial of random size with random coefficients. The constructor creates the matching **Polynomial** instance and populates an internal array with randomly-generated values. The degree of the polynomial should range from 0 to 1000. The coefficient should range from -1000 to 1000. To generate a random value between 0 and 1000, use **rand() % 1001**. Also, call **srand(time(0))** when appropriate to randomize the seed for random values. Alternatively, you may use the **<random>** library from C++11.

Polynomial(string fileName);

Implement a class constructor that takes as input one parameter: name of the file in which polynomial coefficient are stored, **string fileName**. Inside the file, the first line contains the polynomial size while other lines contain polynomial coefficients, with one coefficient stored per line. The constructor creates the matching **Polynomial** instance and populates an internal array with inputted values.

~Polynomial(); // destructor that performs cleanup if needed

Step2.

bool operator==(const Polynomial& target); // performs *this == target

void print(); // prints the polynomial (e.g., 2x^3 + 1x^0)

Polynomial operator+(const Polynomial& target); // performs *this + target

Polynomial operator-(const Polynomial& target); // performs *this - target

Polynomial operator*(const Polynomial& target); // performs *this * target

Polynomial derivative(); // computes the derivative $\frac{d}{dx}$ of *this

Implement the overloaded operators and functions defined above as member functions of the class.

For example, if **p1 = $2x^3$ + $1x^0$** and second polynomial is **p2 = $1x^4$ + $2x^2$**, then the result of **p1 * p2** should be **$2x^7 + 4x^5 + x^4 + 2x^2$**, which should be stored as **{0, 0, 2, 0, 1, 4, 0, 2}**.

Step3.

Create a class called **PolynomialTest** and insert methods that test each **Polynomial** method implemented in Step1 and Step2. Set **PolynomialTest** as friend of **Polynomial,** so you can access attributes directly.

Test regular operation along with boundary and special cases, such as empty polynomial, negative exponents, and so on. Do not rely only on visual inspection. Instead, automate your tests as much as possible.

For example, to test if a constructor works correctly, individually compare the coefficients set inside the **Polynomial** instance with the expected array of coefficients. Furthermore, once you have verified your **operator==** implementation, you may use it to check the result of a specific operations such as addition.

If a test passed, print **"[TestName] Passed"**; else, print **"[TestName] Failed"**. You may also use assertions from the **<cassert>** library or define your own assertion template.

Run all the included tests using a method called **run()** and invoke this method from your **main()** function.

```
class PolynomialTest {
public:
	bool test_constructors1() {
		…
		return true;
	}

	…

	void run() {
		if (test_constructors1())
			cout << "Test Constructors1 Passed" << endl;
		else
			cout << "Test Constructors1 Failed" << endl;

		…
	}
};

int main() {
	PolynomialTest my_test;
	my_test.run();

	return 0;
}
```