Fall 2010 EE 380L Final: NAME: _____

1. (25 pts) A standard rule of thumb for C++ goes as follows. "If a class has one or more virtual functions, then the class should also have a virtual destructor".
   a) (5 pts) Recall that in C++ a member template is a method which is a template function that has template arguments in addition to the class template arguments (if any). We used member templates in project 2 (the Valarray project) for our assignment operator since the argument on the right hand side of the assignment could have (almost) arbitrary type. In C++ member templates cannot be virtual functions. Why? In what way would virtual member templates make the implementation for C++ virtual functions difficult (or impossible)?

   b) (5 pts) Write a program that demonstrates why a virtual destructor is useful (or necessary). Your program should be as simple as possible, but should have a class for which when the destructor is virtual the program works correctly, but when the destructor is not virtual the program has some sort of defect. For full credit, your destructor should be fairly normal (i.e., please avoid writing destructors or functions that are obviously going to cause a crash if/when they're called). i.e., this rule of thumb exists because the problems with C++ programs and non-virtual destructors can be subtle. So, a perfect answer to this question will demonstrate a subtle mistake that can occur.

c) (5 pts) A related rule of thumb is that delete[] should never be used on a pointer to a type that has a virtual destructor. Write a program that demonstrates why delete[] causes an error. The classes used in your program should have a virtual destructor, and once again, for credit your program must demonstrate the subtle mistake that can occur (e.g., allocating a singleton object and then deleting it as if it's an array is not subtle and has nothing to do with object-oriented programming).

d) (5 pts) C++ places restrictions on the return type of virtual functions.
   i. Specifically, if doit is a virtual function defined in a class Base and overridden in a derived class Derived, and if Base::doit returns type T1 while Derived::doit returns type T2, what relationship must exist between T1 and T2? (chose 1 of the following)
      a. T1 must be the same as T2
      b. T1 must be a subtype of T2
      c. T1 must be a supertype of T2
      d. T2 cannot be void unless T1 is also void

e) (5 pts) Write a simple program that illustrates why C++ places the restriction you selected. Specifically, write a program that could not be statically type checked unless the rule you selected were enforced (be sure to explain (briefly) why you think your example demonstrates the problem).

2. (15 pts) One of the limitations of C++ is that the size of an object must be determined at compile time. This limitation forces C++ code to be recompiled even when relatively minor changes are made to class definitions.
   a) (7 pts) Which of the following changes will force a program to be recompiled (circle each correct response)
      i. Adding a virtual function to a class that did not previously have any virtual functions
      ii. Adding an additional virtual function to a class that had one or more virtual functions
      iii. Writing a definition for a virtual function that had previously been pure virtual (i.e., making an abstract function concrete, but still virtual)
      iv. Adding a static method to the class
      v. Adding a non-static method to the class
      vi. Adding a static data member to the class
      vii. Adding a non-static data member to the class.

   b) (8 pts) It would be useful if it were possible to know the size of the largest subtype of an object. In that way, we could allocate storage for an object of type T* and know that the amount allocated was always sufficient to hold an object of that type. Unfortunately, there's no way to solve this problem statically (at compile time) in C++, at least not with a compilation policy that allows files to be compiled into libraries. However, it can be solved at load time (well, at run time, but before main starts running) using a combination of programming conventions and C++ mechanics. Design such a scheme for a class Shape and all its subtypes. Sketch an implementation of Shape and at least two subtypes Triangle and Rectangle and write a function Shape* Shape::alloc(); that returns a pointer to a region of memory on the heap large enough to hold any subtype of shape.

3. Garbage collection
   a) (5 pts) Write a simple C++ program that generates an unbounded amount of garbage but (if run with a garbage collector) would use a bounded amount of heap storage.

   b) (5 pts) Write a simple C++ program that consumes an unbounded amount of heap storage but generates zero garbage.

4. In project 1 and in our Vector case study (for copy-on-write), we used reference counting to ensure our SequenceStrings cleaned up any memory they allocated. While the reference counting in project 1 was correct, in more general applications reference counting cannot prevent memory allocation errors. Let's assume a type Pointer<T> that provides all the functionality of a T* but also performs reference counting.
   a) (10 pts) Build the class Pointer<T> so that it maintains a reference count. You do not need to implement operator+, operator++, or operator== (no pointer arithmetic, array indexing or pointer comparison operations are required for this problem). Please implement operator*, a constructor that takes a T* object, a destructor and a copy constructor. You do not need to implement an assignment operator, as I can deduce your design from your copy constructor and destructor.

   b) (5 pts) Design a class Foo and create one or more Pointer<Foo> objects on the heap for which all the reference counts are correct, but for which there is some sort of memory error (either a reference count of zero and an object that is not garbage, or a garbage object with a reference count of non-zero).

5. (20 pts) For this problem, I'd like you to design a "personality" object. We'll keep things simple. There are three types of personalities, Grumpy, Happy and Indifferent. Personalities can greet other personalities and can say goodbye.

```
int main(void) {
   Personality p = makePersonality();
   Personality q = makePersonality();
   p.greet(q);
   q.greet(p);
   p.farewell();
   q.farewell();
}
```

   a) (10 pts) For part a, I want you to focus on the classes, the constructors and destructors, the makePersonality factory and the farewell method. We'll write the greet method in part b. For this problem, you are permitted to use an "if" (or a switch) statement in the makePersonality factory only (no other conditional branches are permitted – that includes the ? operator in C).

      Implement an object-oriented personality with the three types described above. Write makePersonality so that it randomly creates an personality of the appropriate type. Write constructors, destructors and any other functions you need so that the main program above will run and compile. For part (a), the greet method does not need to work. A Grumpy personality says farewell by outputting "Leave me alone" to the console. A Happy personality says farewell by outputting "Have a nice day!" to the console. An Indifferent personality says farewell by outputting "yeah, whatever" to the console.

(space for your answer to question 5, parts a and b)

b) (10 pts) For part b, I want you to write the greet method so that we obtain the following behavior. A Grumpy personality greets any Personality outputting “Go Away” to the console. Indifferent personalities greet other Personalities by outputting “yeah, whatever” to the console and Happy Personalities greet other Personalities by outputting “Hi there!” In addition, Personalities affect each others’ moods. When a Happy Personality greets an Indifferent Personality, the Indifferent personality becomes Happy. When a Grumpy Personality greets another Grumpy Personality, the Grumpy Personality becomes Indifferent. When a Grumpy Personality greets an Indifferent Personality, the Indifferent Personality becomes Grumpy, and when a Grumpy Personality greets a Happy Personality, the Happy Personality becomes Indifferent (as you can see, happiness is fleeting). Any other combination of moods results in no change after the greeting.To make it specific, the method a.greet(b) will influence object b, possibly causing b to change type (depending on the type of a and b).

   i. Indifferent personalities never change the mood of the personality they greet
   ii. Happy personalities make Indifferent personalities Happy, but have no effect on Grumpy or Happy personalities.
   iii. Grumpy personalities always have an effect. Grumpy Personalities make both Happy and Grumpy personalities become Indifferent, and make Indifferent personalities become Grumpy.

6. (15 pts) Design a class (or classes) to support Number. For this problem, we'll only consider division (we won't worry about addition or other arithmetic operations). Also, we'll our consideration to two types of Numbers: Integers and Doubles. An Integer represents the Number using 64-bit integer (you may use the type "long" to represent the C++ native type for this problem). A Double represents the Number using IEEE 64-bit floating point (you may use the type "double" to represent the C++ native type for this problem). When arithmetic is performed using two Double numbers, or on one Double and one Integer, the arithmetic should be performed using double-precision floating point arithmetic. When arithmetic is performed using two Integers, then integer arithmetic should be used. However, If the result of the division is not an integer (i.e., if there is a remainder), then the object returned by the division operator should be a Double. Consider the following example. In the program the variable z is initially assigned an Integer (since 30 / 6 is an integer). On the second division operation, however, z is assigned a Double (since 5/6 is not an integer). Implement the Number class (and any other classes that you need) so that the following program compiles and runs as described. You will be evaluated in part based on the elegance of your design and the performance of your implementation. You may use templates and/or virtual functions in any combination you desire.

```
int main(void) {
   Number x,y,z;
   x = 30; // x is an Integer
   y = 6; // so is y
   z = x / y; // z is assigned an Integer (5)
   z = z / y; // z is assigned a Double (approx: 0.1666666)
}
```

(space for your answer to Q6)