Iterators

Iterators

- Essentially, an iterator is a **pointer** to a value in a container
  - Does not require an &, accomplished with other operators
    - In fact, iterators are objects!
  - Common across all containers
  - Only way to effectively get access to **every** container as not all containers allow `.at` or `[]` (non-sequences)

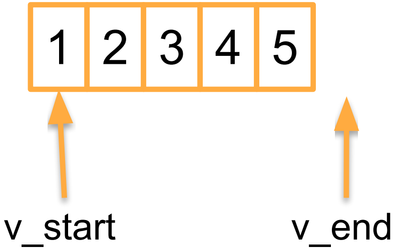
Common Interface

- The result of iterators being a common to all containers, many of the **generic algorithms** depend on iterators
  - Generic algorithms work on a container of every type
  - Access to how the generic algorithms work is via iterators

Creating an iterator

```
vector<int> v = {1, 2, 3, 4, 5};
auto v_start = v.begin();
auto v_end = v.end();
string s = “hi mom”;
auto s_start = s.begin();
```

- `begin()` and `end()` respectively:
  - Return an iterator to first element
  - Return an iterator to **one past** the last element




```
vector<int> v = {1, 2, 3, 4, 5};
auto v_start = v.begin();
auto v_end = v.end();
```

`v_last` one past the end

type `vector<int>::iterator`

Half-open range

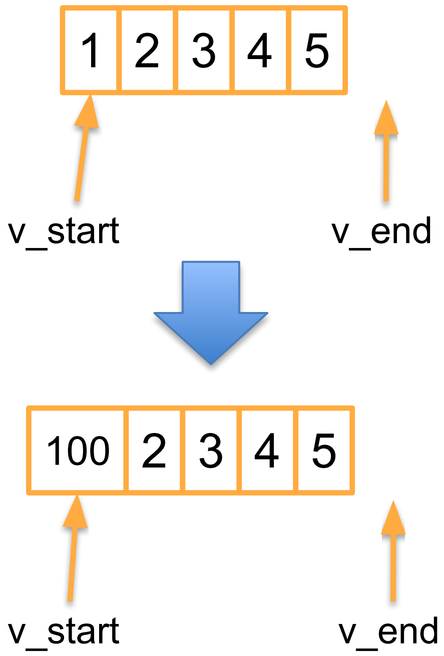
- We saw this in Python as well.
- The reasoning is
  - Have a stopping point (is your iterator less than the end)
  - For an empty range, `begin() == end()` so no special testing required

What type

- Iterator type is **dependent on the container** they point to (huge surprise)
- `v_start`, `v_end` are of type `vector<int>::iterator`
- `s_start` is of type `string::iterator`

Accessing Elements

```
vector<int>::iterator v_start;
vector<int> v{1, 2, 3, 4, 5};
v_start = v.begin();
cout << *v_start;   // first element, 1
*v_start = 100;   // assign first to 100
cout << *v_start;   // first element, now 100
```



3 ways to iterator (one more coming)

```
vector<int> v = {1, 2, 3, 4, 5};
for (int i = 0; i < staic_cast<int>(v.size()); i++)
    cout << v[i] << endl;

for (auto element : v)
    cout << element << endl;

for (auto ptr = v.begin(); ptr != v.end(); ++ptr)
    cout << *ptr << endl;
```

Pointer Arithmetic

- So what does `++ptr` mean?
- For some (more on that later) iterators and all pointers, adding one means **go to the next element**
- We don't add one to the address (which is what a pointer has as a value), we add enough to the address to get to the next value

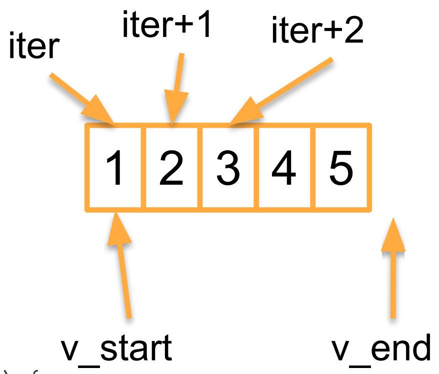
How does it know how much to add?

- Types of course!
- If it is a `long`, add 8 to the address (8 bytes to a long)
- If it is a `double`, add 8
- A `vector` of `int`, add whatever (the compiler knows!)
- Because of the type, pointer arithmetic changes based on that type, adding or subtracting to move to the next element!

More iteration

```
vector<int> v = {1, 2, 3, 4, 5};
auto v_start = v.begin();
auto v_last = v.end();

for (auto itr = v.begin(); itr != v.end(); ++itr){
    cout << *itr << endl;
} // of for
```



True for “just pointers”

- Pointers (initialized via the & operator) behave the same way using pointer arithmetic
  - Address is incremented to the “next” element, based on type
  - When we get to “good old-fashioned arrays”, this will be useful

for-each is shortcut for iterator

- for-each is really a convenience, gets translated to a ptr based loop

```
for (type element : collection) {
    …
}
for (auto pos = collection.begin();
     pos!=collection.end();
     ++pos) {
    type element = *pos;
    …
}
```

Efficiency considerations

- Which is more efficient: `++pos` or `pos++`?
  - `++pos` since previous value does not need to be stored
- Why `pos != end` instead of `pos < end`?
  - Not every collection supports `<` in their iterators (more later)
  - `!=` is more general but more susceptible to error
  - Programmer's call

Iterators (continued)

Type of the auto element

- `auto` is a great way to declare a variable, but it does have its drawbacks
  - it **does not** preserve `const`
  - it **does not** preserve `&`
- You have to add this back yourself

auto

```
for (auto pos = collection.begin(); pos != collection.end();
++pos) {
  auto element = *pos
  …
}
```

```
for (auto element : collection)
{
      …
}
```

- What type, `auto element`?
  - If it is a standard type, `*pos` derefs and makes a **copy** to element
  - Change to element does **not** change the underlying collection
  - May or may not be what you want

auto&

```
for (auto pos = collection.begin(); pos !=
collection.end(); ++pos) {
  auto & element = *pos
  …
}
```

```
for (auto & element : collection) {
    …
}
```

- What if it is `auto & element`?
  - If we add `&` to the auto type, `*pos` derefs and `element` is an **alias** to that deref
  - Change to element **does** change the underlying collection
  - May or may not be what you want

const auto&

```
for (auto pos = collection.begin(); pos !=
collection.end(); ++pos) {
  const auto & element = *pos
  …
}
```

```
for (const auto & element : collection) {
    …
}
```

- What if it is `const auto &element`?
  - If we add `&` to the auto type, `*pos` derefs and `element` is an **alias** to that deref
  - No copy, but **cannot** change the underlying collection
  - May or may not be what you want

dereference and parens

- What is the difference between the code below:

```
vector<int> v = {5, 4, 3, 2, 1};
auto v_start = v.begin();
cout << *v_start + 1; // 6
cout << *(v_start + 1); // 4
```

- deref, add one to the value
- add one to the pointer, deref
- `*` has operator precedence!

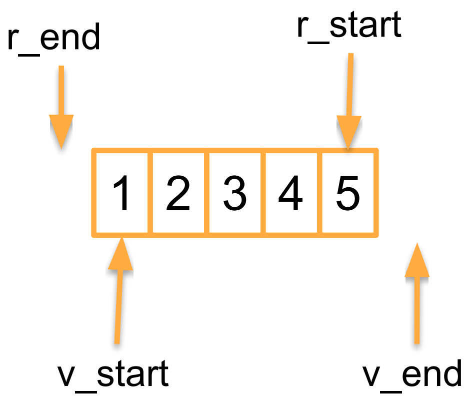
Some iterator types

- `begin(), end()`
  - like we discussed
- `cbegin(), cend()`
  - constant iterators. You can read, but you **cannot write** to the ptr
- `rbegin(), rend()`
  - reverse iterators
- `crbegin(), crend()`
  - constant reverse

reverse

```
vector<int> v = {1, 2, 3, 4, 5}
auto v_start = v.begin();
auto v_last = v.end();
auto r_start = v.rbegin();
auto r_last = v.rend();
```

- half-open range is now reversed



Reverse a string

```
string my_str = “hi mom”, rev_str = “”;
for (auto pos = my_str.rbegin(); pos <
my_str.rend(); ++pos)
   rev_str += *pos;
```

- Weirdly, `++pos` mean **go backwards** one (because it is a reverse iterator)

General classes of iterators

- There are classes of iterators based on the kinds of operations you can perform on them.
- These restrictions (or allowances) are dictated by their associated containers
  - Forward iterators
  - Bi-directional iterators
  - Random access iterators

Forward iterators

- Given an iterator `itr` on a containers, only allow `++itr`;
  - Cannot go backward, `--itr`
  - Cannot go to a particular index, cannot do pointer math
  - No < compare, but `!=` ok
  - Associated with `forward_list`, output iterators, input iterators

Bi-directional iterators

- For a particular iterator `itr`, can go both forward (`++itr`) and backward (`--itr`)
  - Cannot go to a particular index or do pointer arithmetic
  - **Cannot** do `<`, **can** do `!=`
  - Associated with maps, sets, lists

Random Access

- Can do all of the things covered before
  - Can use subscripting (`pos[3]`)
  - Can do pointer arithmetic (`pos += 3`)
  - Can do relational operations (`pos > pos2`)
    - Associated with string, vectors (sequence containers)