Fall 2009 EE 380L Final: NAME:________________________________

1. (15 pts) Write an object_traits<T> template (well, collection of templates) so that object_traits<T> provides the following members (e.g., object_traits<T>::is_pointer)
   a) is_basetype – true if T is an int, a float or a double. False otherwise. Please don't support all the basetypes, as that's just too tedious.
   b) is_pointer – true if T is any pointer type (i.e., T is S* for some type S). False otherwise
   c) is_classtype – true if T is not a pointer and not a basetype (e.g., given our simplifications, it'd be true if T were char). In a full implementation, is_classtype would be true if and only if T were a class or a struct.
   d) deref_type – if T is not a pointer, then deref_type is T. If T is a pointer then deref_type is the type that T points to.

2. (15 pts) For this problem, I want you to write a Universal Copy Factory (UCF) class. The goal of this problem is to write a static member function UCF::copy<T>(T x) that will, for any type of object x, produce a reasonable copy. Here are the rules
   a) If T is a base type (e.g., int) or if T is an object, then the copy function should invoke the copy constructor for T (using x) to create the copy.
   b) If T is a pointer to an ordinary object, then the copy function should invoke new and the copy constructor for *x to create a new pointer that points to a copy of x. You may use iterator_traits<T> if that is helpful.
   c) If T is a pointer to an object that has at least one virtual function, then you should invoke x->copy() and return the object produced by this function. You can assume that x->copy is virtual and the return type is the same as the type of x.

   You can write more than one copy method (i.e., you can use overloading) in your solution if you'd like. Please use object_traits (from Q1) in your solution. Note: when you get to the point of asking x, "do you have a virtual function?", you'll need to be careful not to even ask the question unless is_classtype is true. Asking the question (which usually involves creating a subclass) is not possible unless x is itself a class.

3. (25 pts) This is a rather long question. My hope is that if you understand expression templates (and if you did Project 3), you'll be able to figure out the bits and pieces that I provide and complete the problem.

   First of all, let me explain the goal. The goal is to use expression templates that allow us to substitute a different "Algebra" for arithmetic operations. We'll limit this problem to scalar integers, and we'll use a class (actually, a typedef) called **Int** to work with. Basically, Int will be a wrapper around an **int** value for which we have defined operator+ and operator* overloads. We'll also have a template type called Algebra that allows us to redefine the mathematical meaning of multiplication and addition. In the example below, I use three algebras. In the first cout statement, I use the default algebra (in which "+" means normal addition and "*" means normal multiplication). In the second cout statement I explicitly create an algebra which has precisely the same behavior as the default (again, "+" means addition, "*" means multiplication). However, in the third cout statement, I replace the meaning of "+" to be "maximum" and I replace "*" to mean addition.

```
template <typename Add, typename Mul>
class Algebra {
public:
      typedef Add Add;
      typedef Mul Mul;
};

class Max {
public:
      int operator()(int x, int y) const {
            if (y > x ) { return y; }
            else { return x; }
      }
};

class Plus {
public:
      int operator()(int x, int y) const {
            return x + y;
      }
};

class Times {
public:
      int operator()(int x, int y) const {
            return x * y;
      }
};

int main(void) {
      Int x = 10;
      Int z = 2;
      cout << x + x * z + z << endl; // 32 is printed
      cout << (x + x * z + z).with(Algebra<Plus, Times>())<<endl; // 32
      cout << (x + x * z + z).with(Algebra<Max, Plus>()) << endl; // 12
}
```

OK, so that's the goal. I want you to solve this problem using a specific design, so I'm going to provide you pieces of the solution in the hopes that you will be able to make sense of it.

a) Your first task is to build up the expression templates. You'll be ignoring the "with" functionality in part a. PLEASE, you must use expression templates (even though they provide no value). Once again, for part a) you'll be completing the expression templates so that they perform the default algebra (normal multiplication and addition), but only do so when the assignment operator is called (or cout is used).

   The following design relies on a *getVal* method existing in the "proxy" objects that are created by the expression templates. In fact, the Integer<R> class expects to inherit getVal from type R. Your proxies will have to provide this getVal function (i.e., getVal replaces the [] operator used on Valarrays).

```
template <typename R=void>
class Integer : public R {
public:
   Integer(const R& base) : R(base) {}
};

template <typename R>
ostream& operator<<(ostream& out, const Integer<R>& x) {
   return out << x.getVal();
}

template <>
class Integer<void> {
public:
   Integer(int x = 0) { val = x; }
   int val;
   int getVal(void) const { return val; }

   template <typename R>
   void operator=(const Integer<R>& x) { val = x.getVal(); }
};

typedef Integer<> Int;
```

   (15 pts) Write operator+ and operator* to complete the expression templates (on the next page).

   DO NOT SOLVE BOTH (a) and (b) at this time. Only do what is NECESSARY to solve part (a) as described (note that does require expression templates and that arithmetic is delayed until assignment or ostream::operator<< is invoked). **Do not provide support for with or getWith!**

Now, in part b) we'll finish up the example by writing the *with* method. I want your with method to use the proxy object provided below. This proxy is intended to be used as R in an Integer<R> object created by a (x + y *z).with(…) statement. The With proxy requires an Algebra (type A) and an expression (type T). We expect type T to be an object returned by operator+ or operator*. I've also provided a revised Integer<R> template for you that demonstrates how I expect the *with* method to be implemented.

```
template <typename A, typename T>
class With {
      T val;
public:
      With(const T& x) : val(x) {}
      int getVal(void) const { return val.getWith<A>(); }

      template <typename T>
      int getWith(void) const { return getVal(); }
};

template <typename R=void>
class Integer : public R {
public:
      Integer(const R& base) : R(base) {}

      template <typename A>
      Integer<With<A, Integer<R> > > with(const A&) {
            With<A, Integer<R> > t(*this);
            return Integer<With<A,Integer<R> > >(t);
      }
};
```

As you can see, the design requires a second method in the proxies (getWith) in addition to getVal. The getWith method is a member template that is instantiated with an Algebra type. The With proxy will ignore the algebra argument provided to getWith. That's done so that (z * (x + y).with(Algebra1())).with(Algebra2()) will work. Algebra1 will be used for the + operator and Algebra2 will be used for * in this example. Note that in this example, the root of the expression tree is a With object (using A = Algebra2). That tree passes Algebra2 into the multiplication proxy when it calls getWith<Algebra2>() on the multiplication proxy. The multiplication proxy, in turn, invokes getWith<Algebra2>() on each of its two children (or it least it should), and then invokes the required Mul operator form Algebra2 to combine the values from the left and right children. Well, you'll have to figure out the rest.

To complete this solution, you'll need to modify your proxy objects from part (a) so that they provide a getWith method. You may need to revise your getVal methods as well. Note that you'll have to modify the Integer<void> specialization so that it also provides a getWith. However, you shouldn't have to modify your operator+ or operator* functions. Please rewrite those three classes below so that the original "main" program works as described. Make as few changes as possible. Place your template classes on the next page.

(10 pts) Write your new proxy objects and your new Integer<void> specialization on this page

4. (15 pts) Write a wrapper that uses techniques similar to “speculative inlining” for an hypothetical base class ShapeBase (shown below). Note that there are two draw functions in the wrapper (and similarly, there should be two move methods). The drawSpec method should work correctly regardless of the actual type of shape. However, if the actual shape is a Triangle, then drawSpec<Triangle> should be faster than the ordinary draw. Your solution can use dynamic_cast<T*> or typeid(). If you don’t recall how typeid works (and I don’t), then please recall that dynamic_cast<T*> will return null if the argument is not actually T*. Please be sure to implement draw, drawSpec, move and moveSpec in your wrapper, and please make sure that drawSpec and moveSpec will work for any subtype of shape (including those that may not have been written yet).

```
class ShapeBase {
public:
   virtual ShapeBase* copy(void) const = 0;
   virtual void draw(void) const = 0;
   virtual void move(int x, int y) const = 0;
};

void doit(Shape s) {
   s.draw(); // normal dynamic dispatch
   s.drawSpec<Triangle>(); // always works, but inlined if s is a Triangle
}
```

5. (30 pts) Short answer questions:
   a. Let T be an abstract class. Let x be an instance of T. The virtual function table pointer in the object x should be:
      i. A null pointer
      ii. A pointer that points to a table of null pointers
      iii. A pointer that points to a table in which at least one pointer is null (some of the pointers in the table may not be null)
      iv. Object? What object?

   b. Assume we have an application that uses the following function:
```
void doit(T x) {
  x.foo();
}
```
      What if we changed the function to pass the argument by constant reference:
```
void doit(const T& x) {
  x.foo();
}
```
      If the program compiles with the new version of doit, but behaves differently than it had before, what can we conclude about type T and method foo? For each of the following **write "yes", "no" or "maybe"** depending on whether the statement is definitely true, definitely false, or possibly true or false.
      i. foo() is virtual
      ii. foo() is pure virtual
      iii. T has a superclass
      iv. T has a subclass
      v. T is abstract

   c. Assume that reference counting is used as part of the implementation of a garbage collector.
      i. Is it possible that an object will have a reference count of zero and **not** be garbage?
      ii. Is it possible that an object will be garbage and have a reference count larger than zero?

d. I have a class T, and I'm trying to decide if I should make T::~T virtual. I never call T::~T directly (i.e., I don't do p->~T(), or x.~T()). I do not use dynamic_cast or other form of run-time type identification. Which of the following must be true in order for the virtual destructor to affect my program?

   i. T must have at least one subtype

   ii. T must not be abstract (i.e., T must be "concrete")

   iii. At least one object of some subtype of T must be allocated on the heap

   iv. All of the above

   v. Exactly two of the above (circle which two)

   vi. None of the above

e. Assume an inheritance hierarchy with type Base that has *F* virtual functions, *D* different derived types, assume that each derived type overrides all of the virtual functions.

   i. What is the time complexity to invoke one of the virtual functions for a pointer p of type Base*?

   ii. What is the space complexity of the system if there are *O* objects total (equally divided among the types) and all objects have the same data members.

f. Declare the variables p, q and x to "win" the following puzzle

```
p = &x;
q = &x;
if ((unsigned) p == ((unsigned) q) {
    cout << "You Lose\n";
} else {
    cout << "You Win\n";
}
```