COP 3330 - Programming Assignment #7

Due: Mon, Nov 21 (revised)

Objective: Upon completion of this program, you should gain experience with operator overloading, as well as further practice with dynamic memory allocation inside a class.

The code for this assignment should be portable -- make sure you test with g++ on program.cs.fsu.edu before you submit.

Note: Due to a bug... ahem, an "Undocumented Feature"... in Visual C++ 6.0, you may need to use the following format in your header file if you want to test with Visual C++ 6.0:

```
using std::ostream;
using std::istream;
```

For some reason, when doing overloads of the insertion and extraction operators, Visual C++ does not like it if you just use the blanket statement:

```
using namespace std;
```

(Although, g++ will handle the latter form just fine).

Task: One common limitation of programming languages is that the built-in types are limited to finite ranges of storage. For instance, the built-in `int` type in C++ is 4 bytes in most systems today, allowing for about 4 billion different numbers. The regular `int` splits this range between positive and negative numbers, but even an `unsigned int` (assuming 4 bytes) is limited to the range 0 through 4,294,967,295. Heavy scientific computing applications often have the need for computing with larger numbers than the capacity of the normal integer types provided by the system. In C++, the largest integer type is `long`, but this still has an upper limit.

Your task will be to create a class, called `MyInt`, which will allow storage of any non-negative integer (theoretically without an upper limit -- although there naturally is an eventual limit to storage in a program). You will also provide some operator overloads, so that objects of type `MyInt` will act like regular integers, to some extent (To make them act completely like regular integers, it would take overloading most of the operators available, which we will not do here). There are many ways to go about creating such a class, but all will require dynamic allocation (since variables of this type should not have a limit on their capacity).

The class, along with the required operator overloads, should be written in the files `"myint.h"` and `"myint.cpp"`. I have provided starter versions of these files, along with some of the declarations you will need. You will need to add in others, and define all of the necessary functions. You can get a copy of the starter files here: myint.h and myint.cpp. I have also provided a sample main program to assist in testing, along with a link to some sample test runs from my version of the class. The sample main program is linked at the bottom of this specification. Details and Requirements are listed below, followed by some hints and tips on some of the items.

Details and Requirements:

1) Your class must allow for storage of non-negative integers of any (*theoretical*) size. While not the most efficient in terms of storage, the easiest method is to use a dynamic array, in which each array slot is one decimal "digit" of the number. (This is the suggested technique). You may go ahead and assume that the number of digits will be bound by the values allowed in an unsigned int varible (it would be a good idea to keep track of the number of digits in such a variable). You should create appropriate member data in your class. All member data must be private.

2) There should be a constructor that expects a regular int parameter, with a default value of 0 (so that it also acts as a default constructor). If a negative parameter is provided, set the object to represent the value 0. Otherwise, set the object to represent the value provided in the parameter. There should be a second constructor that expects a C-style string (null-terminated array of type char) as a parameter. If the string provided is empty or contains any characters other than digits ('0' through '9'), set the object to represent the value 0. Otherwise, set the object to represent the value of the number given in the string (which might be longer than a normal int could hold).

Note that the two constructors described above will act as "conversion constructors" (as discussed previously in lecture class). Recall that such constructors will allow automatic type conversions to take place -- in this case, conversions from `int` to `MyInt` and from c-style strings to type `MyInt` -- when appropriate. This makes our operator overloads more versatile, as well. For example, the conversion constructor allows the following statements to work (assuming appropriate definitions of the assignment operator and + overloads described later):

```
MyInt x = 1234;
MyInt y = "12345";
MyInt z = x + 12;
```

3) Since dynamic allocation is necessary, you will need to write appropriate definitions of the special functions (the "automatics"): destructor, copy constructor, assignment operator. The destructor should clean up any dynamic memory when a MyInt object is deallocated. The copy constructor and assignment operator should both be defined to make a "deep copy" of the object (copying all dynamic data, in addition to regular member data), using appropriate techniques. Make sure that none of these functions will ever allow memory "leaks" in a program.

4) If you use a dynamic array, you will need to allow for resizing the array when needed. There should never be more than 5 unused array slots at any given time.

Operator overloads: (The primary functionality will be provided by the following operator overloads).

5) Create an overload of the insertion operator << for output of numbers. This should print the number in the regular decimal (base 10) format -- and no extra formatting (no newlines, spaces, etc -- just the number).

6) Create an overload of the extraction operator >> for reading integers from an input stream. This operator should ignore any leading white space before the number, then read consecutive digits until a non-digit is encountered (this is the same way that >> for a normal `int` works, so we want to make ours

work the same way). This operator should only extract and store the digits in the object. The "first non-digit" encountered after the number may be part of the next input, so should not be extracted. You may assume that the first non-whitespace character in the input will be a digit. i.e. you do not have to error check for entry of an inappropriate type (like a letter) when you have asked for a number.

Example: Suppose the following code is executed, and the input typed is `"  12345  7894H"`**.**

```
MyInt x, y;
char ch;
cin >> x >> y >> ch;
```

The value of `x` **should now be 12345, the value of** `y` **should be 7894 and the value of** `ch` **should be 'H'.**

7) Create overloads for all 6 of the comparison operators (`< , > , <= , >= , == , !=` **). Each of these operations should test two objects of type MyInt and return an indication of true or false. You are testing the MyInt numbers for order and/or equality based on the usual meaning of order and equality for integer numbers.**

8) Create an overload version of the + operator to add two MyInt objects. The meaning of + is the usual meaning of addition on integers, and the function should return a single MyInt object representing the sum.

9) Create an overload version of the * operator to multiply two MyInt objects. The meaning of * is the usual meaning of multiplication on integers, and the function should return a single MyInt object representing the product. The function needs to work in a reasonable amount of time, even for large numbers. (So multiplying x * y by adding x to itself y times will take way too long for large numbers -- this will not be efficient enough).

10) General Requirements: as usual, no global variables other than constants, all member data should be private, use appropriate good programming practices as denoted on previous assignments. Since the only output involved with your class will be in the << overload, your output should match mine exactly when running test programs.

Extra Credit:

A) Create an overloaded version of the - operator to subtract two MyInt objects. The meaning of - is the usual meaning of subtraction on integers, with one exception. Since MyInt does not store negative numbers, any attempt to subtract a larger number from a smaller one should result in the answer 0. The function should return a single MyInt object representing the difference.

B) Create an overloaded / operator and a % operator for division of two MyInt objects. The usual meaning of integer division should apply (i.e. / gives the quotient and % gives the remainder). Both functions should return their result in a MyInt object.

Hints and Tips:

1) Storage: The suggested storage mentioned above is a dynamic array, in which each array slot represents one digit of the number. There are multiple ways to manage this. One method would be to

use an array of integers, another would be an array of characters. Both have advantages and disadvantages. An integer array would be easier for arithmetic calculations, but a character array would be easier for input purposes. You may choose your preference, but remember that an integer digit, like 6, and it's corresponding character representation, '6', are not stored as the same value! '6' is stored with its corresponding character code (usually ASCII).

In case you want to use them (not required), I have provided two functions for converting easily between single digit integers and their corresponding character codes. (These functions are only for use with single digits). You may use them if you like, as long as you cite the source in your comments. These are stand-alone functions (not members of any class). Copies of these functions are already placed in your starter `"myint.cpp"` file.

```
int C2I(char c)
// converts character into integer (returns -1 for error)
{
   if (c < '0' || c > '9')      return -1;      // error
   return (c - '0');                            // success
}

char I2C(int x)
// converts single digit integer into character (returns '\0'
for error)
{
   if (x < 0 || x > 9)          return '\0';    // error
   return (static_cast<char>(x) + '0');         // success
}
```

6) Input: For the >> operator overload, there are some issues to be careful about. You will not be able to just use the normal version of >> for integers, because it attempts to read consecutive digits, until a non-digit is encountered. The problem here is that we will be entering numbers that go beyond the capacity of a normal int (like 6 trillion). So, you will probably find it necessary to read one digit at a time (which means one byte, or character, at a time). Because of this, you may find the above conversion functions useful.

You already know of some istream member functions (like getline), and you have used many versions of the >> operator on basic types. One thing worth keeping in mind is that when the >> operator is used for built-in types, any leading white space is automatically ignored. However, there are a couple of other istream functions you might find useful, including the ones with prototyes:

```
int get();
int peek();
```

The `get()` function extracts and returns the next single character on the stream, even if it is a white space character, like the newline. This function does not skip white space (or any other characters), like the built-in >> functions do. The `peek()` function just looks at and returns the next character on the stream, but it does not extract it. This function is good for seeing what the next character is (i.e. like whether it is a number or not), without actually picking it up.

7) Comparison Overloads: Writing all 6 comparison overloads may sound like a big task, but it really isn't. Always remember that once a function is written, other functions can call it. For example, once you have written the details of comparing two numbers with the less-than operator, it should be very easy to define the > operator (by calling upon the < operator to do the work). You can't write them ALL this way, because you don't want to get stuck in an infinite loop (i.e. don't make both < and > call

each other -- you'll never get out!) But, you can certainly make the job easier by defining some of these in terms of others that are already written.

8) Addition operator: This is a more difficult function than it sounds like, in the MyInt class. However, the algorithm should be conceptually easy -- it can follow exactly the way you learned to add back in grade school! You'll probably find it most helpful to break this algorithm down to the grade school step-by-step level, performing an addition digit by digit. And don't forget that some additions can cause a carry!

9) Multiplication operator: This one will be a little more complex than the addition overload, but again, you should think back to the alrogithm you learned for multiplying numbers in grade school. The same algorithm can be applied here. Again, don't forget the carries!

Sample main program:

The sample main program that is provided can be found here: main.cpp

A couple of sample test runs can be viewed here:

- **Sample run 1**
- **Sample run 2**

Submit the following files (using the usual web submission procedure):

```
myint.h
myint.cpp
```