More STL

- map provides a associative container
- multimaps map to multiple objects
- STL algorithms work on many things
- Predicates make algorithms more flexible

std::map

- Supports association
- A map has a key type and a value type
- Fast access to a value given a key.
- Uses trees, so fast means $O(\log(\text{num items in map}))$
- Must be able to compare keys (operator$<$)
- Also supports iteration in order of keys
std::map construction

• Declarations look like this:

map<string, int> mymap;

• Other constructors for copying, or construction from a range of pairs (e.g. a vector).

vector< pair < string, int > > myvect;
...

map<string, int> mymap2(myvect.begin(), myvect.end());

Adding to a map

• Use operator[] to access items

map<string, int> agemap;
string name = “fred”;
agemap[name] = 45;
int age = agemap[name];
ageMap[name]++;  // now fred --> 46

• Note: use of operator[] will put items in if they aren’t there!
• Generally, this is very useful, occasionally a pain.

map<string, int> visitCount;
string name = “fred”;
visitCount[name]++;  // Works!!
Looking at a map

• Use count(key) to see if key is in the map
• For a map, count() will always return 0 or 1

```cpp
map<string, int> agemap;
string name = "fred";

// Always succeeds, might return 0
int age = agemap[name];

if ( agemap.count(name) == 0 )
{
    // name not in map
}
```

map iterators

• Use iterators to specify items or ranges

```cpp
typedef map<string, int> Map;
typedef Map::const_iterator MapIter;

Map amap;

// Print out map in alphabetical order
for( MapIter mi = amap.begin(); mi != amap.end(); mi++ )
{
    cout << mi->first << " " << mi->second << endl;
}
```
map::find

• Use find to get an iterator for a specific key

```cpp
typedef map<string, int> Map;
typedef Map::const_iterator MapIter;

Map amap;

MapIter result = amap.find( "Fred" );
if ( result != amap.end() )
{
    // Print out map in alpha. order, starting with Fred
    for( MapIter mi = result; mi != amap.end(); mi++ )
    {
        cout << mi->first << " " << mi->second << endl;
    }
}
```

map::insert

• Use insert to put in a new item only if it isn’t there

```cpp
typedef map<string, int> Map;
typedef Map::const_iterator MapIter;

Map amap;

pair< MapIter, bool> result =
    amap.insert( make_pair( "Fred", 45 ) );

if ( ! result.second)
{
    // Fred was already in the map, result.first points
    // there now.
}
```
map::erase

• Use erase to remove an item

typedef map<string, int> Map;
typedef Map::const_iterator MapIter;

Map amap;
...
int numErased = amap.erase("Fred");
if ( numErased == 1 )
{
	// Fred was in the map, now he’s not.
}

std::multimap

• Similar to map, but allows multiple values for one key
• Doesn’t provide []
• insert() returns an iterator, since it can’t fail
• Still supports iteration in order of keys…
• …no order assumed on values
• Now count() can return any size
• Has find(), but this is not as useful as equal_range
std::multimap::equal_range

typedef multimap<string, int> MMap;
typedef MMap::const_iterator MMI;

MMap amap;
pair<MMI,MMI> result = amap.equal_range( "Fred" );

// Print out all values for "Fred"
for( MMI mi = result.first; mi != result.second; mi++ )
{
    cout << mi->first << " " << mi->second << endl;
}

Other STL containers

• priority_queue
• set
• multiset
• valarray (optimized for numeric use, supports slicing)
• bitset
• hash_map (not part of the Standard, but widely avail.)
STL Algorithms

- Containers are nice, but we want more
- Want to find, remove, sort, e.g.
- Could have these functions for each container
- But STL takes another path...
- ... making heavy use of iterators
- What’s an iterator?

Iterators

- If it looks like one and acts like one...
- ... it is one!
- Iterators have flavors like:
  - Output, input, forward, bidirectional, random access
  - A regular C pointer to an array is a random access iterator!
## Iterator flavors

<table>
<thead>
<tr>
<th></th>
<th>Out</th>
<th>In</th>
<th>For</th>
<th>Bi</th>
<th>Ran</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>*p</td>
<td>*p</td>
<td>*p</td>
<td>*p</td>
<td>*p</td>
</tr>
<tr>
<td>Access</td>
<td>- &gt;</td>
<td>- &gt;</td>
<td>- &gt;</td>
<td>- &gt;</td>
<td>- &gt;</td>
</tr>
<tr>
<td>Write</td>
<td>*p=</td>
<td>*p=</td>
<td>*p=</td>
<td>*p=</td>
<td>*p=</td>
</tr>
<tr>
<td>Iteration</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Comparison</td>
<td>==</td>
<td>==</td>
<td>==</td>
<td>!=</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

### std::sort

- Sort the range between two iterators
- Iterators must be random access
- Items pointed to must have operator<

```cpp
template<class RanIt>
void sort(RanIt first, RanIt last);

template<class RanIt, class Pred>
void sort(RanIt first, RanIt last, Pred pr);
```
std::sort

#include< algorithm>
int arr[100];
vector v1;
...
std::sort( arr, arr+100 );
std::sort( v1.begin(), v1.end() );

Predicates

• A function returning a bool is a *predicate*
• An object which overloads operator() to return bool is also a predicate!
• Some algorithms take predicates and do useful things with them.
std::count_if (part 1)

```cpp
#include< algorithm>
bool less_than_7( int v )
{
    return v < 7;
}
vector<int> v1;
int numLess =
    std::count_if(v1.begin(),
                  v1.end(), less_than_7 );
```

std::count_if (part 2)

```cpp
class less_than
{
    public:
        less_than( int t ) : m_thresh( t ){}
        bool operator()(int v){ return v < m_thresh; }
    private:
        int m_thresh;
};

vector<int> v1;
int x = 14;
...
int numLess = std::count_if(v1.begin(), v1.end(),
                             less_than(x ));
```
std::count_if (part 3)

#include<functional>
using std::less
using std::bind2nd

vector<int> v1;
int x = 14;
...
int numLess = std::count_if(v1.begin(), v1.end(),
  bind2nd(less<int>(),x));

---

Boost libraries

• Boost.org
• An amazing development effort
• Good and free (!)
• Currently 86 libraries in 20 categories
• Several libraries accepted for next C++ standard
TR1 library additions

• Reference wrappers
• Smart pointers
  • Resource Acquisition Is Initialization (RAII)
• Function objects:
  • bind generalizes bind1st
• Regexps
• Containers:
  Hash tables
  Tuples
  Fixed-size arrays
• Metaprogramming
• Math and numerics:
  Random numbers
  Special functions

Boost library categories

• String and text processing
  Regexps
• Containers
  Graph
• Iterators
• Algorithms
  foreach
• Function Objects and higher-order programming
• Generic Programming
• Template Metaprogramming
• Preprocessor Metaprogramming
• Concurrent Programming
  Threads
• Math and numerics
  Random numbers

• Correctness and testing
• Data structures
• Image processing
• Input/Output
• Inter-language support
• Memory
  Smart pointers
• Parsing
• Programming Interfaces
• Miscellaneous
• Broken compiler workarounds
Advice

• Try out STL classes and algorithms…
• …you’ll probably get hooked.
• All implementations are now in pretty good shape.
• If your compiler provides the STL and works with the Boost libraries, it can handle almost anything (esp. WRT templates)
• But wait, there’s more!! (streams)