Spring 2013 EE 380L Final: NAME: _____

1. (28 pts) Recall that a wrapper can make working with inheritance hierarchies more convenient. In our examples from class, actual objects were stored on the heap (allocated with new) and the wrapper contained a pointer to the object. Consider the following abstract base class. To keep things simple, I've only included one virtual function for the Base class (it is pure virtual in the Base class).
   a) Create a copy constructor for the wrapper class. If you need to make any changes to the Base class (or either Derived1 or Derived2), please make those changes.
   b) I want to be able to invoke doit on the wrapper using . syntax (i.e., I do not want to use * or ->) Provide a doit method for the wrapper
   c) Write a destructor for your Wrapper. Note that the Base class is abstract and has no obvious need for a destructor. I believe it actually does need a destructor. Write one that is appropriate for the class.
   d) Write an assignment operator so that I can assign a Derived1 or Derived2 object (not a pointer) to a Wrapper.

   The main program shown below requires all four parts (a-d) to be implemented correctly so that it prints, "the answer is 42" with no errors or memory leaks.

```
void fun(Wrap x) { // x is pass-by-value (i.e copy)
   x.doit();
}

int main(void) {
   Wrap w;
   w = Derived1("the answer is");
   fun(w);
   w = Derived2(42);
   fun(w);
}

class Base {
public:
   virtual void doit(void) const = 0;









}; // more code on the next page
```

```
class Derived1 : public Base {
   string x;
public:
   Derived2(string x) { this->x = x; }
   virtual void doit(void) { cout << x; }









};

class Derived2 : public Base {
   int x;
public:
   Derived1(int x) { this->x = x; }
   virtual void doit(void) { cout << x << "\n"; }









};




class Wrap {
   Base* ptr; // ptr will be nullptr or will point to the heap
public:
   Wrap(void) { ptr = nullptr; }









};
```

2. (12 pts) For this example, I'm using the same Base, Derived1 and Derived2 examples from question 1 which do not have multiple inheritance. In the code below, I have two constructors and two assignment operators. The doit() function works correctly when a PolyBase is created using either constructor (when constructed with a string argument, the doit method prints a string, when constructed with an integer argument, the doit method prints an integer. However, if I attempt to change the type of the PolyBase using an assignment operator (e.g., construct the object with Derived1 but assign the object a Derived2), then the doit method produces garbage output.
   a) Why do the assignment operators not work? Specifically, what aspect of the object needs to be changed that isn't changed by the assignment operator? (write your answer with 10 words or fewer).

   b) In the space below, rewrite one of the assignment operators so that the object properly changes type. After assigning a PolyBase a Derived2 object, doit should print an int (even if the object was constructed originally with a string).
   Hint: If you forget the syntax for placement new, it's
   new (p) T(args);
   to construct an object of type T at the address pointed to by p.

```
class PolyBase {
   union {
      Derived1 d1;
      Derived2 d2;
   };
public:
   PolyBase(const string& s) : d1(s) {}
   PolyBase(int x) : d2(x) {}
   void doit(void) const {
      ((Base*) &d1).doit();
   }

   PolyBase& operator=(const Derived1& d1) {
      this->d1 = d1;
      return *this;
   }
   PolyBase& operator=(const Derived2& d2) {
      this->d2 = d2;
      return *this;
   }
}; // re-write the assignment operator below so that it works correctly (changes type)
PolyBase& PolyBase::operator=(const Derived2& x) {
```

3. (24 pts) The purpose of this question is to ensure that you understood object-oriented programming methods. Your solution to this problem, cannot contain any "if" statements, and must not use "?" operator or use any arrays.

   As you are undoubtedly aware, human children come in two basic designs, "Boy" and "Girl". I'd like to have a general type Child which could be either Boy or Girl. All children will play. In this problem I want to consider children that play either by themselves, or children that play with another child. To represent playing, a Child object should have a member function (i.e., a method) called "play". There should be two overloads for this function; play(void) and play(Child*). The play methods must provide the following behavior.

   When a Boy plays by himself, he plays with action figures (print "Action")
   When a Girl plays by herself, she plays with dolls (print "Dolls")
   When a Boy plays with another Boy, they play rough (print "Roughhouse")
   When a Girl plays with another Girl, they play nice (print "Tea party")
   When a Boy and Girl play together, they play tag (print "Cooties")

a) Define the types Child, Boy and Girl.
b) Create a factory method that has a 50% chance of creating and returning a Boy and a 50% chance of creating and returning a Girl.
c) Write a main program that creates two Child objects Alex and Ryan. In your program Alex and Ryan should be initialized by calling your factory method. They could be either boys or girls. Write your program so that it contains three method invocations: Alex plays alone (Alex invokes play(void)), Ryan plays alone (Ryan invokes play(void)) and Alex and Ryan play together (Alex invokes play(Child) where the Child argument is Ryan).
d) Use a wrapper only if it makes your solution shorter or simpler than the code without a wrapper. Do not have any memory leaks.

Additional space on the next page

This page intentionally left blank (so you can answer question 3 here)

4. (10 pts) I want to create a random die object (e.g., a six-sided die) that can be combined with other dice objects to create random number generators. For this problem, I want you to use the library function rand() as the primitive method to create a random number. Assume rand() returns a random uint32_t . I want you to build a type called Die and operators for addition (i.e. operator+) so that I can achieve the following. Implement the Die class (or classes) and the operator+ function(s) you need. A good solution to this problem allows every function to be inlined by the compiler.

```
uint32_t roll; // the result of throwing a die or dice
Die basic_die = Die(6);
roll = basic_die(); // roll is random between 0 and 5
auto die6 = die_mod_6 + 1;
roll = die6(); // roll is random between 1 and 6
auto dice = die6 + die6;
roll = dice(); // roll is random between 2 and 12
```

5. (16 pts) Consider the following struct (based on the std::pair<> template from the C++ standard library.)
   a) I've written a template function called "get". This template function uses compile-time static analysis to select between the first or second element of the Pair struct depending on whether the integer argument to get<> is a zero (select first argument) or a one (select second argument). The get template relies on a template class called Getter. Write Getter. See the get template function and the main function below for information.

```
template <typename T1, typename T2>
struct Pair {
   T1 first;
   T2 second;
   Pair(const T1& f, const T2& s) : first(f),second(s) {}
};

template <uint32_t idx, typename T1, typename T2>
typename Getter<idx, T1, T2>::Type
get(const Pair<T1, T2>& p) {
   Getter<idx, T1, T2> getter;
   return getter.get(p);
}
int main(void) {
   Pair<int, double> p(42, 3.14);
   cout << get<0>(p); // prints integer 42
   cout << get<1>(p); // prints double 3.14
}
```

b) Extend your Getter struct so that the example will now work with Pairs of Pairs. Your solution should work with Pair<T1, Pair<T2, T3>> (as shown below), and ideally should also work with Pair<T1, Pair<T2, Pair<T3, T4>>> etc., etc. See the main function below for an illustration of how get<> should work with the type Pair<T1, Pair<T2, T3>>. Please note, you cannot change the get function. You can only change (or extend) the Getter class you used in part a.

```
int main(void) {
    Pair<int, double> p1(42, 3.14);
    Pair<const char*, Pair<int, double>> p2("hello", p1);
    cout << get<0>(p2); // prints string "hello"
    cout << get<1>(p2); // prints integer 42;
    cout << get<2>(p2); // prints double 3.14
}
```

6. (10 pts) Write a template meta function that lets me declare a type to be the larger of type T1 and T2 (use sizeof). I want to be able to use the meta function in at least the following way:

```
using BiggerType = Bigger<T1, T2>::Type;
```