The University of Chicago
Department of Computer Science

CMSC 15200 – Introduction to Computer Science 2
Summer Quarter 2007
Homework #4 (08/03/2007)
Due: 08/08/2007 @ 1:30pm

Name:

Student ID:

Instructor: Borja Sotomayor

Do not write in this area

A.1	A.2	A.3	A.4	A.5	A.6	B.1	B.2	B.3	**TOTAL**

Maximum possible points: 75 + 10

This homework has two parts: Part A (Polynomial ADT) and Part B (Doubly Linked List). You are only required to do part A if you are taking this class for credit.

Part A: Polynomial ADT

In this part you will implement a *Polynomial* ADT (Abstract Data Type) using C++ classes. In particular, we will only consider second-order polynomials:

$$ax^2 + bx + c$$

The class declaration is the following (poly.h in the homework files):

```
class Polynomial
{
	private:
		/* Member variables */
		int a, b, c;
		/* Static member variable */
		static int numPolynomials;

	public:
		/* Constructors */
		Polynomial();
		Polynomial(int a, int b, int c);
		Polynomial(const Polynomial &p); // Copy constructor

		/* Destructor */
		~Polynomial();

		/* Member functions */
		bool hasRealSolution();
		double getRealSolution1();
		double getRealSolution2();
		string str();

		/* Overloaded operators */
		Polynomial& operator=(const Polynomial &p);
		Polynomial  operator+(const Polynomial &p);
		bool        operator==(const Polynomial &p);

		int operator()(int x);

		/* Friends */


		/* Static member function */
		static int getNumPolynomials();
};
```

A poly.cpp file is provided that includes a partial implementation of the constructors, and the implementation of the static getNumPolynomials() function.

To test your implementation, a main.cpp is provided in the homework files. Running this program with a correct implementation should yield the following:

```
Number of polynomials is 5
p2 and p3 are the same. Good!
p2 and p5 are the same. Good!
p2 and p4 are not the same. Good!
p2 has real solutions x1=3 , x2=-5
p4 has no real solutions.
p2 is 2x^2 + 4x - 30
p2+p3 is 4x^2 + 8x - 60
p2 is 2x^2 + 4x - 30
p5 is 4x^2 + 8x - 60
p2(3) = 0
p2(-5) = 0
p2(0) = -30
Number of polynomials is 4
```

Exercise 1 <<5 points>>

A partial implementation of these constructors is provided:

```
Polynomial();
Polynomial(int a, int b, int c);
```

However, these constructors do not modify the static *numPolynomials* member variable (which keeps a count of the number of *Polynomial* instances created). Modify the constructors so they will correctly change the value of *numPolynomials*, and implement the destructor:

```
~Polynomial();
```

Also, you must make sure that the static *numPolynomials* member variable is correctly initialized.

Exercise 2 <<5 points>>

Implement the copy constructor:

```
Polynomial(const Polynomial &p);
```

Exercise 3 <<10 points>>

Implement the following member functions:

```
bool hasRealSolution();
double getRealSolution1();
double getRealSolution2();
```

In these functions, you will consider the polynomial as a quadratic equation (P(x)=0). *hasRealSolution* returns true if the equation has a real solution (or two), and false otherwise. *getRealSolution1* and *getRealSolution2* assume that a real solution exists, and return each of the real solutions (if the equation has a unique solution, they return the same value).

If you only vaguely remember the quadratic formula, you can get up to speed here:
http://en.wikipedia.org/wiki/Quadratic_equation

Extra credit (10 points): Modify these functions to consider all possible solutions (two complex solutions, one single real solution, two real solutions) and all possible error conditions (e.g. what if a=0? What if a=b=0? What if a=b=c=0?). You should not do this by adding more functions ("getComplexSolution1", ...) but by writing a single function that returns an error code *and* two complex numbers (using the ComplexNumber ADT seen in class).

Exercise 4 <<10 points>>

Overload the following operators:

```
Polynomial& operator=(const Polynomial &p);
Polynomial  operator+(const Polynomial &p);
bool        operator==(const Polynomial &p);
```

Exercise 5 <<10 points>>

Overload the function call operator:

```
int operator()(int x);
```

Note: We have not discussed the function call operator in class. You will have to read about it on your own.

You must overload the function call operator in such a way that using the parentheses operator on a *Polynomial* object will return the value of that polynomial when *x* is equal to the integer value supplied as a parameter. For example:

```
Polynomial p(1,2,3);

cout << p(2); // Prints out “11” (Why? ---> 1*2^2 + 2*2 + 3 = 11)
```

Exercise 6 <<5 points>>

Implement the following function:

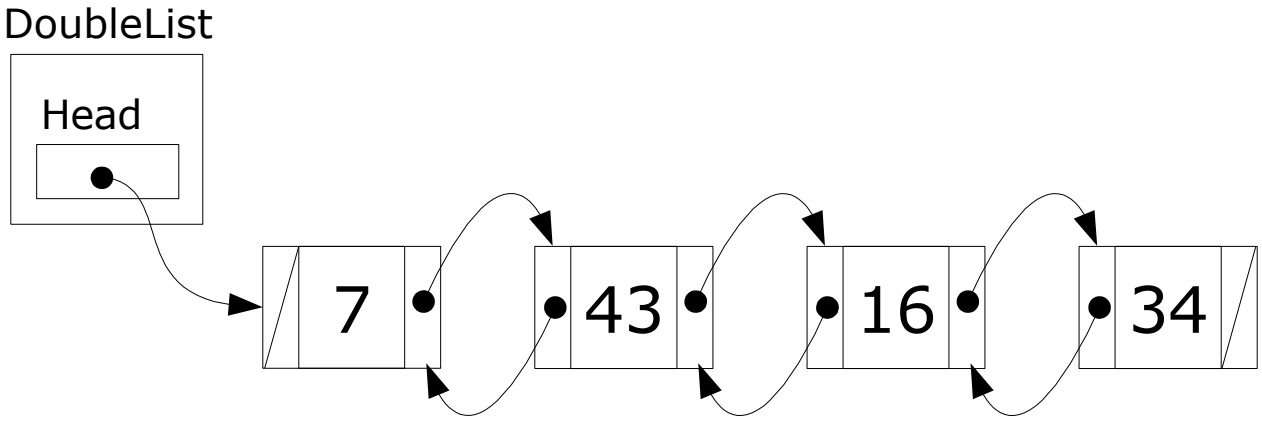
```
string str();
```

This function returns a string representation of the polynomial. You must make sure that you print out the polynomial *correctly.* Hint: Careful with terms that have a negative coefficient and terms that have a zero coefficient.

Part B: Doubly-Linked List

NOTE: You only need to do this exercise if you are taking CMSC 15200 for credit.

You will implement a doubly linked list (as described in class).



The structure and function declarations are the following (dlist.h in the homework files):

```
struct ListNode {
       int data;
       ListNode *next;
       ListNode *prev;
};

struct DList {
       ListNode *head;
};

void createList(DList &l);
ListNode* first(DList &l);
void insertHead(DList &l, int data);
void insertAfter(ListNode* node, int data);
void printData(DList &l);
bool find(DList &l, int data);
void deleteHead(DList &l);
void deleteAfter(ListNode* node);
void deleteData(DList &l, int data);
void deleteList(DList &l);

// New functions
/* Inserts a new node with provided data before the specified node */
void insertBefore(DList &l, ListNode* node, int data);
/* Deletes the specified node */
void deleteNode(DList &l, ListNode* node);
```

Don't reinvent the wheel. Reuse as much code as possible from the list implementation seen in class! In fact, you should only tweak the existing functions to make sure that the "prev" pointer always has a valid value, and then implement the new "insertBefore" and "deleteNode" functions.

Note that, although the book describes how to implement a doubly-linked list, you *cannot* take code directly from the book. This exercise also evaluates your ability to read code (both from the book and the provided list implementation) and adapt it to your own needs.

To test your list implementation, a main_double.cpp is provided in the homework files. Running this program with a correct doubly linked list implementation should yield the following:

```
5 4 3 2 1 5 4 3 2 1
5 1 4 3 2 1 5 4 3 2 1
1
0
1 4 3 2 1 5 4 3 2 1
1 3 2 1 5 4 3 2 1
1 2 1 5 4 3 2 1
42 1 2 1 5 4 3 2 1
42 37 1 2 1 5 4 3 2 1
37 1 2 1 5 4 3 2 1
37 1 2 1 4 3 2 1
37 2 4 3 2
List is empty!
```

Note: Debugging programs that use data structures is no easy matter, and you are likely to encounter run-time errors due to dangling pointers. One good strategy is to draw the data structure on paper, and for each operation see how the "next" and "prev" pointers are affected. Don't forget to consider special cases (e.g. "What happens if I try to delete the *first* node?") Also, if your program crashes unexpectedly (the hallmark of a dangling pointer), the Eclipse debugger can come in handy to pinpoint the dangling pointer (the debugger will pause execution at the exact line that caused the crash).

Note 2: In your code, make sure you explicitly point out (with comments) what code you had to add/modify to turn the list into a doubly linked list.

The point weights for this exercise are the following: <<10 points>> for correctly updating the "prev" pointer, <<10 points>> for implementing insertBefore, <<10 points>> for implementing deleteNode.