Autumn CS193D Final Examination

This is a closed-book, closed-note exam. Your exam should have 14 pages, and there are three questions totaling 50 points. You have three hours to complete the final exam, but the exam is designed so that three hours should feel like an infinite amount of time relative to what's being asked. Unless the exam says otherwise, details of syntax are not important so long as it is clear what you mean. Comments are not required but may help you receive partial credit if your intentions are correct but your code is wrong. Unless otherwise noted, you are always free to write helper/support routines or embedded classes as needed. If you are running out of ideas, pseudo-code sketches may earn some partial credit.

Good luck! And take your time!

```
leland
```
username: _____

Last Name: _____

First Name: _____

I accept the letter and spirit of the honor code. I have neither given nor received aid on this exam.

(signed) _____

		Score	Grader
1. Gossip	(10)	_____	_____
2. Stanford Students	(20)	_____	_____
3. Randomized Iterators	(20)	_____	_____
Total	(50)	_____	

Problem 1: All About Talk Shows

Inspect the following nonsense code where all implementations have been `inlined` for convenience (assume all methods are `public` and all instance methods are `const`):

```
class montel {
   static void jenny() { cout << "montel::jenny" << endl; }
   virtual void oprah() = 0;
   virtual void ricki() { cout << "montel::ricki" << endl; jenny(); }
   void rosie() { cout << "montel::rosie" << endl; ricki(); }
   virtual void sally() = 0;
};

class jerry : public montel {
   static void jenny() { cout << "jerry::jenny" << endl; }
   virtual void oprah() { cout << "jerry::oprah" << endl;}
   virtual void ricki() { cout << "jerry::ricki" << endl; sally(); }
};

class conan : public jerry {
   void rosie() { cout << "conan::rosie" << endl; jenny(); }
   virtual void sally() { cout << "conan::sally" << endl; rosie(); }
};

class dave : public montel {
   static void jenny() { cout << "dave::jenny" << endl; }
   virtual void oprah() { cout << "dave::oprah" << endl; }
   virtual void ricki() { cout << "dave::ricki" << endl;
                              montel::ricki(); }
   virtual void sally() { cout << "dave::sally" << endl; }
};
```

The `gossip` function is designed to take a `const montel` reference as its only parameter:

```
static void gossip(const montel& host)
{
   host.rosie();
}
```

What are the possible types that `host` may be referencing at runtime? For each possibility, trace through a call to `gossip` and present its output. Please use the next page for your answers.

Problem 2: Stanford Students

Students are classified as either Workaholics, PartyAnimals, and Introverts. Each student has a list of friends (who are other students) and tracks two quantities: the amount of work a student has to do for his classes and his sleep deficit, both expressed as an integer number of hours. The list of friends is maintained using the STL `vector` class you've come to know and love; note that a student's friends are not necessarily of the same type: Workaholics are friends with PartyAnimals and Introverts as well as other Workholics.

We're interested in what happens when you try to invite a student to a party by sending an `invite(Student *host)` message to a Student object. The host parameter is another Student who is extending the party invitation.

First off, all Students have a threshold on the maximum sleep deficit. For Workaholics, the threshold is 6, for PartyAnimals it is 12, for Introverts it is 4. If a Student's sleep deficit is above his threshold, he won't go to the party. Also if a student has too much work to do (over 10 hours), they won't go the party—except of course for PartyAnimals who never skip a party because of any work they have to do. Introverted folks don't often go to parties, even if they're all caught up on their work and sleep. However, on every 10th invitation they receive, if they meet the usual sleep and work criteria and the host of the party is one of their friends, they will go to the party.

If a Student decides to accept the invitation, then he goes to the party. Going to the party increases a Student's sleep deficit by 2 hours since they stay out late. When a Workaholic comes home from a party, he stays up even later to do some work, so his sleep deficit goes up another 3 hours, but his work goes down by 3 hours. When a PartyAnimal goes to a party, he invites all of his friends to come with him (making it clear who the real host of the part is.)

We recommend thinking through the entire design before making any decisions.

Just like you did for cacti, draw a little tree modeling the student class hierarchy. List the instance variables and methods (including their types/prototypes) for each class. This drawing will serve as your .h file, so include all the necessary type information and method headers. You will not need to mention or define constructors or other setup code at all—just assume that a miracle occurs and your objects are all set up at run time. You do not need to deal with `virtual`, `private`, `public`, or `protected` specifiers.

Provide implementation code for the `invite` methods of all `Student`s along with any necessary helper methods. You have this and the next two pages to do this. Make sure you are happy with your class hierarchy before you do this. (Don't worry about any infinite recursion resulting from friends inviting friends inviting friends inviting friends and any invitation cycles that result.)

Problem 3: Defining your own Randomized Iterator

For this problem, you are going to design and implement your own iterator class—specifically, the `RandomizedIterator`. Traditional iterators normally reference an element within a container, and in response to `operator++`, advance to the next element in sequence. Your `RandomizedIterator`, given a range of elements, will traverse through the range in what appears to be a random order. Rather than advance to the next element in the sequence, `operator++` will update a `RandomizedIterator` to reference any one of the elements not previously referenced. Additional applications of `operator++` will continue to access previously unseen elements. If all elements have been referenced, `operator++` will advance to the past-the-end state.

For example, the following program would print out the first 10 prime numbers in increasing order:

```
int main()
{
   int primes[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
   int *start = primes;
   int *end = primes + 10;
   while (start != end) {
      cout << *start << endl;
      ++start;
   }
}
```

Should we want to produce a permutation of the first 10 primes, we could leverage the behavior of our `RandomizedIterator` to write the following:

```
int main()
{
   int primes[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
   RandomizedIterator<int *> start(primes, primes + 10);
   RandomizedIterator<int *> end;
   while (start != end) {                 // calls operator!=
      cout << *start << endl;             // calls operator*
      ++start;                            // calls prefix operator++
   }
}
```

The output of the above program would vary from run to run, but each would list the 10 primes in any one of the 10! possible orderings.

The RandomizedIterator need not traverse over every single element in the range. The following program prints out 6 lottery numbers from a range of 1 to 40:

```
int main()
{
    vector<int> countingSet(40);

    for (int i = 0; i < 40; i++)
        countingSet[i] = i + 1;

    RandomizedIterator<vector<int>::iterator> generator(countingSet.begin(),
                                                        countingSet.end());
    cout << "Today's lottery numbers are:"
    for (int j = 0; j < 6; i++) {
        cout << '\t' << *generator++ << endl;
    }
}
```
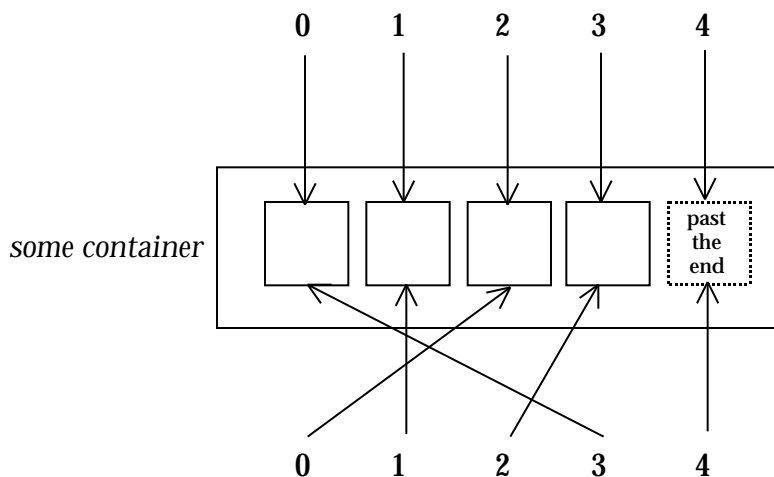
Note that `generator` sees the full spectrum of possibilities, any subset of 6 numbers could be generated in any order whatsoever, all with equal probability.

The `start` and `end` iterators passed to `RandomizedIterator`'s constructor would access elements in this order.

The order in which the elements of a `RandomizedIterator` **might** access the elements, provided the same `start` and `end` are passed to its constructor.

Presented here is the public interface for the `RandomizedIterator` template class you are expected to design (by specifying all the `private` implementation data you need to support these operations) and implement (within the function and method stubs provided over the next few pages.) You are responsible for `everything`, and the strength of your design is just as important as the correctness of your code. <u>Bottom line</u>: you must ensure that your iterator behaves as any built-in iterator would.

Your implementation will benefit from the use of the `random_shuffle` algorithm, which takes a range of elements, bracketed by two iterators (just as `for_each` does, for example), and shuffles the elements, leaving them in one of the n! different possible permutations.

```cpp
template <class BidirectionalIterator>
void random_shuffle(BidirectionalIterator start, BidirectionalIterator end);

template <class BidirectionalIterator>
class RandomizedIterator {

    typedef iterator_traits<BidirectionalIterator>::value_type
            value_type;

    friend bool operator==(const RandomizedIterator<BidirectionalIterator>& lhs,
                           const RandomizedIterator<BidirectionalIterator>& rhs);
    friend bool operator!=(const RandomizedIterator<BidirectionalIterator>& lhs,
                           const RandomizedIterator<BidirectionalIterator>& rhs);

    public:
        RandomizedIterator(BidirectionalIterator start,
                           BidirectionalIterator end);
        RandomizedIterator();

        const value_type& operator*() const;
        value_type& operator*();
        const RandomizedIterator& operator++();
        const RandomizedIterator operator++(int);

    private:



    };
```

specify all of your fields here, using whatever is necessary

Provide implementations for all of the functions and methods, making sure that the suite of operations that result make your `RandomizedIterator` class compatible with all of the STL algorithms we've discussed in lecture, including `for_each` and `accumulate`. In short, your resultant class should be a proper iterator.

```
/*
 * Constructor: RandomizedIterator(start, end)
 * -------------------------------------------
 * Constructs an instance of a RandomizedIterator, where the
 * RandomizedIterator knows to traverse over elements in the range
 * [start, end).  The RandomizedIterator, rather than iterating over
 * the elements sequentially, seemingly iterates over the elements
 * of the range in a random order, making a point to never revisit
 * a previously visited element, and visit every element once.  When
 * all elements have been accessed, the RandomizedIterator will, in the
 * operator== sense, be equal to the past-the-end RandomizedIterator
 * returned by the zero-argument constructor.
 */

template <class BidirectionalIterator>
RandomizedIterator<BidirectionalIterator>::
RandomizedIterator(BidirectionalIterator start, BidirectionalIterator end)
{
```

```
/*
 * Constructor: RandomizedIterator()
 * --------------------------------
 * Constructs an instance of the part-the-end RandomizedIterator,
 * an iterator which represents the situation where no remaining
 * elements may be legally accessed.
 */

template <class BidirectionalIterator>
RandomizedIterator<BidirectionalIterator>::RandomizedIterator()
{
```

```
/*
 * Dereference operators: const and non-const forms
 * ------------------------------------------------
 * Returns a reference (either const or non-const) to the
 * client element that the RandomizedIterator is currently
 * pointing to.  Note that a BidirectionIterator isn't what
 * gets returned, but instead a reference to one of the elements
 * pointed to by one of the original BidirectionalIterators.
 */

template <class BidirectionalIterator>
const typename RandomizedIterator<BidirectionalIterator>::value_type&
RandomizedIterator<BidirectionalIterator>::operator*() const
{
```

```
template <class BidirectionalIterator>
typename RandomizedIterator<BidirectionalIterator>::value_type&
RandomizedIterator<BidirectionalIterator>::operator*()
{
```

```
/*
 * Increment operators: both pre and post-fix forms
 * ------------------------------------------------
 * Advances the RandomizedIterator to some other element
 * of the original sequence, that has not been visited
 * previously.  If the RandomizedIterator has previously
 * references all other elements in the sequence, then
 * it should advance to a state such that future calls
 * to operator++ are ignored and such that it compares
 * (in an operator== sense) equally to any past-the-end
 * iterator returned by the zero-arg constructor.
 */

template <class BidirectionalIterator>
const RandomizedIterator<BidirectionalIterator>&
RandomizedIterator<BidirectionalIterator>::operator++()
{
```

```
template <class BidirectionalIterator>
const RandomizedIterator<BidirectionalIterator>
RandomizedIterator<BidirectionalIterator>::operator++(int)
{
```