CMSC 202 — Fall 2015 — Prof. Marron

Project 3

Due: Thursday, October 29, 2015 9:00PM

Objectives

1. Practice using C++ class syntax
2. Practice using C++ dynamic memory
3. Practice using linked lists

Background

For this project, you will implement member functions of a C++ `Polynomial` class that stores univariate polynomials in linked lists. You will also complete the implementation of the `Node` class which describes linked list node. You will use C++ dynamic memory (`new` and `delete`) to allocate and destory nodes in the linked lists.

Polynomials are of fundamental importance in mathemtics as even very complex functions can often be approximated by polynomials. In addition, multi-precision numbers can be represented by polynomials. For example, the 192-bit number

```
1985008943614989521094070429257356021098061753208738173457
```

is represented by the quadratic polynomial

```
5833416998877842037 x^2 + 4432272967969985261 x + 10368951529917665809
```

with x equal to 2^{64}. Each coefficient of the polynomial requires at most 64 bits of storage, so this representation allows us to store and manipulate very large numbers on a computer with a 64-bit word size. This is especially useful for public key cryptographic algorithms which typically require arithmetic computations with large integers.

In addition to storing polynomials in linked lists, you will implement functions to add, subtract, and mutiply polynomials; to determine the degree of a polynomial; and to evaluate a polynomial for a given value of x. Lastly, you will overload the insertion operator (`<<`) to display a polynomial and the assignment operator (`=`) to make a deep copy of a `Polynomial` object.

Assignment

Your assignment is to implement the member functions of the classes `Polynomial` and `Node`, defined below. A `Polynomial` object has a single variable, `m_head`, which is a pointer to the first node in the linked list. Your linked list implementation must be space-efficient, only storing what is necessary to represent the polynomial. The first node of a linked list must be a "dummy" node; the code will be cleaner and easier to write if you can assume that there is always at least one node in the linked list.

You must implement polynomial addition and subtraction (the `add()` and `subtract()` functions) efficiently, making a single pass through the linked lists of the two operands. Implementation of the remaining functions is fairly straight-forward.

```
#ifndef POLYNOMIAL_H
#define POLYNOMIAL_H

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

using namespace std;

class Polynomial {

 public:

  // Do not change the member function prototypes for
  // any public member function.

  // See documentation in Project 3 description.

  Polynomial();
  Polynomial(const Polynomial& p);
  ~Polynomial();

  void insertMonomial(long coeff, unsigned int deg);

  Polynomial add(const Polynomial& p) const;
  Polynomial subtract(const Polynomial& p) const;
  Polynomial multiply(const Polynomial& p) const;

  Polynomial modulo(const Polynomial& p) const;  // Extra Credit

  unsigned int degree() const;
  long evaluate(long x) const;

  Polynomial& operator=(const Polynomial& p);
  friend ostream& operator<<(ostream& sout, const Polynomial& p);

 private:

  // Do not change the declaration of m_head

  Node *m_head;

  // Declarations for Additional private member functions
  // may be added below. Fully document these.

};

#endif
```

Here are the requirements for the `Polynomial` member functions:

- `Polynomial();`
  The default constructor should set up the host object as an empty linked list, containing only a "dummy" node.
- `Polynomial(const Polynomial& p);`

The copy constructor creates a new `Polynomial` object from an existing one; it must make a "deep" copy.

- `~Polynomial();`
  The destructor must delete *all* the nodes of the linked list in a `Polynomial` object.
- `void insertMonomial(long coeff, unsigned int deg);`
  The function inserts the monomial (*coeff* x^{deg}) into the linked list of the host object. If a monomial of the given degree is already present, it prints a warning message and leaves the host object unchanged.
- `Polynomial add(const Polynomial& p) const;`
  The function returns the mathematical sum of the host object and `p`. The implementation must be efficient, requiring only a single pass through each linked list.
- `Polynomial subract(const Polynomial& p) const;`
  The function returns the mathematical difference of the host object and `p` (`host - p`). The implemenation must be efficieint (see `add()`, above).
- `Polynomial multiply(const Polynomial& p) const;`
  The functions returns the mathematical product of the host polynomial and `p`.
- `unsigned int degree() const;`
  The function returns the degree (highest power of *x*) of the host object.
- `long evaluate(long x) const;`
  The function evaluates the polynomial for the given value of *x*.
- `Polynomial& operator=(const Polynomial& p);`
  The overloaded assignment operator performs a "deep copy" of *p*.
- `friend ostream& operator<<(ostream& sout, const Polynomial& p);`
  The overloaded insertion operator can be used to display a `Polynomial` object in standard mathematical notation. See the test output for a sample of acceptable output.
- **Extra Credit:** `Polynomial modulo(const Polynomial& p) const;`
  The function returns the remainder when dividing the host polynomial by *p*.

```
#ifndef NODE_H
#define NODE_H

using namespace std;

// Do not change any part of this file

// See documentation in Project 3 description.

class Node {

 public:

  Node();
  Node(long coeff, unsigned int deg);

  long m_coefficient;
  unsigned int m_degree;
  Node *m_next;
};
```

```
#endif
```

The `Node` class has two constructors, but no other functions. The variable `m_coefficient` stores the coefficient of a monomial and `m_degree` stores the degree (power of *x*) of a monomial; `m_next` is a pointer to the next node in the linked list.

Here are the requirements for the `Node` member functions:

- `Node();`
  The default constructor initializes `m_next` to `NULL`.
- `Node(long coeff, unsigned int deg);`
  The construtor initializes `m_ceofficient` to `coeff`, `m_degree` to `deg`, and `m_next` to `NULL`.

Implementation Issues
Here are some issues to consider and pitfalls to avoid. (You should read these before coding!)

- **Reminder:** do not develop your programs in the submission directories. Develop your programs in some other location in GL or on your own computer. The files that you have in the submission directory should not be your only copy of your code.
- Your linked list implementation must be space-efficient, so you should not store any terms with coefficient zero. In particular, the zero polynomial is represented by a linked list having *only* the dummy node.
- It is a good idea to store the terms of your polynomial in order of *decreasing degree*. Standard mathematical notation writes polynomials in this order; storing the terms in the same order makes the implementation of the overloaded insertion operator easier. Also, in this case the degree of the polynomial is just the degree of the first non-dummy node in the linked list.
- The variables of the `Node` class are all public, so they can be accessed from within the `Polynomial` class without the need for accessors or mutators.
- The header files `Polynomial.h` and `Node.h` and a test program `test.cpp` and sample output `test.out` can be copied to your working directory on GL:

  ```
  cp /afs/umbc.edu/users/c/m/cmarron/pub/cmsc202/proj3.zip .
  unzip proj3.zip
  ```

  You should put your implementations of the `Polynomial` and `Node` member functions in the files `Polynomial.cpp` and `Node.cpp`, respectively. You should not have a `main()` function in `Polynomial.cpp` or `Node.cpp`; any `main()` functions should be in separate test or driver files.
- On GL, you can compile the test program with your implementation with one command, like this:

  ```
  linux3% g++ -Wall -g test.cpp Polynomial.cpp Node.cpp
  ```

  or separately, like this:

  ```
  linux3% g++ -Wall -g -c Node.cpp
  linux3% g++ -Wall -g -c Polynomial.cpp
  linux3% g++ -Wall -g -c test.cpp
  linux3% g++ -g test.o Polynomial.o Node.o
  ```

Submitting your program
You should submit these files to the `proj3` subdirectory:

- `Polynomial.h` The only change from the provided file should be the addition of private member function prototypes, if used.

- `Polynomial.cpp` This should contain your implementations of all of the `Polynomial` member functions. This file should **not** include a `main()` function.
- `Node.cpp` This should contain your implementations of all the `Node` member functions. This file should **not** include a `main()` function.
- `mytest.cpp` A main program that exercises the *working* parts of your submission. This is where you tell us what works. If you could not get some member function to work, don't test it here. We will test all functions with other programs.

If you followed the directions for setting up the shared directories, then you can copy your code to the submission directory with:

```
cp Polynomial.h Polynomial.cpp Node.cpp mytest.cpp  ~/cs202proj/proj3/
```

You can check that your program compiles and runs in the `proj3` directory, but please clean up any `.o` and executable files. Again, do not develop your code in this directory; you need to keep the main copy of your program elsewhere.
