Lecture 6: Iterators

CS 106L, Fall ‘20

Today’s Agenda

- Recap: **Collections**
- Iterators
- Iterator Practice

Question

What type lets you insert at back and front equally efficiently?

(Answer: a deque)

Question

What type(s) require a ***comparison operator*** defined over the type of its elements? (For example, the type of the elements of a std::vector<string> is string).

(Answer: sets and maps)

What can we do to avoid this?

(Answer: unordered_set, unordered_map)

Question

Which one is faster, a **set** or an **unordered_set**?

Recap: Collections

Stanford vs STL Vector (a Review)

```
// Stanford
Vector<char> vec{'a', 'b', 'c'};

vec[0] = 'A';
cout << vec[vec.size()-1];

for (int i = 0; i < vec.size(); i++) {
  vec[i]++;
}

for (auto& elem : vec) {
  elem--;
}
```

```
// STL
std::vector<char> vec{'a', 'b', 'c'};

vec[0] = 'A';
cout << vec[vec.size()-1]; // or vec.back()

for (size_t i = 0; i < vec.size(); i++) {
  vec[i]++;
}

for (auto& elem : vec) {
  elem--;
}
```

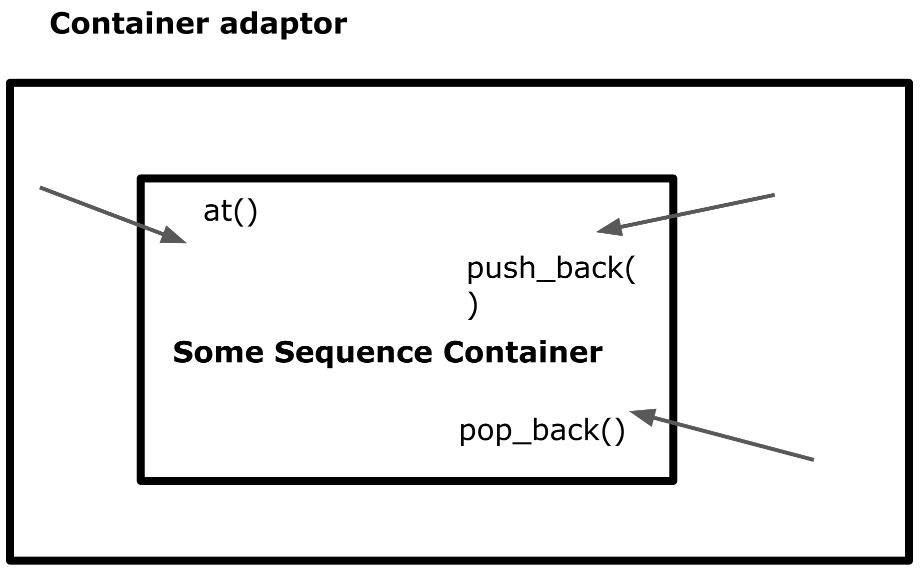
std::deque provides fast insertion anywhere

`std::deque` has the exact same functions as `std::vector` but also has `push_front` and `pop_front`.

```
std::deque<int> deq{5, 6};      // {5, 6}
deq.push_front(3);              // {3, 5, 6}
deq.pop_back();                 // {3, 5}
deq[1] = -2;                    // {3, -2}
```

How do you design a stack?

- **Container adaptors** provide a different interface for sequence containers. You can choose what the underlying container is!



Lööps

Looping over Collections

Fill in the four blanks for std::vector in the chat!

```
std::vector<int> vector{3, 1, 4, 1, 5, 9};
for (initialization; termination condition; increment) {
    const auto& elem = retrieve element at index;
    cout << elem << endl;
}

std::set<int> set{3, 1, 4, 1, 5, 9};
for (initialization; termination condition; increment) {
    const auto& elem = retrieve element at index;
    cout << elem << endl;
}
```

🤔 Why is **elem** by reference, and why is it **const**?

Looping over Collections

Fill in the four blanks for std::vector in the chat!

```
std::vector<int> vector{3, 1, 4, 1, 5, 9};
for (size_t i = 0; i < vector.size(); i++) {
    const auto& elem = vector[i];
    cout << elem << endl;
}

std::set<int> set{3, 1, 4, 1, 5, 9};
for (initialization; termination condition; increment) {
    const auto& elem = retrieve element at index;
    cout << elem << endl;
}
```

Looping over Collections

Fill in the four blanks for std::vector in the chat!

```
std::vector<int> vector{3, 1, 4, 1, 5, 9};
for (size_t i = 0; i < vector.size(); i++) {
    const auto& elem = vector[i];
    cout << elem << endl;
}

std::set<int> set{3, 1, 4, 1, 5, 9};
for (uhh; umm; something++?) {
    const auto& elem = idk;
    cout << elem << endl;
}
```

Iterators

Iterators allow iteration over any container

whether ordered or unordered

An iterator is like a “claw”

An iterator is like “the claw”

Iterators (“the claw”) can:

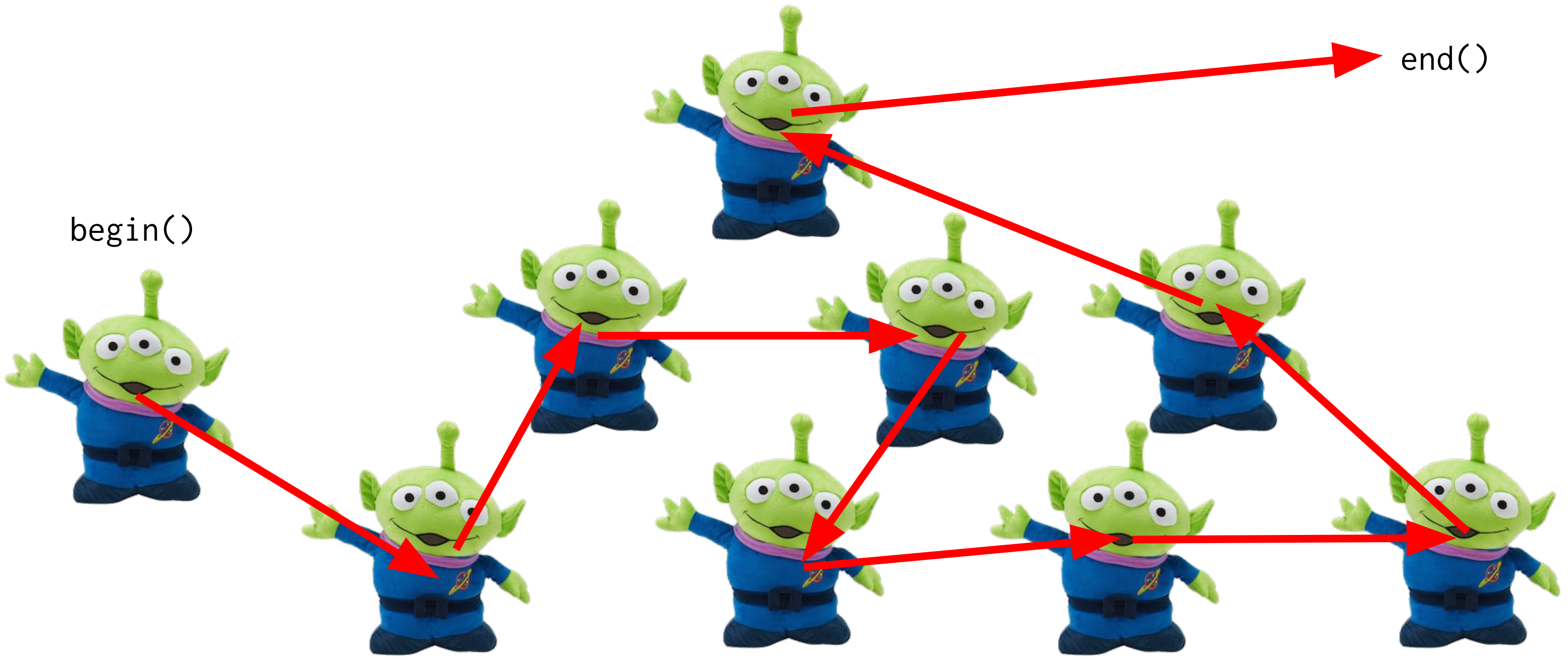
- move “forward”
  - according to some order…
- retrieve element
- check if two claws are in the same place

Containers (“the machine”) provide:

- the bounds (begin and end)

Key idea: iterator has ordering over elems

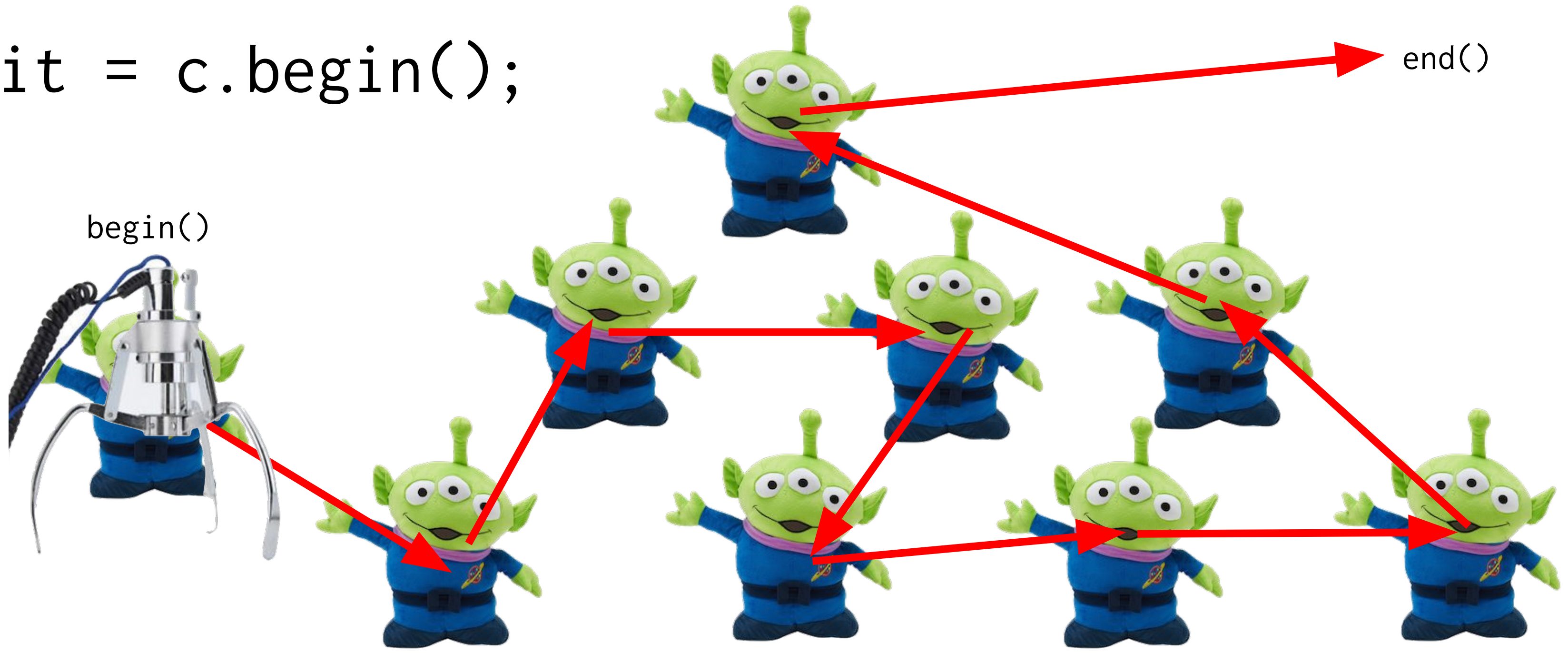
i.e. it always knows what the “next” element is



Key idea: iterator has ordering over elems

i.e. it always knows what the "next" element is

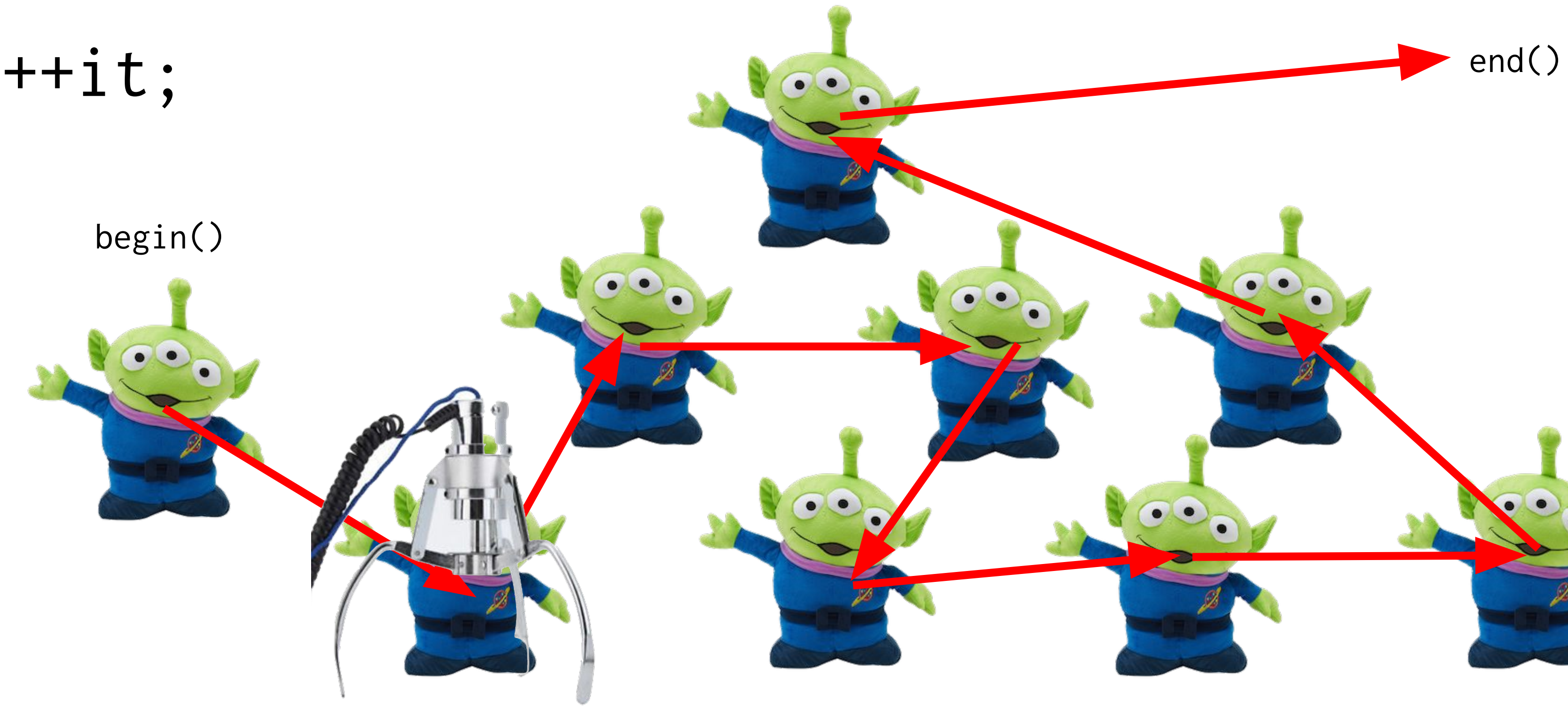
```
it = c.begin();
```



Key idea: iterator has ordering over elems

i.e. it always knows what the "next" element is

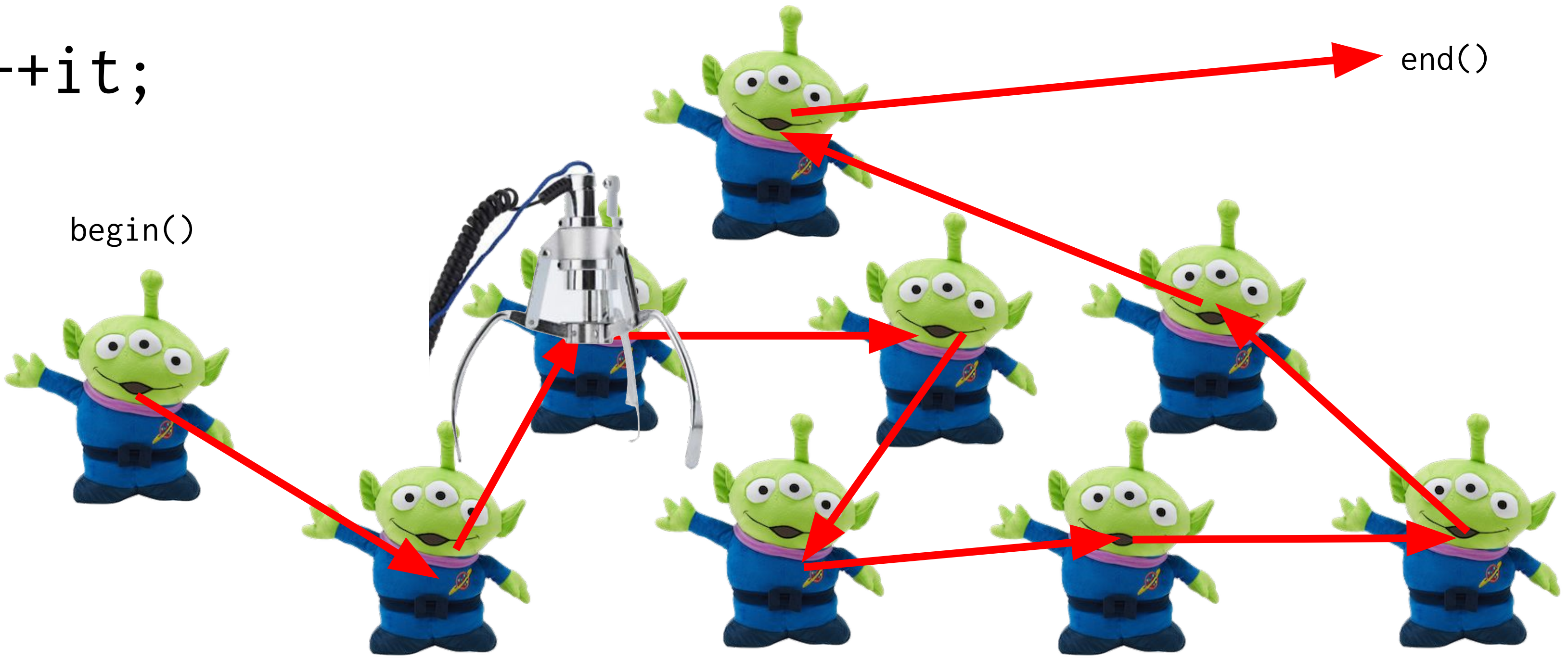
```
++it;
```



Key idea: iterator has ordering over elems

i.e. it always knows what the “next” element is

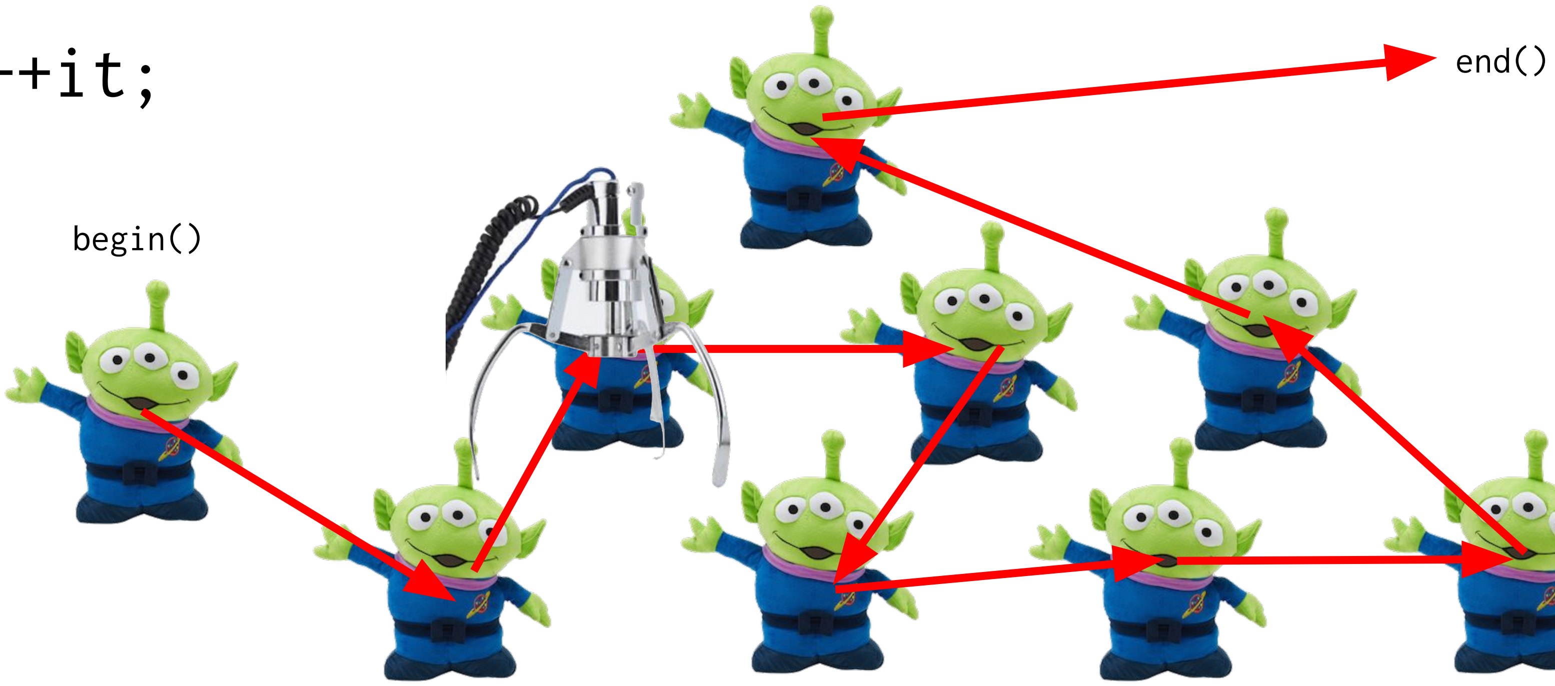
```
++it;
```



Key idea: iterator has ordering over elems

i.e. it always knows what the "next" element is

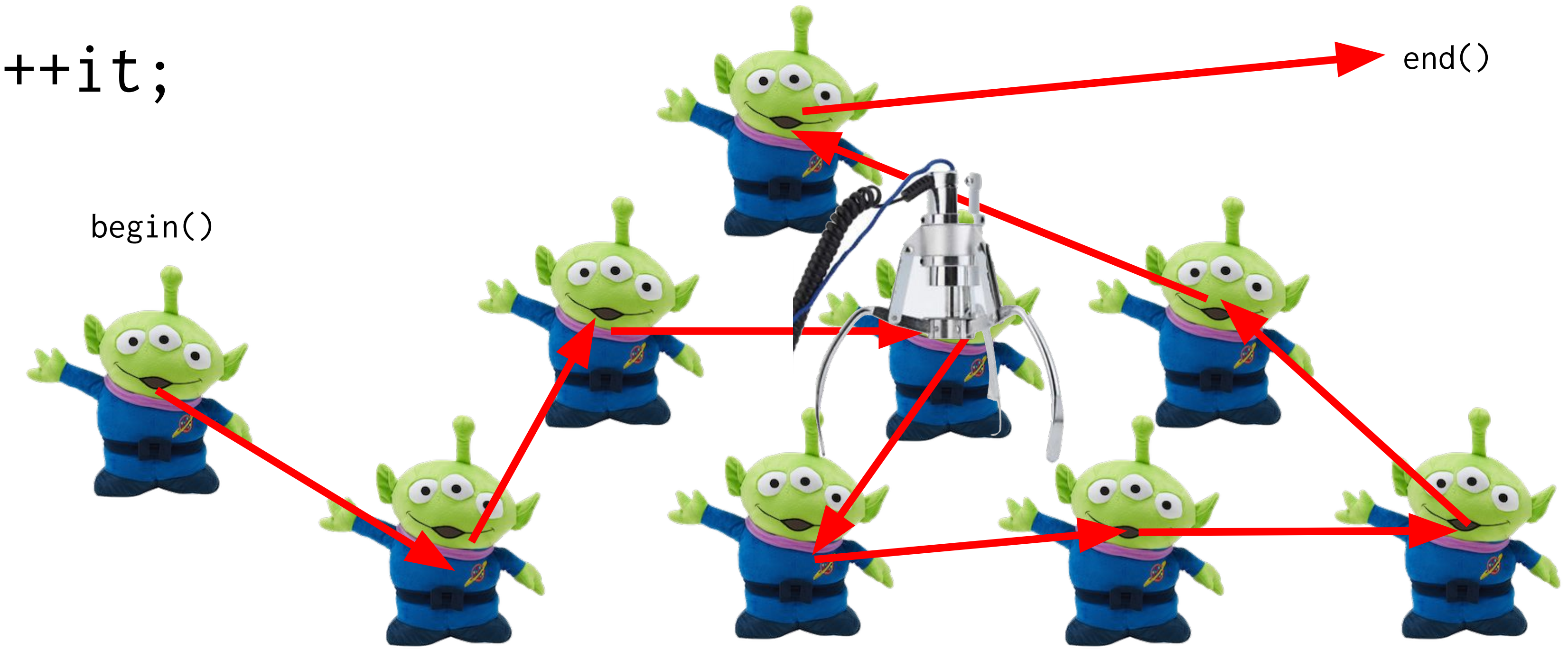
```
++it;
```



Key idea: iterator has ordering over elems

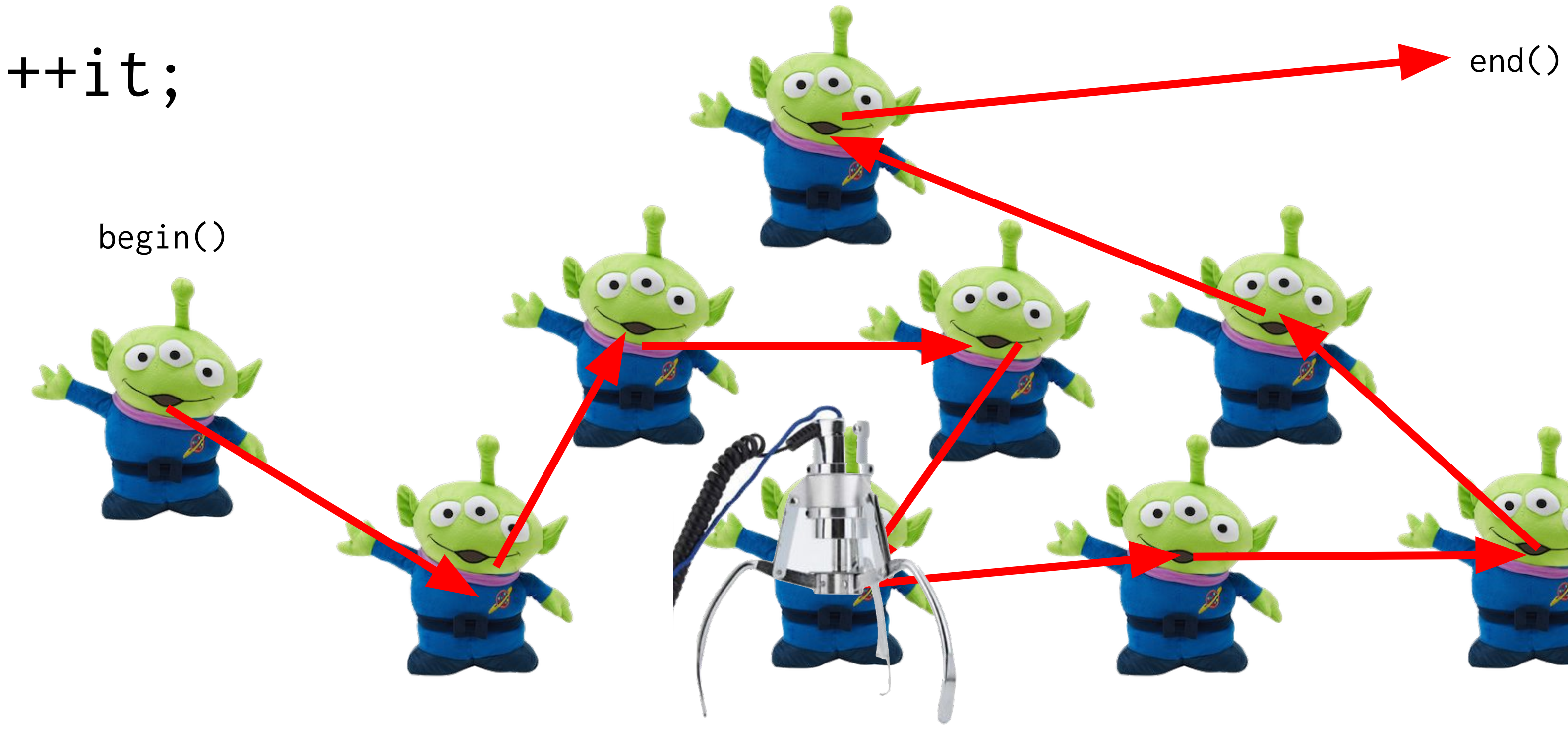
i.e. it always knows what the "next" element is

```
++it;
```



Key idea: iterator has ordering over elems

i.e. it always knows what the “next” element is

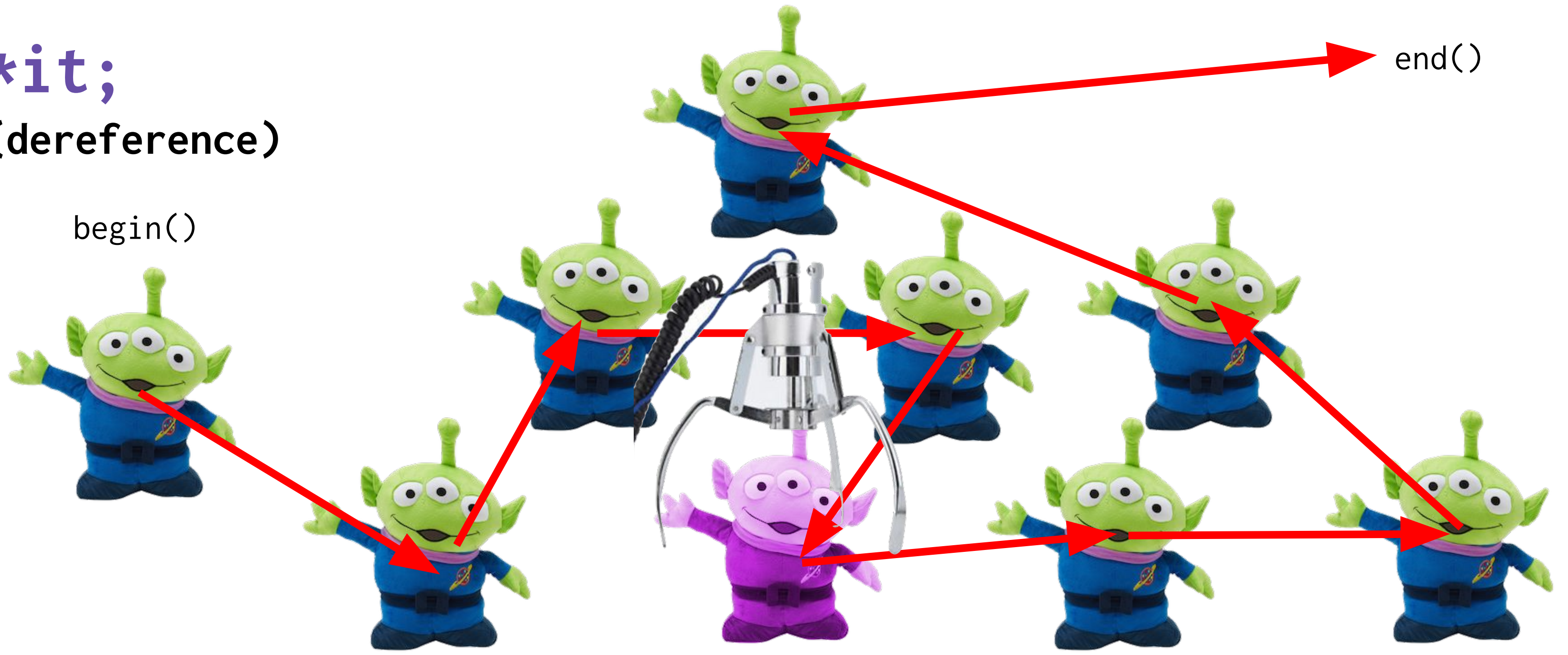


Key idea: iterator has ordering over elems

i.e. it always knows what the "next" element is

`*it;`

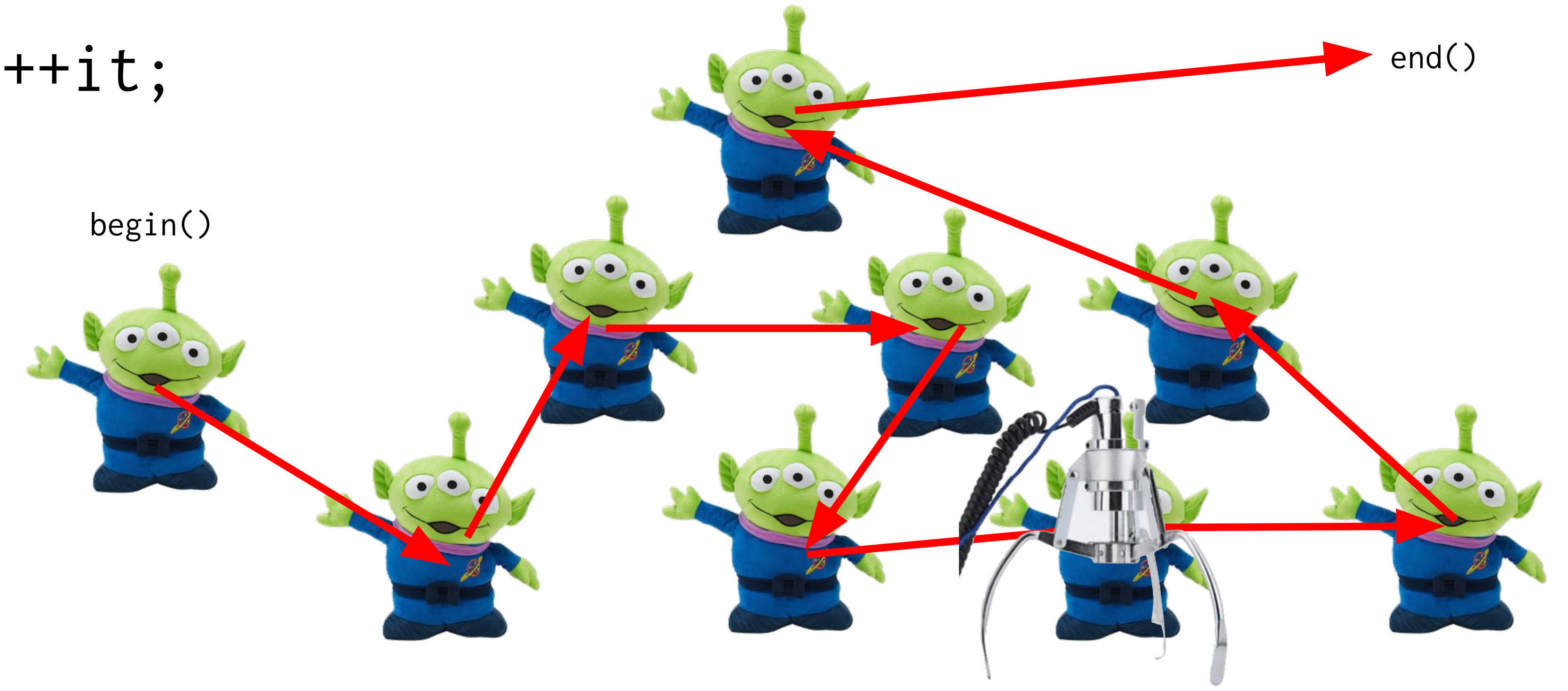
(dereference)



Key idea: iterator has ordering over elems

i.e. it always knows what the “next” element is

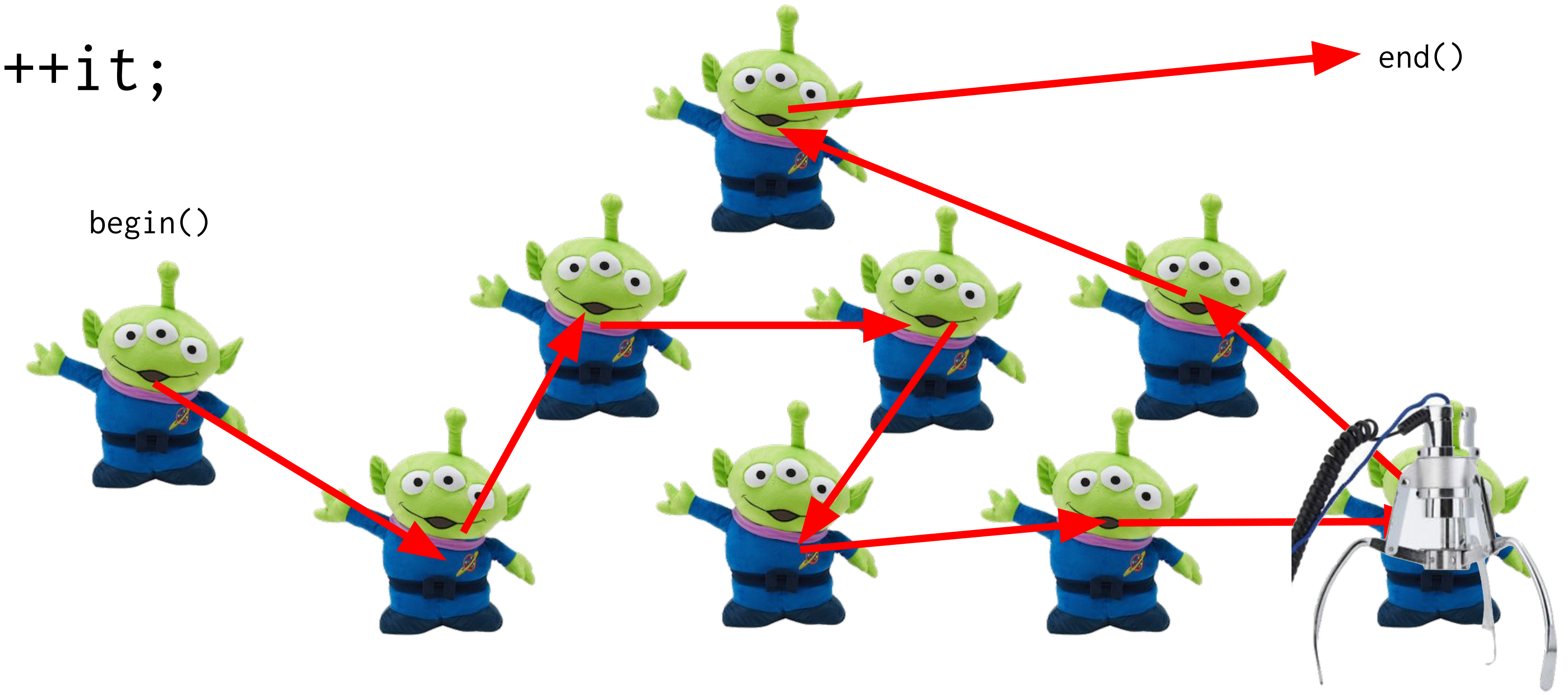
```
++it;
```



Key idea: iterator has ordering over elems

i.e. it always knows what the “next” element is

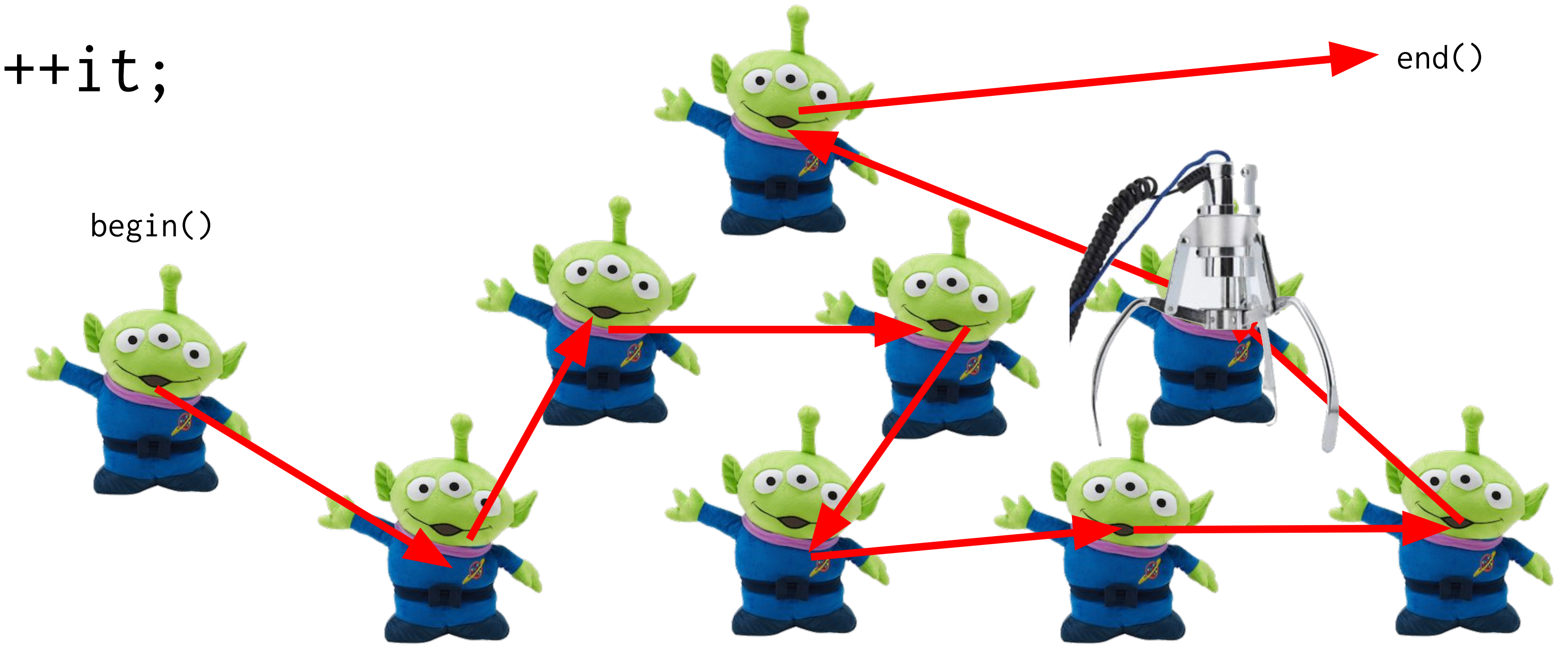
```
++it;
```



Key idea: iterator has ordering over elems

i.e. it always knows what the “next” element is

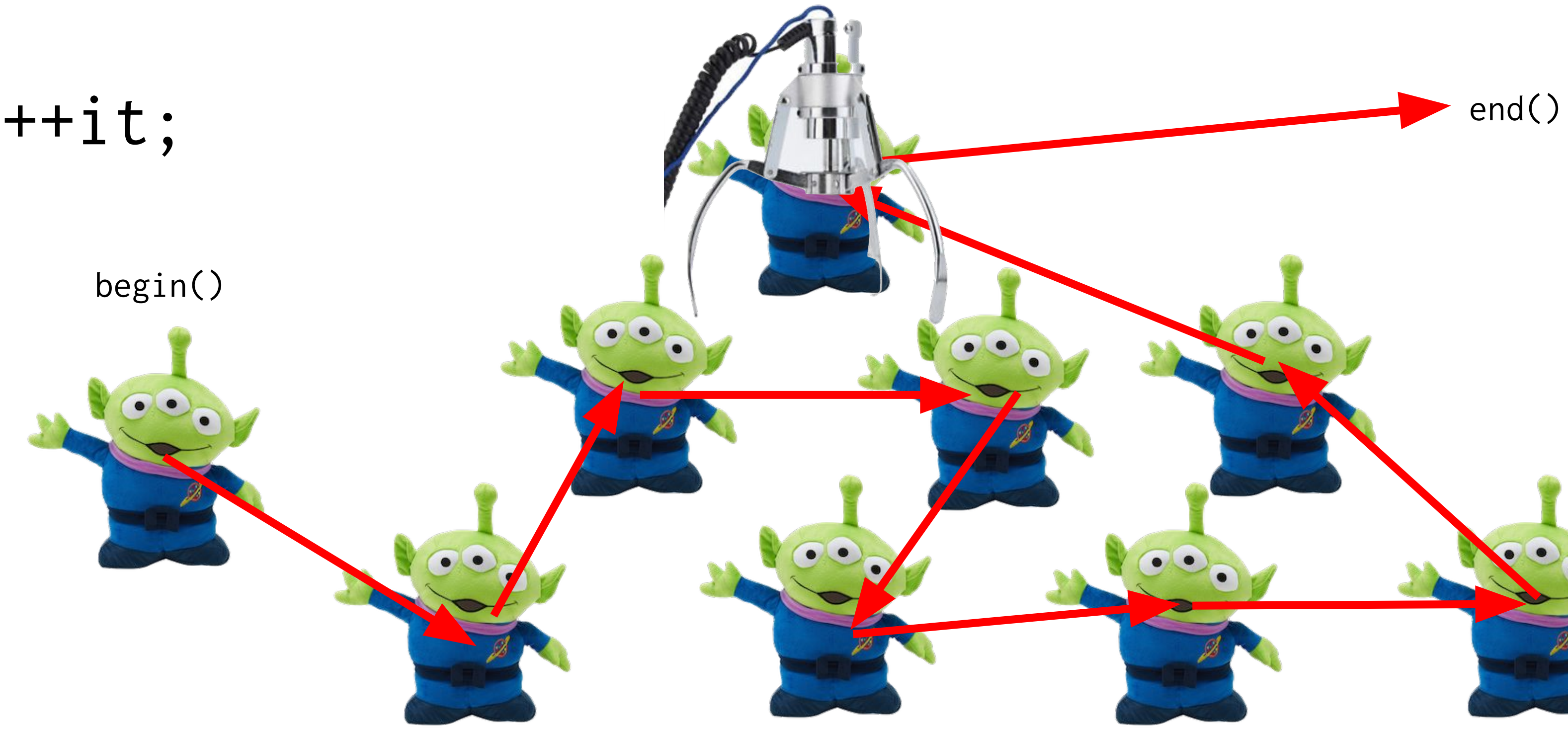
```
++it;
```



Key idea: iterator has ordering over elems

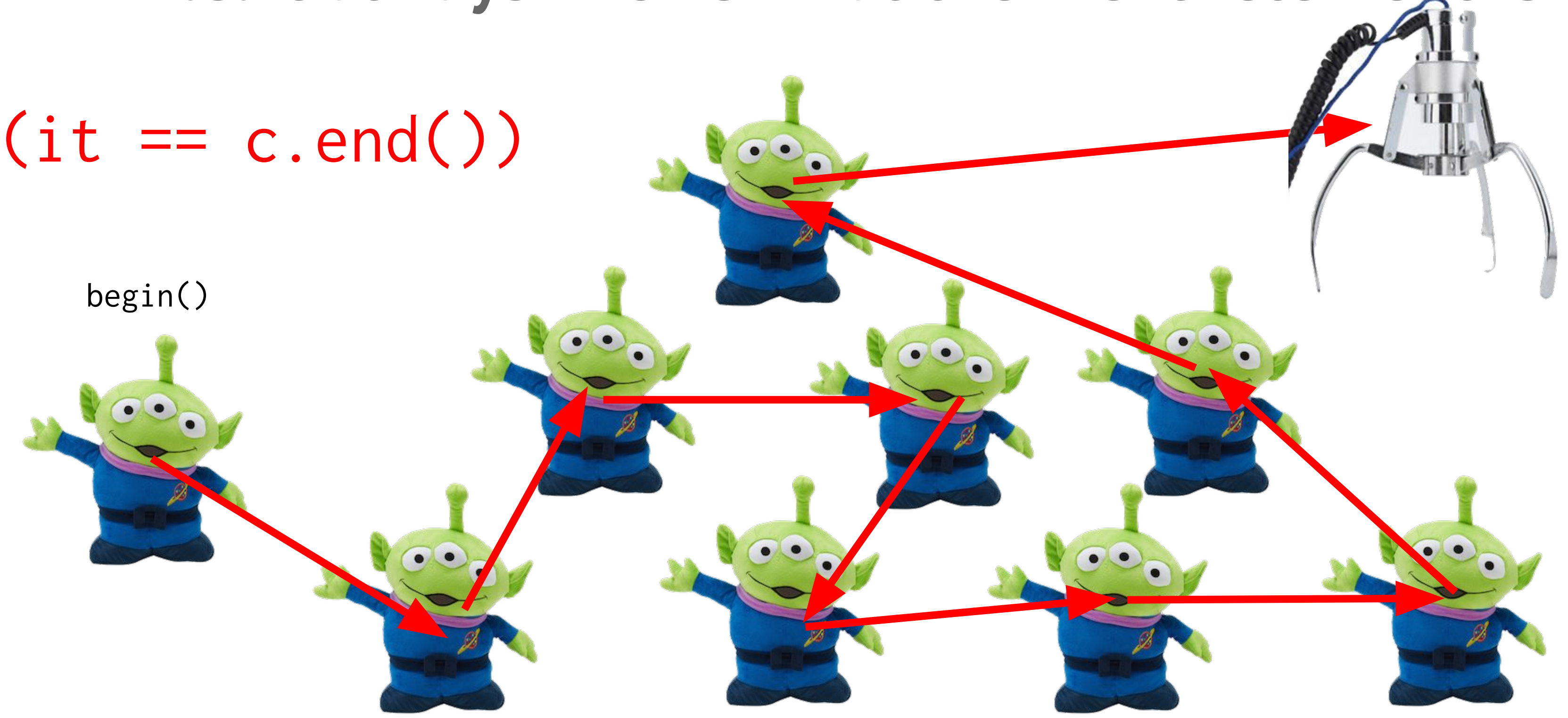
i.e. it always knows what the “next” element is

```
++it;
```



Key idea: iterator has ordering over elems

i.e. it always knows what the “next” element is



STL Iterators

Generally, STL iterators support the following operations:

```
std::set<T> s;
auto iter = s.begin();
iter++;                                  // increment; prefix operator is faster (why?)
*iter;                                   // dereference iter to get curr value
(iter != s.end());                       // equality comparison

iter = another_iter                      // copy construction
```

STL sets have the following operations:

```
s.begin();                               // an iterator pointing to the first element
s.end()                                  // one past the last element
```

Why use ++iter and not iter++?

Answer: ++iter returns the value *after* being incremented, so there's no need to store the old value of the iterator!

Iterator Practice

What type is this?

```
std::map<int, int> map {{1, 2}, {3, 4}};
auto it = map.first();                                  // what type is this?
auto map_elem = *it;                                    // how about this? guess in the chat!
```

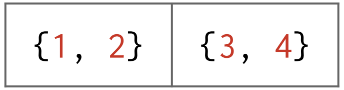
Quiz: Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
// note that dereferencing a std::map::iterator returns a std::pair
auto iter = map.begin();                      // what is *iter?
++iter;                                       // what is (*iter).second now?
auto iter2 = iter;
++iter;                                       // what does (*iter).first return?

// ++iter: go to the next element
// *iter:  retrieve what’s at iter’s position
// copy constructor: create another iterator pointing to same thing
```

Quiz: Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
// note that dereferencing a std::map::iterator returns a std::pair
auto iter = map.begin();                  // what is *iter?
++iter;                                   // what is (*iter).second now?
auto iter2 = iter;
++iter;                                   // what does (*iter).first return?

// ++iter: go to the next element
// *iter:  retrieve what's at iter's position
// copy constructor: create another iterator pointing to same thing
```

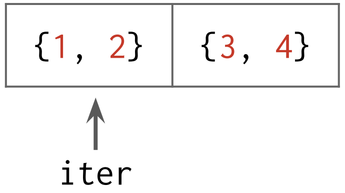
Declare a map.



Quiz: Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
// note that dereferencing a std::map::iterator returns a std::pair
auto iter = map.begin();                    // what is *iter?
++iter;                                     // what is (*iter).second now?
auto iter2 = iter;
++iter;                                     // what does (*iter).first return?

// ++iter: go to the next element
// *iter:  retrieve what's at iter's position
// copy constructor: create another iterator pointing to same thing
```

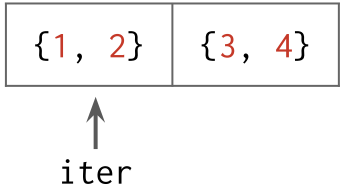
iter is a copy of begin iterator



Quiz: Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
// note that dereferencing a std::map::iterator returns a std::pair
auto iter = map.begin();                      // what is *iter?
++iter;                                       // what is (*iter).second now?
auto iter2 = iter;
++iter;                                       // what does (*iter).first return?

// ++iter: go to the next element
// *iter:  retrieve what's at iter's position
// copy constructor: create another iterator pointing to same thing
```

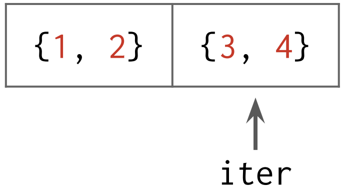
***iter returns {1, 2}**



Quiz: Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
// note that dereferencing a std::map::iterator returns a std::pair
auto iter = map.begin();                   // what is *iter?
++iter;                                    // what is (*iter).second now?
auto iter2 = iter;
++iter;                                    // what does (*iter).first return?

// ++iter: go to the next element
// *iter:  retrieve what's at iter's position
// copy constructor: create another iterator pointing to same thing
```

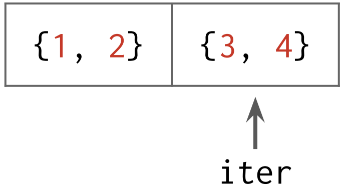
`*iter` incremented to next element



Quiz: Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
// note that dereferencing a std::map::iterator returns a std::pair
auto iter = map.begin();                      // what is *iter?
++iter;                                       // what is (*iter).second now?
auto iter2 = iter;
++iter;                                       // what does (*iter).first return?

// ++iter: go to the next element
// *iter:  retrieve what's at iter's position
// copy constructor: create another iterator pointing to same thing
```

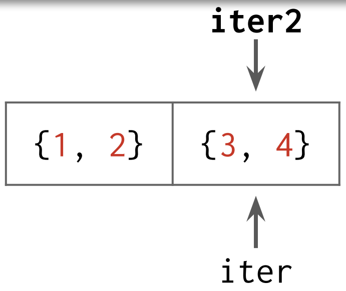
`*iter.second` is 4



Quiz: Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
// note that dereferencing a std::map::iterator returns a std::pair
auto iter = map.begin();                          // what is *iter?
++iter;                                           // what is (*iter).second now?
auto iter2 = iter;
++iter;                                           // what does (*iter).first return?

// ++iter: go to the next element
// *iter:  retrieve what's at iter's position
// copy constructor: create another iterator pointing to same thing
```

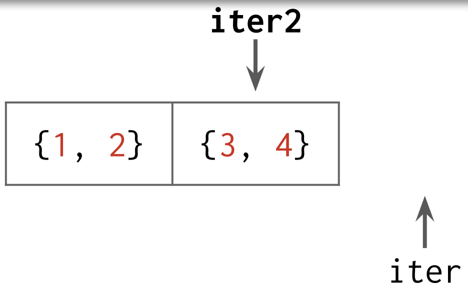
create an independent copy of `iter` pointing to same thing



Quiz: Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
// note that dereferencing a std::map::iterator returns a std::pair
auto iter = map.begin();                    // what is *iter?
++iter;                                     // what is (*iter).second now?
auto iter2 = iter;
++iter;                                     // what does (*iter).first return?

// ++iter: go to the next element
// *iter:  retrieve what's at iter's position
// copy constructor: create another iterator pointing to same thing
```

increment `iter`
(`iter2` not impacted...)

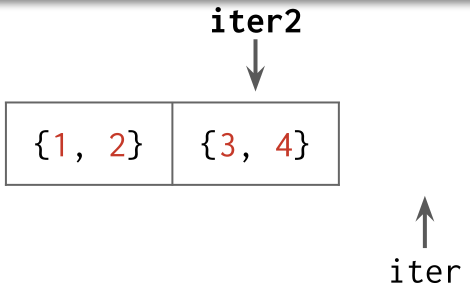


Quiz: Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
// note that dereferencing a std::map::iterator returns a std::pair
auto iter = map.begin();                    // what is *iter?
++iter;                                     // what is (*iter).second now?
auto iter2 = iter;
++iter;                                     // what does (*iter).first return?

// ++iter: go to the next element
// *iter:  retrieve what's at iter's position
// copy constructor: create another iterator pointing to same thing
```

undefined!

(iter == map.end())



Exercise: print all elements in these collections

Fill in the blanks in chat! Should be the same for set/map.

```
std::set<int> set {3, 1, 4, 1, 5, 9};
for (initialization; termination-condition; increment) {
  const auto& elem = retrieve-element;
  cout << elem << endl;
}

std::map<int> map {{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (initialization; termination-condition; increment) {
  const auto& [key, value] = retrieve-element;  // structured binding!
  cout << key << ":" << value << endl;
}
```

Exercise: print all elements in these collections

Fill in the blanks in chat! Should be the same for set/map.

```
std::set<int> set {3, 1, 4, 1, 5, 9};
for (auto iter = set.begin(); iter != set.end(); ++iter) {
  const auto& elem = *iter;
  cout << elem << endl;
}

std::map<int> map {{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (auto iter = map.begin(); iter != map.end(); ++iter) {
  const auto& [key, value] = *iter; // structured binding!
  cout << key << “:” << value << endl;
}
```

What?
Iterator is evolving!

Exercise: print all elements in these collections

🎉 You discovered **For-Each Loop!**

```
std::set<int> set {3, 1, 4, 1, 5, 9};
for (const auto& elem : set) {

  cout << elem << endl;
}


std::map<int> map {{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (const auto& [key, value] : map) {

  cout << key << “:” << value << endl;
}
```

Iterator Shorthand

These are equivalent:

```
auto key = (*iter).first;
auto key = iter->first;
```

We'll find out more as to why this exists under "Pointers" in CS106B.

Types of Iterators

Types of Iterators

- All iterators are **incrementable** (++)
- **Input** iterators can be on the right side of =:

```
auto elem = *it;
```

- **Output** iterators can be on the left side of =:

```
*elem = value;
```

- **Forward** iterators can be traversed multiple times:

```
iterator a;
b = a;
a++; b++;
assert (*a == *b)          // true
```

Can you think of an example of an iterator that *should not* be a forward iterator?

Types of Iterators

- **Random access** iterators support indexing by integers!

```
it += 3;                          // move forward 3
it -= 70;                         // move backwards by 70
auto elem = it[5];                // offset by 5
```