Fall 2010 EE 380L Midterm: NAME: _____

1. (15 pts) Declare and write an STL-style function called “reverse” that satisfies the following requirements
   a. reverse operates upon a container (using the C++ conventions for containers).
   b. reverse must reorder the elements in the container such that the sequence of the elements is reversed from their original sequence. E.g., a container with the sequence of values { 1, 3, 5, 2, 4, 6 } would have the values { 6, 4, 2, 5, 3, 1 } after reverse.
   c. For full credit, you should not assume the container supports random access (although any other reasonable assumption is fine).

2. (15 pts) I want to improve the performance of my heap in a C++ program. As part of my strategy I'm going to restrict most memory allocations to be a power of 2 in size. So, for example, if I had a 12-byte object, I would like the heap to allocate 16-bytes to hold this object. Consider the following template class that I intend to use as a base class for my objects. Note that the purpose of Allocator is to provide a customized "new" function that allocates a block of memory large enough to hold an object of type T, and also that has a size constrained to be a power of 2.

```
template <typename T> struct Allocator {
      void* operator new(size_t) const { // parameter is not used
            size_t sz = 1;
            while (sz < sizeof(T)) { sz *= 2; }
            return ::operator new(sz);
      }
};
```

I'm concerned about the run-time overhead of the loop in this class. Rewrite the Allocator class so that the value for *sz* is calculated entirely at compile time. You may use as many template classes, template functions or ordinary classes and functions as you need. Note that *sz* must be equal to the smallest power of 2 that is not smaller than sizeof(T).

3. (10pts) The Allocator<T> base class is intended to be used as follows:

```
class Foo : public Allocator<Foo> {
      int x;
      int y;
      int z;
};
Foo* p = new Foo; // Allocator<Foo>::new is used
```

a. What sort of problems occur if we create a derived class that inherits Foo? (e.g.,)

```
class Bar : public Foo {
      int x;
      int y;
};
Bar* q = new Bar; // ???
```

b. Can we solve the problem by using multiple inheritance, e.g.: (explain)?

```
class Bar : public Foo, Allocator<Bar> {
      int x;
      int y;
};
```

4. (5 pts) There's a perception among some C++ experts that recursive functions are faster when the functions are declared as static member functions (instead of ordinary member functions). Explain why this might be true (in, for example, the following).

```
class Tree {
      Node* p;
public:
      (static??) int height(Node* n) {
            if (n == 0) { return 0; }
            int lh = height(n->left);
            int rh = height(n->right);
            if (lh > rh) { return lh + 1; }
            else { return rh + 1; }
      }
};
```

5. (10 pts) Using any feature, function or template from the standard library, write a function isRandom(p) that, assuming p is some type of iterator, returns true if p is a random access iterator. Your function must work for pointer types and all iterators that conform to the standard library.

6. (15 pts) In functional programming the term “to curry” is sometimes used to describe a technique where a binary function (with two arguments) is converted into a unary function (with one argument) by specifying one of the two arguments. In the following C++ example, *curry* is invoked on *std::multiplies* and specifies that the first argument to multiplies should be the value 5. The result of invoking *curry* is a function object (which is, in the example below, invoked on the value 2). Using techniques similar to what you’ve learned for expression templates, write a *curry* function in C++. You may create as many template classes or template functions as you require.

```
curry(std::multiplies(), 5)(2); // evaluates to 5 * 2 -- i.e., 10
```

7. (15 pts) Continuing on with the functional programming example, write an operator+ that performs function composition. Specifically, the expression (f + g) when f and g are unary function objects should produce a unary function that first invokes f and then invokes g – i.e., (f + g)(x) should be equivalent to g(f(x)). Assuming a correct implementation of curry, your operator+ should cause the following expression to evaluate to 28. Again, create as many template classes or template objects as you require.

```
(curry(std::multiplies(), 5) + curry(std::plus(), 3))(5); // 5 * 5 + 3
```

8. Three short answer questions (5 pts each)
   a. Assume I have a program with class Base and class Derived. Derived uses single inheritance and has Base as its base class. The code for Base is in a file Base.cpp (and Base.h). All of the other code (including the code for Derived) is in main.cpp. Assuming I recompile Base.cpp but DO NOT recompile main.cpp, which of the following changes to class Base will permit the program to continue working without errors. **Circle all correct responses:**

      i. A new data member is added to class Base

      ii. A new member function is added to class Base

      iii. A new static data member is added to class Base

      iv. A new static member function is added to class Base

      v. None of the above

   b. Please provide type definitions for the pointers p and q in the following (HINT: p and q are different types) such that the program contains a serious error that is likely to result in corrupting the heap. Please provide a definition of class Foo and any other classes (hint) that your solution references indirectly.

```
p = new Foo;
q = p;
delete q;
```

   c. For most types the statements "++x;" and "x += 1;" are equivalent. In fact, outside of C++, I never use the ++ operator since I do not like the semantics of "pre increment" vs "post increment". Unfortunately, in C++ for some types, ++x and x += 1 have very different behavior. What is an example of a type in C++ where I should expect ++x and x += 1 to behave differently (hint: getting an error in one case while not getting an error in the other would count as "behaving differently). Explain your answer (very briefly).