Fall 2005 EE 380L Midterm: NAME:________________________________

RULES: No internet searches of any kind. You may work on this examination for 4.5 hours (no longer). When you elect to work on the examination, you must be in a private work environment with no outside assistance of any kind. Text books are permitted, as are all EE380L materials (including the EE380L web sites). You are not permitted to consult any "research papers" or published articles of any kind (only text books).

By attaching your name to this paper, you affirm the following statement:
I have fully complied with both the letter and the spirit of the academic honesty policies for this examination. I recognize that this examination is intended to be a challenging exercise, but that it is fundamentally an evaluation of my ability to develop my own designs. Therefore, I recognize that searching for and/or reproducing designs, approaches or solutions developed by others (whether published or not) is in violation of the academic honesty rules. I attest that the following log accurately reflects the time I spent working on this examination:

Date	time started	time completed

1. (15 pts) I want you to write a program that plays the famous Rock/Paper/Scissors game. In this game two players independently select one *Option* which can be either *Rock*, *Paper*, or *Scissors*. The winner of the game is determined as follows. If both players select the same type of *Option*, it is a draw. Otherwise, *Rock* beats *Scissors*, *Paper* beats *Rock*, and *Scissors* beats *Paper*. Design and implement an inheritance hierarchy where *Option, Rock, Scissors* and *Paper* are all classes. Write a function *play* which takes two *Options*, *x* and *y* respectively, and returns 0 if the game is a draw, -1 if *x* beats *y* and +1 if *y* beats *x*. You may not use any **if** statements, switch statements or make use of the *?* operator of C/C++. Your solution must work with the following code fagment (don't forget to define the four types, *Option*, *Rock*, *Scissors*, and *Paper*). Your solution can contain memory leaks (i.e., you're encouraged to assume the existence of a garbage collector).

```
int main(void) {
   while (true) {
      int res = play(Player1Select(), Player2Select());
      if (res == 0) { cout << "it's a draw\n"; }
      else if (res < 0) {
        cout << "Player 1 wins!\n";
        break;
      } else {
        cout << "Player 2 wins!\n";
        break;
      }
   }
```

Your solution to this problem requires complete definitions of the four Types and the play function. A neatly written solution (hand-written) takes less than a full page. Please understand that unnecessarily complicated responses reflect a lack of understanding of the material and will be given a low score even if the program "works". You should place your response to question one on the following page

2. One of the nuisances of generic programming is needing to remember the order in which the template arguments must be specified. For example, a hash table data structure might have template arguments for the key type (Key), the value type (Value), the hash function (Hash) and a function for comparing two keys (KeyCompare). Solve this problem. This is a multi-part question, and is somewhat complicated. The parts are intended to be independent of each other. So, although part a is the most difficult, you should be able to receive full credit for parts b and c even if you do not figure out part a.
   a. (10 pts) We're going to pass template parameters to our HashTable class using a parameter list (which happens to be a linked list). Note that the parameter list is a list of *types*, not a list of objects. The ParamList struct is shown below. Note that we're using a *void-terminated* singly-linked list of types.

      ```
      template <typename Head, typename Tail>
      struct ParamList {};
      ```

      A Param in the ParamList is represented using the following struct

      ```
      template <typename ParamName, typename ParamValue>
      struct Param {};
      ```

      Here's what we want in a nutshell. We will use a family of four types to represent the "names" of the parameters. Then, we'll construct a parameter list using the Param and ParamList structs as follows.

```
struct Key {}; // the "name" for the Key parameter
struct Value {}; // the "name" for the Value parameter, etc.
struct Hash {};
struct KeyCompare {};

struct MyHashFun { // an example of a type for the Hash param
  int operator()(int) { return 1; } // a lousy hash function, no?
  static string typeName(void) { return "MyHashFun"; }
};

struct MyKeyCompare { // an example for the KeyCompare param
  bool operator()(int x, int y) { return y < x; }
  static string typeName(void) { return "MyKeyCompare"; }
};

typedef ParamList<Param<Value, double>,
            ParamList<Param<Key, int>, void> > ParamList1;
typedef ParamList<Param<Key, double>,
            ParamList<Param<Value, int>, void> > ParamList2;
typedef ParamList<Param<Value, double>,
            ParamList<Param<KeyCompare, MyKeyCompare>,
                ParamList<Param<Key, int>,
                    ParamList<Param<Hash, MyHashFun>,
                        void> > > > ParamList3;
```

You'll note that the order that the params are listed varies in our three parameter lists. In ParamList1, the *Value* is specified first, and the *Key* second. In ParamList2, the *Key* comes first and the *Value* second. Note also that neither ParamList1 nor ParamList2 provide all four of the possible parameters. ParamList3 does specify all four parameters. Please keep in mind that the types *Key*, *Value, Hash,* and *KeyCompare* are used as names. They are empty structs, and have no purpose other than to identify a specific parameter within a parameter list.

When we're finally done, our template for a HashTable object will have just one template argument, a ParamList constructed as above. We should be able to instantiate a HashTable<ParamList1>, a HashTable<ParamList2> or a HashTable<ParamList3>.

For part a) of this problem, you need to write the template metaprogram FindParam that extracts one of the parameters from the param list. Your metaprogram must not involve any execution time (it must "run" entirely at compile time) and must work with the following program

```
template <typename T>
struct WhatType {
  static string doit(void) { return T::typeName(); }
};
template <>
struct WhatType<int> {
  static string doit(void) { return "int"; }
};
template <>
struct WhatType<double> {
  static string doit(void) { return "double"; }
};
template<>
struct WhatType<void> {
  static string doit(void) { return "void"; }
};

template <typename Params>
void showList(void) {
  typedef typename FindParam<Params, Key>::ANSWER KeyParam;
  typedef typename FindParam<Params, Value>::ANSWER ValueParam;
  typedef typename FindParam<Params, Hash>::ANSWER HashParam;
  typedef typename FindParam<Params, KeyCompare>::ANSWER CompareParam;
  cout << " Key is: " << WhatType<KeyParam>::doit() ;
  cout << " Value is: " << WhatType<ValueParam>::doit();
  cout << " Hash is: " << WhatType<HashParam>::doit();
  cout << " KeyCompare is: " << WhatType<CompareParam>::doit();
}


int main(void) {
  cout << "ParamList1"; showList<ParamList1>(); cout << endl;

  cout << "ParamList2"; showList<ParamList2>(); cout << endl;
  cout << "ParamList3"; showList<ParamList3>(); cout << endl;
}
```

Using the ParamList and Param structs defined above, and the following IfThenElse struct, write the FindParam metaprogram. Please be sure your solution fits in the space provided (should not be more than 50 lines long).

```
template <bool P, typename T, typename F> struct IfThenElse {};

template <typename T, typename F>
struct IfThenElse<true, T, F> { typedef T ANSWER; };

template <typename T, typename F>
struct IfThenElse<false, T, F> { typedef F ANSWER; };
```

b. (5 pts) For our HashTable example, assume that the Hash and KeyCompare arguments are optional, and that there are default template DefaultHash<T> and DefaultCompare<T> that should be used if the ParamList does not contain arguments for Hash and Compare respectively. Write a template struct HashArgs that defines four nested types *MyKey, MyValue¸MyHash,* and *MyCompare* based on the ParamList argument to the HashArgs template. Use the FindParam and IfThenElse metaprograms from part a so that if the parameter's value is void, then the default argument will be used. You can assume that Key and Value are never void.

c. (5 pts) Finally assume that there is a hash table template __HashTable (note the leading underscore) that takes four arguments in the following order, a Key, a Value, a KeyComparison function and a HashFunction. Assume that this template is already written, well tested, etc. In this part write a HashTable template that takes one template parameter (a ParamList) and then instantiates __HashTable using the correct arguments from part b. The __HashTable should be the base class for HashTable, and we should inherit all the functionality of this class into the HashTable.

3. (10 pts) Assume we have a *LinkedList* class that has a template argument T (the element type). We want to implement the iter_swap function as efficiently as possible. Note that with a *LinkedList*, it is possible to swap two elements in either of two ways. We can swap the values themselves, or we can updating the pointers to the link cells so that the cells appear in the swapped order. Updating the pointers requires updating eight pointers (in a doubly-linked list) in general. Write the iter_swap function so that it invokes *pointer_swap* if the size of the element type is at least 16 bytes, and invokes value_swap if the size of the element is smaller than 16 bytes. For full credit,
   - solve this problem with no if statements, and
   - ensure that you iter_swap function (which should not be a member function or member template) can be used only with *LinkedList* iterators (and not with iterators from other classes).
   - (Note: I do not need you to write the pointer_swap or value_swap functions. Just assume that these functions each take two *LinkedList* iterators as arguments. Also note that I'm not interested in error checking for the possibility that one or both of the iterators are outside the bounds of the *LinkedList*)

4. (10 pts) I want to have a self-morphing *Sequence* class. The *Sequence* class has an operator[] and insert and remove methods (ignore iterators for this problem). If insert() or remove() are invoked on the middle of the sequence, the *Sequence* should morph into a Linked List. If they are invoked on the end of the sequence, the *Sequence* should morph into a Vector. For purposes of this question, assume that the position argument to *insert* is an integer (not an iterator), and that the "middle of the sequence" is any position other than the first position (position 0) and the last position (position size() – 1). Do not write push_front, push_back, etc, just insert and remove. Your linked list implementation should be derived from the std::list<T> class. Your vector implementation should be derived from the std::vector<T> class. Your operator[] cannot contain any **if** statements, and your solution should be as "object oriented" as is reasonably possible.

5. Write a duplicate-skipping iterator, Skipper<T>. The template argument, T, will be an iterator type. The constructor to a Skipper<T> takes two objects (iterator) of type T (b and e). If a Skipper is dereferenced, then it returns the object that it's underlying iterator points to (i.e., dereference the T stored inside the Skipper). Initially, this underlying iterator is the "b" argument to the Skipper constructor. If the Skipper is incremented, then the underlying iterator is incremented as many times as necessary until either it "points to" an object with a different value, or until the value "e" is reached (e is the second argument passed to the Skipper constructor).
   a. (5 pts) Write the Skipper<T> template class

   b. (5 pts) Write a binary subtraction (operator-) for Skipper<T> that returns the number of positions between the underlying iterators in the two Skippers. You can assume that T is a random-access iterator.

c. (5 pts) Write a template function rle that takes four arguments, the first two arguments, b and e, are random-access iterators of type T. The next argument is an output iterator (you can assume forward iterator) into which a sequence of **int**s can be written. The fourth argument is an output iterator (again, you can assume forward iterator) into which a sequence of objects can be written (iterator_traits<T>::value_type objects to be more specific). The behavior of rle should be to run-length-encode the sequence described by [b, e). The length of each "run" will be written into the third argument. The object for each run should be written into the fourth argument.

d. (5 pts) Using the following back inserter, and your RLE function (and an infinite loop), write a main function that prints the famous sequence: 1, 11, 21, 1211, 111221, … . The easiest way to do this is to use two vectors, x and y. Initially x contains just "1". Each iteration of the loop, compute the RLE of x and store both parts of the resulting sequence in y. Print x, set x equal to y and repeat.

```
class BackInserter {
  vector<int>& v;
public:
  BackInserter(vector<int>& _v) : v(_v) {}
  BackInserter& operator++(void) { return *this; }
  BackInserter& operator*(void) { return *this; }
  int operator=(int x) { v.push_back(x); return x; }
};
```

6. (15 pts) Design an array-like class called *Array* that provides the following functionality:

- The class holds a sequence of objects that are subtypes of *Base*. Part of this question is to decide what virtual methods will be required for type *Base*. So, you'll need to write a simple class *Base* that demonstrates what functions are required.
- Has a constructor that creates an empty sequence (size() == 0).
- Has method *push_back* that accepts a *const Base&* as the argument and appends that argument to the end of the sequence. The *push_back* method should run in amortized O(1) time.
- Has method *get* that returns an *const Base&*. The argument to *get* is an **int** indicating the position in the sequence of the object to be returned. *Get* must run in O(log N) time when there are N elements in the sequence.
- Has a *const_iterator* nested type and methods *begin* and *end* (as per the C++ standard library conventions). The *const_iterator* should return a *const Base&* when dereferenced. The const_iterator should be a forward iterator and should provide constant time increment, dereference and equivalence methods.
- Provides spatial locality. Object i+1 should be stored as close as possible to object i (i.e., their addresses should be as close as possible).
- You cannot assume that all objects are the same type, just that each object is some subtype of Base. You can define any pure-virtual methods in Base that you think are required.

  Hints: You do not need to write copy constructors or assignment operators. They would ordinarily be required, but they'll just clutter up your response to this examination question. I suggest you start by writing a method Array::offset(int k) that returns the offset (in bytes) of the kth object in the sequence. Keep in mind that you may need to use "placement new" to solve this problem correctly, and you'll almost certainly require at least one virtual method for the *Base* class.

7. (10 pts) Consider the difficulty in managing “plug ins” to an application. The specific problem we have is that at run time, new classes will be added to our application. We need to learn what methods these objects have, and then invoke these methods. This problem shows up as follows, I have an object pointed to by a pointer of type void* (I know nothing else about the object), and I have a string that contains the name of the objects true type. Assume I need to determine if the object has a doit() method (not static) and if so, invoke the doit() method on the object. I’m considering trying to solve this problem in one of two ways. Both techniques rely on a combination of programming conventions and C++ language mechanisms. Comment on the difficulties I’m likely to have in fleshing out both of the following approaches (one or both may ultimately prove impossible, but that’s for you to decide).

   a. My first plan is to require that all developers create a specialization of the Class template for each class *Foo* that they write. When they make the specialization of *Class<Foo>* they will include a map<string, Foo::*> that I can use at run time to examine what members are contained in the Foo class. To use a “plug in” that is actually of type *Foo* all I need to do is to instantiate the Class template for the *Foo* type and then I can access all the information I need through the specialization that the Foo author created (i.e., the *Class* template is essentially a traits class that can be used to discover information about any type). I’ll look up “doit” in the map, get the Foo::* pointer-to-member-function back from the map, and then apply the function to the object.

b. My second plan is not use templates at all, but rather to have an abstract class *Class*, and to require that all developers write a subclass *ClassFoo* for each class *Foo* that they write. All of the subclasses of *Class* will be placed into a large hash table that I can access with a ”factory method”. Essentially, if I want to find the *ClassFoo* object at runtime, I just call the factory method with the string “Foo”, which looks up the string in the hash table and returns the *ClassFoo* object to me. C*lass* has a number of virtual methods. One of the methods is *memberLookup* which takes a string as an argument and returns a Foo::* pointer-to-member-function as the return value. So, once I’ve looked up the *ClassFoo* object using the factory method, I can call the *memberLookup* function with the argument “doit” to find the doit function, and (if it exists) invoke doit on the object.