Iterators and Ranges

CS 1044

Types of Collections

- C++ supports a few types of collections
  - `vector` and `list` – ordered, sequential collections of data
  - `set` – unordered collection (forbids duplicates)
  - `map` – unordered “dictionary” (look up values based on a “key” rather than an index”

Iteration

- If we have a collection of data, it’s natural to want to perform an operation on **every element** in the collection
- We use iteration to perform this repetitive task
- Remember: Iteration = loops

Iteration with Vectors

```
vector<string> v;

// ... push_back some stuff onto v ...

for (int i = 0; i < v.size(); i++)
{
    cout << v[i]; // do something with element
}
```

Collections without Order

- Sets and maps **don't have a natural order** or numerical positions for elements
- We can't say `some_set[i]`, for example
- So how do we use a loop to get at the elements?
- C++ introduced **iterators** to make access to collections the same, regardless of the type of collection

Iterators

- When iterating over a collection, we really only need the following information:
  - Where do we **start**?
  - Where do we **stop**?
  - How do we get from one element to the **next** one?
  - How do we get the element at our **current** position?

Vector Example, Again

```
vector<string> v;
// ... push_back some stuff onto v ...

for (int i = 0; i < v.size(); i++)
{
    cout << v[i]; // do something with element
}
```

Where we start

Where we stop

How we get to

How we get the

Vector Example w/ Iterators

```
vector<string> v;

// ... push_back some stuff onto v ...


vector<string>::iterator it;
for (it = v.begin(); it != v.end(); it++)
{
    cout << (*it); // do something with element
}
```

First, declare a variable that represents an iterator for the vector

Vector Example w/ Iterators

```
vector<string> v;

// ... push_back some stuff onto v ...

vector<string>::iterator it;
for (it = v.begin(); it != v.end(); it++)
{
    cout << (*it); // do something with element
}
```

Where we start

Where we stop

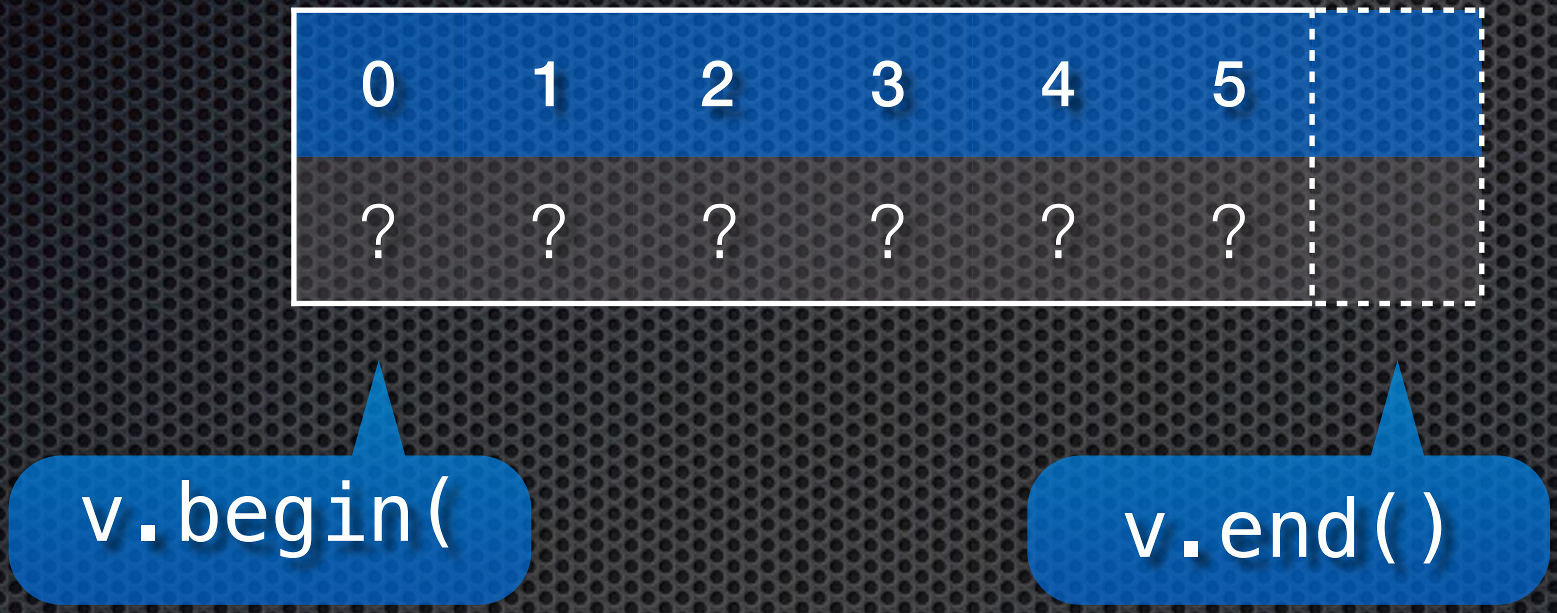
How we get to

How we get the

Iterator Details

- If your collection `v` is a `vector<T>` (fill in `T` with whatever you want), then the type of its iterator is `vector<T>::iterator`
- `v.begin()` returns an iterator pointing to the first element (`v[0]`)
- What does `v.end()` return?

End Iterators



- `v.end()` returns an iterator that points to a “fake” location **one past the last element** in the collection
- Must work this way for loops to work correctly (think about it)

Accessing Elements

- If we have an iterator called `it`, then we refer to the element that it points to by writing `*it`

```
cout << (*it); // parens for extra safety
```

- We can also overwrite the element the iterator points to

```
(*it) = 50;
```

Moving Among Elements

Code	Description
`it++`	Move forward to the next element in the collection.
`it--`	Move backward to the previous element in the collection.
`it = it + N`	Move forward N elements.
`it = it - N`	Move backward N elements.

The last two supported by vectors only

Comparing Iterators

Code	Description
`it1 == it2`	True if both iterators point to the same element.
`it1 != it2`	True if each iterator points to a different element.
`it1 < it2`	True if `it1` points to an element before `it2` in a vector.
<=, >, and >= work similarly	

Iterators as Locations

- As we already saw, vectors use iterators, rather than numerical indices, for `insert` and `erase`
- Examples:
  - `v.insert(v.begin() + 3, "foo");`
  - `v.erase(v.begin() + 2);`
- Why? Consistency with other collections that don't have numerical indexing

Iterators as Ranges

- Two iterators define a **range** of elements in a collection
- Assume we have two iterators, `first` and `last`
- They define the range `[first, last)` – in other words, **including** `first` but **excluding** `last`
- So, `v.begin()` and `v.end()` define the range containing **all elements** in v
- A range where both iterators are the same is **empty**

Range-based Algorithms

- C++ provides a huge number of **pre-written algorithms** that work with ranges of data defined by iterators
- Advanced computations such as **inspecting**, **sorting**, **searching**, **shuffling**, and **transforming** data in collections
- Always best to use proven solutions instead of reinventing the wheel – we'll see some later