Fall 2006 EE380L Quiz3: NAME: _____

RULES: No internet searches. You may work on this examination for 2 hours (no longer). The time can be distributed over at most two contiguous intervals where each interval is at least 15 minutes long. When you elect to work on the examination, you must be in a private work environment with no outside assistance of any kind. Text books are permitted, as are all EE380L materials (including the EE380L web sites). You are not permitted to consult any "research papers" or published articles of any kind (only text books). You may use a compiler to write and test possible solutions. However, it is not necessary that your solution compile or run in order to receive a perfect score (be wary, as the compiler can be a time sink).

Addendum: You are permitted a grand total of four hours on this assignment. You can spend at most 2 hours actively working on your solution. In addition to the two hours of active work, you are permitted to "daydream" about the problem for at most an additional two hours. That means, if you are thinking about this problem on your ride home today, then that time spent in the car/bus will not count against the time you have available to spend sitting at your desk working on the exam. You can think about the problem while you're in the shower, lay awake at night thinking about it, etc. However, you must not abuse this license. After two hours of stewing over the problem, any additional time you spend thinking on the problem counts against the time allotted. Time spent in class today does not count against your four hours.

By attaching your name to this paper, you affirm the following statement:
I have fully complied with both the letter and the spirit of the academic honesty policies for this examination. I recognize that this examination is intended to be an evaluation of my ability to develop my own designs. Therefore, I recognize that searching for and/or reproducing designs, approaches or solutions developed by others (whether published or not) is in violation of the academic honesty rules. I attest that the following log accurately reflects the time I spent working on this examination:

Date time started time completed

_____

_____

I want to build a convenient infinite precision integer class, and I'm thinking about using some object-oriented programming in my design. My plan is to actually use three concrete classes and to have some kind of abstract base class. My three concrete classes will be SmallInteger, MediumInteger and LargeInteger. The SmallInteger will represent only those values that **int** can represent. The MediumInteger will represent only those values that **int64** (i.e, "long long") can represent. The LargeInteger will represent all possible integers (limited only by the available memory of the computer). The small and medium integer classes will just have one data member (a base type variable holding the value). The LargeInteger will use an array of **char** variables (not very efficient, but easy), with one **char** being used to store each decimal digit in the number. For this problem, we'll assume that all integers are non-negative.

I want you to implement the three concrete classes so that integers behave polymorphically. Specifically, I'd like to be able to declare a variable of type Integer that could refer to an instance of any one of the three concrete types. I'd like to be able to perform basic arithmetic using variables of type Integer. However, the arithmetic being performed should use the most appropriate form. For example, if the two Integers *x* and *y* refer to SmallIntegers, then the expression *x*+ *y* should involve only 32-bit arithmetic. If the Integers *x* and *y* were MediumIntegers, then *x* + *y* should be computed using 64-bit arithmetic. If *x* and *y* were LargeIntegers, then *x* + *y* should be computed using the appropriate decimal addition (i.e., in a loop).

Here's an example program that you should make sure works.

```
int main(void) {
  Integer x = 1; // x is a reference to a SmallInteger
  Integer y = 1;
  while (true) {
    y = y + 1;
    if (y == x) {
      cout << y << endl;
      x = x + x;
    }
  }
}
```

In this program, y should start out being represented with only 32 bits, then should be represented with 64 bits, and should (at least in theory) eventually be represented with a LargeInteger (although we probably don't have time to actually run the program that long). The same goes for x. Note that 1 + 1 should be performed using 32-bit arithmetic.

Note: I want object-oriented programming techniques used here. I do not expect to see any templates in your solution. You do not need to support arbitrary arithmetic for your Integer class and its descendants. You just need to support the functions used above (I need to create and assign to them, print them, compare them for equality and add them)

Some hints:

- You can get the expression y + 1 to work by having a (non-explicit) constructor for the Integer class that takes an **int** as an argument. It is desirable if this constructor is the only way to create Integer variables. It is also desirable if this type conversion is the only way to make y + 1 work (i.e., please don't write more than one operator+ function for the Integer class).
- I do expect the expressions x == y and x + y to work when *x* and *y* are different types of Integers. You can use typeid(), or dynamic_cast to determine if the two objects have the same type (assuming there's a virtual function somewhere).
- Note that since we don't have subtraction, then you can assume that if *x* and *y* are of different types, then *x* and *y* cannot have the same value (the only reason to create a MediumInteger is because the value is too large to fit into a SmallInteger). So, your operator== is pretty straightforward to write. Be object-oriented (no switch statements or complex case analysis).
- Since you only have to do addition, we only need to worry about promoting objects – i.e., the object produced as the result of x + y may not be the same type as either x or y, but it will definitely be at least as large as x and y. Since you can assume that all integers are non-negative, then a negative result for x + y means that you had overflow, and that you should promote the objects and try the arithmetic again.
- I want to grade your solution to type promotion separately from your solution to the rest of this quiz. Doing type promotion elegantly is a major part of the challenge (and will be a substantial part of the grade). Use as few if statements as possible, and be as object-oriented as you can. Please have a function (or functions, if you have to) dedicated to doing type promotion and nothing else. Call that function inside your operator+ function (and/or wherever you need to promote your types). Please do not embed any type promotion directly in your operator== or operator+ code, as that will only make your solution hard to grade.
- You must completely implement all classes necessary to run the program above.
- Your solution will be graded in part based on my subjective evaluation of how "object-oriented" it is. For example, your solution should easily permit the introduction of a 128-bit integer class in between MediumInteger and LargeInteger with a minimum of additional code (please don't write this fourth class, I'm using that as an example of a hypothetical future modification that might be made).
- Excessively object-oriented is good.