

Standard Libraries

- In C++ there is the *Standard Library*, formerly known as the *Standard Template Library* or *STL*
 - 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, deque, 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, analagous 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

```
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++;}
```

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?

STL overview

- **STL implements generic programming in C++**
 - Container classes, e.g., vector, stack, deque, set, map
 - Algorithms, e.g., search, sort, find, unique, match, ...
 - Iterators: pointers to beginning and one past the end
 - Function objects: less, greater, comparators
- **Algorithms and containers decoupled, connected by iterators**
 - Why is decoupling good?
 - Extensible: create new algorithms, new containers, new iterators, etc.
 - Syntax of iterators reflects array/pointer origins, an array can be used as an iterator

STL examples: wordlines.cpp

- How does an iterator work?
 - Start at beginning, iterate until end: use [first..last) interval
 - Pointer syntax to access element and make progress

```
vector<int> v; // push elements
vector<int>::iterator first = v.begin();
vector<int>::iterator last  = v.end();
while (first < last) {
    cout << *first << endl;
    ++first;
}
```

- Will the while loop work with an array/pointer?
- In practice, iterators aren't always explicitly defined, but passed as arguments to other STL functions

Review: what's a map

- **Maps keys to values**
 - Insert key/value pair
 - Extract value given a key
 - Overloads operator[], so can be used like an array
 - Find returns an iterator that refers to item
 - Returns iterator::end if item is not in map
- **In Java, there's an inheritance hierarchy of AbstractMap, TreeMap, HashMap**
 - STL uses red-black tree, guaranteed $O(\log n)$
 - STL unofficially has a hash_map, see SGI website
 - STL also has multimap