Admin

- Assign 1 due & Assign 2 out
  - No rest for the weary :-)
- Today's topics
  - CS106 class library: Map, Set, Iterator, function callbacks
- Reading
  - Handout 14 (today), Reader ch. 4 (next)
- Got strep?
  - My office hours today canceled

More containers!

- Sequential containers
  - Vector, Grid, Stack, Queue
  - Store/retrieve elements based on sequence of insert/update
  - Doesn't examine elements, just store/retrieve them
- Associative containers
  - Map, Set
  - Not based on sequence, instead on relationships
  - Efficient, smart access
  - Do examine/compare elements to store/retrieve efficiently

Map class

- Collection of key-value pairs
  - Associate a key with a value
  - Retrieve value previously associated with key
- Usage
  - Constructor creates empty map
  - Use add to insert new pair (or overwrite previous value)
  - Access elements using getValue
  - Shorthand operator[]
  - Browse pairs using iterator or mapping function
- Useful for:
  - dictionary, database, lookup table, document index, ...

Example maps

- Dictionary: word mapped to definitions

  cup -> small open container used for drinking
  dog -> a domesticated carnivorous mammal
  hop -> to move with light bounding steps
  fun -> a source of enjoyment or pleasure
- Thesaurus: word mapped to synonyms

  happy -> pleased, joyful, content, delighted
  walk -> saunter, stroll, hike, amble, toddle
  exit -> door, outlet
- DMV: license mapped to registration info

  2A0B130 -> 1985, Datsun B210, gray
  3CHT473 -> 1992, Honda Civic, red
  4XHR875 -> 2002, Mini Cooper, blue

Map interface

```
            // any type of Value, but always string key
template <typename ValueType>
class Map {
   public:

      Map();
      ~Map();

      int size();
      bool isEmpty();

      void add(string key, ValueType value);
      void remove(string key);

      bool containsKey(string key);
      ValueType getValue(string key);
   };
```

- What happens if getValue of non-existent key?

Client use of Map

```
void MapTest(ifstream & in)
{
   Map<int> map;
   string word;

   while (true) {
      in >> word;
      if (in.fail()) break;
      if (map.containsKey(word)) { // already seen word
         int count = map.getValue(word);
         map.add(word, count + 1); // incr & update
      } else
         map.add(word, 1);         // first occurrence
   }
   cout << map.size() << " unique words." << endl;
}
```

More on Map

- Shorthand operator[]
  - Can be used to get/set/update value for key
  - Applied to non-existent key will add pair with "default" value
- Why must keys always be string type?
  - Map internally uses known structure of strings to store efficiently
  - Can convert type to string to use as key (e.g. IntegerToString)
- What if more than one value per key?
  - Add new value will overwrite previous
  - Use Vector as value type for one-to-many relationship
- How can you summarize/browse entries?
  - e.g., printing all entries, summing all frequencies, finding the word with largest number of synonyms, and so on
  - Map provides access all elements in turn via an *iterator*

Iterating over Map

- Iterator is nested type, declared within Map class

  Full name is `Map<type>::Iterator`
- Usage
  - Ask map to create iterator
  - Walk through keys using hasNext/next on iterator
  - Iterator will visit all keys, no guarantee on which order

```
void PrintFrequencies(Map<int> & map)
{
    Map<int>::Iterator itr = map.iterator();
    while (itr.hasNext()) {
      string key = itr.next();
      cout << key << " = " << map[key] << endl;
    }
}
```

Set class

- Unordered collection of unique elements
  - {3, 5} is same set as {5, 3}, no duplicate elements
- Usage
  - Constructor creates empty set
  - Add/remove/contains to operate on members
  - High-level ops: unionWith, intersect, subtract, isSubsetOf, equals
  - Iterator to browse members
- Useful features:
  - Fast membership operations
  - Coalesce duplicates
  - High-level ops
    - Unioning our friends to create party invite list
    - Checking if set of courses meets requirements to graduate
    - Intersecting my desired pizza toppings with yours, subtracting things we both hate
    - Compound boolean queries, AND/OR/NOT

Set interface

```
template <typename ElemType>
class Set {
  public:
       Set(int (cmpFn)(ElemType, ElemType) = OperatorCmp);
       ~Set();

       int size();
       bool isEmpty();

       void add(ElemType element);
       void remove(ElemType element);
       bool contains(ElemType element);

       bool equals(Set & otherSet);
       bool isSubsetOf(Set & otherSet);
       void unionWith(Set & otherSet);
       void intersect(Set & otherSet);
       void subtract(Set & otherSet);

       Iterator iterator();
};
```

Client use of Set

```
void RandomTest()
{
   Set<int> seenSoFar;
   while (true) {
      int num = RandomInteger(1, 100);
      if (seenSoFar.contains(num)) break;
      seenSoFar.add(num);
   }
   cout << seenSoFar.size() << " unique before repeat.";
}


void PrintSet(Set<string> &set)
{
   Set<double>::Iterator itr = set.iterator();
   while (itr.hasNext())
      cout << itr.next() << " ";
   // Set iterator visits elements in order (unlike Map's)
}
```

Set higher-level operations

```
struct personT {
   string name;
   Set<string> friends, enemies;
};

Set<string> MakeGuestList(personT one, personT two)
{
   Set<string> result = one.friends;   // one's friends
   result.unionWith(two.friends);      // add two's friends
   result.subtract(one.enemies);       // remove one's enemies
   result.subtract(two.enemies);       // remove two's enemies
   return result;
}
```

Why Set is different

- Other containers store/retrieve elements, but Set truly examines them — why?
  - Non-duplication for add
  - Find element for contains, remove
  - High-level ops compare elements for match
- But Set is written as a template!
  - ElemType is just a placeholder
  - How to compare two things of unknown type?

Default element comparison

- Some types can be compared using < and ==
- Set uses a default function to compare two elements that looks like this:

```
{
  if (one == two) return 0;
  else if (one < two) return -1;
  else return 1;
}
```

- What happens if this default comparison doesn't make sense for the client's type?
  - E.g. == and < don't work on this type

Template compilation error

```
struct studentT {
   string first, last;
   int idNum;
   string emailAddress;
};

int main()
{
   Set<studentT> students;
```

- The above code will generate a compile error that is reported something like this:

```
 Error: no match for 'operator==' in 'one == two'
Error   : illegal operands 'studentT' == 'studentT'
 (point of instantiation: 'main()')
  (instantiating: OperatorCmp<studentT>(studentT, studentT)')
cmpfn.h line 25       if (one == two) return 0;
```

- < and == don't work for structs!

Client callback function

- Client writes function that compares two elements
  - Must match prototype as specified by Set
- Body of function does comparison
  - As appropriate for type
- Pass this function to Set
  - Set will hold onto it, and "call back" to client whenever it needs to compare two elements

Supplying callback function

```
struct studentT {
   string first, last;
   int idNum;
};

int CmpById(studentT a, studentT b)
{
   if (a.idNum < b.idNum) return -1;
   else if (a.idNum == b.idNum) return 0;
   else return 1;
}

int main()
{
    Set<studentT> set(CmpById); // ok!
```

Building things: ADTs rock!

- Map of Set
  - Google's web index (word to matching pages)
- Vector of Queues
  - Grocery store checkout lines
- Set of sets
  - Different speciality pizzas
- Stack of Maps
  - Compiler use to enter/exit nested scopes