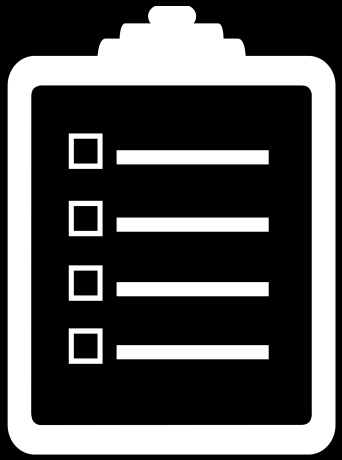
Advanced Associative Containers

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.

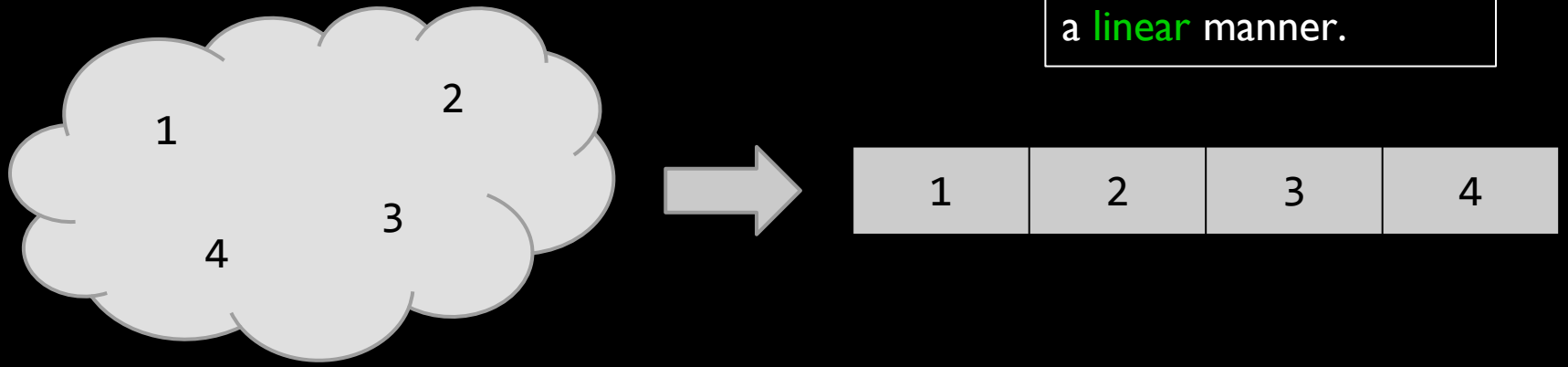
```
std::map
    map<string, int> directory;     // name -> phone number

std::set
    set<string> dict;               // does it contains a word?
```

Iterators

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.



Iterators

A standard interface to iterate through any collection.

```
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:

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

The `std::pair` Class

Quicker way to make a pair

```
std::pair<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

(MapIterators.`pro`)

Map Iterators

Example:

```
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

Iterator Uses

Iterators are useful for more than just looping through things!

We saw some uses already!

Iterator Uses

(Iterator Uses.`pro`)

Iterator Uses - Sorting

For example, we sorted a vector using

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

Iterator Uses - Find

Finding elements

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

Iterator Uses - Ranges

Finding elements

```
set<int>::iterator i = mySet.lower_bound(7);
set<int>::iterator end = mySet.lower_bound(26);
while (i != end) {
    cout << *i << endl;
    ++i;
}
```

Iterator Uses - Ranges

We can iterate through different ranges

	[a, b]	[a, b)	(a, b]	(a, b)
begin	lower_bound(a)	lower_bound(a)	upper_bound(a)	upper_bound(a)
end	upper_bound(b)	lower_bound(b)	upper_bound(b)	lower_bound(b)

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`

```
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`)

auto

auto

Writing iterator types can be unsightly.

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

auto

Where might you use this? 🤔

Writing iterator types can be unsightly.

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

auto

Writing iterator types can be unsightly.

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

Where might you use this? 🤔

File Location	Data
to be | or not to be just ...	`map = {}` `window = {to, be}`
to be or | not to be just ...	`map = { {to, be} : {or} }` `window = {be, or}`

to be or not | to be just ...	`map = { {to, be} : {or},`
	`{be, or} : {not} }`
	`window = {or, not}`

to be or not to | be just ...	`map = { {to, be} : {or},`
	`{be, or} : {not},`
	`{or, not} : {to} }`
	`window = {not, to}`

auto

Writing iterator types can be unsightly.

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

```
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);
}
```

auto

How can we clean this up?

Writing iterator types can be unsightly.

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

```
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);
}
```

auto

How can we clean this up?

Writing iterator types can be unsightly.

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

Use an alias for the collection type!

```
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);
}
```

auto

How can we clean this up?

Writing iterator types can be unsightly.

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

Use an alias for the collection type!

```
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);
}
```

auto

How can we clean this up?

Writing iterator types can be unsightly.

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

Use an alias for the collection type!

```
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);
}
```

auto

How can we clean this up?

Writing iterator types can be unsightly.

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

Use an alias for the collection type!

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

    doSomething(*(iter).first, *(iter).second);
}
```

auto

How can we clean this up?

Writing iterator types can be unsightly.

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

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

    doSomething(*(iter).first, *(iter).second);
}
```

auto

How can we clean this up better?

Writing iterator types can be unsightly.

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

```
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);
}
```

auto

How can we clean this up better?

Writing iterator types can be unsightly.

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

The auto keyword!

```
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);
}
```

auto

How can we clean this up better?

Writing iterator types can be unsightly.

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

The auto keyword!

```
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);
}
```

auto

How can we clean this up better?

Writing iterator types can be unsightly.

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

The auto keyword!

```
map<deque<string>, vector<string>> myMap;
for(auto iter = myMap.begin(); iter != myMap.end(); ++iter) {

    doSomething(*(iter).first, *(iter).second);
}
```

`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)

Range Based `for` Loop

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

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

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

Range Based `for` Loop

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

6.5.4 The range-based `for` statement **[stmt.ranged]**

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