ECE 2574 ♦ Supplementary Notes for Homework 2

This document contains the following contents:

1. Polynomials
2. File I/O
3. Command line arguments
4. Storing a polynomial using a linked list-based ADT
5. Exception handling
6. Copy Constructor (Understanding deep copy)
7. Overloading operators for polynomials

A short review of each of these concepts is present below, along with example code snippets which should help you get started.

Warning: There may be errors/typos in the example code snippets. I strongly urge you to use them as a reference to help design/debug your program, rather than simply inserting them directly in your program.

1. Polynomials:

A polynomial is a mathematical expression involving a sum of powers in one or more variables multiplied by coefficients.

A polynomial in one variable (i.e., a univariate polynomial) with constant coefficients is given by

$$a_n x^n + a_{n-1} x^{n-1} + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + a_2 x^2 + a_1 x + a_0 .$$

For example: $2x^5 – 5x^3 – 10x + 9$, $2x^{12} - 5x^9 – 12$

2. File I/O

C++ has two basic classes to handle files, *ifstream* and *ofstream*. To use them, include the header file *fstream*. The class *ifstream* handles file input (reading from files), and *ofstream* handles file output (writing to files). The way to declare an instance of the *ifstream* or *ofstream* class is:

```
ifstream a_file;
```

or

```
ifstream a_file ( "filename" );
```

The constructor for both classes will actually open the file if you pass the name as an argument. Both classes have an open command (`a_file.open()`) and a close command (`a_file.close()`). You are not required to use the close command as it will automatically be called when the program terminates, but if you need to close the file before the program ends, it is useful.

The beauty of the C++ method of handling files rests in the simplicity of the actual functions used in basic input and output operations. Because C++ supports overloading operators, it is possible to use << and >> in front of the instance of the class as if it were cout or cin. In fact, file streams can be used exactly the same as *cout* and *cin* after they are opened.
For example:

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
  char str[10];

  //Creates an instance of ofstream, and opens example.txt
  ofstream a_file ( "example.txt" );

  // Outputs to example.txt through a_file
  a_file << "This text will now be inside of example.txt";

  // Close the file stream explicitly
  a_file.close();

  //Opens for reading the file
  ifstream b_file ( "example.txt" );

  //Reads one string from the file
  b_file >> str;

  //Should output 'this'
  cout << str << "\n";

  // wait for a keypress
  cin.get();

  // b_file is closed implicitly here
}
```

The default mode for opening a file with `ofstream`'s constructor is to create it if it does not exist, or delete everything in it if something does exist in it.

For this assignment, the part of your code that reads the data from a given file may look similar to the following:

```
. . .
int coef;
int exponent;

ifstream input( “infile.txt” );
ofstream output( “outfile.txt” );
```

```
. . .
while( input >> coef >> exponent )
{
  //add this term to the polynomial;
  . . .
}
```

To write the data to a file, use "<<".

For example,

```
. . .
ofstream output( "outfile.txt" );
. . .
int result1 = 5;
int result2 = 10;
. . .
output << result1 << " " << result2 << endl;
. . .
```

You should keep in mind that the purpose of an ADT is to encapsulate all functionality required to understand and maintain the data stored in an instance of the ADT, and nothing more. For example, code to obtain a particular term's coefficient can be written as a method of the ADT, but code that writes said coefficient to a file should not be a method of the ADT. Think of the ADT as a data type – the methods of the ADT only process the contents of the ADT, thus preserving the generality of the ADT.

For details, read the material in the following link:
http://www.cplusplus.com/doc/tutorial/files/

3. Command line arguments

In C++, it is possible to accept command line arguments. Command-line arguments are given after the name of a program in a command-line operating systems like DOS or Linux, and are passed in to the program from the operating system. To use command line arguments in your program, you must first understand the full declaration of the main function. In fact, main can actually accept two arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.

The full declaration of main looks like this:

```
int main ( int argc, char *argv[] )
```

The integer, argc is the ARGument Count (hence argc). It is the number of arguments passed into the program from the command line, including the name of the program. The array of character pointers is the listing of all the arguments. argv[0] is the name of the program, or an empty string if the name is not available. After that, every element number less than argc is a command line argument. You can use each argv element just like a string, or use argv as a two dimensional array. argv[argc] is a null pointer.

How can this be used? Any program that wants its parameters to be set when it is executed would need to use this. One common use is to write a function that takes the name of a file and outputs the entire text of it onto the screen. For example:

```
#include <fstream>
#include <iostream>

using namespace std;

int main ( int argc, char *argv[] )
{
  if ( argc != 2 ) // argc should be 2 for correct execution

       // We print argv[0] assuming it is the program name
    cout<<"usage: "<< argv[0] <<" <filename>\n";
  else {

    // We assume argv[1] is a filename to open
    ifstream the_file ( argv[1] );

    // Always check to see if file opening succeeded
    if ( !the_file.is_open() )
      cout<<"Could not open file\n";
    else {
      char x;
      // the_file.get ( x ) returns false if the end of the file
      // is reached or an error occurs
      while ( the_file.get ( x ) )
        cout << x;
    }

    // the_file is closed implicitly here
  }
  return 0;
}
```

The above program is fairly simple. It uses the full declaration of main. Then it first checks to ensure that the user enters two arguments. The program then checks to see if the file is valid by trying to open it. This is a standard operation that is effective and easy. If the file is valid, it gets opened.

For more details, please read the material in the following link:
http://www.learncpp.com/cpp-tutorial/713-command-line-arguments/

4. Storing a polynomial using a linked list-based ADT

The Poly ADT stores the terms of its polynomial as items in a linked list. The Poly class contains a pointer to the head of the linked list which holds the terms of the polynomial. The terms themselves are nodes in polyTerm instances.

The Poly class also maintains a current pointer, pointing to the current element. This pointer can be used by Poly's methods to maintain the state of processing of the object.

Your header file will need to include a number of declarations. An example of a partial header file is given below. *Note that this is just an example; your header file may have more or less declarations.*

```
#include <iostream>
#include <iomanip>
#include <stdlib.h>

using namespace std;

// Here the user can specify a different type;
// Your implementation should work for double, float, and int
typedef int coefType;
typedef int exponentType;


class Poly
{
   public:

// Constructor: create an empty polynomial linked list.
        Poly();

// Copy constructor: create a deep copy
        Poly(const Poly& original);

// Destructor: deallocate memory that was allocated dynamically
        ~Poly();

// Get corresponding coefficient for an exponent
        coefType getCoef( exponentType exponent );

// Set the coefficient for an exponent
        void setCoef( exponentType exponent, coefType coef );

// Find the term for an exponent, returns a pointer to the term.
// If there is no term with that exponent, make a new term and set
// the coefficient to zero.
        void findTerm( exponentType exponent );

// Insert a new term. This is placed directly after the current
// item.
        void newTerm( exponentType exponent, coefType coef );

// Overloaded assignment operator
        Poly& operator = (const Poly& orig);
```

```
// Find the highest exponent in the polynomial
        exponentType findDegree();

// Overloaded math operators
        Poly operator+(const Poly& orig);
        Poly operator*(const Poly& orig);


   private:

// List node, called polyTerm for Polynomial Term
        struct polyTerm
        {
              coefType coef;
              exponentType exponent;
              polyTerm *next;
        };


// Points to the first term
        polyTerm *head;

// Points to the current term – the tail end term of the linked list
        polyTerm *current;

};
```

A NULL value of the next pointer indicates the end of the linked list. It is important to assign the NULL value to the next pointer when adding a new Polyterm.

5. Exception Handling

Exception handling is a programming language construct designed to handle the occurrence of exceptions, special conditions that change the normal flow of program execution. An example of a simple exception handling routine is given below.

Example:

```
some_class some_class::some_function(input)
{
   try
   {
        if(check for exception type 1)
              throw 1;

        .....

        if(check for exception type 2)
```

```
            throw 2;

      .....

   }
   catch(int err)
   {
         cerr << "Some message on the screen to help user";
         if (err == 1)
               cerr << "Statement explaining exception type 1"<<endl;
         if (err == 2)
               cerr<<" Statement explaining exception type 2"<<endl;
   }


}
```

For this assignment, your implementation should use try/throw/catch to handle a number of different types of exceptions. For example, your program needs to handle the exceptions caused by scenarios where any coefficient of the input polynomials is zero or when any exponent of the input polynomials is negative. The code for handling such exceptions may look similar to the following:

```
. . .
try
{
while( input >> coef >> exponent )
{
// Check for negative exponent or zero coefficient
         if( exponent < 0 ) throw(1);
         if( coef == 0 ) throw(2);

               . . .
   }
}
catch( int error )
{
   if( error == 1 )
   {
         cout << "ERROR: Input exponent is negative!" << endl;
   }
   if( error == 2 )
   {
         cout << "ERROR: Input coefficient is zero!" << endl;
   }
   . . .
}
```

The new statement immediately exits if there is an error and it cannot allocate memory. Hence, handling this type of exception is different from other typical exception handling routines. To prevent it from exiting, we must catch the error. But we also throw the error again so that surrounding catch blocks are also told about the error. When checking for failed allocations, you need to catch a bad allocation ("bad_alloc") error in case

of failed allocations to prevent the program from exiting. Here is an example code that can be used to handle a bad allocation error.

```
try {

      . . .

      try {
           data = new double [100];
      }

      catch(bad_alloc& err) {
                      throw(3);
      }

      . . .

} catch(int err) {
      if err == 3:
      cerr <<”Allocation Failed” << endl;
}
```

If you would like to test exception handling for a new statement, you could supply a very large or a negative size for the allocation. If an error occurs, you may need to return a value from a location outside your try block. In such a situation, it is better to return a value which indicates that the error occurred. If such a value does not exist or cannot be used, you should try returning a value which causes the least impact to the execution of the rest of your code.

The following example code shows how to handle exceptions when input/output files are not provided on command line, input file does not exist, or input file does not have string XXX

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include "PolytermsP.h"

int main(int argc, char* argv[])
{
      ifstream inFile;
      ofstream outFile;
      string line;

      try
      {
            if (argc == 3)
            {
                  inFile.open(argv[1]);
                  outFile.open(argv[2]);

                  if (inFile.is_open() && outFile.is_open())
                  {
                        bool separator = false;
```

```
                        try
                        {
                              while(!inFile.eof())
                              {
                                    getline(inFile, line);
                                    istringstream iss (line);
                                    if(line == "XXX")
                                          separator = true;
                              }
                              if (separator == false)
                              {
                                    throw 1;
                              }

                        }
                        catch(int x)
                        {
                              if (x == 1)
                                    outFile << "bad input file: did not have 'XXX'
   to separate polynomials" << endl;
                        }
                        inFile.close();
                        outFile.close();
                  }
                  else
                  {
                        throw 2;
                  }
            }
            else
            {
                  throw 3;
            }
      }
      catch(int x)
      {
            if (x == 3)
                  outFile << "input or output file not provided on command line"
   << endl;
            else if (x == 2)
                  outFile << "input file does not exist" << endl;
      }
      return 0;
}
```

6. Copy Constructor

In the previous assignment we learnt that overloading copy constructor is important to copy values of one objects into another. There are two types in which a memory can be copied.

a.) Shallow Copy: When you create a shallow copy of a dynamically allocated object **a** in **b**, you are only copying the memory location being pointed to by **a**. Essentially, **b** and **a** now point to the same memory location. This can have unforeseen consequences if you are not

careful. If you change the object **a**, then **b** automatically gets changed because **b** points to the same memory location.

b.) Deep Copy: Deep copy as the name suggests, creates a "new" memory dynamically and then copy over the object element by element. This means , when you change object **a**, object **b** does not get affected because **b** is pointing to its own memory location.

Let's look at an example:

```
1.  // Author: Harsh Agrawal
2.  class MyClass {
3.    public:
4.    int *a;
5.    int size;
6.    MyClass(int size) {
7.      this->size = size;
8.      a = new int[size];
9.      for( int i=0;i<size;i++)
10.       a[i] = i;
11.   }
12.   /* Copy Constructor - Shallow Copy - Wrong
13.   MyClass(const MyClass& other) {
14.     a = other.a; // creates a shallow copy
15.     size = other.size; //doesn't matter, static primitive data type
16.   }
17.   */
18.
19.   /* Copy Constuctor - Deep Copy - Correct  */
20.   MyClass(const MyClass& other) {
21.     size = other.size;
22.     a = new int[size];                  // Step 1: ALLOCATE new memory
23.     for( int i = 0; i < size; i++) {    // Step 2: Copy the values element by element
24.       a[i] = other.a[i];                // Copy one element
25.     }
26.   }
27.
28. }
29.
30. int main(){
31.   MyClass obj1(10);
32.   MyClass obj2 = obj1; // Copy Constructor Called.
33.   obj1.a[0] = -1;
34.   cout << "obj1.a[0]: "<<obj1.a[0];   // should print -1
```

```
35.   cout << "obj2.a[0]: "<<obj2.a[0];   // Deep Copy. Should print 0;
36.
37. }
```

In previous example, if you comment out the correct implementation, and uncomment out the incorrect implementation, then both obj1.a[0] and obj2.a[0] will be -1 even if you just changed one object.

In this assignment, make sure you implement a deep copy constructor. These are difficult to debug so be careful while implementing.

7. Overloading Operators for Polynomials

Polynomial arithmetic is used in a variety of applications, especially cryptography. The operator methods can thus be written with different targets in mind – whether we would like to reduce space complexity (i.e., memory use) or time complexity (i.e., improve speed).

If you are aiming for simplicity, the order and structure of each polynomial can be preserved by using a sort() and a simplify() method. The sort() method would sort the terms into increasing or decreasing exponent, and simplify() would combine terms which have the same exponent. The operators for polynomials can thus be defined by accumulating relevant terms into a new polynomial, and then calling sort() and simplify() on the new polynomial. For example, the + operator may be implemented as given below. The + operator adds all terms from both operand polynomials into a new polynomial. It then sorts the new polynomial using a sort() call, and then simplifies it using a simplify() call. The sort() function will arrange the terms in ascending or descending order. The simplify() function will combine all terms which have the same exponent. The actual addition is carried out by the simplify() function, since calling simplify() for the result polynomial will be the same as adding all terms of an equal exponent together. Example code is provided below. Note that this is only an example; your code may look different from this.

```
Poly Poly::operator+(const Poly& orig)
{
   Poly n;           //Polynomial which stores the result of addition

   //cur and cur2 are iterator pointers which point to polyterm being considered
   polyterm * cur;
   polyterm * cur2;

   //cur and cur2 are initialized to the first Polyterm object of each polynomial
   cur = head;
   cur2 = orig.head;

   //Add all the terms from the Left Hand polynomial
   while(cur != NULL)
   {
         n.newterm(cur->exponent, cur->coefficient);
         cur = cur->next;
   }
```

```
   //Add all the terms from the Right Hand polynomial
   while(cur2 != NULL)
   {
         n.newterm(cur2->exponent, cur2->coefficient);
         cur2 = cur2->next;
   }
   n.sort(); //sort terms in ascending or descending order of exponents
   n.simplify();//combine terms with equal exponents into a single term by addition

   return n;  //Return the result polynomial n to the calling function
}
```

The multiplication operator can also be implemented using the same approach. Example code is provided below.

```
Poly Poly::operator*(const Poly& orig)
{
   //n stores the result of the multiplication
   Poly n;

   // new_coef and new_expo store the coefficient and exponent of the next term to
   // be added to n
   coeftype new_coef;
   expotype new_expo;

   //The loops below take each term in the Left Hand polynomial, multiply it with
   //all the terms of the right hand polynomial, and store term into n

   for(polyterm * cur = head; cur != NULL; cur= cur->next)
   {
         for(polyterm * cur2 = orig.head; cur2 != NULL; cur2 = cur2->next)
         {
               new_coef = cur->coefficient * cur2->coefficient;
               new_expo = cur->exponent + cur2->exponent;
               n.newterm(new_expo, new_coef);
               }
   }

   n.sort();     //sort terms in ascending or descending order of exponents
   n.simplify(); //combine terms with equal exponents

   return n;  //Return the result polynomial n to the calling function
}
```

Example code for the = operator is given below.

```
   Poly Poly::operator=(const Poly& orig)
   {
         polyterm * cur; //iterator pointer for Right hand polynomial
         cur = orig.head;

         polyterm * cur2; //iterator pointer for Left Hand Polynomial
         cur2 = this->head;

         //This loop deletes all Polyterms currently associated with the Left hand
         //Polynomial
```

```
        while(cur2 != NULL)
        {
              cur2 = cur2->next;
              delete cur2;
        }

        //Once all Polyterms are deleted, set current and head to NULL
        this-> head = NULL;
        this-> current = NULL;

        //This loop adds a new Polyterm to the Left Hand Polynomial for
        //every term in the Right Hand Polynomial
        while(cur->next != NULL)
        {
              cout << "found" << endl;
              (*this).newterm(cur->exponent, cur->coefficient);
              cur = cur-> next;

        }

        //The this pointer is returned so operators can be chained
        return * this;
  }
```

Again, Compile and run your code in Visual Studio 2012 before turning it in. Make sure you follow the naming conventions specified in the homework description document. Follow the output file format specified in the document.

The above implementations are sufficient for this project. But it should be noted that more efficient implementations for these operations are possible. The multiplication (*) operator implementation given above will be comparatively slow, as you may have to sort n*m elements in certain cases (m and n are the degrees of the polynomials). There are faster implementations, which are based on recursion, such as an implementation using the Karatsuba method/algorithm. Refer to the advanced readings for more information.

Useful References:

1. http://www.bogotobogo.com/cplusplus/linkedlist.php#linkedlistexample10
2. http://mathworld.wolfram.com/Polynomial.html
3. http://en.wikipedia.org/wiki/Linked_list
4. http://en.wikipedia.org/wiki/Exception_handling
5. http://learning-computer-programming.blogspot.com/2007/06/introduction-to-dynamic-memory.html
6. http://www.fredosaurus.com/notes-cpp/ds-lists/list-ex1a.html
7. http://www.codeproject.com/Articles/24684/How-to-create-Linked-list-using-C-C

Advanced Reading:

1. http://en.wikipedia.org/wiki/Karatsuba_algorithm
2. http://cslibrary.stanford.edu/105/LinkedListProblems.pdf
3. http://www.codeofhonor.com/blog/avoiding-game-crashes-related-to-linked-lists
4. http://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/llrecursion/index.html