# COMP322: Assignment 1 - Winter 2013

Due at 11:30pm EST, 5 Feb 2013

## 1 Introduction

For this assignment, we will investigate using a *linked list* representation of a polynomial. The purpose of this assignment will be the following:

- Give you practice with C-style pointers
- Give you a bit of practice with memory allocation and management (since C++does not have a built in garbage collector)
- See a couple ideas that are specific to C++
- Learn about an implement a commonly used computer science data structure.

We will not be exploiting any advanced features of the C++ language yet.

The assignment is designed to be challenging. If you are having trouble with it, please contact the instructor, a TA, or post on the discussion boards your specific issues. Collaboration is strongly encouraged as long as your final submission is your own individual work. Part of the purpose of the assignment will be to make sure you feel comfortable enough with the C language.

Unfortunately, as we only have 1 hour per week in class, it is likely we won't get to every topic for this assignment. In this case, there are a few options: a)google search, b)email instructors or TAs, c)post on discussion boards, d)come to office hours, e)chat with each other. That said, the assignment is not due for 3 weeks, so we will be covering some of this in class before the assignment is due.

## 2 Background Information

See the file titled `BackgroundInformation.pdf` on the course webpage for information necessary to complete the assignment. That pdf contains information about pointers, sparse polynomials, structs, and linked lists which will help with the assignment. In addition, we will go over some of this in class.

## 3 Assignment Requirements

On this assignment, you will, in C++, implement a computer science data structure called a *linked list* in order to represent a *sparse polynomial* in one variable.

You should do the following:

## 3.1 Defining a struct for each term (10 points)

The first thing to do is define a new C++ `struct TermElement` which contains 3 properties: 1 int representing the degree, 1 double representing the coefficient and a third property which is a pointer to a TermElement. Each `TermElement` will represent one term of a polynomial (e.g. $3x^{20}$)

## 3.2 Functions to write (90 points)

Once you have defined the struct above, you should write the following functions.

1. A function `evaluate` which takes as input a `TermElement` and a `double x` and returns the result of applying the `TermElement` passed as input to the double passed as input. For example, if the `TermElement` represents $3x^2$ and the double 2 is passed in, you should evaluate the polynomial at the point $x = 2$. Note the function `pow` that is part of the C++ library and you should use this to simplify things.

2. A function `addTerm` which takes as input a `TermElement* root` that is sorted by degree (in decreasing order with the largest degree terms first) an `int degree`, and a a `double coefficient`. The method should add the polynomial term with `degree` and `coefficient` to the polynomial represented by `root` and then return the `root` of the new polynomial (which in some cases is the same as the root of the old polynomial). It must do so in a way that satisfies the following:

    - If `root` is NULL, you should construct a term based on the input values and then return this.
    - If there is NOT a term inside the polynomial represented by `root` with the same degree as the new term, then your function you insert the term into the correct location so as to preserve sorted (by decreasing degree) order. Note that there are a few cases here depending on whether you insert into the beginning, the end, or the middle.
    - If there is already a term with the same degree, then your function should merge the coefficients together. For example, if there is already a term $3x^2$ and you are trying to add the term $4x^2$ rather than creating a new `TermElement` you should modify the existing `TermElement` to produce $7x^2$.

      If, after merging the 2 terms, the coefficient is now 0, you should `delete` this term from your linked list.

    Your function may assume that the `root` is properly sorted to being with. The idea is your function will be called in the following way

    ```
    TermElement* root = addTerm(NULL, 1,2);
    root = addTerm(root,7,8);
    root = addTerm(root,4,5);
    ```

and the polynomial $7x^8 + 4x^5 + x^2$ would be represented by the linked list (and stored in order).

Hint: You must use the new operator to create your TermElement. If you don't you will run into issues after your function finishes.

3. A function `evaluatePolynomial` which takes as input a `TermElement* root` and a `double x` and evaluates the polynomial represented by the linked list starting with root at the point `x`. Beware of potential overflow issues if the coefficients are too large (you don't need to handle these cases but if you're getting bizarre results this could be why).

4. A function `derivative` which takes as input a `TermElement* root` and returns a new linked list of TermElement based on calculating the derivative of the polynomial using the power rule. Remember that to calculate the derivative of a term of a polynomial, you multiply the base by the exponent and reduce the exponent by one. For example the derivative of $3x^4$ is $12x^3$. To compute the derivative of an entire polynomial, simply add up the separate terms.

5. A function `freeList` which takes as input a `TermElement* root` and removes every element in the list. You must make sure you perform proper memory management and do not create any memory leaks in doing this. Your function should return `void`

It is not required, but it is highly recommended you write a `printPolynomial` method for debugging purposes.

## 3.3 Analyzing your results (0 points)

This part is not graded but is useful to see the benefit of what you've written (and for that matter to help test what you've done) and should be fast to do.

To demonstrate the benefit of using the sparse polynomial representation, download from the course webpage the file `SampleTimer.cc` which contains a function `compareTimes` as well as `evaluateArrayPolynomial`. The function `compareTimes` creates a polynomial using the linked list representation as well as the array representation. It then calls the evaluate function you wrote for linked lists and the `evaluateArrayPolynomial` for arrays and times how long it takes, printing the results in milliseconds. If you make just one call, the resulting difference will be negligible, but if you set the program to call each function enough times, the difference will be noticeable. Add these two functions to your code and play around with the counters until you reach the point that the timers display a difference.

Feel free to experiment with various polynomials being tested by changing the variables in the function compareTimes. The neat thing is if you double the degree of the maximum term using the linked list approach, the run time is the approximately the same. However, if you double the maximum degree of the array form then the run time will double. (This isn't surprising but verifies the expected result.)

## 4 Using/installing g++

**g++** is the GNU C++ compiler. A recent version (4.3.2) is available on most of the Ubuntu lab computers. Our solution was developed using version 4.4.1.

We build our solution using the command line:

```
g++ -Wall -o PolynomialLinkedList.exe PolynomialLinkedList.cc
```

To do this, you'll have to add a main method. If you don't want to write a main method (you should as it's the best way to test things!), you could write

```
g++ -Wall -o PolynomialLinkedList.o PolynomialLinkedList.cc
```

to compile.

If you want to install **g++** on your system, you should be able to find instructions around the web.

For the Mac, the instructions here look reasonably accurate and up-to-date:
`http://www.edparrish.com/common/macgpp.php`

For PC/Windows systems, there are multiple choices, but the Cygwin environment mimics most of the Linux command line on Windows, so it is a good option:
`http://www.cygwin.com`

For Ubuntu (and possibly Debian) users, you can install the free `g++` package with the following command:

```
sudo apt-get install g++
```

If you have trouble, or you're running some other operating system, let us know and we'll try to help you.

## 5 Submitting your assignment

### What To Submit

| | |
|---|---|
| `PolynomialLinkedList.cc` | Here you should put all code. |
| `PolynomialLinkedList.h` | (optional) Header file where you can put your function headers |
| `Confession.txt` | (optional) In this file, you can tell the TA about any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit. On the other hand, it also may lead the TA to notice something that otherwise he or she would not. |

## 6 Not for credit bonus

Write the following functions: (These are interesting problems to solve but hard enough, particularly multiply, that they aren't required.)

- A function `addPolynomials` which takes as input two `TermElement* root1` and `root2` both sorted by increasing degree and adds the polynomials represented by these term elements together so that the resulting polynomial is still sorted by increasing degree. (Hint: Use your insert method above)
- Write a function `multiplyPolynomials` which takes as input two `TermElement* root1` and `root2` and multiplies the polynomials together using the distributive law of polynomials. Note that this means each term in the polynomial represented by `root1` by each term in the polynomial represented by `root2`.

You may also be curious to modify `compareTimes` to compare the times of the add methods using arrays vs add methods using linked lists.