Iterators

CS 106L Spring 2020 – Avery Wang and Anna Zeng
Stanford University

22 April 2020

Review Quiz: Which of the following are true?

1. If you need a sequence container, use vector over deque, unless you need to insert to front.

2. stack and queues are container adaptors. Internally, they are implemented using a deque.

3. The std::map<K, V> internally stores std::pairs with member types K and V.

4. The std::set<T> requires a comparison operator defined over T .

Bonus (just fun facts, you aren't expected to know these)

5. The Stanford Vector<T> is a container adaptor for std::vector<T>, with the exception of Vector<bool> which is an adaptor for std::deque<bool>.

6. The Stanford Set<T> is a container adaptor for std::set<T>.

Game Plan

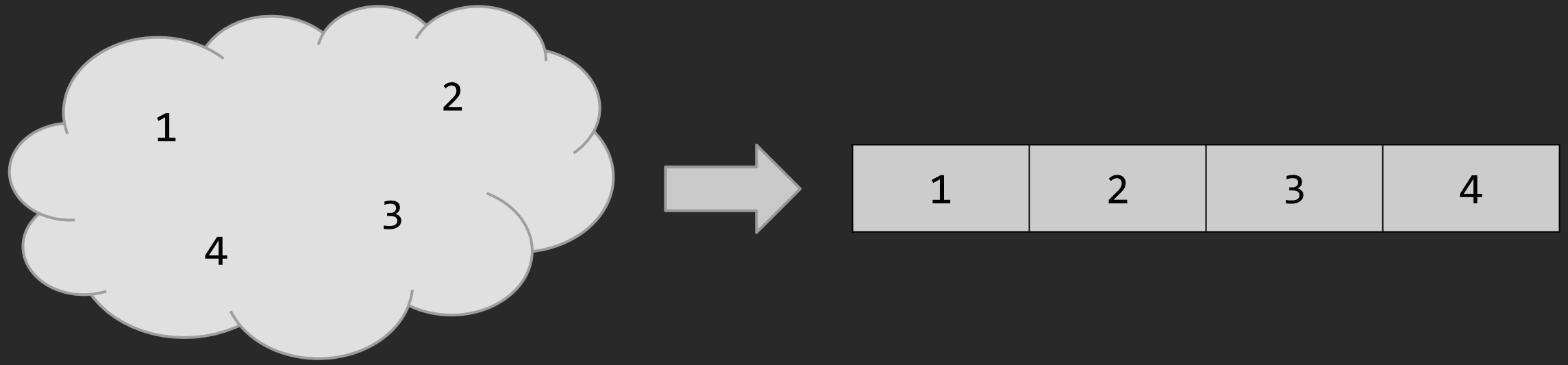
- basics
- iterator categories
- iterator invalidation

Iterator Basics

Iterators

Iterators allow iteration over any container,
whether it is ordered or not.

Iterators allow us to view a non-linear container as if it was a linear one.



How to print the elements of a collection?

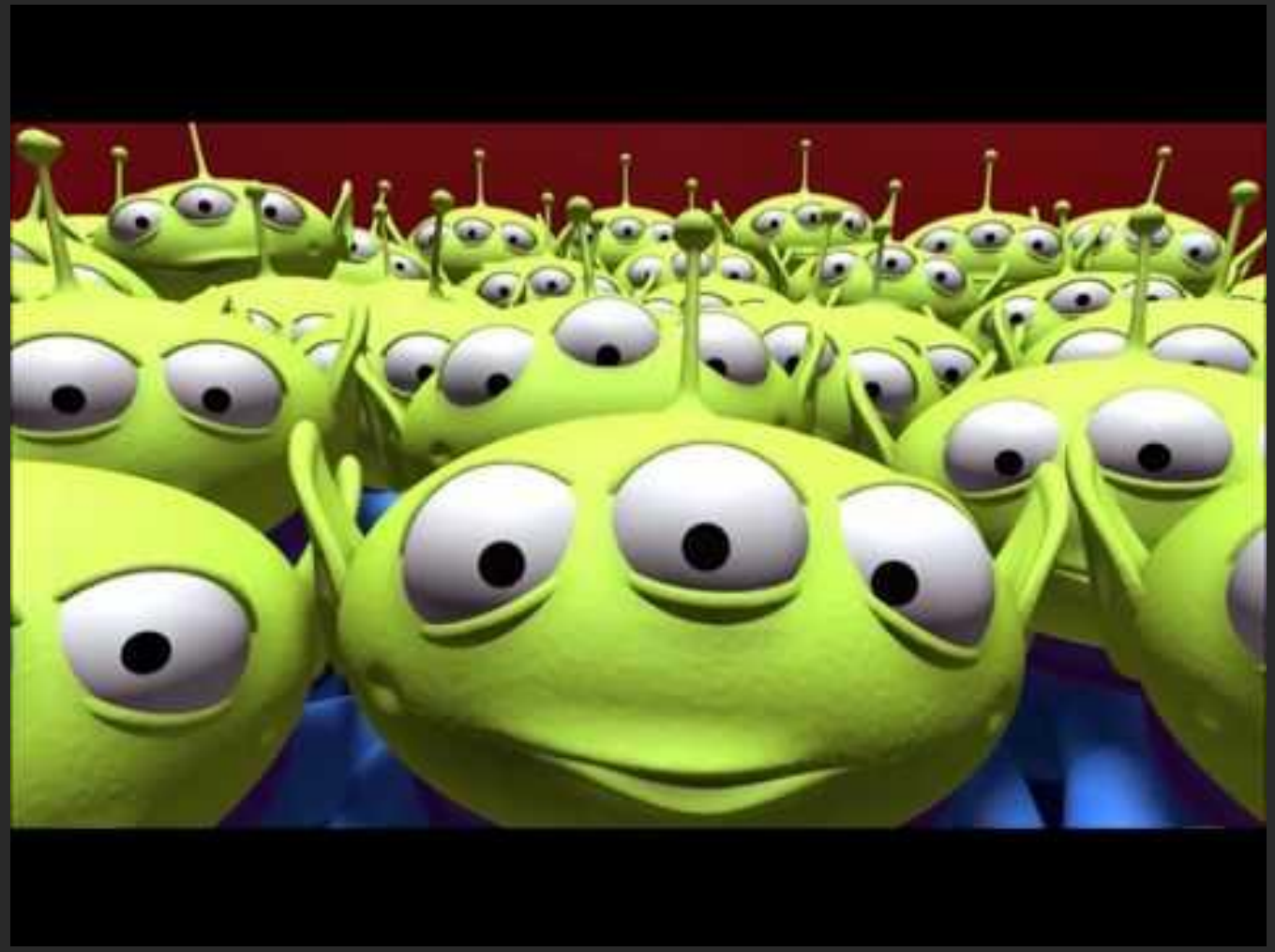
Type out the four blanks for vector in the chat.

```
1.  std::vector<int> vector{3, 1, 4, 1, 5, 9};
2.  for (initialization; termination condition; increment) {
3.    const auto& elem = retrieve element;
4.    cout << elem << endl;
5.  }
6.
7.  std::set<int> set{3, 1, 4, 1, 5, 9};
8.  for (initialization; termination condition; increment) {
9.    const auto& elem = retrieve element;
10.   cout << elem << endl;
11. }
```

How do we print all the elements of a collection?

```
1.  std::vector<int> vector{3, 1, 4, 1, 5, 9};
2.  for (size_t i = 0; i < vector.size(); ++i) {
3.    const auto& elem = vector[i];
4.    cout << elem << endl;
5.  }
6.
7.  std::set<int> set{3, 1, 4, 1, 5, 9};
8.  for (     uhh;   errr;     maybe++?) {
9.    const auto& elem = that thingy;
10.   cout << elem << endl;
11. }
```

How do we loop through an associative container?



An iterator is like "the claw".

Iterators ("the claw") support some key operations:

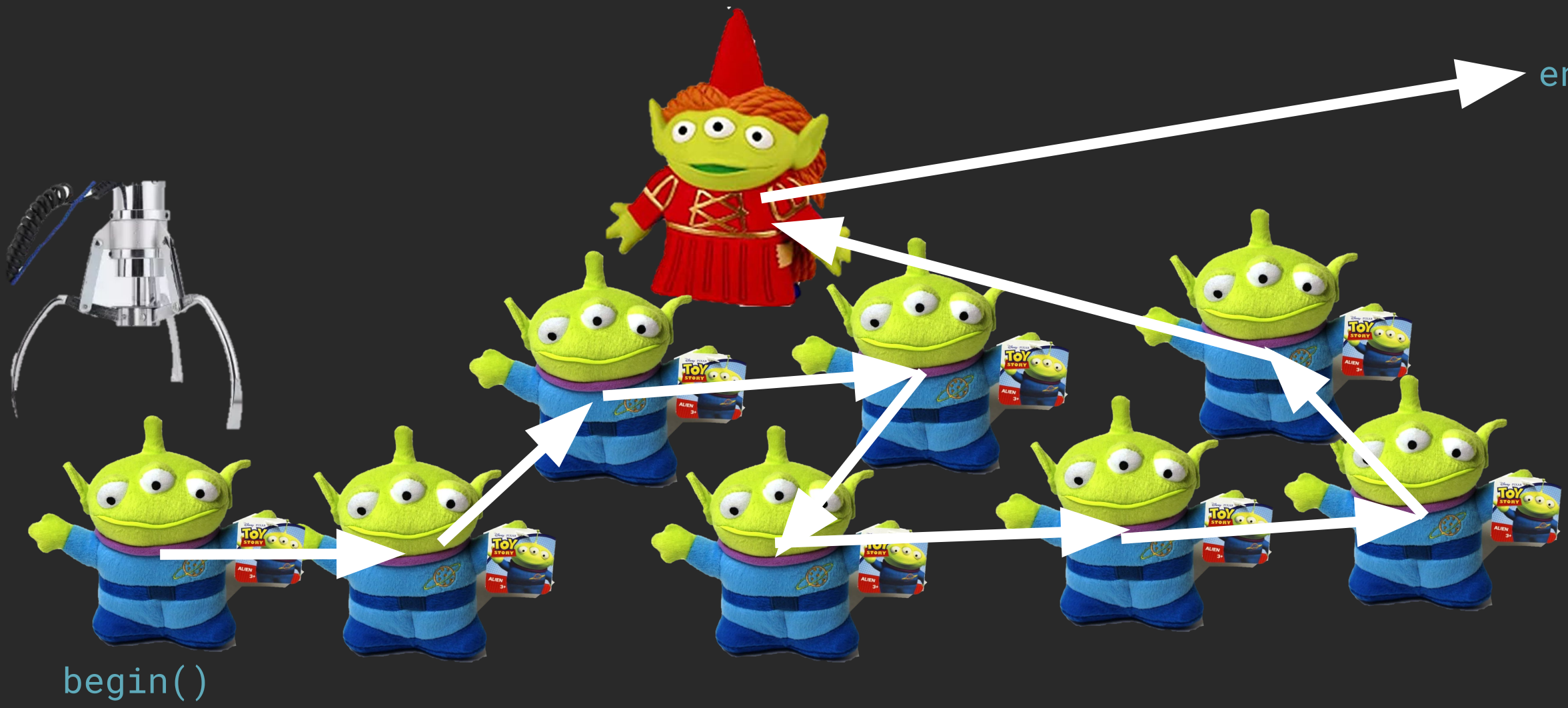
- Move "forward"
- Retrieve element
- Check if two claws are at the same place

Containers ("the machine") support iterators with:

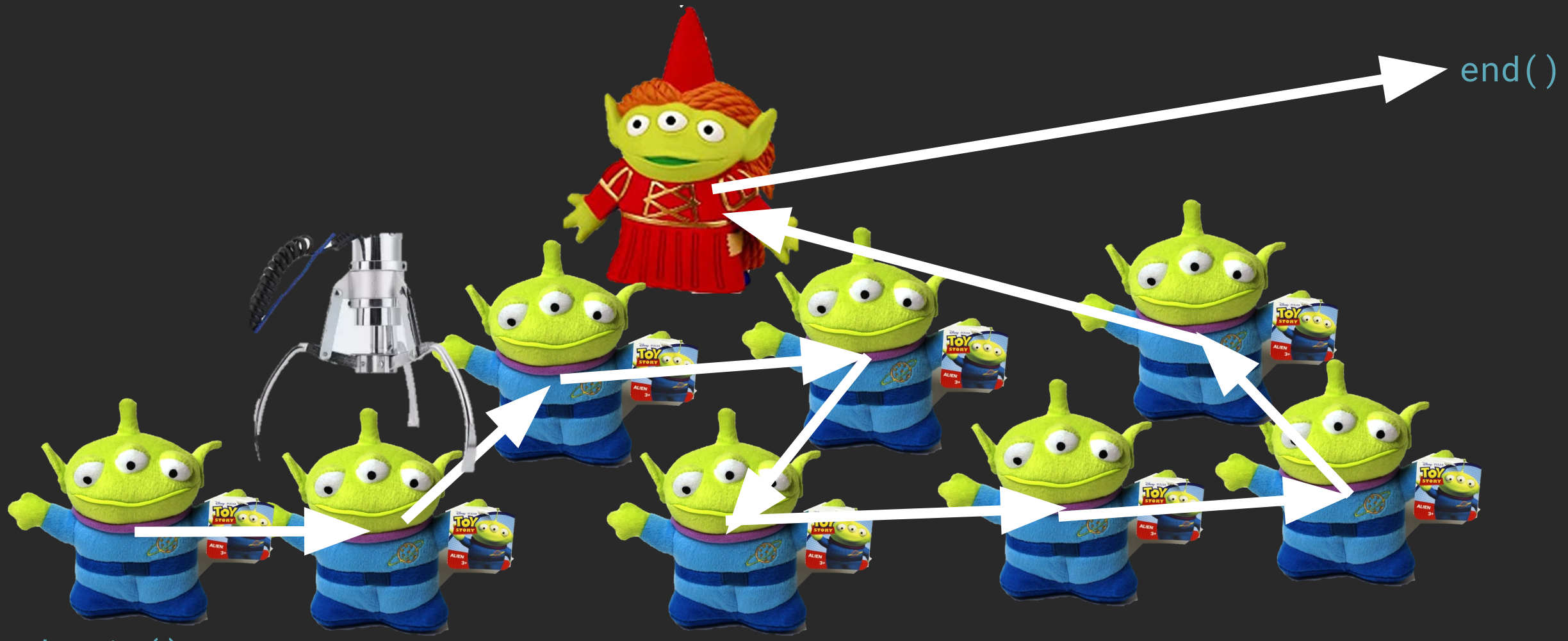
- the bounds (the begin and end iterators)

Key idea: iterator knows where the next element is, even if the elements are stored haphazardly.

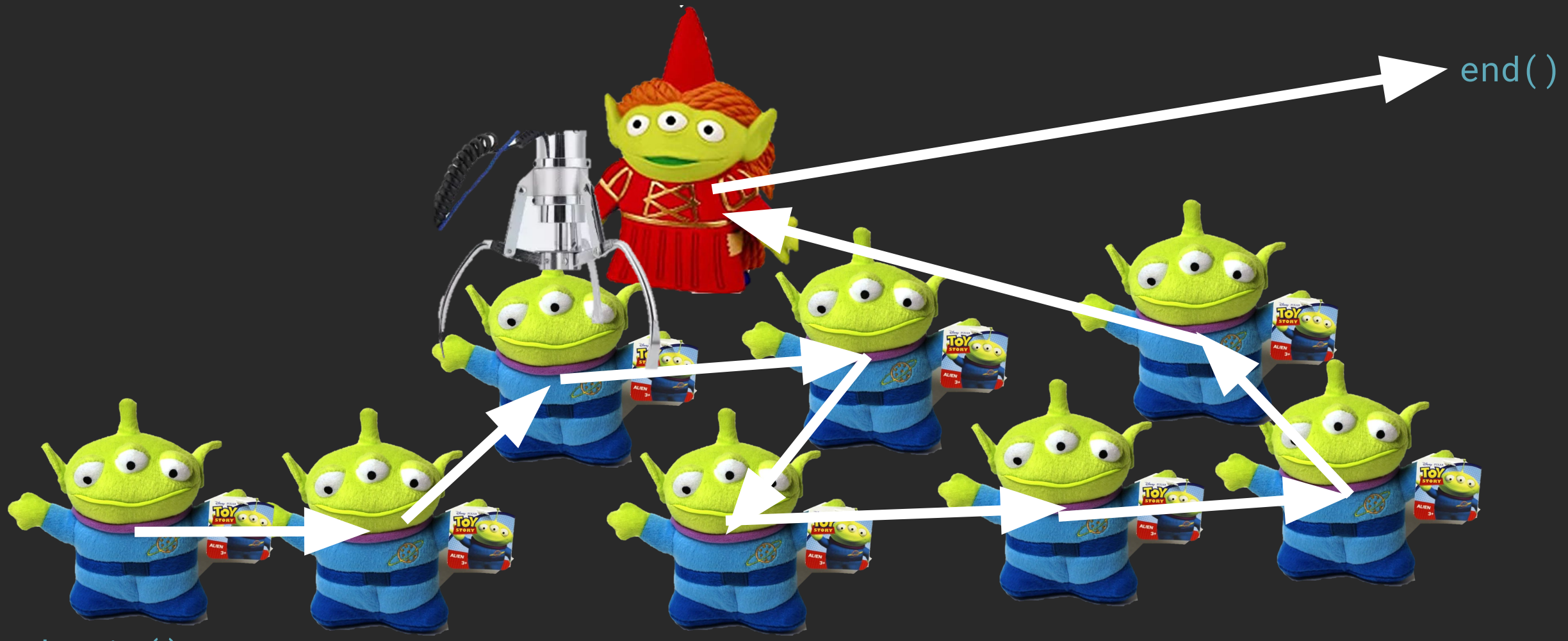
Key idea: iterator knows where the next element is, even if the elements are stored haphazardly.



Key idea: iterator knows where the next element is, even if the elements are stored haphazardly.

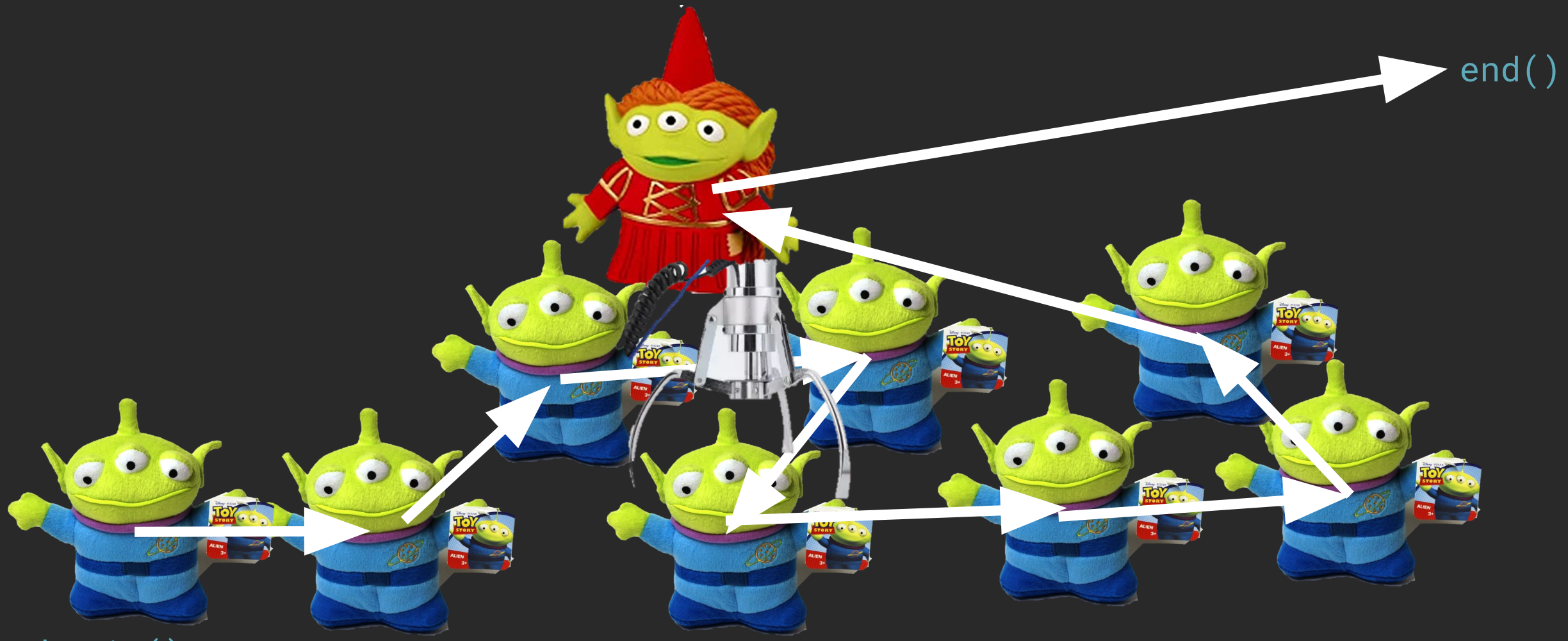


Key idea: iterator knows where the next element is, even if the elements are stored haphazardly.

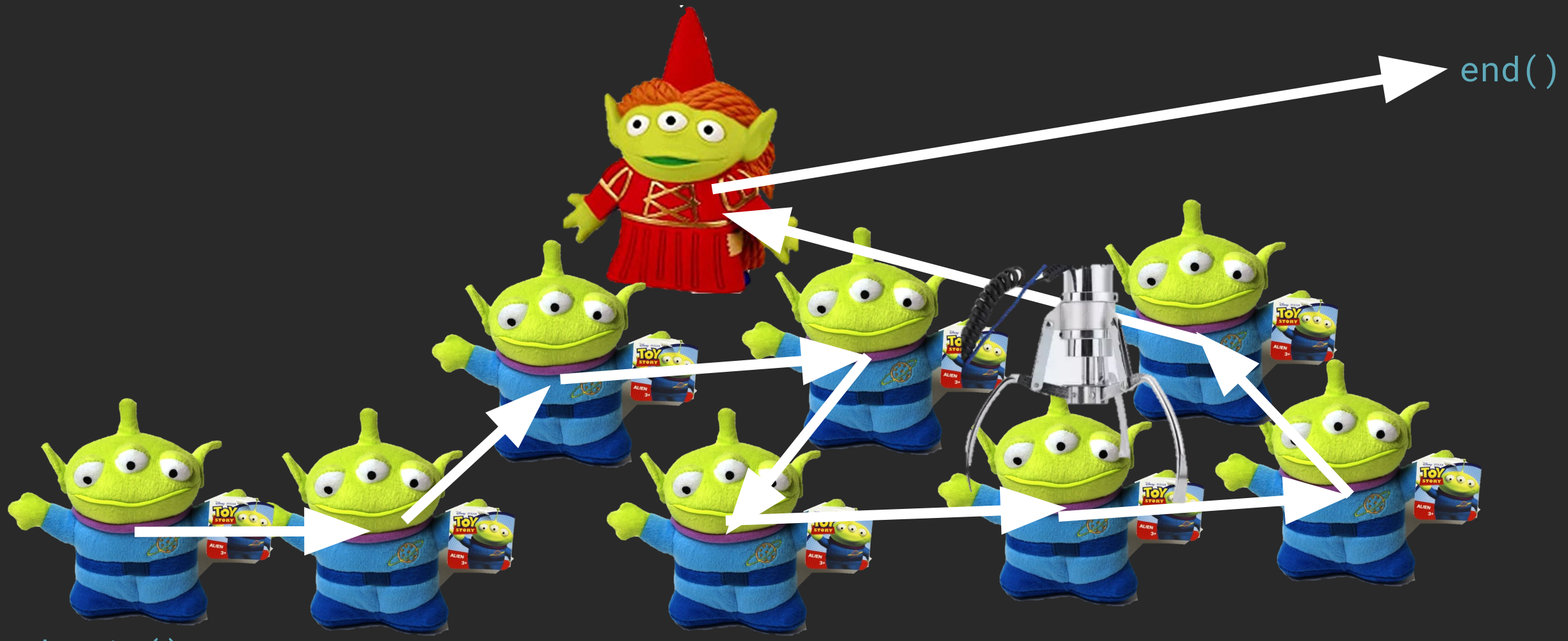


Key idea: iterator knows where the next element is, even if the elements are stored haphazardly.

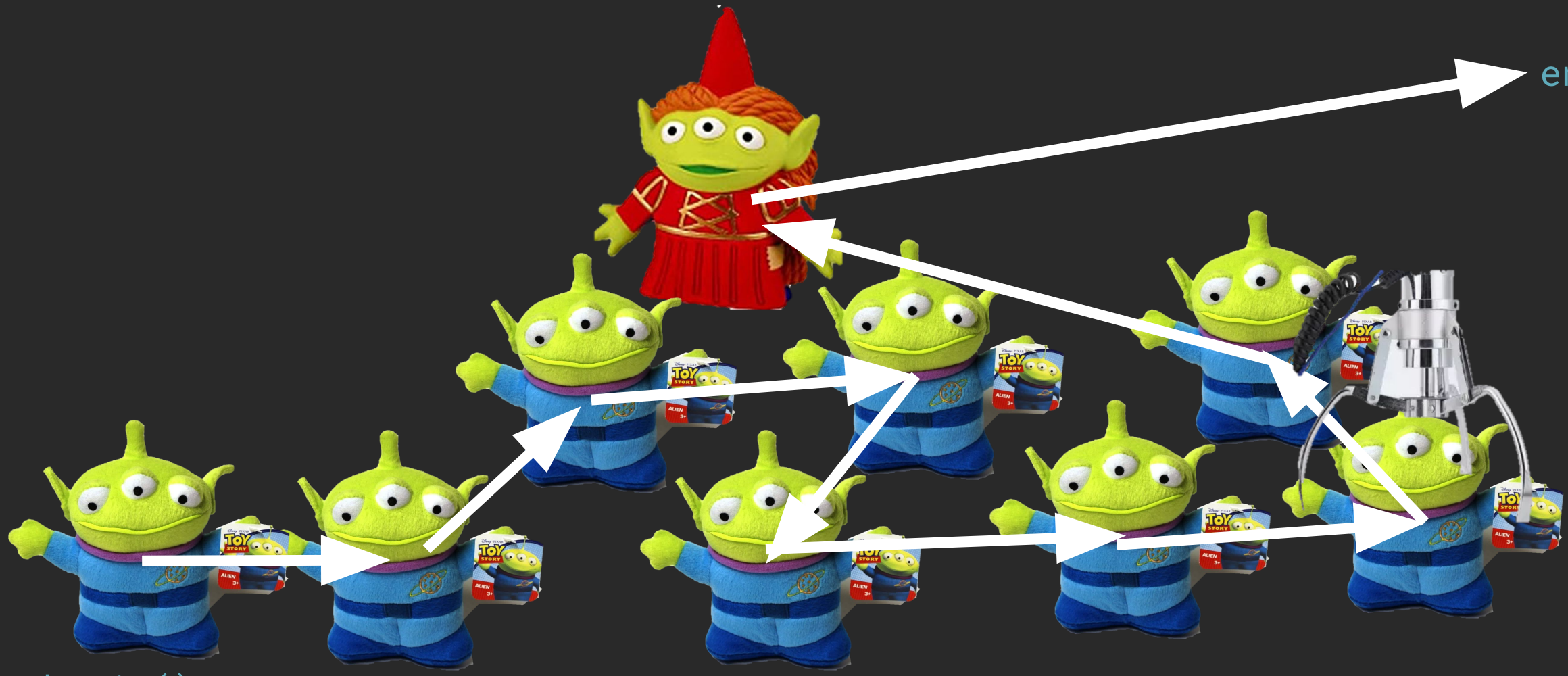
Key idea: iterator knows where the next element is, even if the elements are stored haphazardly.



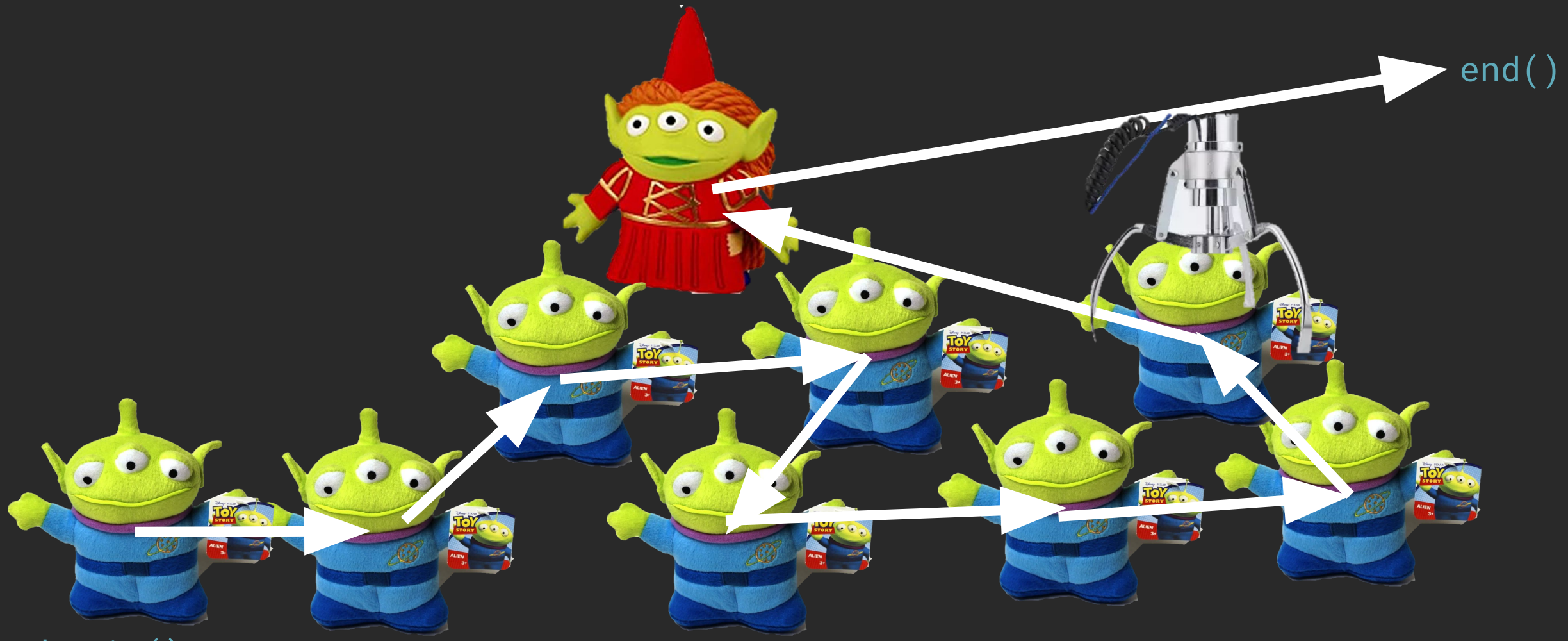
Key idea: iterator knows where the next element is, even if the elements are stored haphazardly.



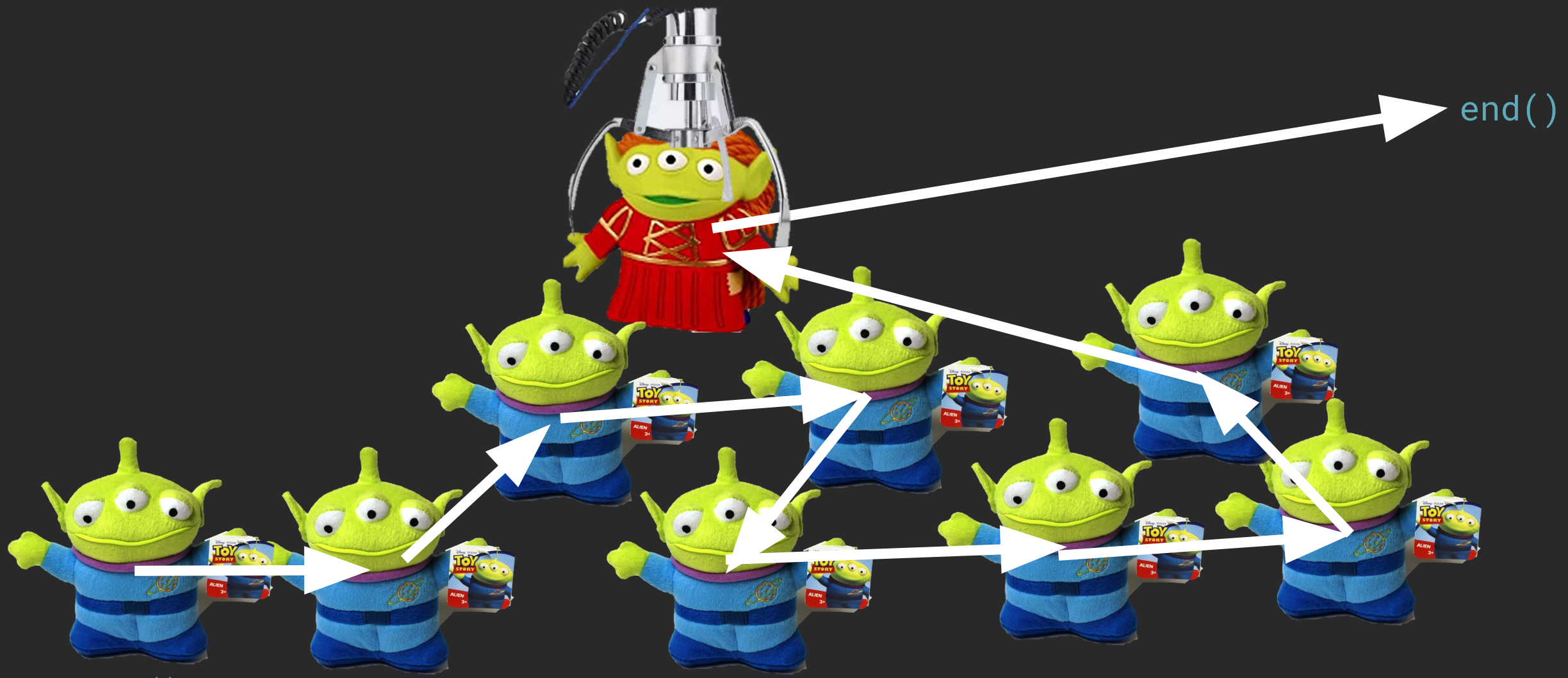
Key idea: iterator knows where the next element is, even if the elements are stored haphazardly.



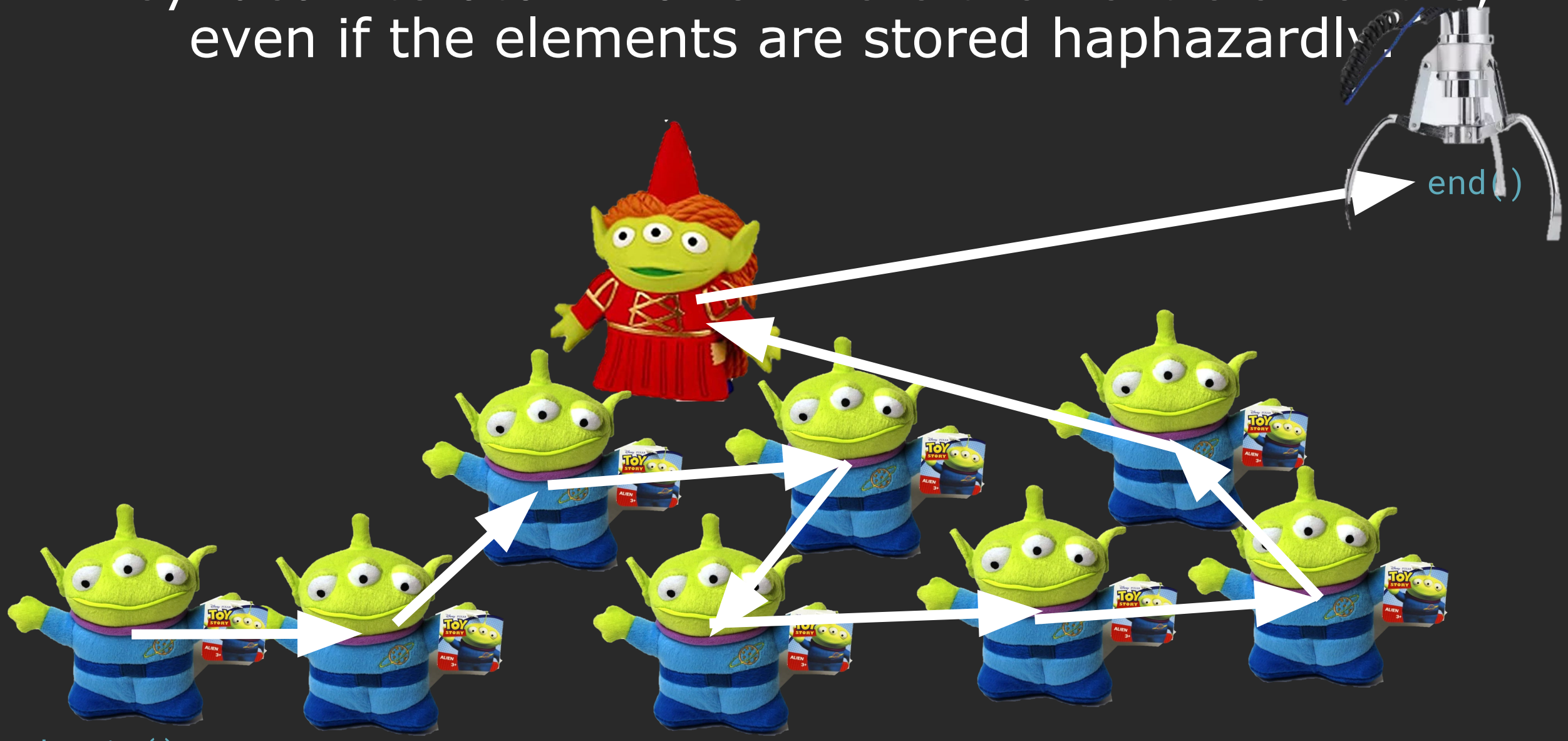
Key idea: iterator knows where the next element is, even if the elements are stored haphazardly.



Key idea: iterator knows where the next element is, even if the elements are stored haphazardly.



Key idea: iterator knows where the next element is, even if the elements are stored haphazardly



Let's iterate through all of the aliens that are scattered throughout our machine.

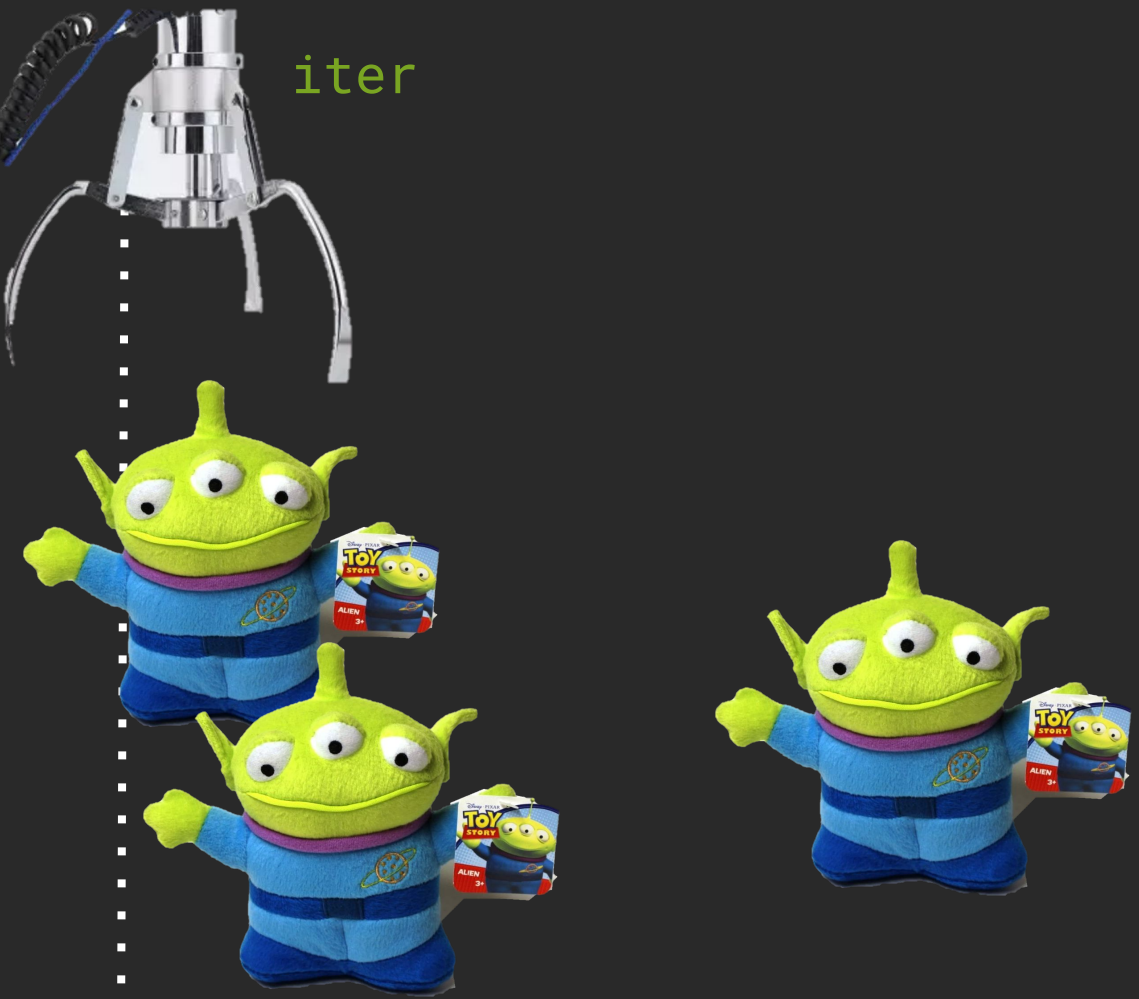
begin()

end()

Just like in a claw machine, the prizes in our set<Alien> are not laid out in a linear fashion.

```
std::set<Alien> machine;
```

The containers tells us where the begin iterator is.

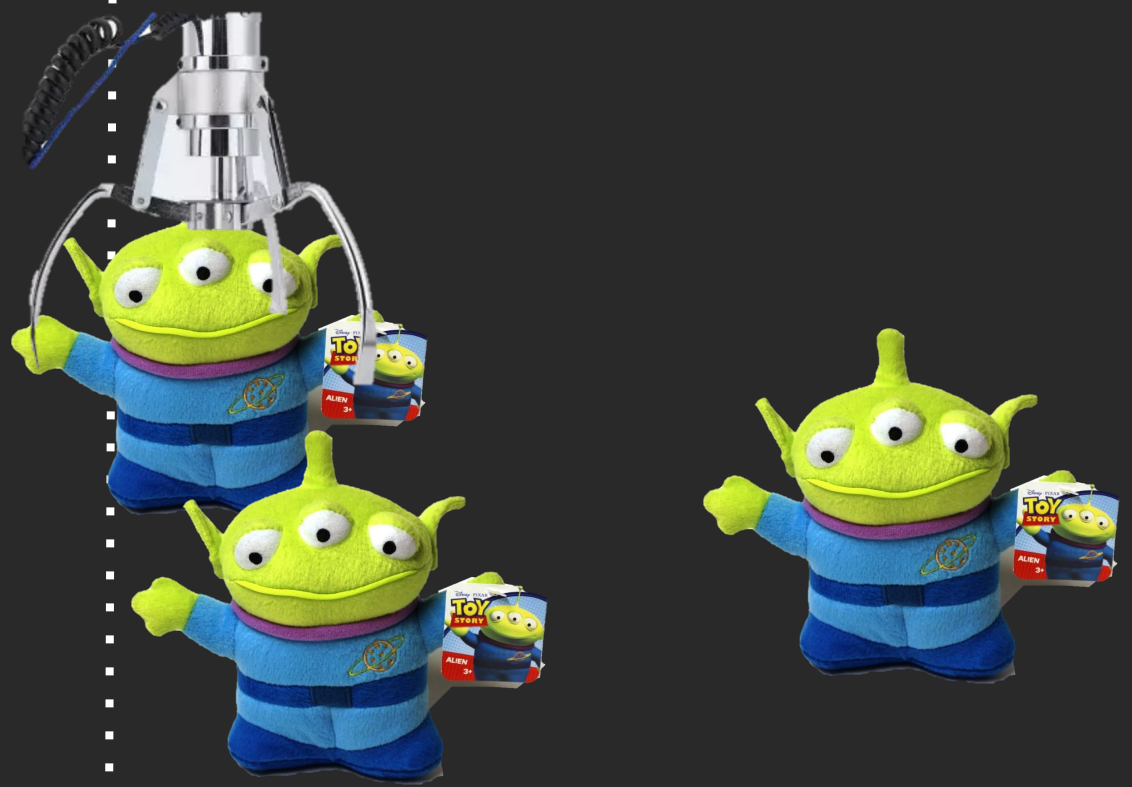


Our machine tells us where the claw initially starts.

```
auto iter = machine.begin();
```

We can retrieve the element using the dereference (*) operator.

iter



begin()

end()

We can retrieve the Alien right below the claw.

```
Alien a1 = *iter;
```

Sometimes, you can even overwrite the element the iterator is pointing to.

iter

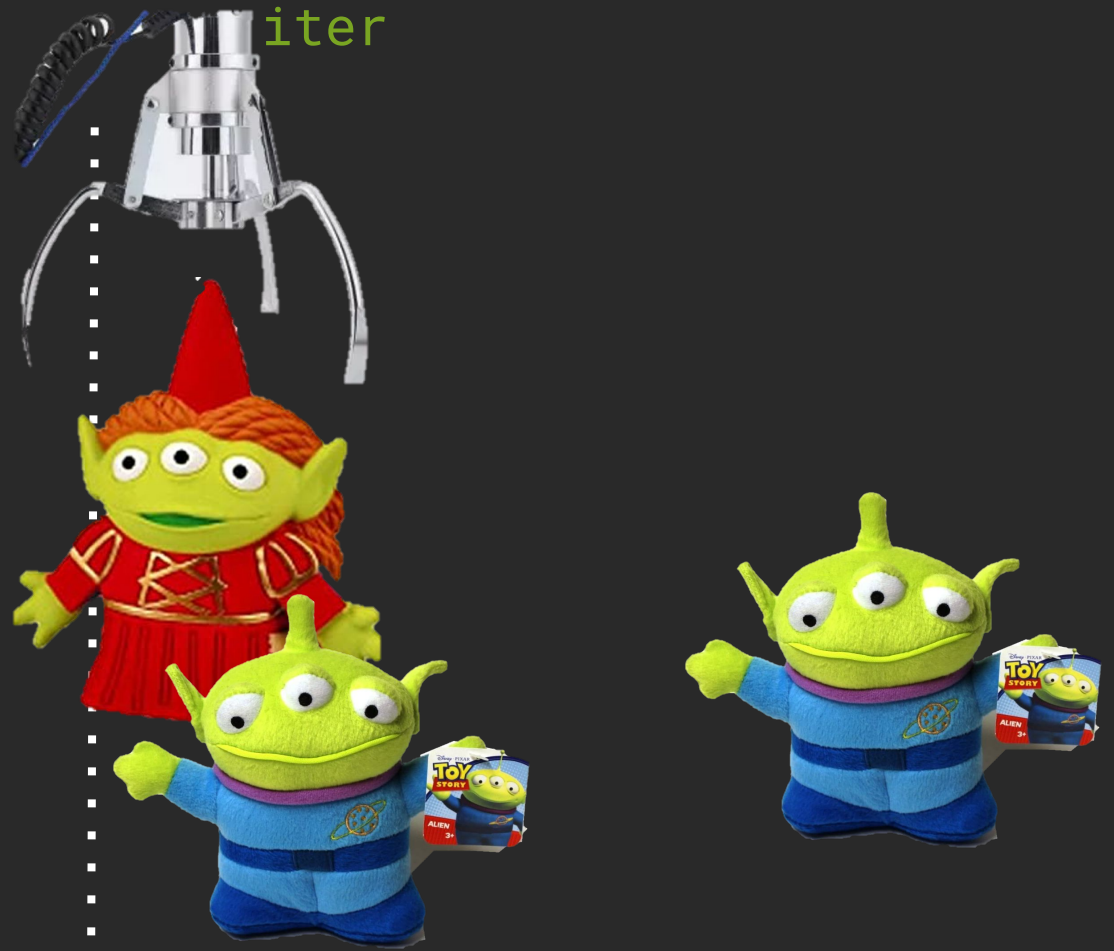
begin()

end()

This might be a bit of a stretch for our analogy...at least the alien is cute.

```
*iter = redAlien;
```

How do we get to the next element?



Remember, we don't know where the next element is stored!

`iter?`

We can get to the next element by using the increment (++) operator.

We can move the claw to the next Alien.

```
++iter;
```

We can retrieve the element using the dereference (*) operator.

iter

begin()

end()

Let's grab the next Alien!

```
Alien a2 = *iter;
```

We can get to the next element by using the increment (++) operator.

We can move the claw to the next Alien.

```
++iter;
```

We can get to the next element by using the increment (++) operator.

We can move the claw to the next Alien.

```
++iter;
```

We can retrieve the element using the dereference (*) operator.

Let's grab the next Alien!

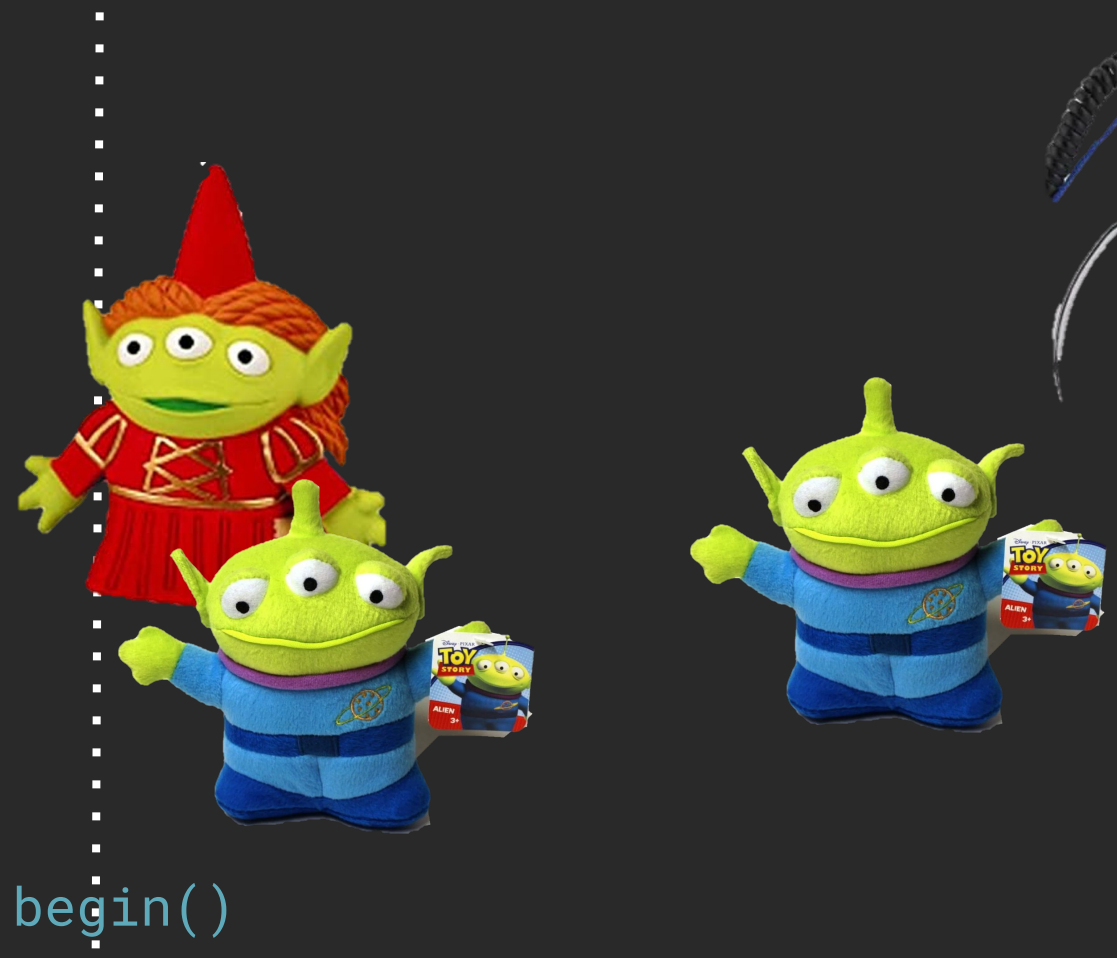
```
Alien a3 = *iter;
```

We can get to the next element by using the increment (++) operator.

We can move the claw to the next Alien.

```
++iter;
```

We can get to the next element by using the increment (++) operator.



We can move the claw to the next Alien.

```
++iter;
```

To check if we've reached the end of the container, we can compare to end().

iter

Did we reach the end of the machine?
Let's double check with the machine.

```
if (iter == machine.end()) return;
```

begin()

end()

An iterator is like "the claw".

The STL iterators support the following operations:

- Copy constructible and assignable (`iter = another_iter`)
- Retrieve current element (`*iter`)
- Advance iterator (`++iter`)
- Equality comparable (`iter != container.end()`)

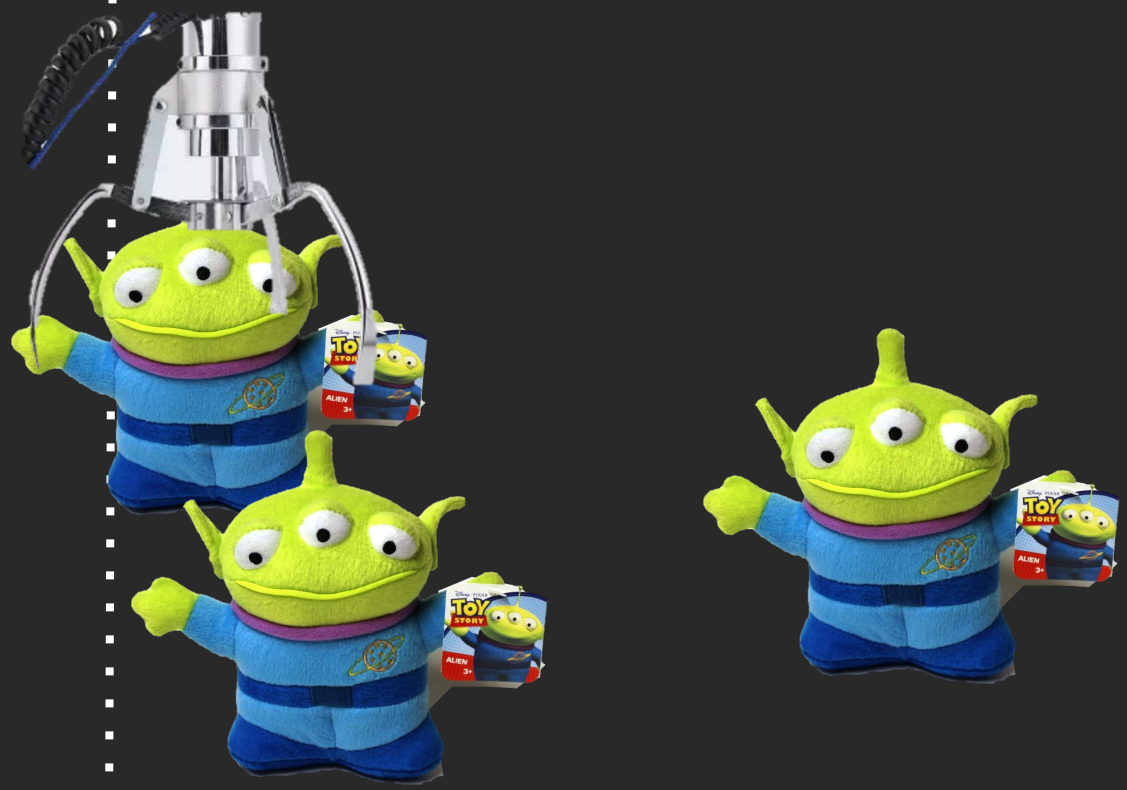
The STL containers support the following operations:

- `begin()` - iterator to the first element
- `end()` - iterator one past the last element

Important! Dereferencing or advancing the end iterator is undefined behavior.

Why did we use this example?

iter



begin()

end()

Switch the Aliens with elements, and you got yourself a Container with Iterators!

iter

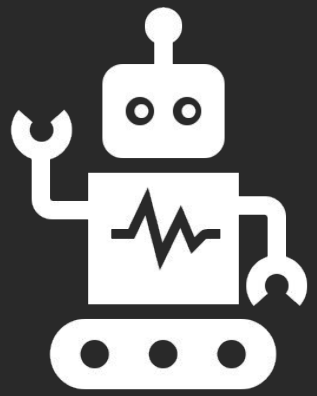
{1, 2}

{3, 4}

{5, 6}

begin()

end()



Questions

Answer a ton of questions.

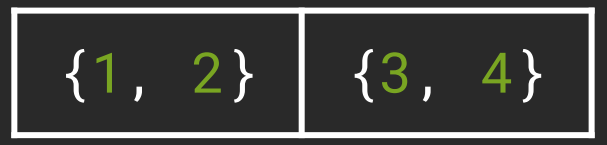
Poll Quiz! Iterator basics.

```
1.  std::map<int, int> map{{1, 2}, {3, 4}};
2.  auto iter = map.begin();
3.  // what is *iter at this line?
4.  ++iter;
5.  // what is (*iter).second at this line?
6.  auto iter2 = iter;
7.  ++iter;
8.  // what is (*iter).first at this line?
9.  // what is *iter2 at this line?

Recall:
- *iter = get what iter is pointing to
- ++iter = go to the next element
- copy = create an another iterator pointing to same thing
```

Declares the map.

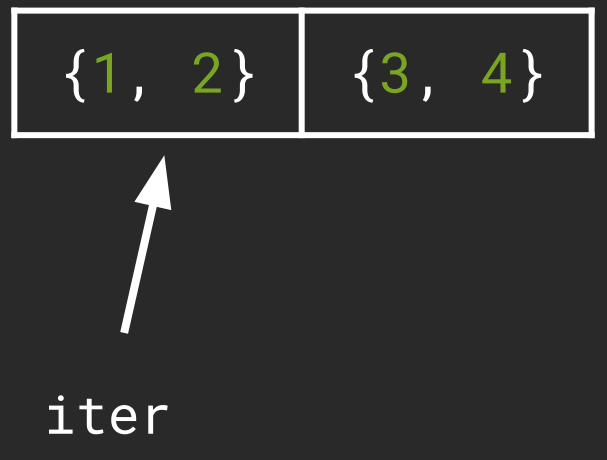
```
1.  std::map<int, int> map{{1, 2}, {3, 4}};
2.  auto iter = map.begin();
3.  // what is *iter at this line?
4.  ++iter;
5.  // what is (*iter).second at this line?
6.  auto iter2 = iter;
7.  ++iter;
8.  // what is (*iter).first at this line?
9.  // what is *iter2 at this line?
```



Recall:
- *iter = get what iter is pointing to
- ++iter = go to the next element
- copy = create an another iterator pointing to same thing

iter is a copy of begin iterator

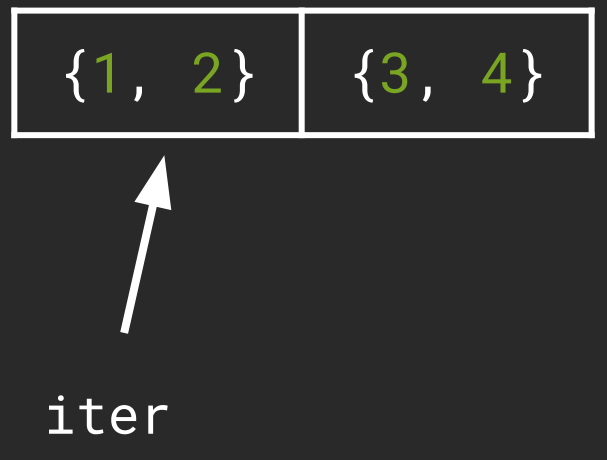
```
1.  std::map<int, int> map{{1, 2}, {3, 4}};
2.  auto iter = map.begin();
3.  // what is *iter at this line?
4.  ++iter;
5.  // what is (*iter).second at this line?
6.  auto iter2 = iter;
7.  ++iter;
8.  // what is (*iter).first at this line?
9.  // what is *iter2 at this line?
```



Recall:
- *iter = get what iter is pointing to
- ++iter = go to the next element
- copy = create an another iterator pointing to same thing

*iter returns the underlying element, which is a pair {1, 2}

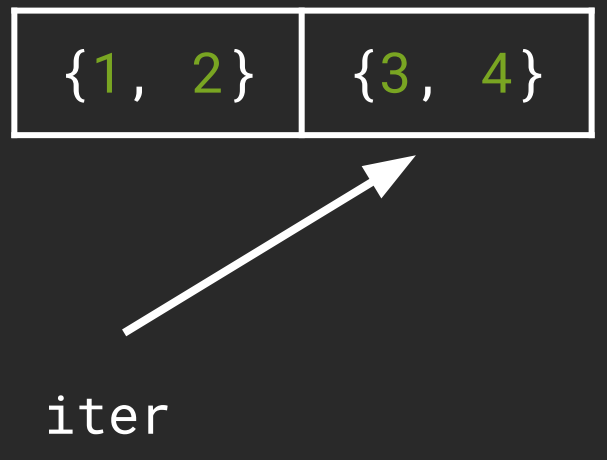
```
1.  std::map<int, int> map{{1, 2}, {3, 4}};
2.  auto iter = map.begin();
3.  // what is *iter at this line?
4.  ++iter;
5.  // what is (*iter).second at this line?
6.  auto iter2 = iter;
7.  ++iter;
8.  // what is (*iter).first at this line?
9.  // what is *iter2 at this line?
```



Recall:
- *iter = get what iter is pointing to
- ++iter = go to the next element
- copy = create an another iterator pointing to same thing

iter incremented to next element

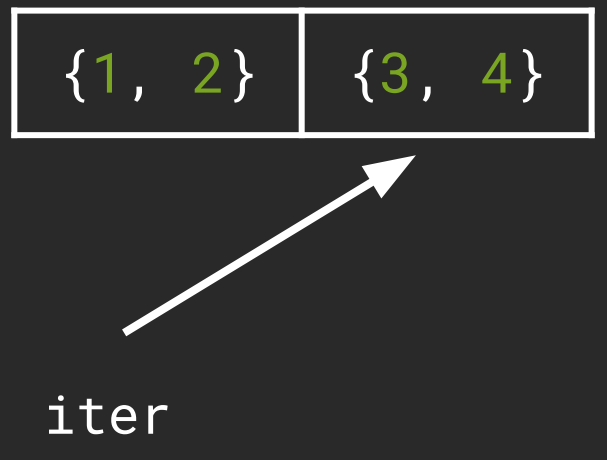
```
1.  std::map<int, int> map{{1, 2}, {3, 4}};
2.  auto iter = map.begin();
3.  // what is *iter at this line?
4.  ++iter;
5.  // what is (*iter).second at this line?
6.  auto iter2 = iter;
7.  ++iter;
8.  // what is (*iter).first at this line?
9.  // what is *iter2 at this line?
```



Recall:
- *iter = get what iter is pointing to
- ++iter = go to the next element
- copy = create an another iterator pointing to same thing

dereference, then access the second field of the pair, which is 4

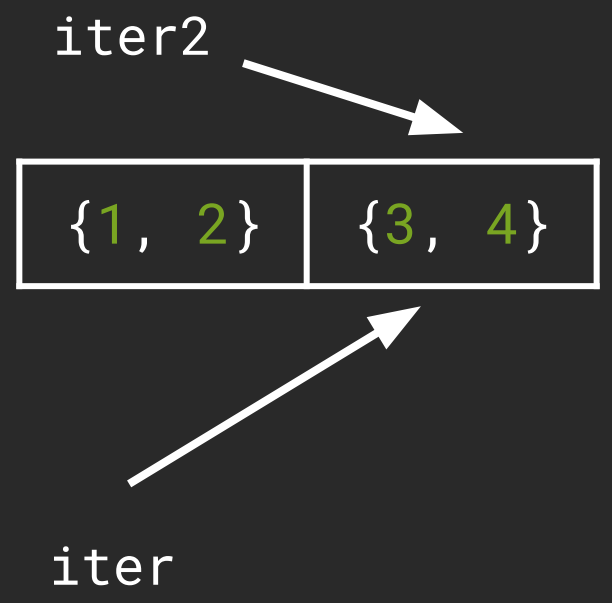
```
1.  std::map<int, int> map{{1, 2}, {3, 4}};
2.  auto iter = map.begin();
3.  // what is *iter at this line?
4.  ++iter;
5.  // what is (*iter).second at this line?
6.  auto iter2 = iter;
7.  ++iter;
8.  // what is (*iter).first at this line?
9.  // what is *iter2 at this line?
```



Recall:
- *iter = get what iter is pointing to
- ++iter = go to the next element
- copy = create an another iterator pointing to same thing

creates an independent copy of iter pointing to same thing

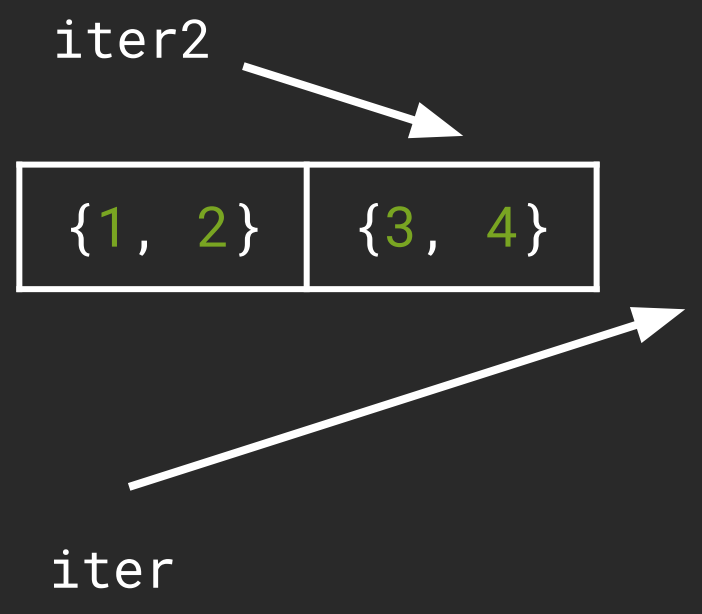
```
1.  std::map<int, int> map{{1, 2}, {3, 4}};
2.  auto iter = map.begin();
3.  // what is *iter at this line?
4.  ++iter;
5.  // what is (*iter).second at this line?
6.  auto iter2 = iter;
7.  ++iter;
8.  // what is (*iter).first at this line?
9.  // what is *iter2 at this line?
```




```
Recall:
- *iter = get what iter is pointing to
- ++iter = go to the next element
- copy = create an another iterator pointing to same thing
```

increments iter, notice iter2 not impacted

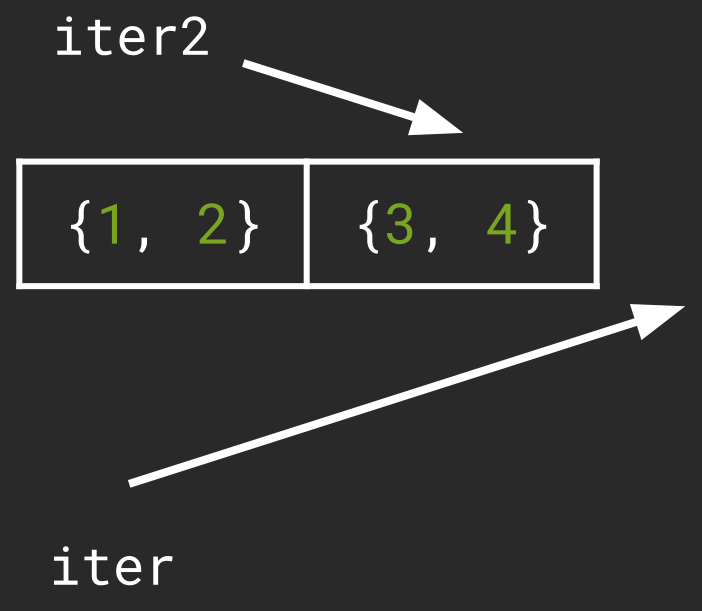
```
1.  std::map<int, int> map{{1, 2}, {3, 4}};
2.  auto iter = map.begin();
3.  // what is *iter at this line?
4.  ++iter;
5.  // what is (*iter).second at this line?
6.  auto iter2 = iter;
7.  ++iter;
8.  // what is (*iter).first at this line?
9.  // what is *iter2 at this line?
```




```
Recall:
- *iter = get what iter is pointing to
- ++iter = go to the next element
- copy = create an another iterator pointing to same thing
```

dereference iter
(undefined behavior - can't dereference end iterator)

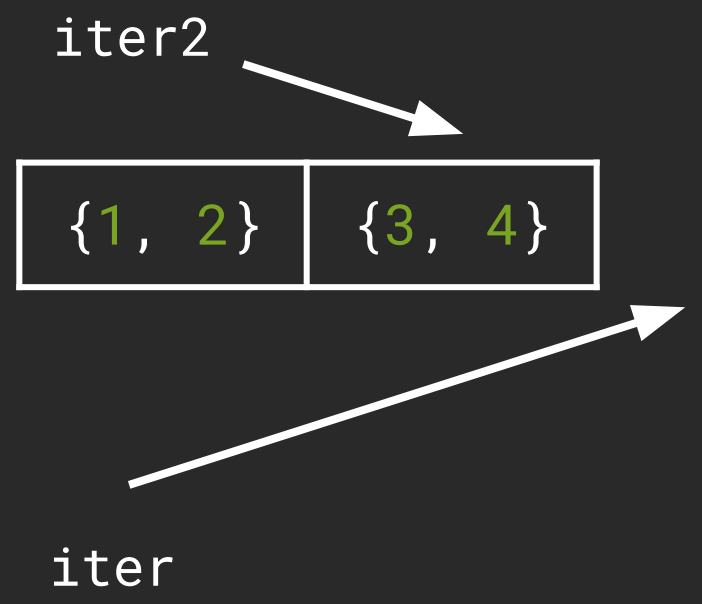
```
1.  std::map<int, int> map{{1, 2}, {3, 4}};
2.  auto iter = map.begin();
3.  // what is *iter at this line?
4.  ++iter;
5.  // what is (*iter).second at this line?
6.  auto iter2 = iter;
7.  ++iter;
8.  // what is (*iter).first at this line?
9.  // what is *iter2 at this line?
```



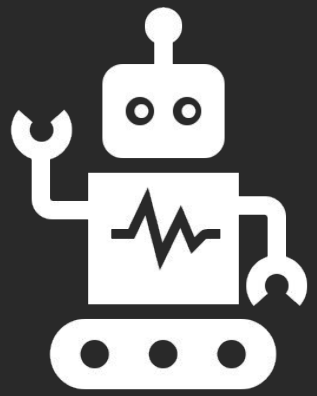
Recall:
- *iter = get what iter is pointing to
- ++iter = go to the next element
- copy = create an another iterator pointing to same thing

dereference iter2, returns the pair {3, 4}

```
1.  std::map<int, int> map{{1, 2}, {3, 4}};
2.  auto iter = map.begin();
3.  // what is *iter at this line?
4.  ++iter;
5.  // what is (*iter).second at this line?
6.  auto iter2 = iter;
7.  ++iter;
8.  // what is (*iter).first at this line?
9.  // what is *iter2 at this line?
```




```
Recall:
- *iter = get what iter is pointing to
- ++iter = go to the next element
- copy = create an another iterator pointing to same thing
```



Questions

Answer 4 questions.

exercise: print all elements the following collections

```
1.  std::set<int> set{3, 1, 4, 1, 5, 9};
2.  // TODO
3.
4.
5.
6.
7.  std::map<int, int> map{{3, 2}, {1, 5}, {7, 7}, {4, 3}};
8.  // TODO
9.
10.
11.
```

exercise: print all elements the following collections

Type out the four blanks for set/map in the chat. Should be the same for set vs. map.

```
1.  std::set<int> set{3, 1, 4, 1, 5, 9};
2.  for (initialization; termination condition; increment) {
3.    const auto& elem = retrieve element;
4.    cout << elem << endl;
5.  }
6.
7.  std::map<int, int> map{{3, 2}, {1, 5}, {7, 7}, {4, 3}};
8.  for (initialization; termination condition; increment) {
9.    const auto& [key, value] = retrieve element;
10.   cout << key << ":" << value << endl;
11. }
```

exercise: print all elements the following collections

```
1.  std::set<int> set{3, 1, 4, 1, 5, 9};
2.  for (auto iter = set.begin(); iter != set.end(); ++iter) {
3.    const auto& elem = *iter;
4.    cout << elem << endl;
5.  }
6.
7.  std::map<int, int> map{{3, 2}, {1, 5}, {7, 7}, {4, 3}};
8.  for (auto iter = map.begin(); iter != map.end(); ++iter) {
9.    const auto& [key, value] = *iter;
10.   cout << key << ":" << value << endl;
11. }
```

You just discovered the for-each loop!

```
1.  std::set<int> set{3, 1, 4, 1, 5, 9};
2.  for (const auto& elem : set) {
3.
4.    cout << elem << endl;
5.  }
6.
7.  std::map<int, int> map{{3, 2}, {1, 5}, {7, 7}, {4, 3}};
8.  for (const auto& [key, value] : map) {
9.
10.   cout << key << ":" << value << endl;
11. }
```

-> is a shorthand for iterators.

These are equivalent

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

```
 auto key = iter->first;
```

```
auto [key, value] = iter;
```

Iterating over the elements of a STL set/map.

- Exactly the same as in CS 106B - no modifying the container in the loop!
- The elements are ordered based on the operator< for element/key.
- Because maps store pairs, each element m is an std::pair that you can use structured binding on.

```
1.  for (const auto& element : s) {
2.    // do stuff with key
3.  }
4.
5.  for (const auto& [key, value] : m) {
6.    // do stuff with key, value
7.  }
```

What is the type of an iterator?

```
1.  std::set<int> s{1, 2, 3};
2.  std::set<int>::iterator iter = s.begin();
```

Scope resolution: tells us that this iterator type is a "member type" of the set<int> class.

Who cares?

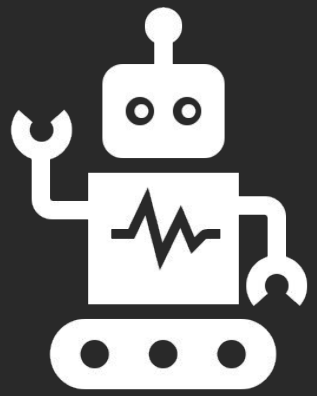
```
1.  std::set<int> s{1, 2, 3};
2.  auto iter = s.begin();
```

Summary of Iterator Basics

Containers may store elements in weird places.

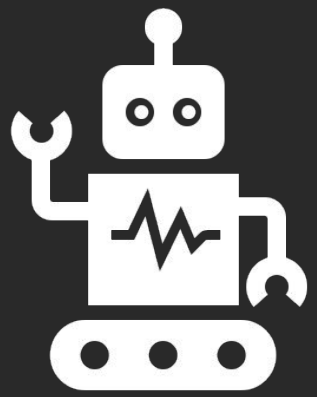
Iterators help you get from one to the next.

The container tells you where to start and end.



Questions

Answer 2 questions.



Example

map::insert and erase, which require iterators

Iterator Categories

let's review the basic functionality of iterators

All iterators must support

- copy construction and assignment
- equality comparable (== and !=)
- dereferencable (*iter) - either readable or writable (not necessarily both)
- incrementable (++iter) - single pass (previous iter invalidated after ++)

```
1.  auto iter = something.begin();
2.  auto copy = iter;
3.  ++iter;                    // single pass: would invalidate copy
4.  *copy;                     // invalid if iter was single pass
```

let's review the basic functionality of iterators

All iterators must support

- copy construction and assignment
- equality comparable (== and !=)
- dereferencable (*iter) - either readable or writable (not necessarily both)
- incrementable (++iter) - single pass (previous iter invalidated after ++)

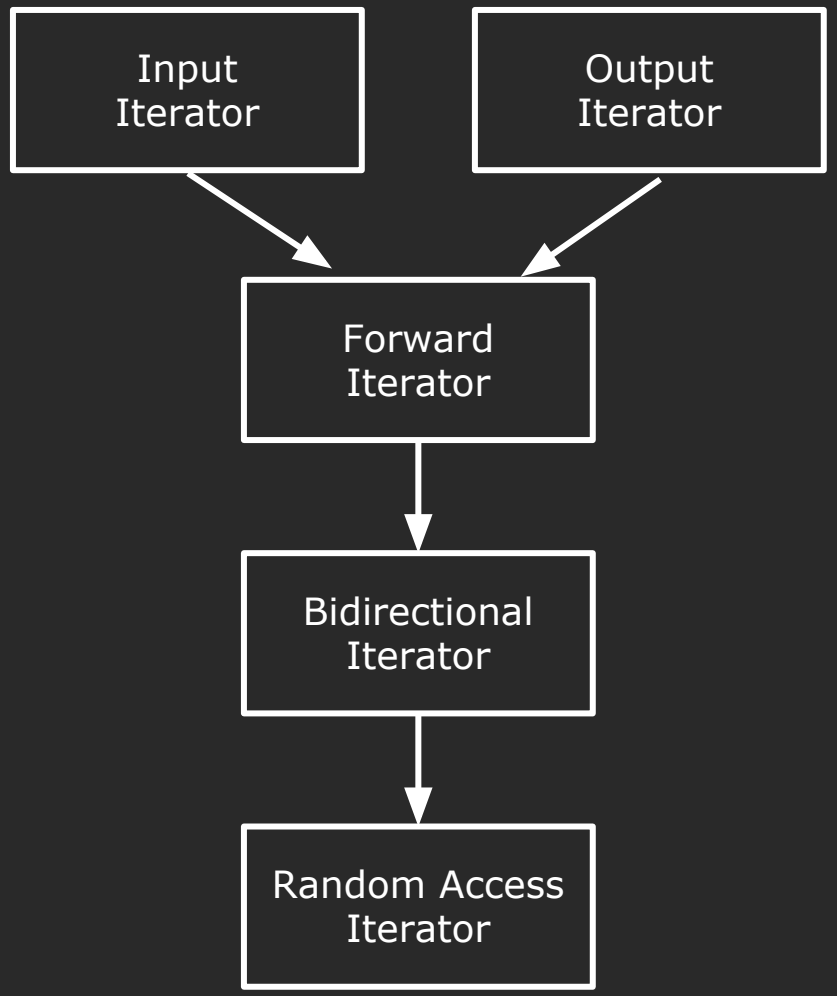
Wouldn't it be nice if the iterator was...

- both readable and writable?
- multi-pass incrementable?
- decrementable? (eg. --iter)
- able to move by an arbitrary amount (eg. iter += 5)

Iterators (by definition)

Supported Operations

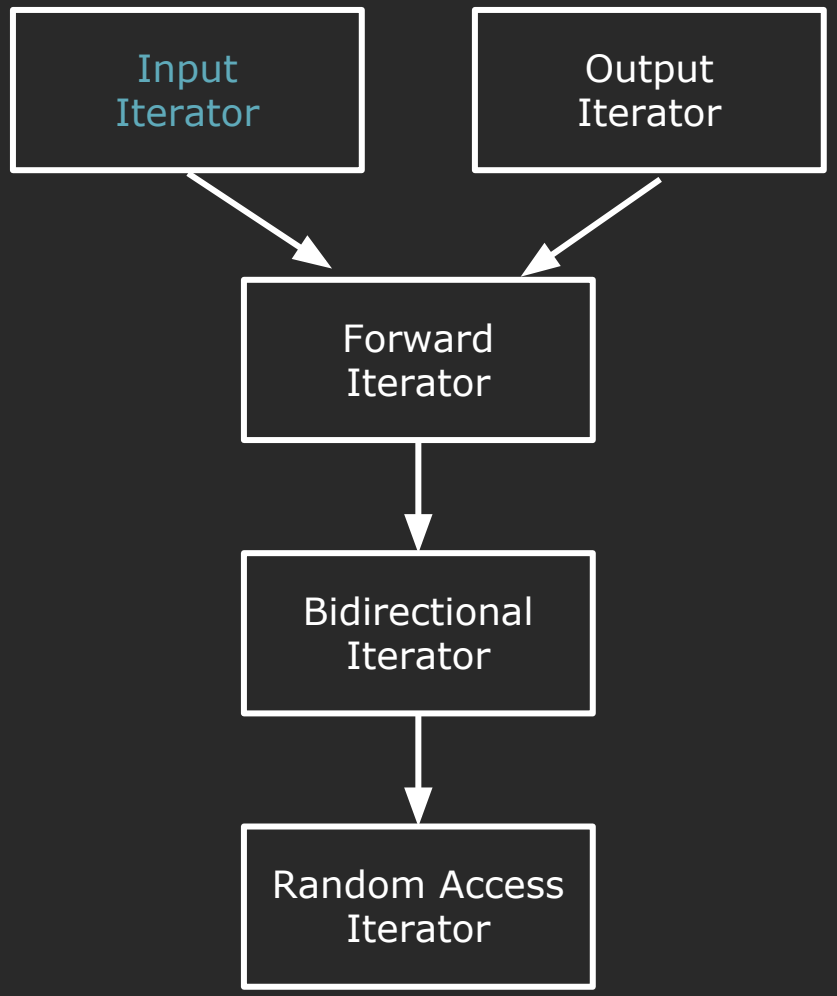
- Copy construct/assign (=) + destructible
- Default constructible
- Equality comparable (==, !=)
- Dereferencable r-value (auto e = *iter, ->)
- Dereferencable l-value (*iter = e)
- Single-pass incrementable (++)
- Multi-pass incrementable
- Decrementable (--)
- Random access (+, -, +=, -=, offset [])
- Inequality comparable (<, >, <=, >=)



Input Iterators (eg. std::istream_iterator)

Supported Operations

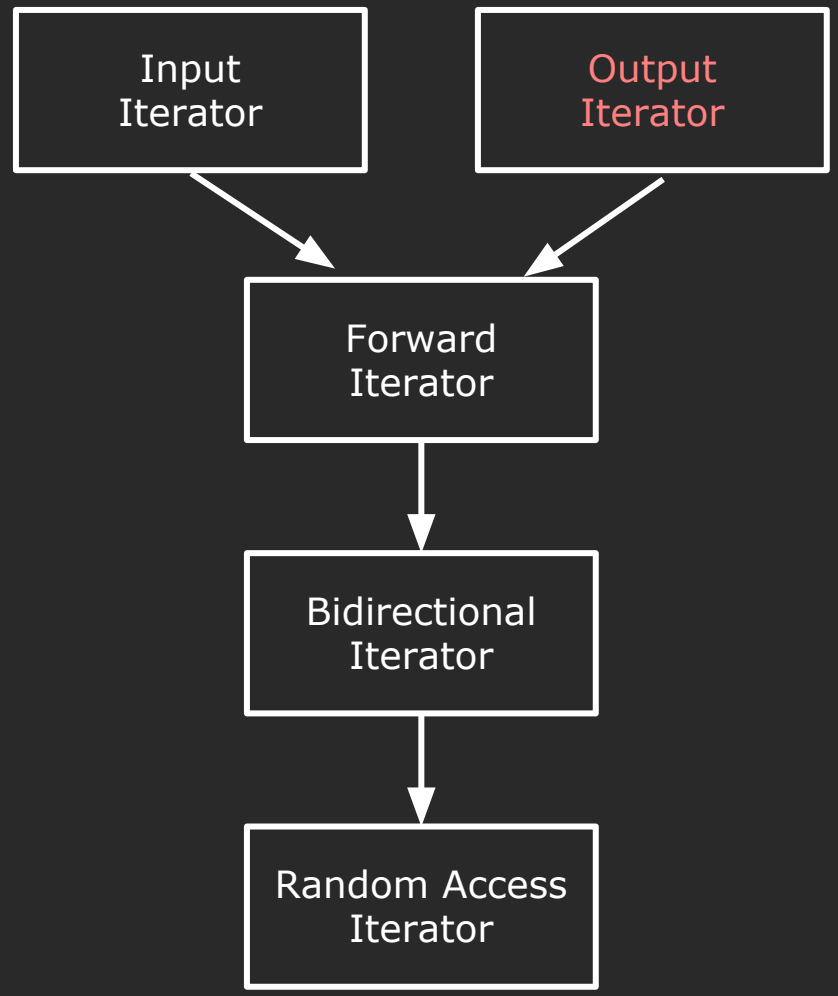
- Copy construct/assign (=) + destructible
- Default constructible
- Equality comparable (==, !=)
- Dereferencable r-value (auto e = *iter, ->)
- Dereferencable l-value (*iter = e)
- Single-pass incrementable (++)
- Multi-pass incrementable
- Decrementable (--)
- Random access (+, -, +=, -=, offset [])
- Inequality comparable (<, >, <=, >=)



Output Iterators (eg. std::ostream_iterator)

Supported Operations

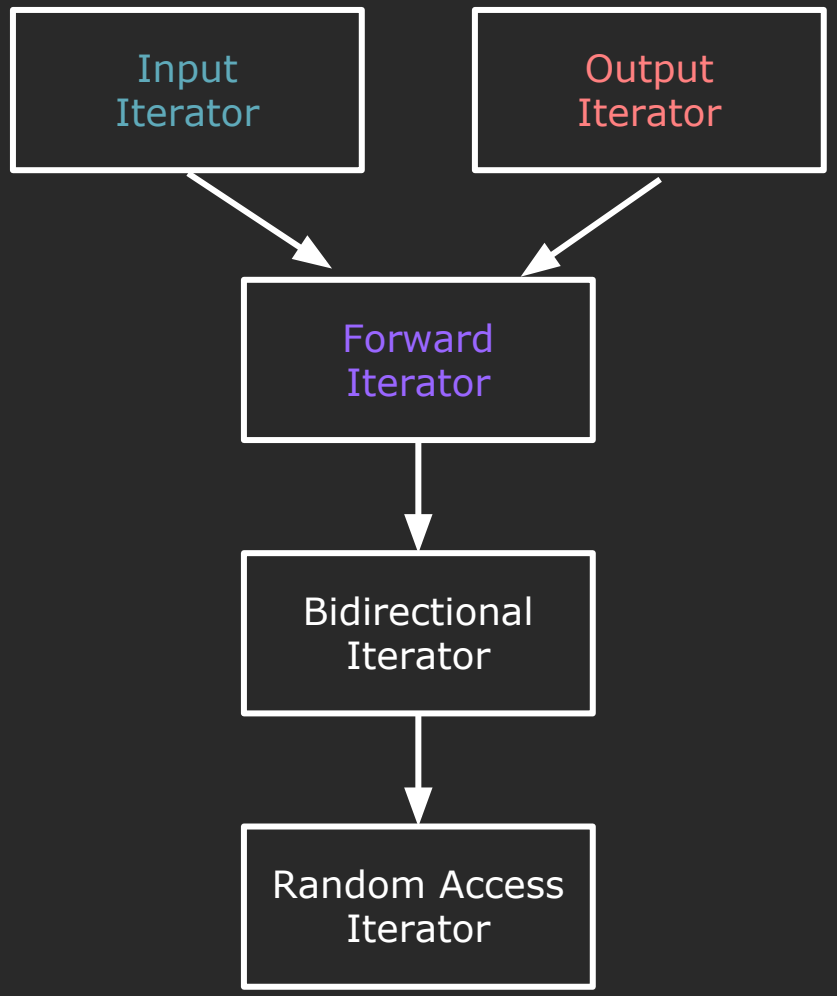
- Copy construct/assign (=) + destructible
- Default constructible
- Equality comparable (==, !=)
- Dereferencable r-value (auto e = *iter, ->)
- Dereferencable l-value (*iter = e)
- Single-pass incrementable (++)
- Multi-pass incrementable
- Decrementable (--)
- Random access (+, -, +=, -=, offset [])
- Inequality comparable (<, >, <=, >=)



Forward Iterators (eg. std::unordered_map)

Supported Operations

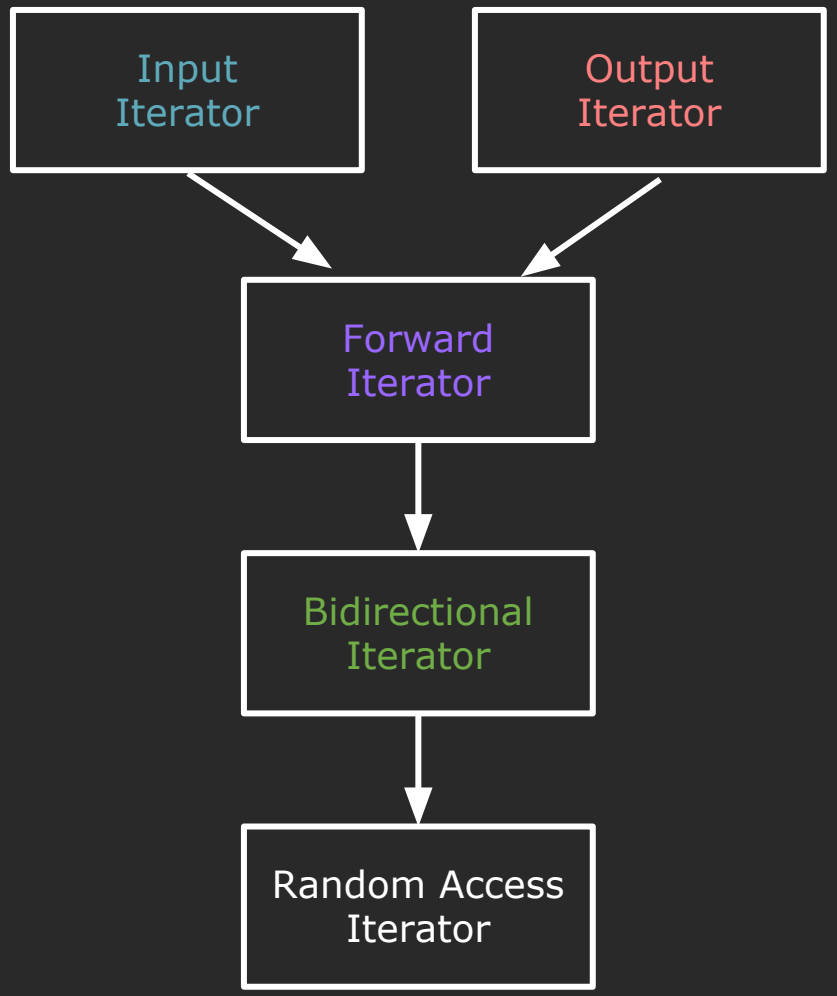
- Copy construct/assign (=) + destructible
- Default constructible
- Equality comparable (==, !=)
- Dereferencable r-value (auto e = *iter, ->)
- Dereferencable l-value (*iter = e)
- Single-pass incrementable (++)
- Multi-pass incrementable
- Decrementable (--)
- Random access (+, -, +=, -=, offset [])
- Inequality comparable (<, >, <=, >=)



Bidirectional Iterators (eg. std::map/set/list)

Supported Operations

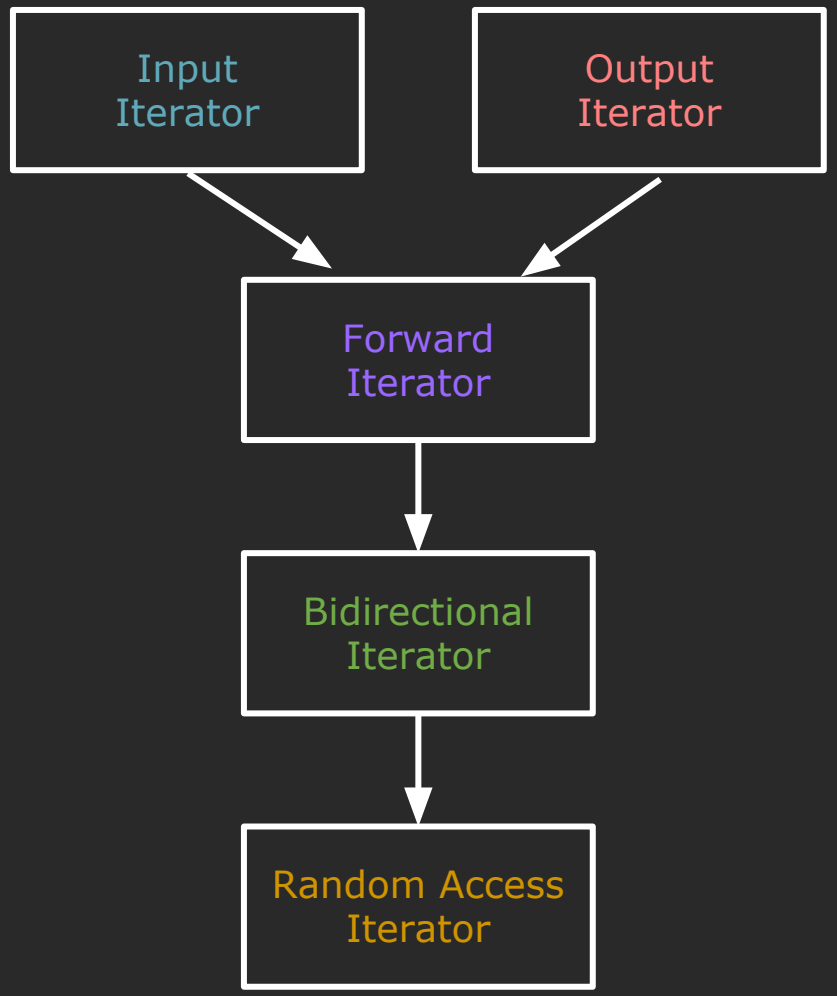
- Copy construct/assign (=) + destructible
- Default constructible
- Equality comparable (==, !=)
- Dereferencable r-value (auto e = *iter, ->)
- Dereferencable l-value (*iter = e)
- Single-pass incrementable (++)
- Multi-pass incrementable
- Decrementable (--)
- Random access (+, -, +=, -=, offset [])
- Inequality comparable (<, >, <=, >=)



Random Access Iterators (eg. std::vector/deque)

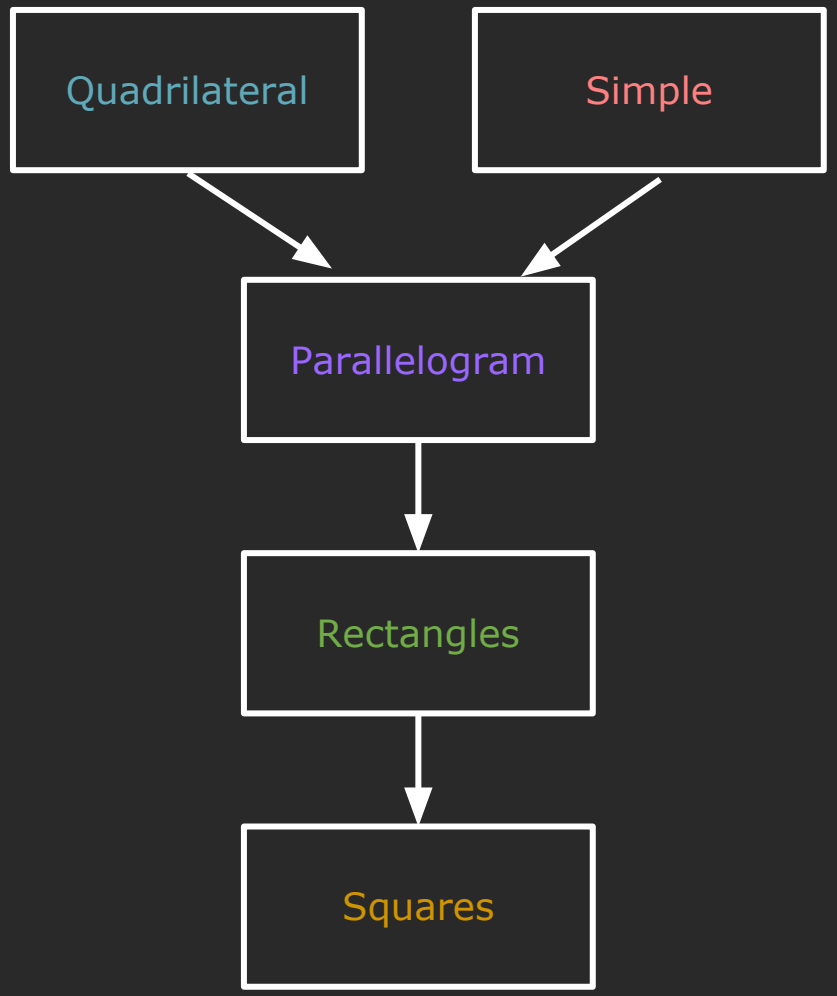
Supported Operations

- Copy construct/assign (=) + destructible
- Default constructible
- Equality comparable (==, !=)
- Dereferencable r-value (auto e = *iter, ->)
- Dereferencable l-value (*iter = e)
- Single-pass incrementable (++)
- Multi-pass incrementable
- Decrementable (--)
- Random access (+, -, +=, -=, offset [])
- Inequality comparable (<, >, <=, >=)



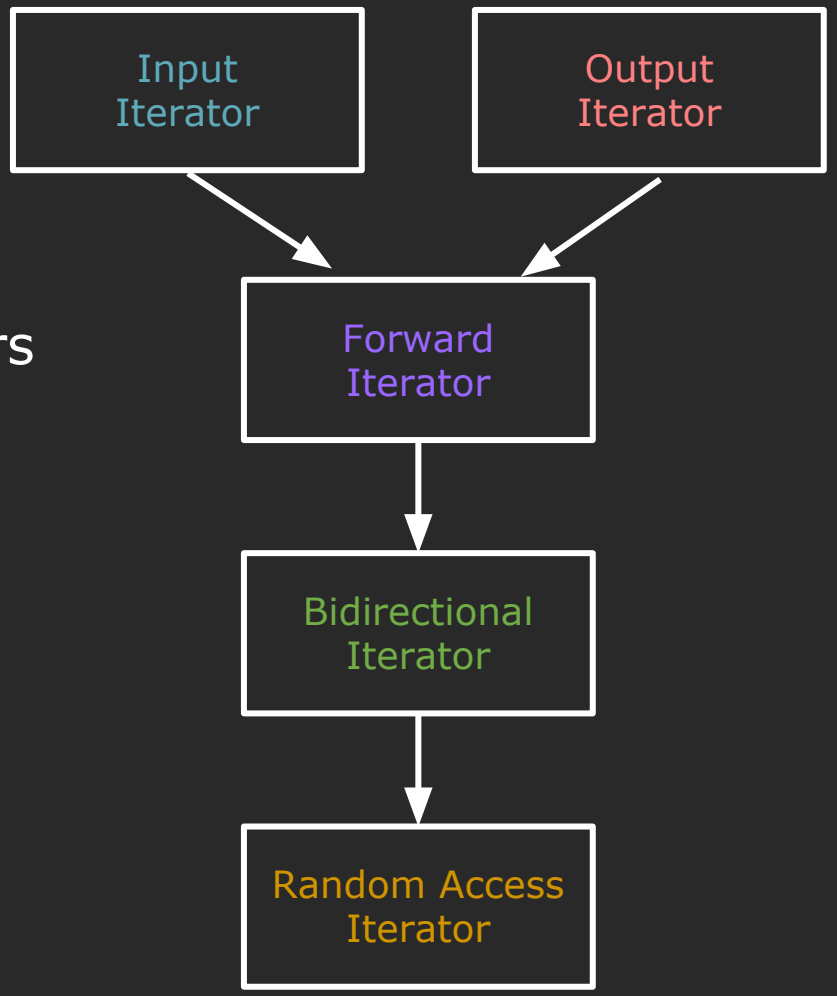
Iterator Categories form a Hierarchy

- All squares are rectangles
- All rectangles are parallelograms
- All parallelograms are simple and are quadrilaterals
- All quadrilaterals are shapes
- All simple shapes are shapes

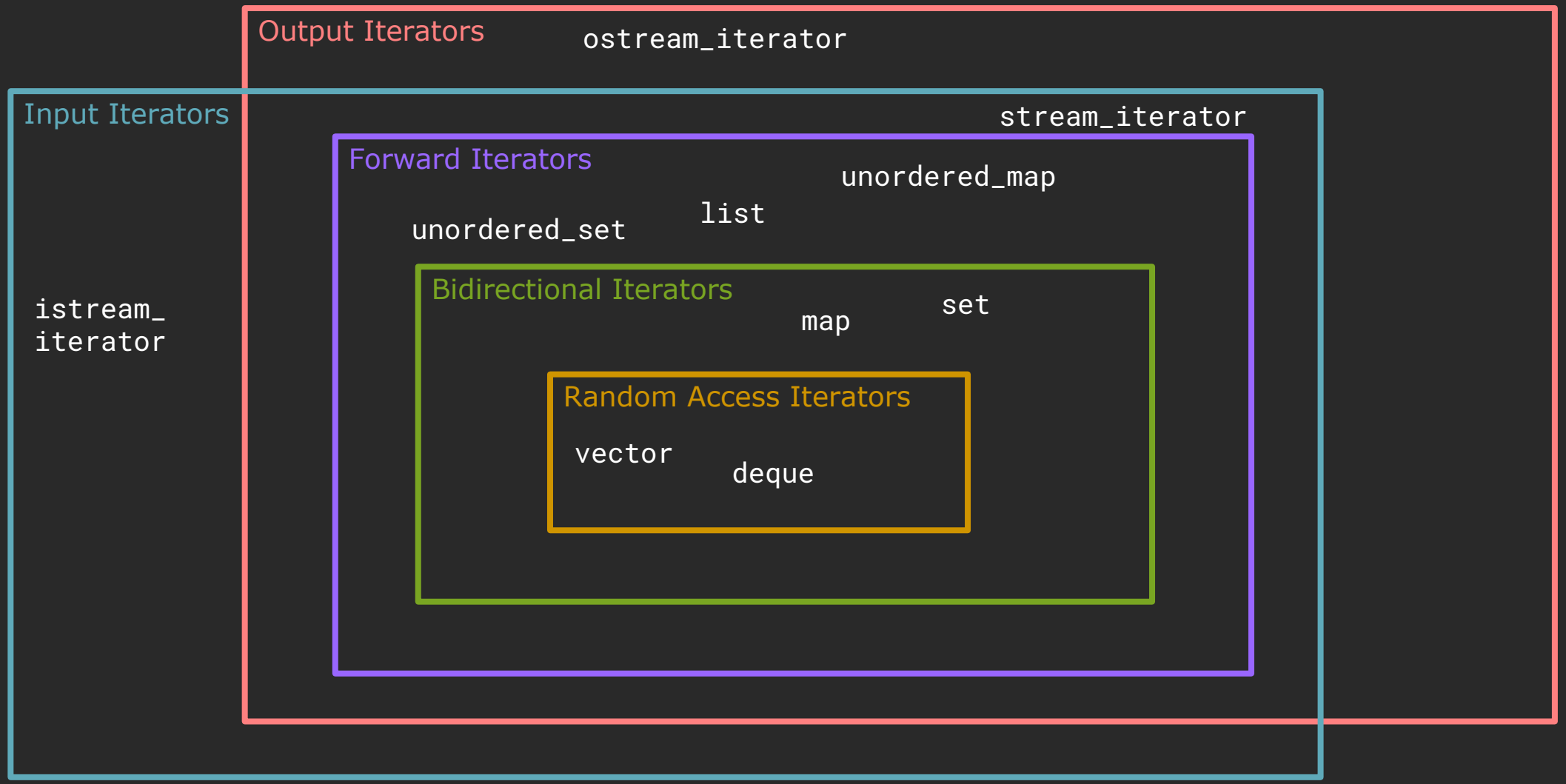


Iterator Categories form a Hierarchy

- All Random Access Iterators are Bidirectional Iterators
- All Bidirectional Iterators are Forward Iterators
- All Forward Iterators are Input and Output Iterators
- All Output Iterators are Iterators
- All Input Iterators are Iterators



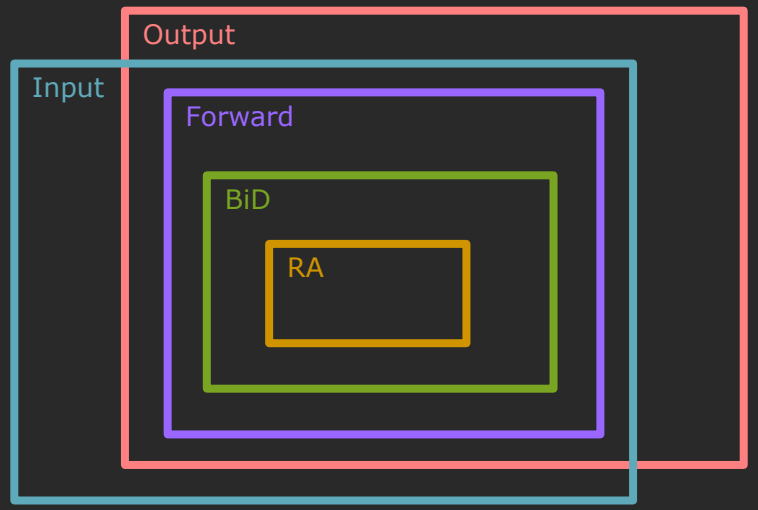
Iterator Categories form a Hierarchy



Answer on the chat:

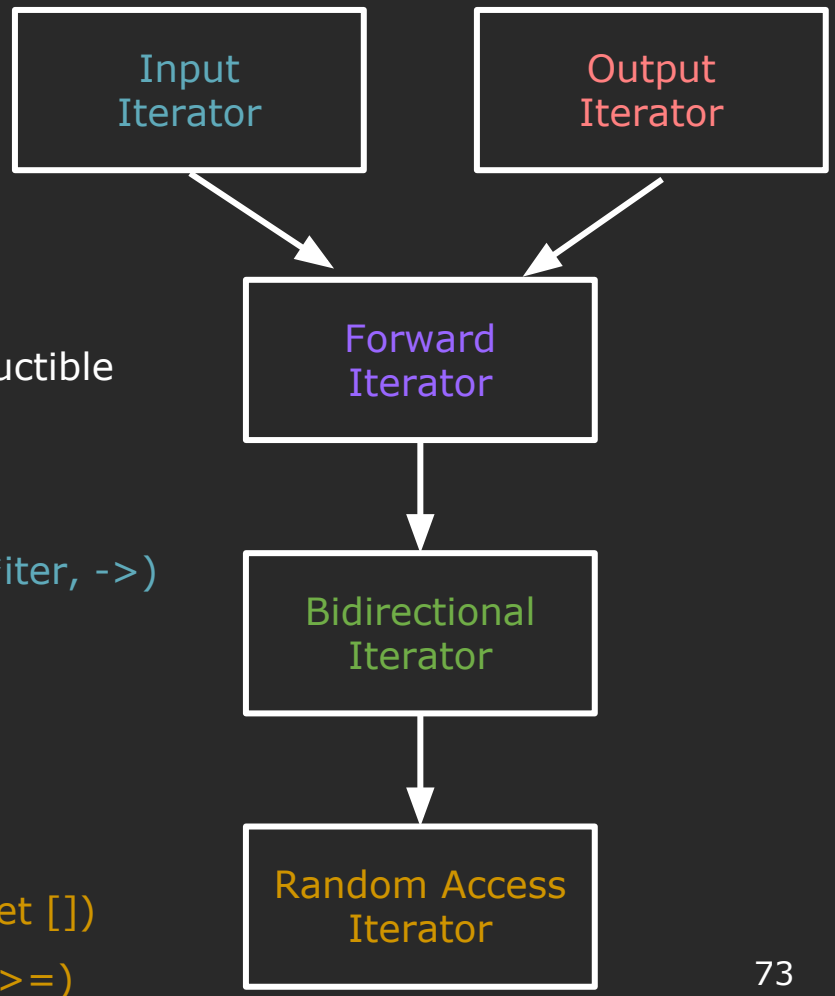
What is the difference between these two lines of code?

```
iter += 2;
++(++iter);
```



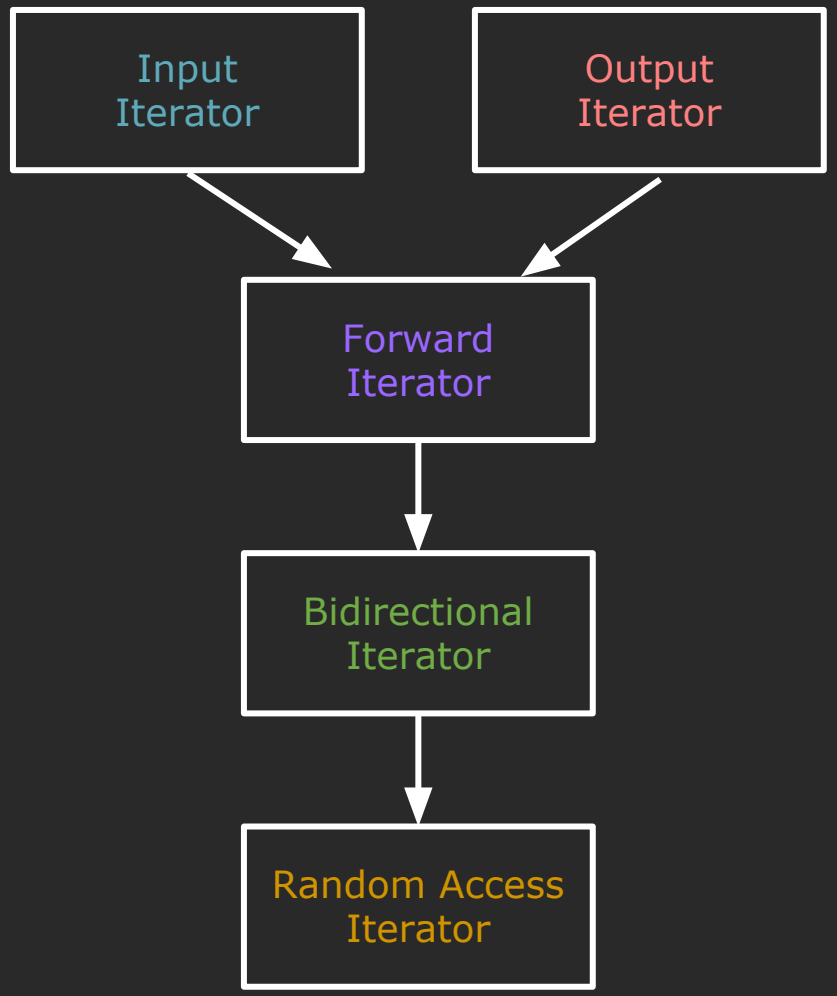
Supported Operations

- Copy construct/assign (=) + destructible
- Default constructible
- Equality comparable (==, !=)
- Dereferencable r-value (auto e = *iter, ->)
- Dereferencable l-value (*iter = e)
- Single-pass incrementable (++)
- Multi-pass incrementable
- Decrementable (--)
- Random access (+, -, +=, -=, offset [])
- Inequality comparable (<, >, <=, >=)



Why do we care about iterator categories?

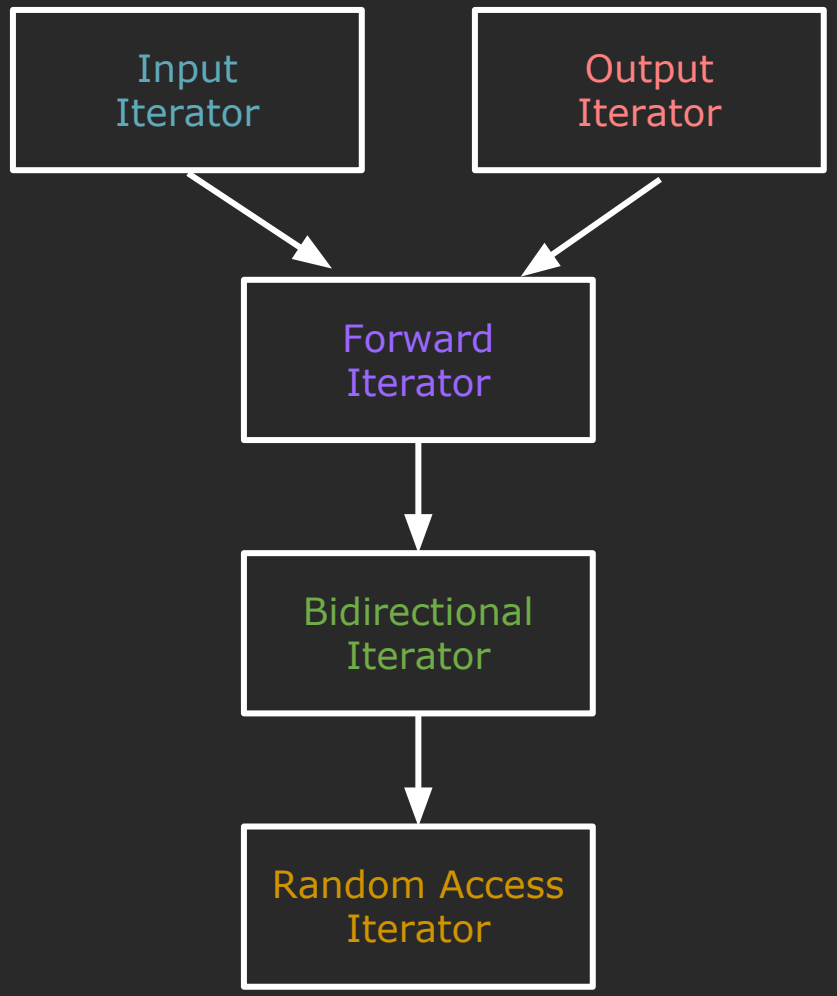
- Iterator operators are always constant-time.
- Any iterator can move forward by an arbitrary number of positions.
- Only a random access iterator can do so efficiently (in constant time).
- A collection does not have random access iterators because it's impossible to "jump forward" efficiently.



Why use iterators?

- We will begin writing generic algorithms in the next lecture.
- These algorithms don't depend on the container. They only depend on the iterator category.

- Example: sort requires random access iterators.
- Example: find requires input iterators.



Summary of Iterator Category

There is a "hierarchy" of iterator types.
The more powerful types have additional operators.
Different containers have different iterator categories.

FAQ: Are iterators just pointers?

- Pointers are one type of iterator, but most iterators are not pointers.
- ++ for pointers is to move to the next location in memory. Most containers do not store their elements contiguously in memory.
- Pointers are random-access iterators for a C-array (which does store element contiguously in memory).

```
1.  int arr[5];
2.  int* iter = arr;    // array decays to pointer
3.  *iter = 1;
4.  ++iter;
5.  *iter = 2;
```

FAQ: How are iterators implemented?

We don't need to know to use them!

That's the power of abstraction.

But we'll implement iterators in a few weeks!

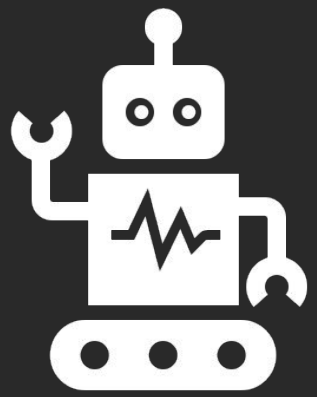
FAQ: How are iterators implemented?

We don't need to know to use them!

That's the power of abstraction.

~~But we'll implement iterators in a few weeks!~~

Actually we won't...but it's a bonus topic + extension on A2!



Deep Dive into Documentation + Example

Range-based methods in STL
Iterator Categories

Erase and Iterator Invalidation

Supplemental Material

last lecture we skipped erase - requires iterators

std::vector<T,Allocator>::erase

```
iterator erase( iterator pos );                                          (1)  (until C++11)
iterator erase( const_iterator pos );                                         (since C++11)

iterator erase( iterator first, iterator last );                         (2)  (until C++11)
iterator erase( const_iterator first, const_iterator last );                  (since C++11)
```

Erases the specified elements from the container.

1) Removes the element at pos.
2) Removes the elements in the range [first, last).

Invalidates iterators and references at or after the point of the erase, including the end() iterator.

The iterator pos must be valid and dereferenceable. Thus the end() iterator (which is valid, but is not dereferencab
cannot be used as a value for pos.

The iterator first does not need to be dereferenceable if first==last: erasing an empty range is a no-op.

example: remove first element that begin with 'A'.

```
1.  void removeOneA(std::vector<string>& vec) {
2.    for (auto iter = vec.begin(); iter != vec.end(); ++iter) {
3.      if (!(*iter).empty() && (*iter)[0] == 'A') {
4.          vec.erase(iter);
5.          return;
6.      }
7.  }
```

exercise: remove all elements that begin with 'A'.

```
1.  void removeOneA(std::vector<string>& vec) {
2.    for (auto iter = vec.begin(); iter != vec.end(); ++iter) {
3.      if (!(*iter).empty() && (*iter)[0] == 'A') {
4.          vec.erase(iter);
5.
6.      }
7.  }
```

That's easy! Just remove the return statement!

That's not correct...here's why!

The iterator knows how to get from one element to the next through the entire range.

Let's find and remove the red Alien!

The iterator knows how to get from one element to the next through the entire range.

Let's find and remove the red Alien!

The iterator knows how to get from one element to the next through the entire range.

Found it!

begin()

end()

The iterator knows how to get from one element to the next through the entire range.

Found it!

begin()

end()

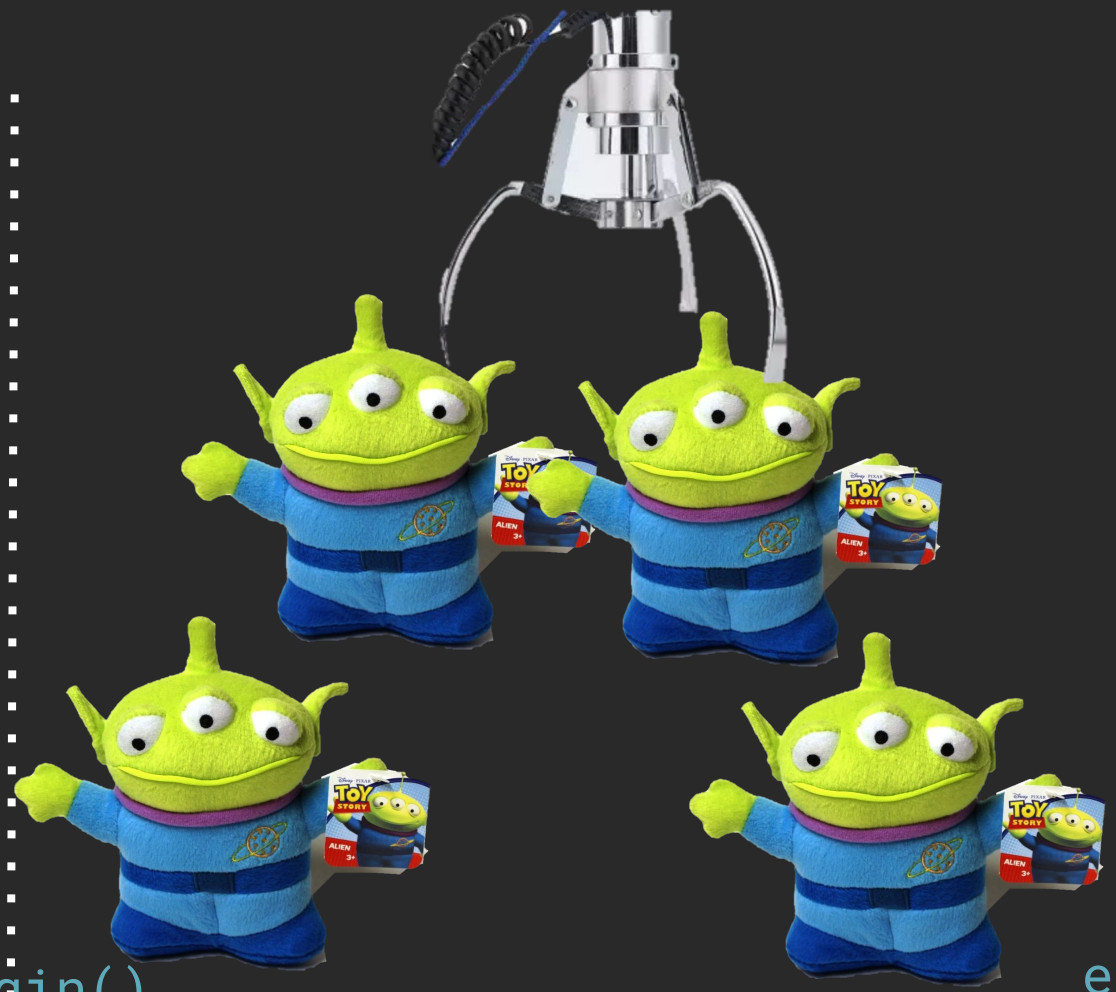
The iterator knows how to get from one element to the next through the entire range.

Found it!

begin()

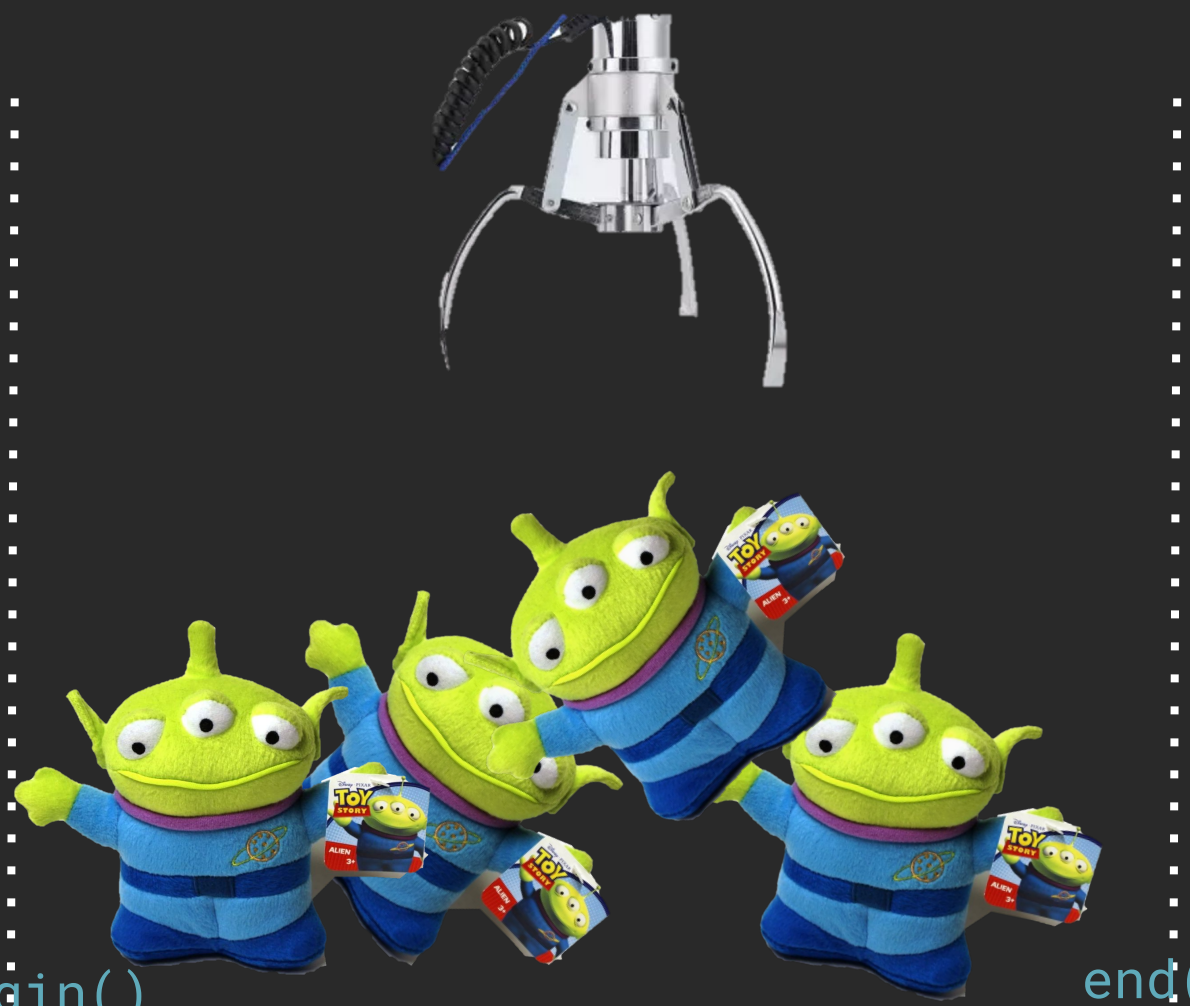
end()

The iterator knows how to get from one element to the next through the entire range.



Gone!

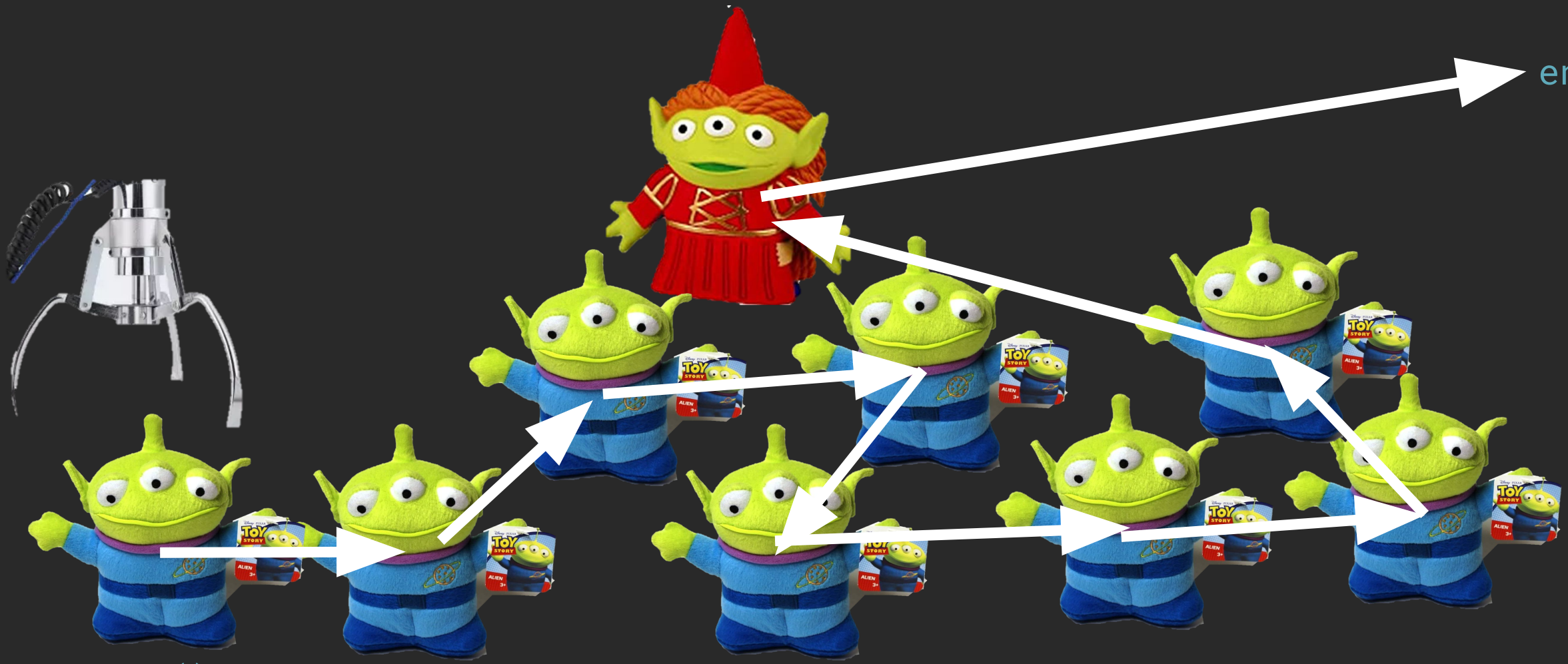
Removing elements can invalidate iterators.



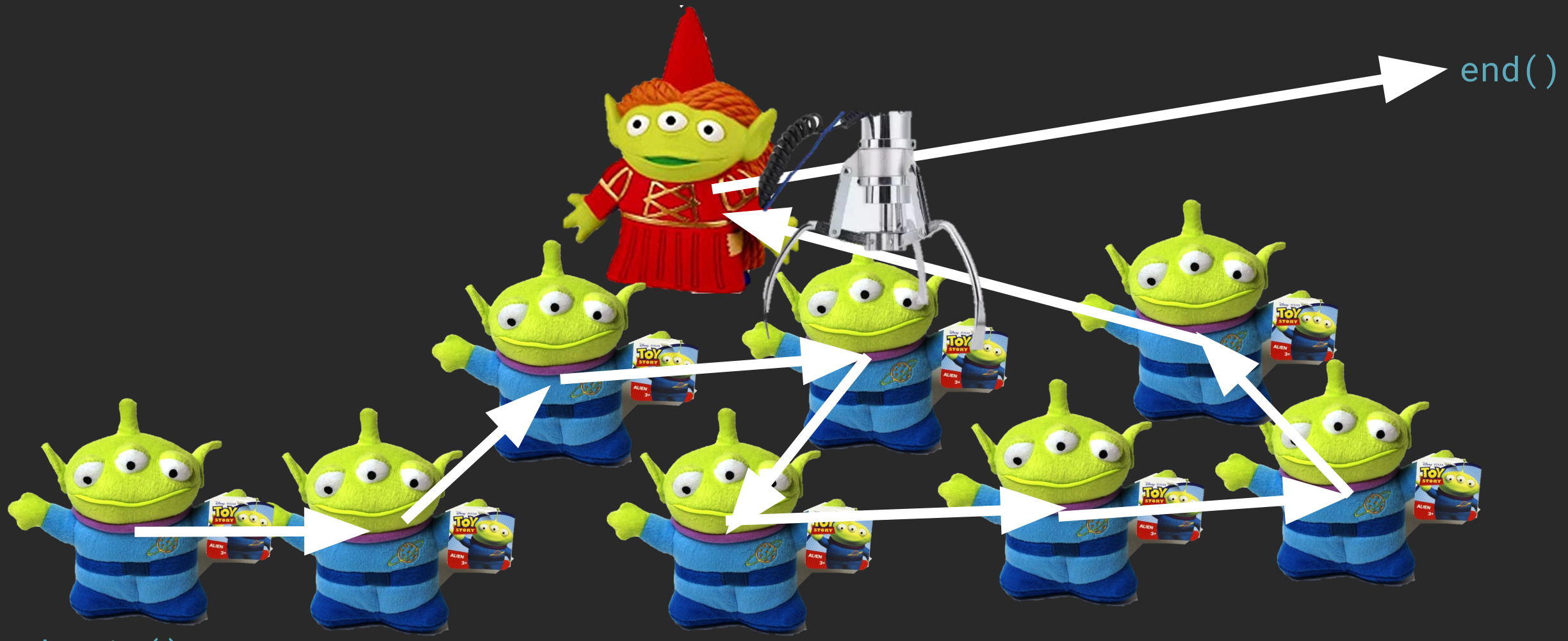
Just like removing the bottom soup can in a soup pyramid, everything falls!

The iterator no longer is able to go through the remaining elements!

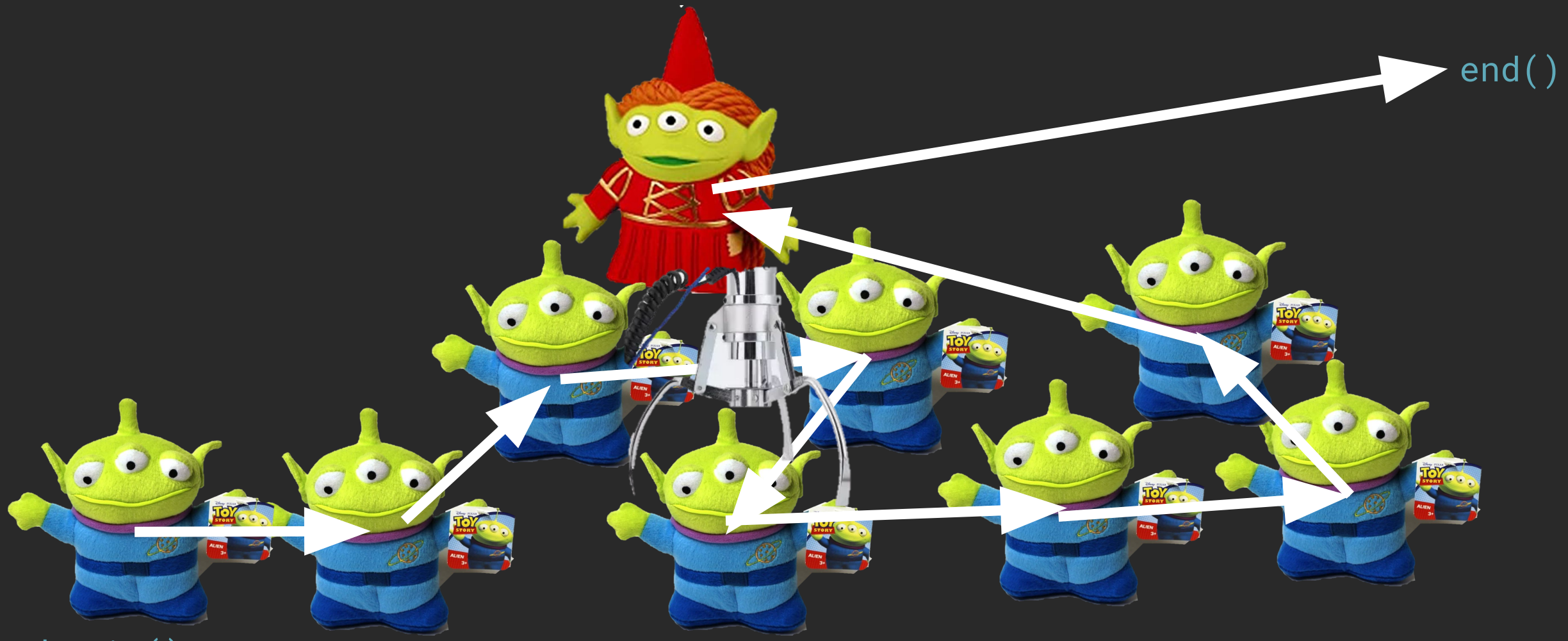
Another more drastic view of iterator invalidation



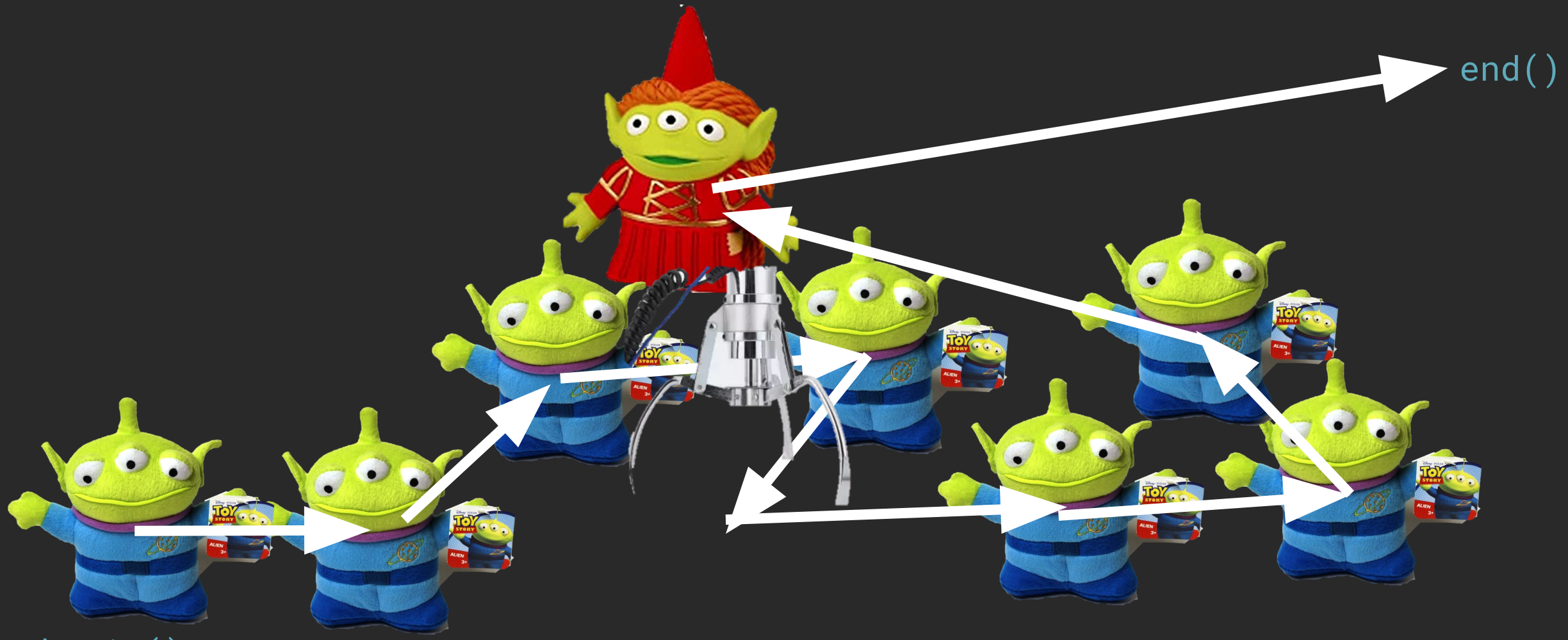
Another more drastic view of iterator invalidation



Another more drastic view of iterator invalidation

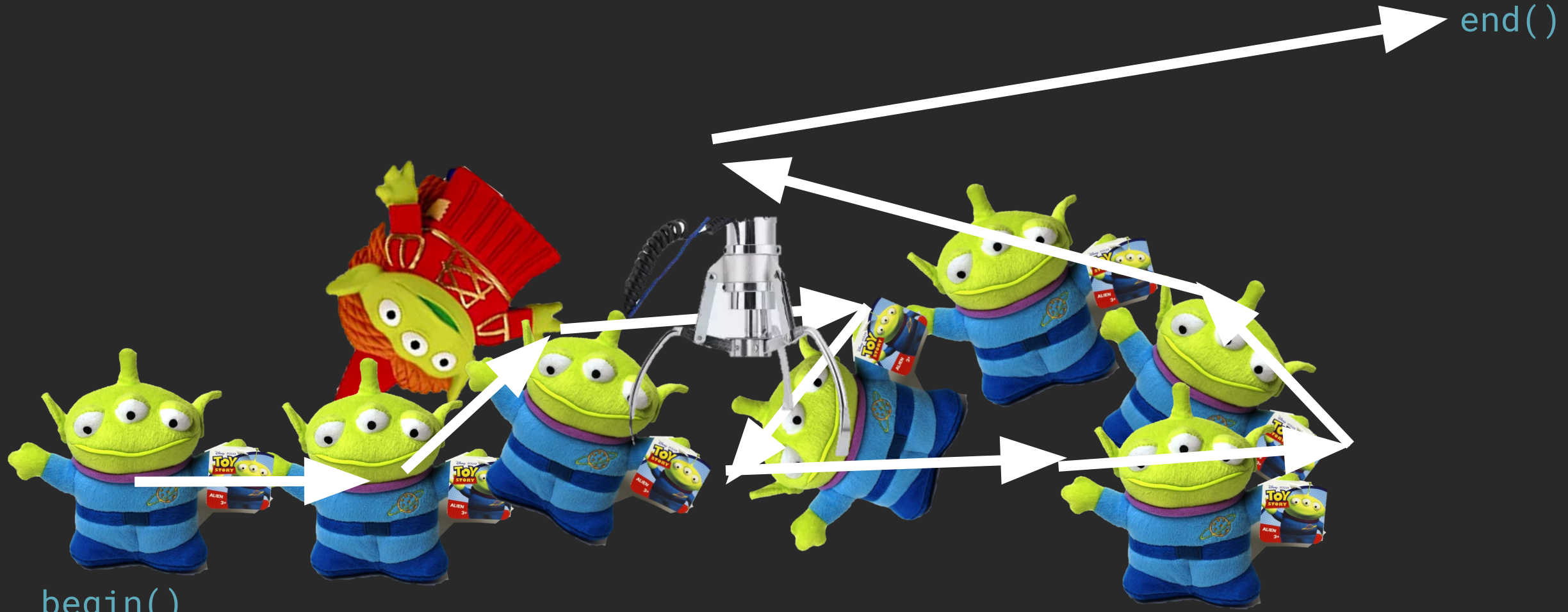


Another more drastic view of iterator invalidation

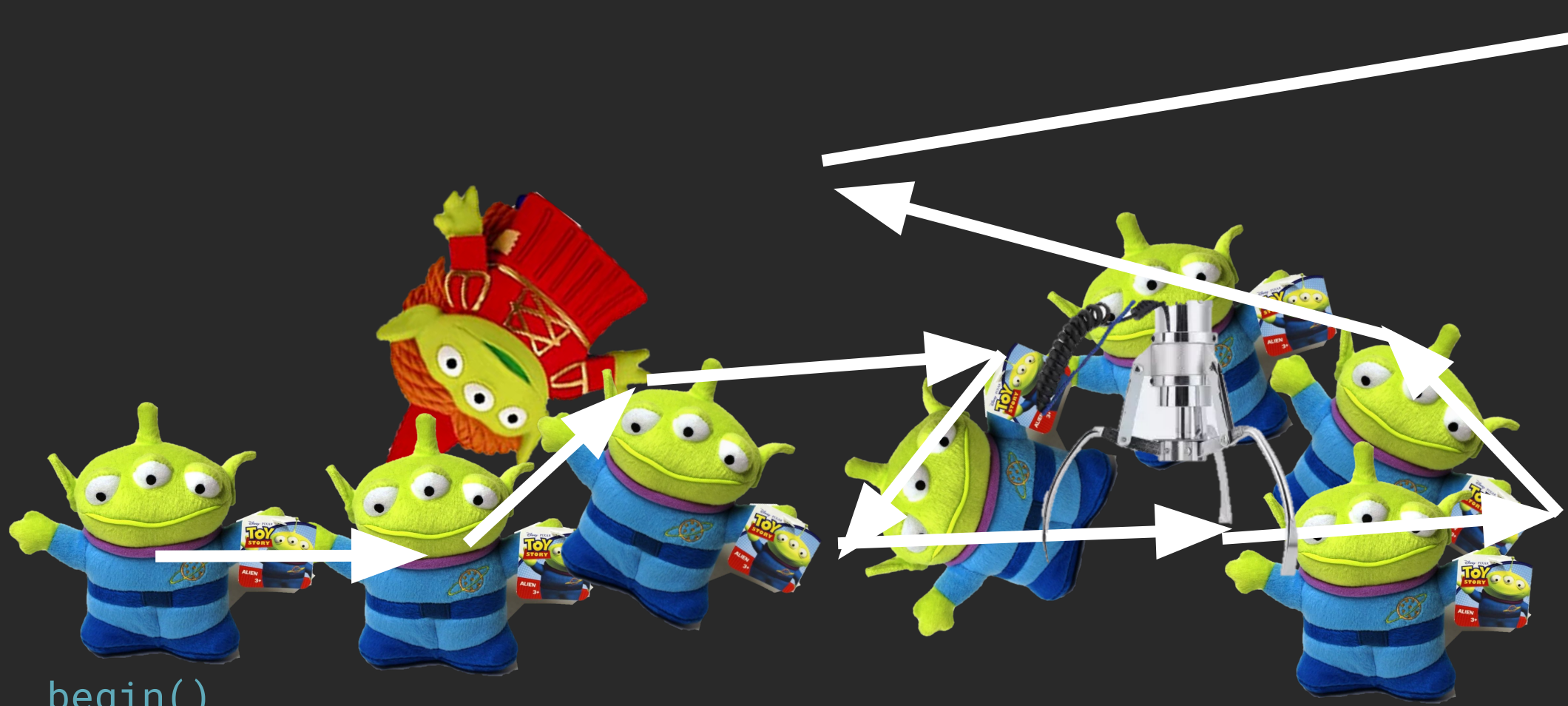


Supplemental Material

When an element is erased, the container may reorganize the layout of the elements.



The iterator still thinks it should follow the arrows to find the next element.



Iterator Invalidation

Some operations on some containers
may invalidate iterators.

std::vector - lots of invalidation

std::vector<T,Allocator>::erase

```
iterator erase( iterator pos );                                  (1)  (until C++11)
iterator erase( const_iterator pos );                                 (since C++11)

iterator erase( iterator first, iterator last );                 (2)  (until C++11)
iterator erase( const_iterator first, const_iterator last );          (since C++11)
```

Erases the specified elements from the container.

1) Removes the element at pos.

2) Removes the elements in the range [first, last).

Invalidates iterators and references at or after the point of the erase, including the end() iterator.

The iterator pos must be valid and dereferenceable. Thus the end() iterator (which is valid, but is not dereferencab
cannot be used as a value for pos.

The iterator first does not need to be dereferenceable if first==last: erasing an empty range is a no-op.

Supplemental Material

std::vector - documentation tells you which operations invalidate which iterators.

Iterator invalidation

Operations	Invalidated
All read only operations	Never
`swap`, `std::swap`	`end()`
`clear`, `operator=`, `assign`	Always
`reserve`, `shrink_to_fit`	If the vector changed capacity, all of them. If not, none.
`erase`	Erased elements and all elements after them (including `end()`)
`push_back`, `emplace_back`	If the vector changed capacity, all of them. If not, only end().
`insert`, `emplace`	If the vector changed capacity, all of them. If not, only those at or after the insertion point (including end()).
`resize`	If the vector changed capacity, all of them. If not, only end() and any elements erased.
`pop_back`	The element erased and `end()`.

This is what is meant by "std::list is stable".

std::list

Defined in header <list>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class list;
```

(1)

```
namespace pmr {
    template <class T>
    using list = std::list<T, std::pmr::polymorphic_allocator<T>>;
}
```

(2) (since C++17)

std::list is a container that supports constant time insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is usually implemented as a doubly-linked list. Compared to std::forward_list this container provides bidirectional iteration capability while being less space efficient.

Adding, removing and moving the elements within the list or across several lists does not invalidate the iterators or references. An iterator is invalidated only when the corresponding element is deleted.

std::list meets the requirements of *Container*, *AllocatorAwareContainer*, *SequenceContainer* and *ReversibleContainer*.

good news: erase returns a new valid iterator!

std::vector<T,Allocator>::erase

`iterator erase( iterator pos );`	(1)	(until C++11)
`iterator erase( const_iterator pos );`		(since C++11)
`iterator erase( iterator first, iterator last );`	(2)	(until C++11)
`iterator erase( const_iterator first, const_iterator last );`		(since C++11)

Return value

Iterator following the last removed element.

If pos refers to the last element, then the end() iterator is returned.

If last==end() prior to removal, then the updated end() iterator is returned.

If [first, last) is an empty range, then last is returned.

exercise: remove all elements that begin with 'A'.

```
1.  void removeOneA(std::vector<string>& vec) {
2.    for (auto iter = vec.begin(); iter != vec.end(); ++iter) {
3.      if (!(*iter).empty() && (*iter)[0] == 'A') {
4.          vec.erase(iter);
5.      }
6.
7.
8.  }
```

This is incorrect: iter is invalidated after the erase.

exercise: remove all elements that begin with 'A'.

```
1.  void removeOneA(std::vector<string>& vec) {
2.    for (auto iter = vec.begin(); iter != vec.end(); ++iter) {
3.      if (!(*iter).empty() && (*iter)[0] == 'A') {
4.          iter = vec.erase(iter);
5.      }
6.
7.
8.  }
```

This is almost correct...but we should not increment iter if we reassign it to the iterator to the next element.

exercise: remove all elements that begin with 'A'.

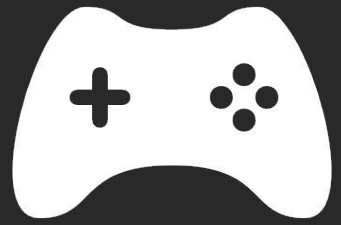
```
1.  void removeOneA(std::vector<string>& vec) {
2.    for (auto iter = vec.begin(); iter != vec.end();) {
3.      if (!(*iter).empty() && (*iter)[0] == 'A') {
4.          iter = vec.erase(iter);
5.      } else {
6.          ++iter;
7.       }
8.  }
```

This is fully correct! If we erased something, then we don't increment. Otherwise, we do increment.

Summary of Iterator Invalidation

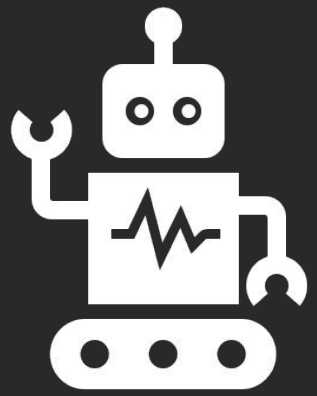
Some operations on some containers invalidate iterators. Read the documentation!

These operations usually will return a valid iterate that you should use instead.



Next time

Templates and Functions



Homework

CS 106B section problems…
but you'll have to use STL containers + iterators.