Associative containers and iterators

Reid Watson
(rawatson@stanford.edu)

Administrivia

- Assignment zero is on the site right now!
- Let’s take a look at a demo of GraphViz
- Due October 22nd
- A fun assignment to do!

Associative Containers

- Unsurprisingly, associative containers are containers (objects you can store data in)
- Associative containers use the idea of a **key**, which is used to lookup a value
- We'll be talking about two associative containers today
  - Maps
  - Sets

STL <set>

- Most methods are fairly similar, so I won't bore you with a list of the methods
  - I posted a comparison of Stanford and STL containers online though, in case you want to peek at it.
- Let's take a quick peek at the code though, so we can see what STL set code looks like

STL <map>

- Most methods are fairly similar, so I won't bore you with a list of the methods
  - I posted a comparison of Stanford and STL containers online though, in case you want to peek at it.
- We will take a quick look at some code which uses maps though

Versions of C++

I'm going to start talking about iterators, but before that, I want to talk about the different versions of C++

Versions of C++

- 1983-1998: Prehistoric C++
  - This was the original form of C++. There was no document describing what C++ was. Any code which would compile and run was valid C++
  - This was fine, but many people wanted to codify the rules of C++ into a standard
- 1998: C++98
  - The C++98 standard is published, which details exactly what valid C++ code is.
- 2003: C++03
  - Minor changes to the rules governing how programs run, but effectively the same as C++98

Versions of C++

- 2011: C++11!
  - C++11 was the largest change to the C++ language since it's creation
  - Clarified a lot of things in the old standard
  - Added **tons** of new features
    - Multithreading
    - Type inference
    - Lambdas
    - Variadic Templates
    - Range based for
    - `constexpr`
    - New initialization syntax
  - Long story short, C++11 has a lot of cool stuff in it!

Versions of C++

- I would love to only ever mention C++11 in this class
  - New features added in C++11 often make code a **lot** easier to read and write

C++03	C++11

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

Versions of C++

- The problem is that not everyone's compiler can handle C++ code
  - Stanford uses Visual Studio 2008 in CS106B, which doesn't support C++11 (released in 2011)
  - We're working on it!
- We're going to learn the long C++03 way of doing things first for two reasons:
  - Really understanding the C++11 way requires understanding the C++03 way
  - The large majority of C++ applications don't run on systems with full C++11 support

Versions of C++

- The problem is that not everyone's compiler can handle C++ code
  - Stanford uses Visual Studio 2008 in CS106B, which doesn't support C++11 (released in 2011)
  - We're working on it
- We're going to learn the wrong C++03 way of doing things first for two reasons:
  - Really understanding the C++11 way requires understanding the C++03 way
  - The large majority of C++ applications don't run on systems with full C++11 support

NOT ANYMORE HAHAHAHA

Versions of C++

Now that we're all using Qt Creator, C++11 is available for us!

Iterators

- How do you iterate through all the elements of a set?
- How do you iterate through all the elements of a map?

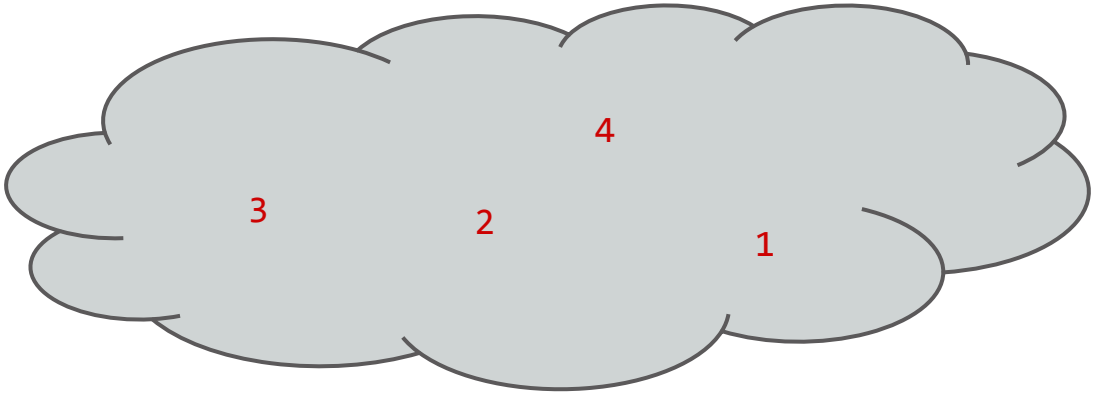
Because maps and sets aren't sequence containers, we can't just go from 0 to vector. size() - 1, or pop elements off of a stack until it's empty.

Iterators

As we first see them, iterators will allow us to iterate through all the elements of an unsequenced collection of elements (like a set or a map)

Iterators

- Let's first try and get a conceptual model of what an iterator is
- Say that we have a `set` of integers. Say the set was named 's'.

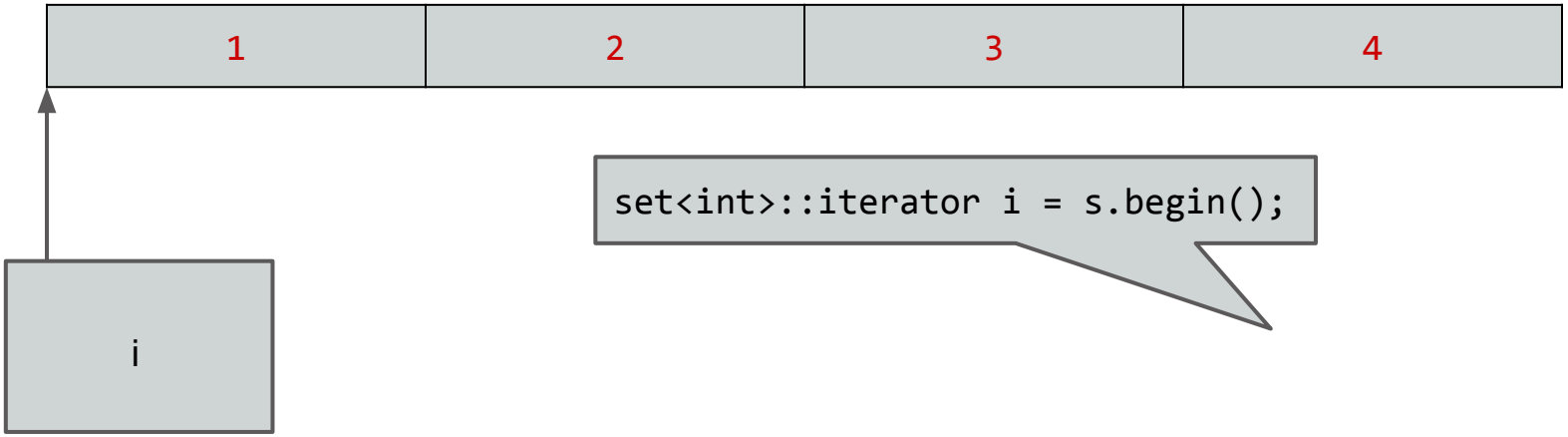


Iterators

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

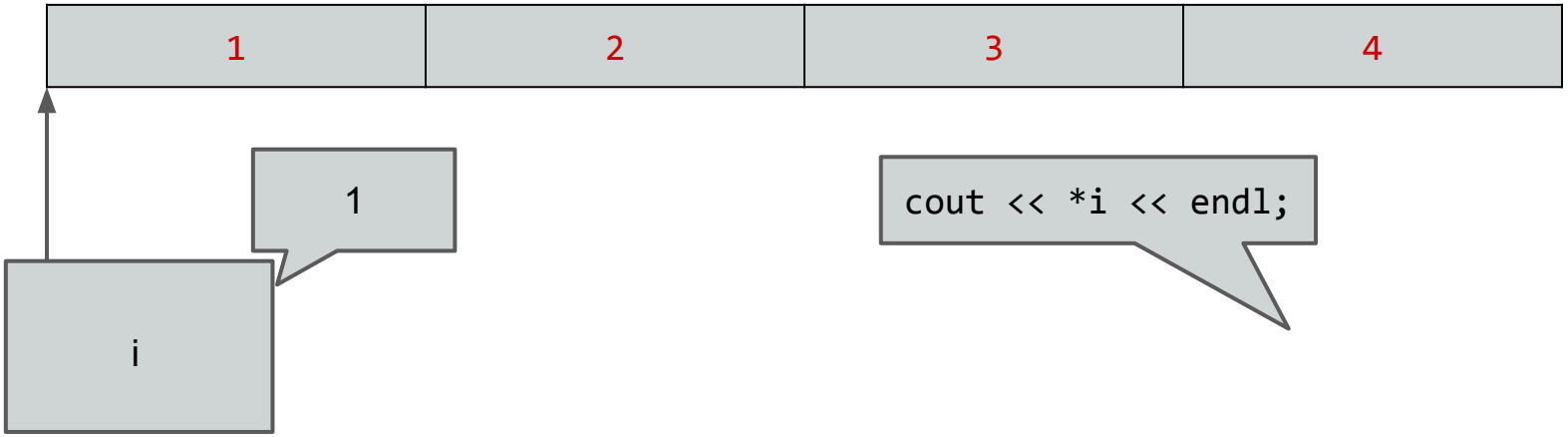
Iterators

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



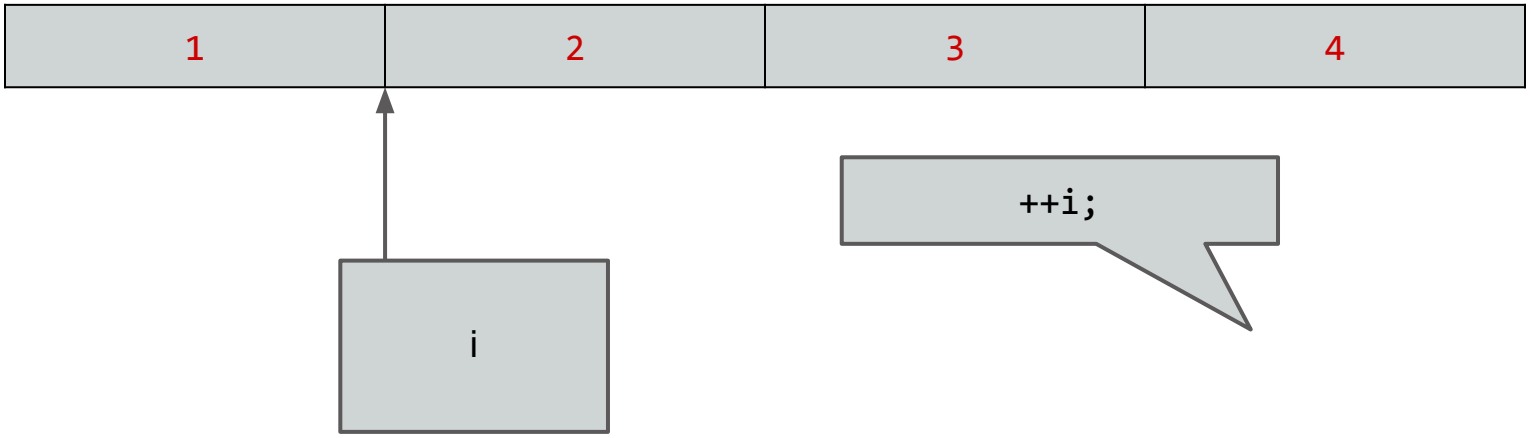
Iterators

- Let's first try and get a conceptual model of what an iterator is
- We can **dereference** our iterator to read the value the iterator is currently on



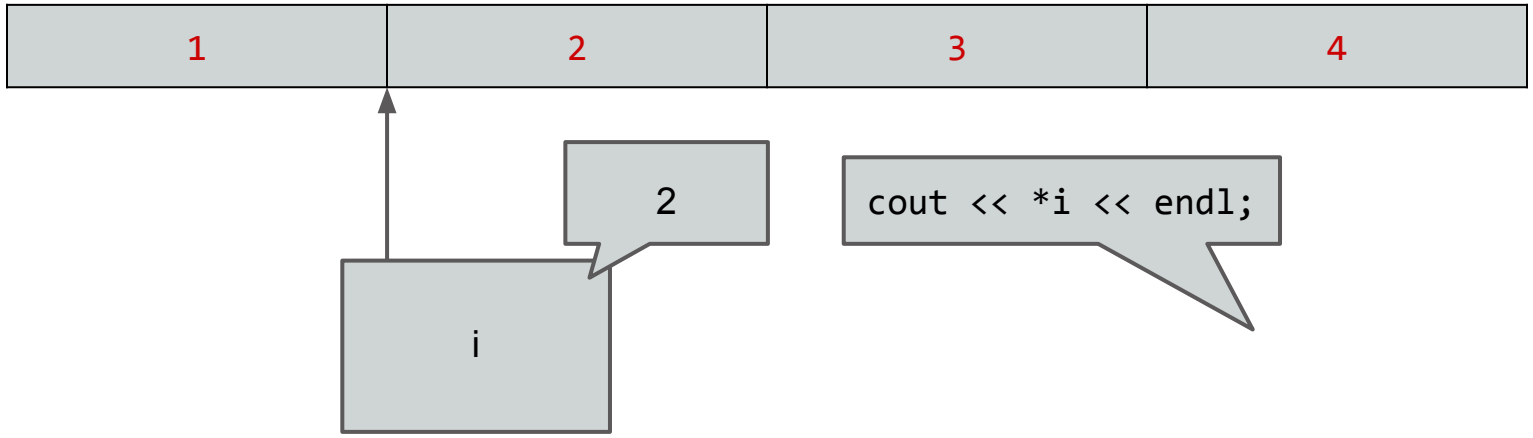
Iterators

- Let's first try and get a conceptual model of what an iterator is
- We can **advance** our iterator



Iterators

- Let's first try and get a conceptual model of what an iterator is
- We can **dereference** our iterator again and read a different value



Iterators

Eventually, we reach the end of a container.

You can check if an iterator has iterated through every element in the container by comparing it to the magical .end() element.

```
if (i == s.end())
  cout << "We're done!" << endl;
```

Iterators

Remember the four most fundamental iterator operations:

- 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 take a look at C++03 and C++11 iterator syntax in a bit of code.

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

Iterator's don't always have to iterate through all of a container.

For example, they could iterate through a range of elements.

Other uses for iterators

For example, here's the code to iterate through all the integers in a set:

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

Other uses for iterators

For example, here's the code to iterate through all the integers **greater than 7** and **less than 23** in a set:

```
set<int>::iterator i = s.lower_bound(7);
set<int>::iterator end = s.upper_bound(23);
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 take a quick look at this in code

Parting Thoughts

- Iterators are used everywhere in C++ code
- When you first look at a C++03 style iterator loop, 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
- Don't forget assignment one!