Associative Containers & Iterators

...

Original from : Mike Precup (mprecup@cs.stanford.edu)
ENJMIN Edition 2016

Associative Containers

- Like Sequence Containers, Associative containers store data
- Unlike Sequence Containers, Associative containers have no idea of an ordering
- Instead, based on a **key**
- There are eight associative containers
  - `map`
  - `set`
  - `multimap`
  - `multiset`
  - `unordered_map`
  - `unordered_set`
  - `unordered_multimap`
  - `unordered_multiset`

STL <map>

- Methods are the same as the Stanford Map except for some syntax differences
  - If you want to see a complete list of methods, google search `std::map` or check out http://www.cplusplus.com/reference/map/map
- Let's see an example using a map (04Map)

STL <set>

- Methods are the same as the Stanford Set except for some syntax differences
  - If you want to see a complete list of methods, google search `std::set` or check out http://www.cplusplus.com/reference/set/set/
- Let's see an example using a set (04Set)
- Key point, a set is just a specific case of a map

Iterators

- How do we iterate over associative containers?

Versions of C++

C++03	C++11

`map<string, int> map;`
 	
`map<string, int>::iterator i;`
`map<string, int>::iterator end`
`  = map.end();`
 	
`for(i = map.begin();`
`    i != end; ++i) {`
`    cout << (*i).first`
`         << " " << (*i).second`
`         << endl;`
`}`
	`map<string, int> map;`
	
	`for (auto& a : map) {`
	`    cout << a.first << " "`
	`         << a.second << endl;`
	`}`

Versions of C++

- If we have C++11, why learn about iterators?

Versions of C++

- If we have C++11, why learn about iterators?
  - Iterators are used for more than just iterating as we will see when we start talking about ranges and algorithms
  - The C++11 code we saw does actually use iterators, they're just being used behind the scenes

Iterators

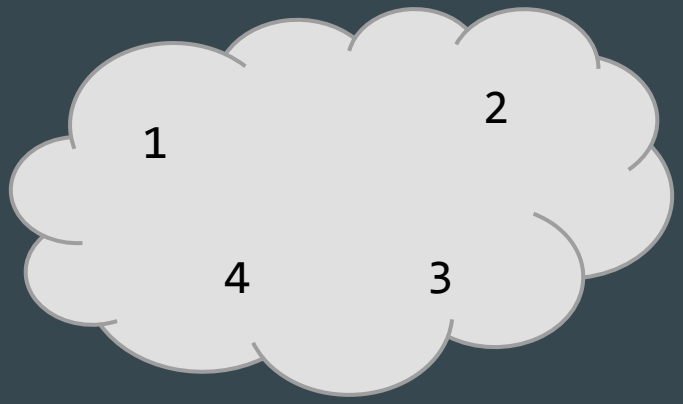
Iterators allow a programmer to iterate

over all the values in any container

whether it is ordered or not

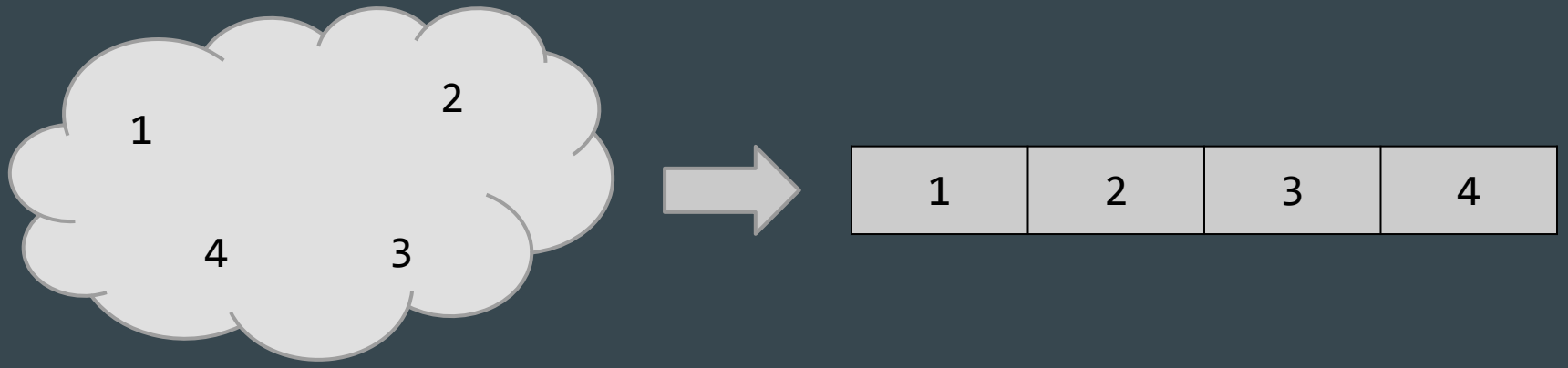
Iterators

- Let's first try and get a conceptual model of what an iterator is
- Say that we have a `set` of integers called `mySet`



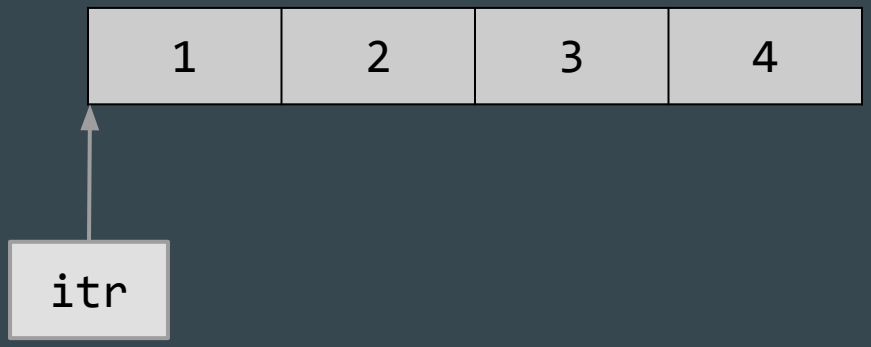
Iterators

- Let's first try and get a conceptual model of what an iterator is
- Iterators allow us to view a non-linear collection in a linear manner



Iterators

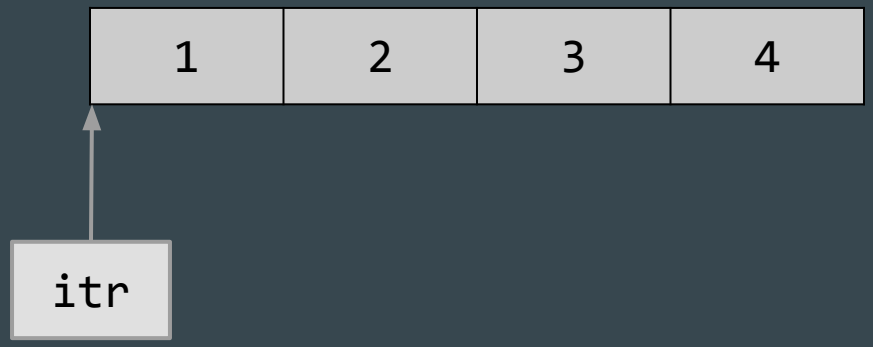
- Let's first try and get a conceptual model of what an iterator is
- We can construct an iterator 'itr' to point to the first element in the set




```
set<int>::iterator itr = mySet.begin();
```

Iterators

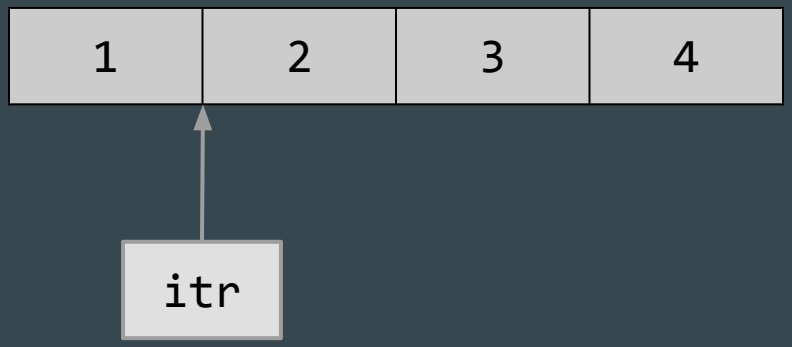
- Let's first try and get a conceptual model of what an iterator is
- We can get the value of an iterator by using the **dereference** operator *




```
cout << *itr << endl; //prints 1
```

Iterators

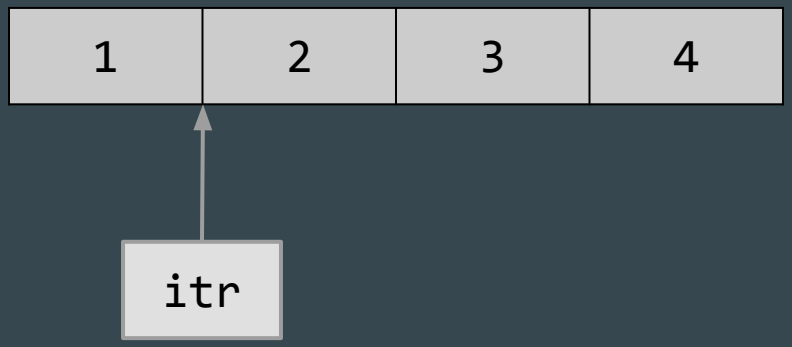
- Let's first try and get a conceptual model of what an iterator is
- We can advance our iterator with **++**
- It's convention to put the **++** before the iterator




```
++itr;
```

Iterators

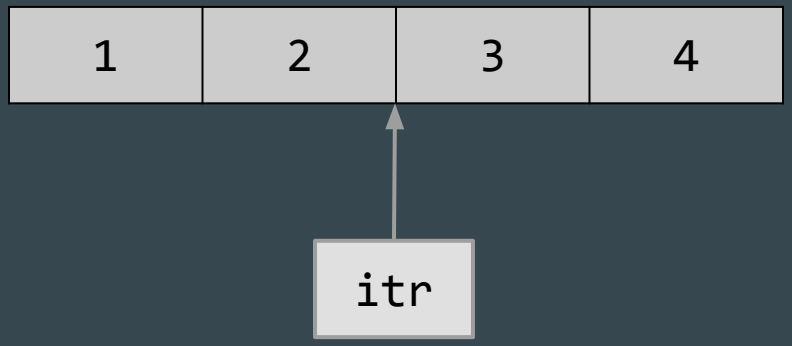
- Let's first try and get a conceptual model of what an iterator is
- We can keep dereferencing and advancing as we wish




```
cout << *itr << endl; //prints 2
```

Iterators

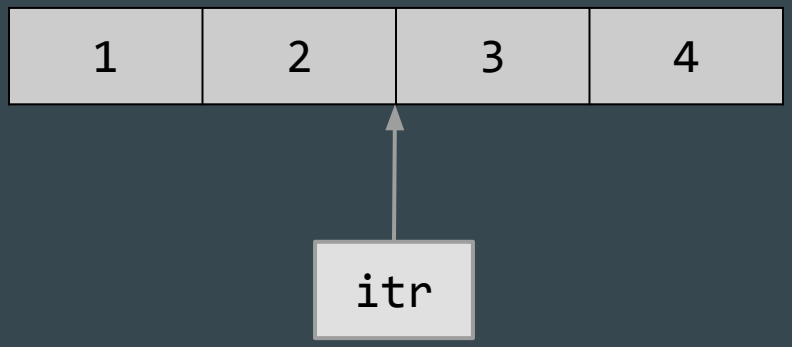
- Let's first try and get a conceptual model of what an iterator is
- We can keep dereferencing and advancing as we wish




```
++itr;
```

Iterators

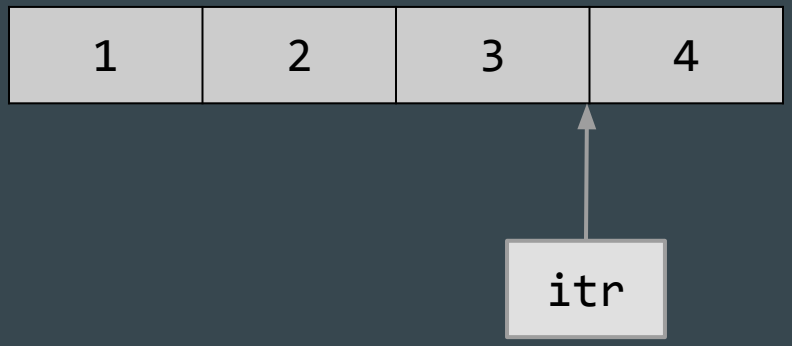
- Let's first try and get a conceptual model of what an iterator is
- We can keep dereferencing and advancing as we wish




```
cout << *itr << endl; //prints 3
```

Iterators

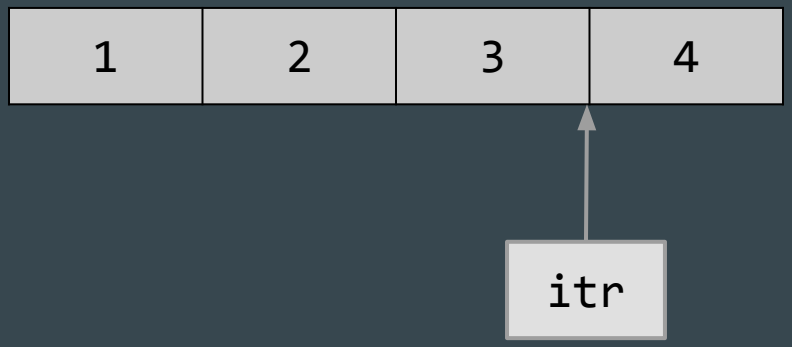
- Let's first try and get a conceptual model of what an iterator is
- We can keep dereferencing and advancing as we wish




```
++itr;
```

Iterators

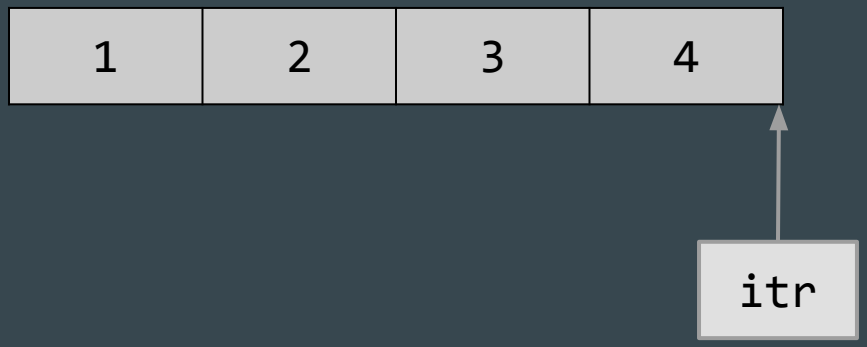
- Let's first try and get a conceptual model of what an iterator is
- We can keep dereferencing and advancing as we wish




```
cout << *itr << endl; //prints 4
```

Iterators

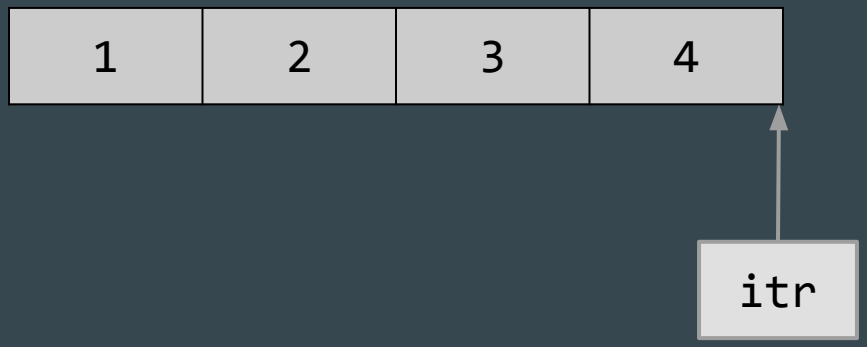
- Let's first try and get a conceptual model of what an iterator is
- We can keep dereferencing and advancing as we wish




```
++itr;
```

Iterators

- Let's first try and get a conceptual model of what an iterator is
- We can check if we've reached the end by comparing to **`.end()`**




```
if (itr == mySet.end()) return;
```

Iterators

Four essential iterator operators:

- **Create** an iterator
- **Dereference** an iterator and read the value it's currently looking at
- **Advance** an iterator
- **Compare** an iterator against another iterator (especially one from the `.end()` method)

Iterators

Let's Do Some Examples

(BasicIterator)

Versions of C++

- If we have C++11, why learn about iterators?
  - Iterators are used for more than just iterating as we will see when we start talking about ranges today and algorithms next week
  - The C++11 code we saw does actually use iterators, they're just being used behind the scenes

Versions of C++

- If we have C++11, why learn about iterators?
  - Iterators are used for more than just iterating as we will see when we start talking about ranges and algorithms
  - The C++11 code we saw does actually use iterators, they're just being used behind the scenes

Other Uses for Iterators

STL containers often use iterators to specify individual elements inside a container.

```
vector<int> v;
for (int i = 0; i < 10; i++) {
    v.push_back(i);
}
v.erase(v.begin() + 5, v.end());
// v now contains 0, 1, 2, 3, 4
```

Other Uses for Iterators

Iterators don't always have to iterate through an entire container

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

Other Uses for Iterators

Here is code that will iterate through all elements **greater than or equal to** 7 and **less than** 26

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

Other Uses for Iterators

Note that we can iterate through various ranges of numbers simply by choosing different values of begin and end

	[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)

Other Uses for Iterators

Let's code up some examples

(IteratorRanges.pro)

Iterating through maps

- All of our iterator examples involved set iterators, but (almost) all C++ collections have iterators
- Sequence Container iterators are straightforward
- Maps are a little more complicated

The Pair Class

- A **pair** is simply two objects bundled together
- Syntax is the following:

```
pair<string, int> p;

p.first = “phone number”;

p.second = 8675309;
```

Iterating through maps

- When iterating through maps, dereferencing returns a pair containing the key and the value of the current element

```
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;

}
```

Multiset

- Sets store unique elements
- If you want to store multiple copies of an element, use a multiset
- Almost all methods are the same

```
multiset<int> myMSet;

myMSet.insert(3);

myMSet.insert(3);

cout << myMSet.count(3) << endl; //prints 2
```

Multimap

- Maps store unique keys
- If you want to store multiple copies of a key, use a multimap
- Syntax is non-trivially different

Multimap

- No [] operator
- Add elements using `insert(make_pair(key, value))`

```
multimap<int, int> myMMap;

myMMap.insert(make_pair(3, 3));

myMMap.insert(make_pair(3, 12));

cout << myMMap.count(3) << endl; //prints 2
```

Unordered Collections

- Can be used almost exactly as ordered collections
- Iterating over them will return elements in unreliable orders (not sorted!)
- Faster lookup time, insertion and deletion times vary but are usually better
- Uses more memory

Unordered Collections

```
unordered_map<int, int> m;

unordered_map<int, int>::iterator i = m.begin();

unordered_map<int, int>::iterator end = m.end();

while (i != end) {

    cout << (*i).first << (*i).second << endl;

    ++i;

}

// Will not reliably print in sorted order
```

The `auto` Keyword

- `auto` is a C++11 feature that uses **type deduction**
- Asks the compiler to figure out the correct type for you
- Let's see an example

`auto` VS `var`

- If you're familiar with some other languages, you might think auto seems a lot like `var`
- Depending on what language, you might be right (C#)
- Depending on what language, you might be wrong (Javascript)
- **auto variables are still statically typed** (example)

When to Use `auto`?

- Active point of contention amongst C++ programmers
- No real accepted answer
- My personal opinion:
  - On nested types where the type itself is still obvious (iterators)
  - In places where only the compiler knows the type (yes, these exist)

`auto` and References

- `auto` drops reference qualifiers
- Add them with `auto&`

```
Widget& getWidget();

auto widget = getWidget();

// type of widget is Widget, not Widget&!
```

`auto` and References

- `auto` drops reference qualifiers
- Add them with `auto&`

```
Widget& getWidget();

auto& widget = getWidget();

// type of widget is Widget&
```

To Whom It May Concern

- Don’t worry if this doesn’t make sense, we’ll cover it in more detail later
- `auto` drops:
  - `&`
  - `const`
  - `volatile`
- `auto&` doesn’t
- Mixing auto with array types, rvalue references, or initializer lists has more complex behavior

Iterating Through Maps (with `auto`)

```
map<int, int> m;

auto i = m.begin();

auto end = m.end();

while (i != end) {

    cout << (*i).first << (*i).second << endl;

    ++i;

}
```

Iterating Through Maps (with `auto`)

```
map<int, int> m;

for (auto i = m.begin(); i != m.end(); ++i) {

    cout << (*i).first << (*i).second << endl;

}
```

Range Based `for` Loops

- The name for the following syntax:

```
vector<int> vec;

// fill vec
// ...

for (int i : vec) {
    cout << i << endl;
}
```

- Works on any container with `begin()` and `end()`

Closing Notes

- Iterators are used everywhere in C++ code
- When you first look at a C++ style iterator, you may find yourself missing foreach, but iterators offer a lot more
- Iterator ranges are just the start. When we start talking about <algorithm> We'll see just how useful iterators can be