Advanced Associative Containers

Ali Malik
malikali@stanford.edu
Game Plan

- Recap
- Map Iterators
- Further Usage
- Multimap
- auto and Range Based for
Recap
Associative Containers

Useful abstraction for “associating” a key with a value.

```cpp
std::map
map<string, int> directory; // name -> phone number

std::set
set<string> dict; // does it contains a word?
```
Let's try and get a mental model of iterators:

Say we have a `std::set<int> mySet`

Iterators let us view a non-linear collection in a linear manner.
int numOccurences(vector<int>& cont, int elemToCount) {
    int counter = 0;
    vector<int>::iterator iter;
    for (iter = cont.begin(); iter != cont.end(); ++iter) {
        if (*iter == elemToCount)
            ++counter;
    }
    return counter;
}
Map Iterators
Map Iterators

Map iterators are slightly different because we have both keys and values.

The iterator of a `map<string, int>` points to a `std::pair<string, int>`. 
The `std::pair` Class

A pair is simply two objects bundled together.

Syntax:

```cpp
std::pair<string, int> p;
p.first = "Phone number";
p.second = 6504550404;
```
The `std::pair` Class

Quicker way to make a pair

```cpp
std::pair<std::string, int> p{"Phone number", 6504550404};
std::make_pair("Phone number", 6504550404);
{"Phone number", 6504550404};
```
Map Iterators

Let’s reuse an example from last time to see how to iterate through a map.

Map Iterators
(MaplIterators.pro)
Map Iterators

Example:

```cpp
map<int, int> m;
map<int, int>::iterator i = m.begin();
map<int, int>::iterator end = m.end();
while(i != end) {
    cout << (*i).first << (*i).second << endl;
    ++i;
}
```
Further Usage
Iterators are useful for more than just looping through things!

We saw some uses already!
Iterator Uses - Sorting

For example, we sorted a vector using

```cpp
std::sort(vec.begin(), vec.end());
```
Finding elements

```cpp
vec<int>::iterator it = std::find(vec.begin(), vec.end());
if(it != vec.end()) {
    cout << "Found: " << *it << endl;
} else {
    cout << "Element not found!" << endl;
}
```
Finding elements

```cpp
set<int>::iterator i = mySet.lower_bound(7);
set<int>::iterator end = mySet.lower_bound(26);
while (i != end) {
    cout << *i << endl;
    ++i;
}
```
We can iterate through different ranges

<table>
<thead>
<tr>
<th></th>
<th>[a, b]</th>
<th>[a, b)</th>
<th>(a, b]</th>
<th>(a, b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>lower_bound(a)</td>
<td>lower_bound(a)</td>
<td>upper_bound(a)</td>
<td>upper_bound(a)</td>
</tr>
<tr>
<td>end</td>
<td>upper_bound(b)</td>
<td>lower_bound(b)</td>
<td>upper_bound(b)</td>
<td>lower_bound(b)</td>
</tr>
</tbody>
</table>
Multimap
Multimap

Maps store unique keys

Sometimes we want to allow the map to have the same key pointing to different values
Multimap

Don’t have [] operator

Add elements by calling .insert on a key value std::pair

```cpp
multimap<int, int> myMMap;
myMMap.insert(make_pair(3, 3));
myMMap.insert({3, 12}); // shorter syntax
cout << myMMap.count(3) << endl; // prints 2
```
Practice Problem (maybe)

An interview problem!

Interview Problem
(InterviewProblem.pro)
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:
Writing iterator types can be unsightly.

Consider a **map of deque of strings to vector of strings**:

<table>
<thead>
<tr>
<th>File Location</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>to be</td>
<td>or not to be just ...</td>
</tr>
<tr>
<td></td>
<td><code>window = </code>{to, be}`</td>
</tr>
<tr>
<td>to be or</td>
<td>not to be just ...</td>
</tr>
<tr>
<td></td>
<td><code>window = </code>{be, or}`</td>
</tr>
<tr>
<td>to be or not</td>
<td>to be just ...</td>
</tr>
<tr>
<td></td>
<td><code>{be, or} : </code>{not}`</td>
</tr>
<tr>
<td></td>
<td><code>window = </code>{or, not}`</td>
</tr>
<tr>
<td>to be or not to</td>
<td>be just ...</td>
</tr>
<tr>
<td></td>
<td><code>{be, or} : </code>{not}`,</td>
</tr>
<tr>
<td></td>
<td><code>{or, not} : </code>{to}`</td>
</tr>
<tr>
<td></td>
<td><code>window = </code>{not, to}`</td>
</tr>
</tbody>
</table>
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```

Use an alias for the collection type!
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
using NGramMap = map<deque<string>, vector<string>>;
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```

How can we clean this up?

Use an alias for the collection type!
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
using NGramMap = map<deque<string>, vector<string>>;
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```

How can we clean this up?

Use an alias for the collection type!
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
using NgramMap = map<deque<string>, vector<string>>;
NgramMap myMap;
for (NgramMap::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
using NgramMap = map<deque<string>, vector<string>>;
NgramMap myMap;
for (NgramMap::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```

How can we clean this up better?
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (map<deque<string>, vector<string>>::iterator iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```
Writing iterator types can be unsightly.

Consider a map of deque of strings to vector of strings:

```cpp
map<deque<string>, vector<string>> myMap;
for (auto iter = myMap.begin(); iter != myMap.end(); ++iter) {
    doSomething(*(iter).first, *(iter).second);
}
```

How can we clean this up better?

The `auto` keyword!
auto

auto is a C++11 feature that uses type deduction.

Asks the compiler to figure out the type for you.

When to use it?

- Use it whenever the type is obvious (e.g. iterators)
- In places where only the compiler knows the type (yes these exist)
A range based `for` loop is (more or less) a shorthand for iterator code:

```cpp
map<string, int> myMap;
for(auto iter = myMap.begin(); iter != myMap.end(); ++iter) {
    auto thing = *iter;
    doSomething(thing.first, thing.second);
}
```

```cpp
map<string, int> myMap;
for(auto thing : myMap) {
    doSomething(thing.first, thing.second);
}
```
Range Based **for** Loop

A range based **for** loop is (more or less) a shorthand for iterator code:

```c
6.5.4 The range-based for statement

1 For a range-based for statement of the form
    for (for-range-declaration : expression) statement
    let range-init be equivalent to the expression surrounded by parentheses\(^{86}\)
    (expression)
    and for a range-based for statement of the form
    for (for-range-declaration : braced-init-list) statement
    let range-init be equivalent to the braced-init-list. In each case, a range-based for statement is equivalent
to

    {
        auto && _range = range-init;
        for (auto _begin = begin-expr,
             _end = end-expr;
             _begin != _end;
             ++_begin ) {
            for-range-declaration = *_begin;
            statement
    }
```
Next Time

Templates