Fall 2009 EE 380L Midterm: NAME: _____

1. (15 pts) Declare and write an STL-style function (i.e., consistent with the C++ standard library conventions) for computing the median value of a generic collection.
   - You cannot change the values (or their sequence) in the collection
   - Your solution can have at most O(N) space complexity (i.e., it's OK to create a copy of the collection, but don't go crazy).
   - You may invoke std::sort() if you'd like. In fact, you probably should use std::sort in your solution.
   - You can (and should) assume that the value_type for the collection has an operator< defined.
   - You should not assume that the collection supports random access (although if you're making a copy of the collection, you can certainly create a copy that has random access).
   - Be sure to define the return type and parameter type(s) correctly for your function. Name your function "median". I'm expecting it to be a template function with exactly one template argument (the function may have more than one argument, the template should have only one). If the collection has an even number of elements, then you can return either of the two "middle" values.

2. (15 pts) NOTE: C++ has a "run time type ID" system built into the language, but in this question, I want you to design and implement a run time type ID from scratch. You are allowed templates, virtual functions, inheritance, etc, but you are prohibited from using "dynamic_cast<>" or other elements of the C++ built-in RTTI system.

   As the head of a large design project, you have mandated that every class will be a subclass of a class TypeIDBase. For this question, I need you to write the TypeIDBase class and two functions (same and is<T>). To help you complete this task, I'm also going to insist that every class will have a static member function *TypeID*. Here is an example class Foo showing you how classes will be defined for this system. Note that assignID is a (public) static member of the TypeIDBase class.

```
int TypeIDBase::assignID(void) {
  static int nextID = 1;
  return nextID++;
}

class Foo : public TypeIDBase {
  static int myTypeID;
public:
  static int TypeID(void) { return myTypeID; }
};
int Foo::myTypeID = assignID();
```

Given this head start, please write the following three things ON THE NEXT PAGE

- Write the base type TypeIDBase
- Write a function SameType(x, y) that returns true of x and y have the same run time typeID.
- Write a template function *is*<*T*>. This template function should be something I can invoke on any subtype of TypeIDBase, and it should return true if that subtype is exactly T, and should return false otherwise. For example I should be able to write

```
if (is<ChocolateCake>(x))
```

  and the expression would evaluate to **true** if an only if the RUN TIME type of x is exactly ChocolateCake (note that the declared type of x might be a supertype, e.g., Cake&).

3. (15 pts) I now want to design a base type called Comparable. All subclasses of Comparable will have a "<" operator that can be used to compare that object. Unlike TypeIDBase Not all classes will be subclasses of Comparable (since for some types, it does not make sense to try and compare their values, e.g., the C++ types cout and cerr – is "cout < cerr"?). However, many classes can be defined to be subclasses of Comparable.
   a. Design a base type for Comparable that ensures that any instance (i.e., any actual object) that is a subtypeof Comparable must have an operator< function. HINTS: be careful with the type of the argument for this function – also, you'll want the method to be "const".

   PLEASE read parts b and c to this question (on the next page) before solving part a

Create three (concrete) classes, each a subtype of Comparable called “Rock”, “Paper” and “Sizzors”. Each of these classes will need to support the run time type ID system from question 1 (if you’d like, you can make Comparable a subclass of TypeIDBase, or you can use multiple inheritance).

- An instance of Rock is less than an instance of Paper but not less than any other type.
- An instance of Paper is less than an instance of Sizzors, but not less than any other type
- An instance of Sizzors is less than an instance of Rock, but not less than any other type.

b. Ensure your solution to (a) and (b) permits the following code to work (only the three conditions labeled “YES” evaluate to true and print anything).

```
int main(void) {
      Rock r;
      Sizzors s;
      Paper p;
      if (r < s) { cout << "Rock < Sizzors\n"; } // NO
      if (s < r) { cout << "Sizzors < Rock\n"; } // YES
      if (r < p) { cout << "Rock < Paper\n"; } // YES
      if (p < r) { cout << "Paper < Rock\n"; } // NO
      if (s < p) { cout << "Sizzors < Paper\n"; } // NO
      if (p < s) { cout << "Paper < Sizzors\n"; } // YES
}
```

4. (25) Assume I now have a std::vector<Comparable*> collection, and I wish to sort it. Note that since the collection is a collection of pointers, the "built-in less than" operator will not result in proper sorting. There are two reasonable approaches to make the sort operator work. Design a solution for both approaches. NOTE: my collection is called "myCollection" and it's a std::vector<Comparable*>.
   - (10 pts) Create a custom function object for comparing Comparable* pointers (the function object would de-reference the pointers and compare the objects that are pointed to). Use this function object as the 3rd argument to std::sort to sort myCollection (PLEASE show me how you would call std::sort, don't just write a function object).
   - (10 pts) Create iterator adaptors, that convert the value_type of vector<Comparable*>::iterator from "Comparable*" to "Comparable&". After defining these adaptors, use the 2-argument std::sort function to sort myCollection. PLEASE call std::sort.
   - (5 pts) Which, if either, of these two approaches (part a, or part b) do you think will result in the faster sorting implementation? Please, limit your consideration to the likelihood that comparison operations will be inlined during compiler optimization. Explain (no credit for random guesses).

5. (15 pts) One of the obstacles for garbage collection is “finding the pointers”. A potential remedy for this problem is to mandate that all programmers on the design team use “Smart Pointers” and then design the Smart Pointers so that they register themselves in some sort of global list. Design a smart pointer template class called Pointer<T> with the following characteristics:
   - Pointer<T> is a subclass of PointerBase. Write both PointerBase and Pointer<T>
   - PointerBase has a static data member std::vector<PointerBase*> allPtrs, and the address of every Pointer<T> is contained in this vector.
   - Additionally, every Pointer<T> has a data member called “myOffset” such that “PointerBase::allPtrs<myOffset> == this” for any Pointer<T>. (note that myOffset can be inherited from PointerBase if you’d like).
   - HINT: to keep the correct set of Pointers in the vector allPtrs, add new pointers to the end (via push_back). When a pointer, x, goes out of scope, replace allPtrs[x.myOffset] with the last value in the vector, then invoke pop_back.
   - Pointer<T> does not support pointer arithmetic. However, dereference and equality (i.e, op==) are supported and have the normal meaning. Also, a Pointer<T> can be assigned the value of a T* and/or assigned the value of another Pointer<T>.

6. Two short answer questions on garbage collection based on Pointer<T>.
   a. (5 pts) Having a list of "all pointers" is nice, but to be Complete, a garbage collector needs to be able to distinguish root pointers for other pointers. Assume you had to add a method "bool isRoot() const" how would you write this function? Explain how (you don't need to write it).

   b. If we used the Pointer<T> exclusively in our application (there are no variables declared of type T*), do you believe it is possible to achieve the following goals (indicate "yes" or "no" and explain). Note, for the purpose of this question, I want you to ignore the memory used to implement the data structure PointerBase::allPtrs and consider only the other memory in the system. (please assume the application is single-threaded).

- (4 pts) Could the garbage collector be used to implement a mark-and-sweep collector that is both "safe" and "complete"? (i.e., collects all garbage and only garbage).

- (3 pts) Could the garbage collector be used to implement a safe and complete copy-collector that defragments the heap as objects are moved (and does not break the program by moving objects)?

- (3 pts) I've tried to avoid arrays in the discussion up to this point. If I wanted to write a vectorGC<T> template using Pointer<T>, what additional pointer operation(s) should I included for Pointer<T> (if any). Please explain any issues you think we might have.