Standard Libraries

- In C++ there is the *Standard Library*, formerly known as the *Standard Template Library* or STA
  - Emphasizes generic programming (using templates)
  - Write a sorting routine, the implementation depends on
    - Elements being comparable
    - Elements being assignable

*We should be able to write a routine not specific to int, string or any other type, but to a generic type that supports being comparable/assignable*

- In C++ a templated function/class is a code-factory, generates code specific to a type at compile time
  - Arguably hard to use and unsafe
STL concepts

- **Container**: stores objects, supports iteration over the objects
  - Containers may be accessible in different orders
  - Containers may support adding/removing elements
  - e.g., vector, map, set, dequeue, list, multiset, multimap

- **Iterator**: interface between container and algorithm
  - Point to objects and move through a range of objects
  - Many kinds: input, forward, random access, bidirectional
  - Syntax is pointer like, analogous to (low-level) arrays

- **Algorithms**
  - find, count, copy, sort, shuffle, reverse, …
**Iterator specifics**

- An iterator is dereferenceable, like a pointer
  - \*it is the object an iterator points to

- An iterator accesses half-open ranges, [first..last), it can have a value of last, but then not dereferenceable
  - Analogous to built-in arrays as we’ll see, one past end is ok

- An iterator can be incremented to move through its range
  - Past-the-end iterators not incrementable

```cpp
vector<int> v; for(int k=0; k < 23; k++) v.push_back(k);
vector<int>::iterator it = v.begin();
while (it != v.end()) { cout << *v << endl; v++; }
```
STL components: see stlcount.cpp

```cpp
 ifstream input(argv[1]);
 int matchCount =
   count(istream_iterator<string>(input),
        istream_iterator<string>(),
       word);
 cout << word << " : " << matchCount << endl;
```

- **Questions about code**
  - What does implementation of `count( .. )` look like?
  - What changes if we count occurrences in vector or map?
  - What properties of iterator are required? Different from sorting via iterators?
  - There are other algorithms, see SGI site for details
From STL to Java

- **In STL an iterator is a concept, there are refinements**
  - Input, output, forward, bidirectional, random access
    - A forward iterator is an input iterator and an output iterator
    - The iterator may be immutable (or const)---read only

  - Refinements not implemented by inheritance, but by design, contract, and subsequently implementation
    - What happens if you try to implement an STL iterator?

- **In Java *Iterator* is an interface (like a base class), similar to Tapestry iterators**
  - Collection(s) are required to have iterators, these are used in some operations like max, min, construct vector, ...
  - Related to STL as algorithm glue, but very different
Wordlines.java, print strings, line #'s

```java
public void print()
{
    Iterator allKeys = myMap.keySet().iterator(); // words

    while (allKeys.hasNext()) {
        String key = (String) allKeys.next();
        System.out.print(key + "\t");
        Iterator lines = ((Set) myMap.get(key)).iterator();
        while (lines.hasNext()) {
            System.out.print((Integer) lines.next() + " ");
        }
        System.out.println();
    }
}
```

- Differences between Java and Tapestry in practice?
  - Must store current element since `next()` does two things
Design patterns

“... describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

Christopher Alexander, quoted in GOF

- **Name**
  - good name provides a handle for the pattern, builds vocabulary

- **Problem**
  - when pattern is applicable, context, criteria to be met, design goals

- **Solution**
  - design, collaborations, responsibilities, and relationships

- **Forces and Consequences**
  - trade-offs, problems, results from applying pattern: help in evaluating applicability
Iterator as Pattern

- (GOF) Provides access to elements of aggregate object sequentially without exposing aggregate’s representation
  - Support multiple traversals
  - Supply uniform interface for different aggregates: this is *polymorphic iteration* (see C++ and Java)

- **Solution:** tightly coupled classes for storing and iterating
  - Aggregate sometimes creates iterator (Factory pattern)
  - Iterator knows about aggregate, maintains state

- **Forces and consequences**
  - Who controls iteration (internal iterator, apply in MultiSet)?
  - Who defines traversal method?
  - Robust in face of insertions and deletions?