Iterators and Pointers

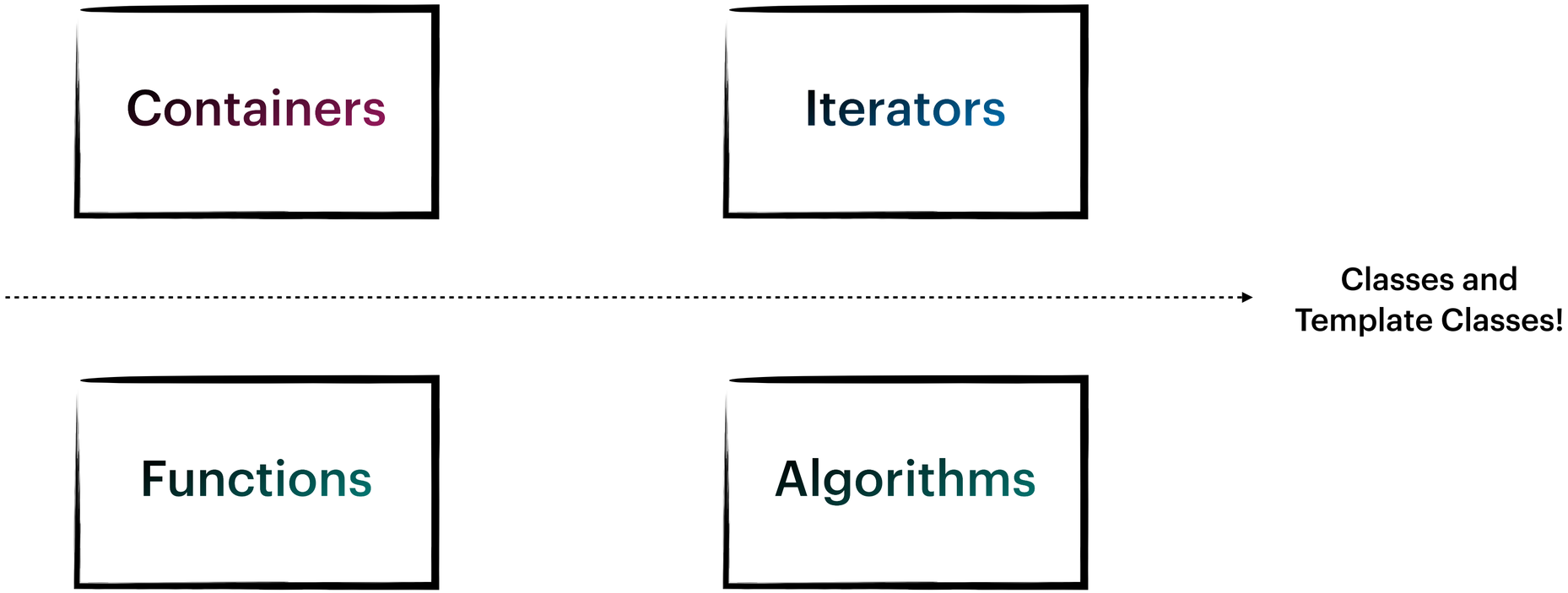
CS 106L, Fall ‘21

How can we programmatically access containers?

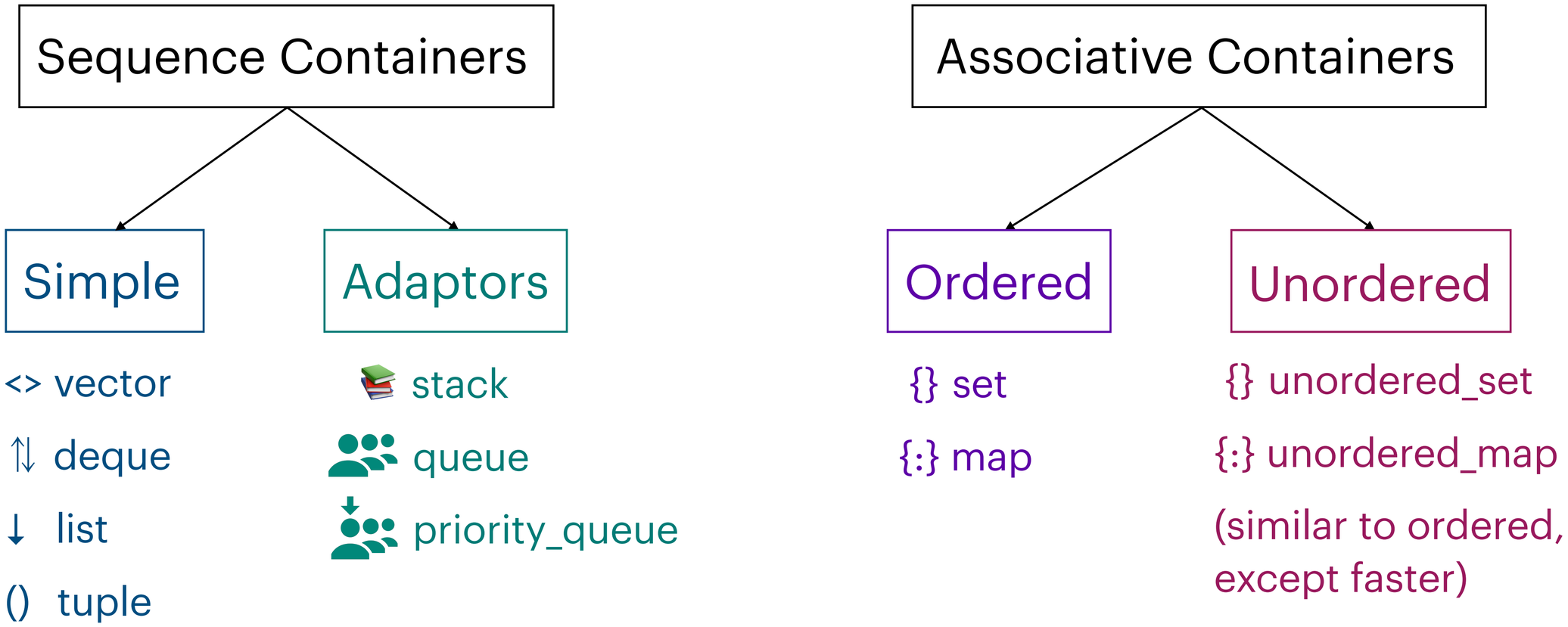
Today's agenda

- Recap: **Collections**
- Iterators!
- Iterator Practice
- Pointers
- Iterators vs. Pointer demo

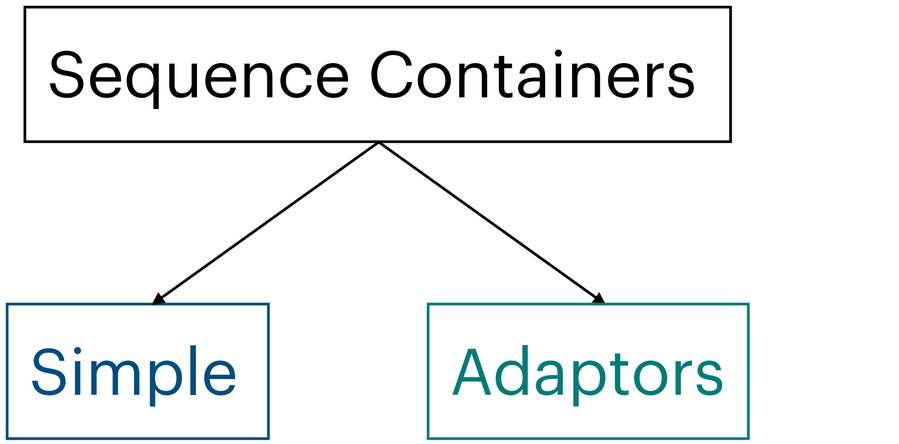
What's in the STL?

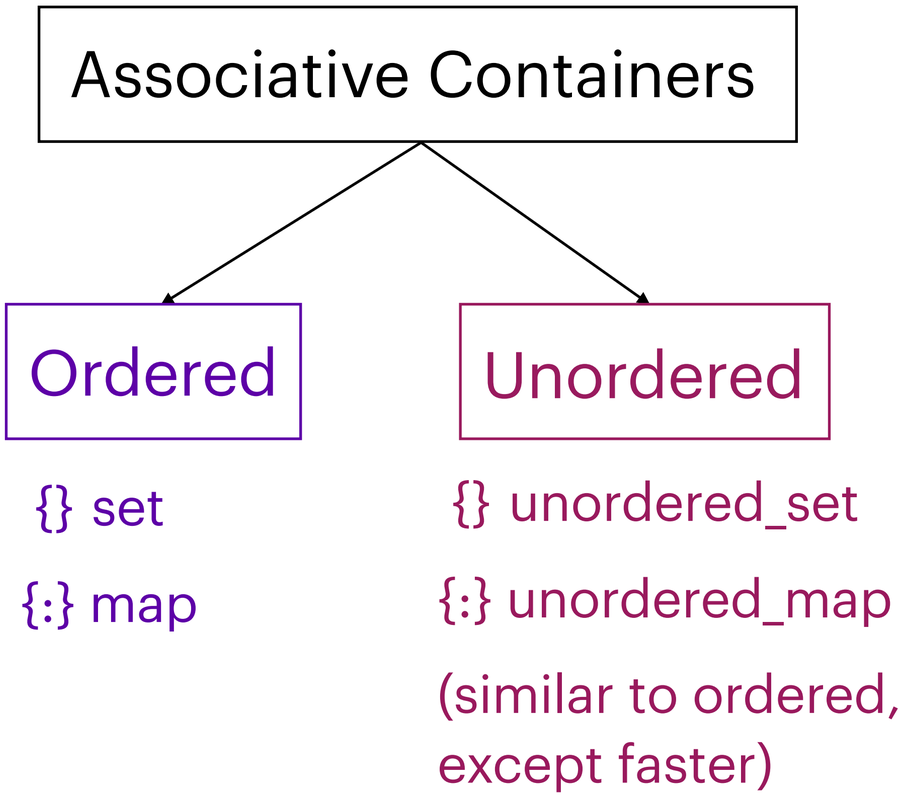


Types of containers

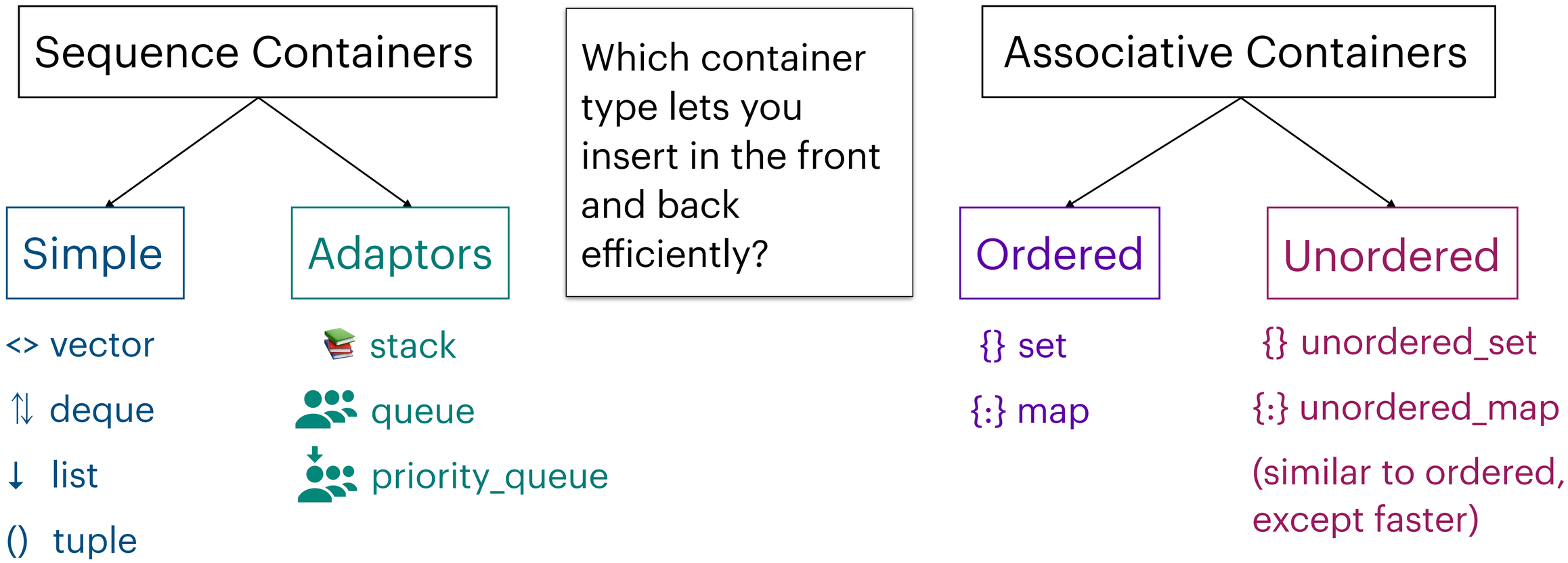


Types of containers

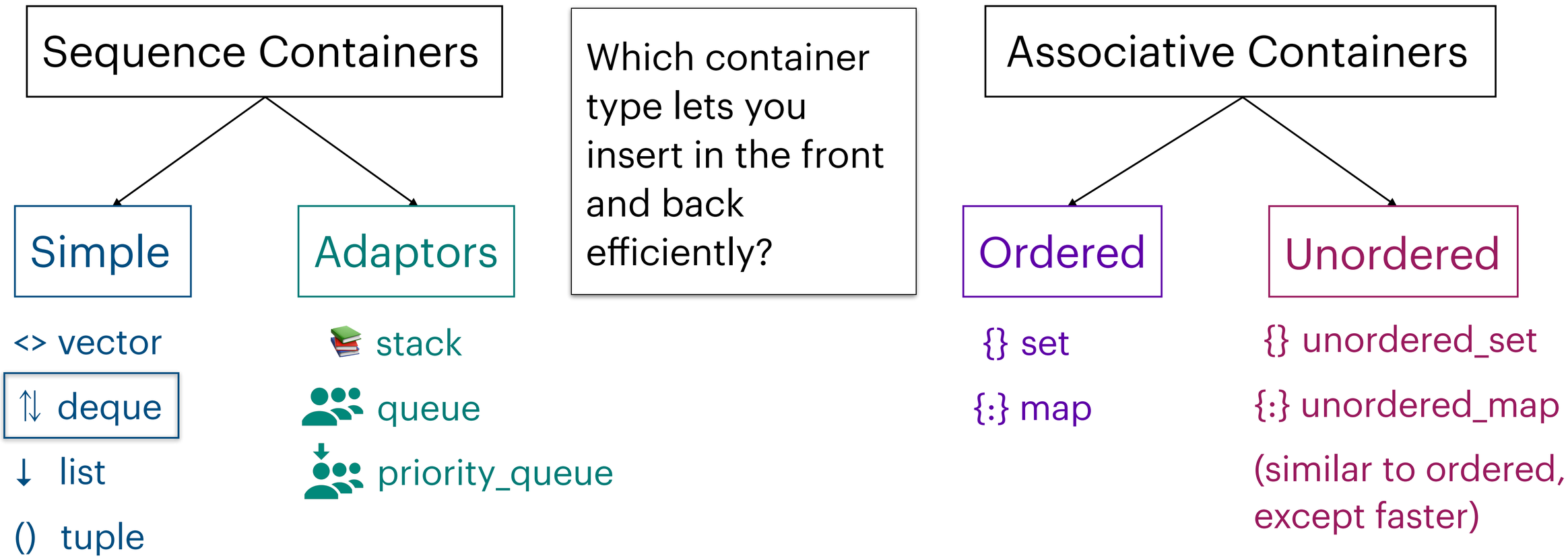




Types of containers



Types of containers



Types of containers

Sequence Containers

Simple

<> vector

⇅ deque

↓ list

() tuple

Adaptors

 stack

 queue

 priority_queue

Which containers require a comparison operator defined over the type of its elements?

Associative Containers

Ordered

{} set

{:} map

Unordered

{} unordered_set

{:} unordered_map

(similar to ordered, except faster)

Types of containers

Sequence Containers

Simple

<> vector

⇅ deque

↓ list

() tuple

Adaptors

 stack

 queue

 priority_queue

Which containers require a comparison operator defined over the type of its elements?

Associative Containers

Ordered

{} set

{:} map

Unordered

{} unordered_set

{:} unordered_map

(similar to ordered, except faster)

Types of containers

Sequence Containers

Simple

<> vector

⇅ deque

↓ list

() tuple

Adaptors

📚 stack

queue

priority_queue

Which containers require a comparison operator defined over the type of its elements?

What can we do to avoid this?

Associative Containers

Ordered

{} set

{:} map

Unordered

{} unordered_set

{:} unordered_map

(similar to ordered, except faster)

Types of containers

Sequence Containers

Simple

<> vector

⇅ deque

↓ list

() tuple

Adaptors

📚 stack

queue

priority_queue

Which containers require a comparison operator defined over the type of its elements?

What can we do to avoid this?

Associative Containers

Ordered

{} set

{:} map

Unordered

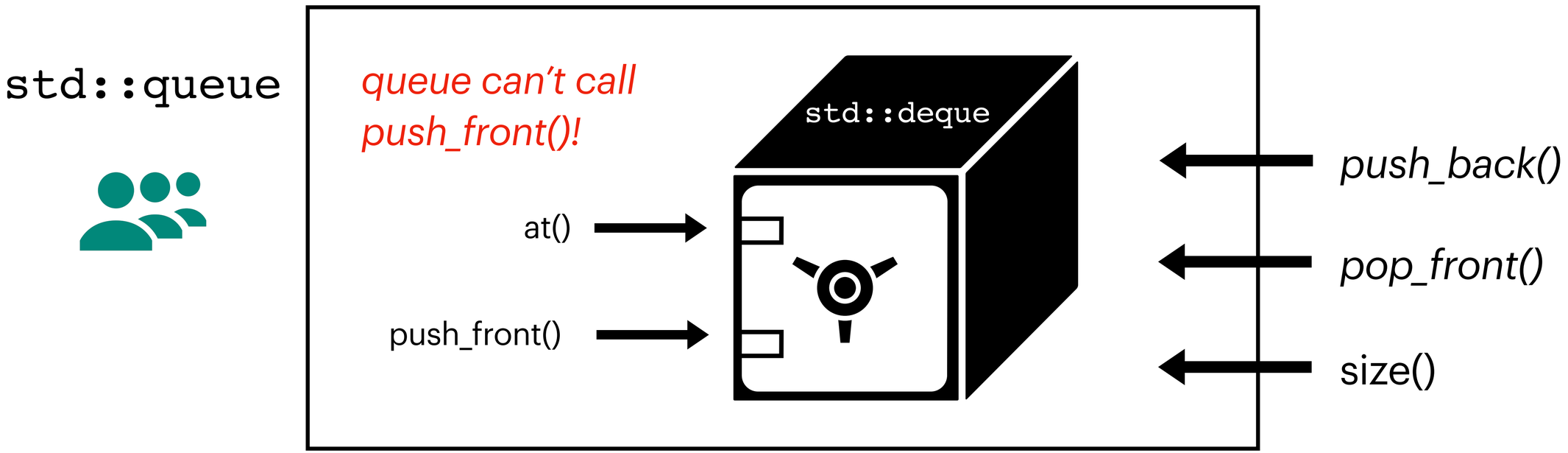
{} unordered_set

{:} unordered_map

(similar to ordered, except faster)

How do you design a STL stack?

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



`std::deque` provides fast insertion anywhere

std::deque has the exact same functions as std::vector + push_front and pop_front.

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

Stanford “Vector” vs STL “vector” (a review)

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

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

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]; // c, or vec.back()

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

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

Looping over collections

- How do we loop over vectors?

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

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

Looping over collections

- Looks like we have a handle on looping over vectors!
- How about sets? (or maps? or queues?)

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

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

Looping over collections

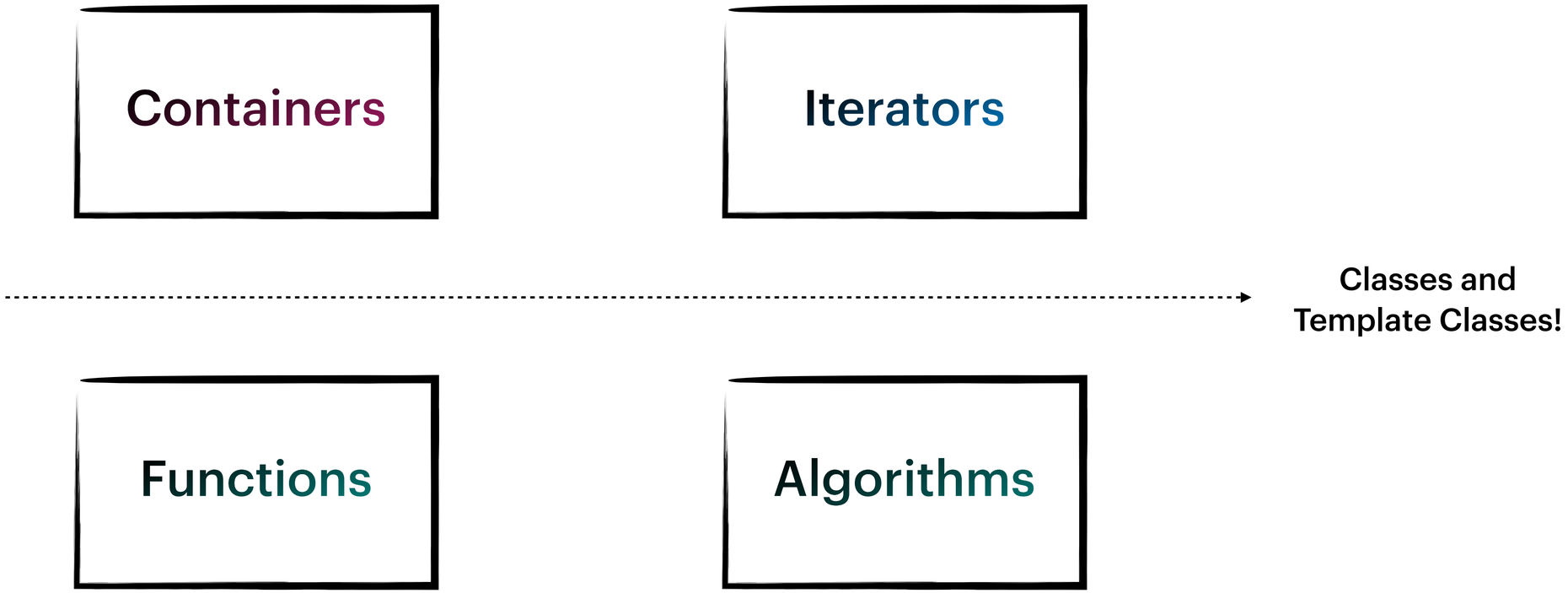
- Looks like we have a handle on looping over vectors!
- How about sets? (or maps? or queues?)

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

std::set<int> s{3, 1, 4, 1, 5, 9};
for (uhh; wait a sec; what++) {
  const auto& elem = idek;
  cout << elem << endl;
}
```

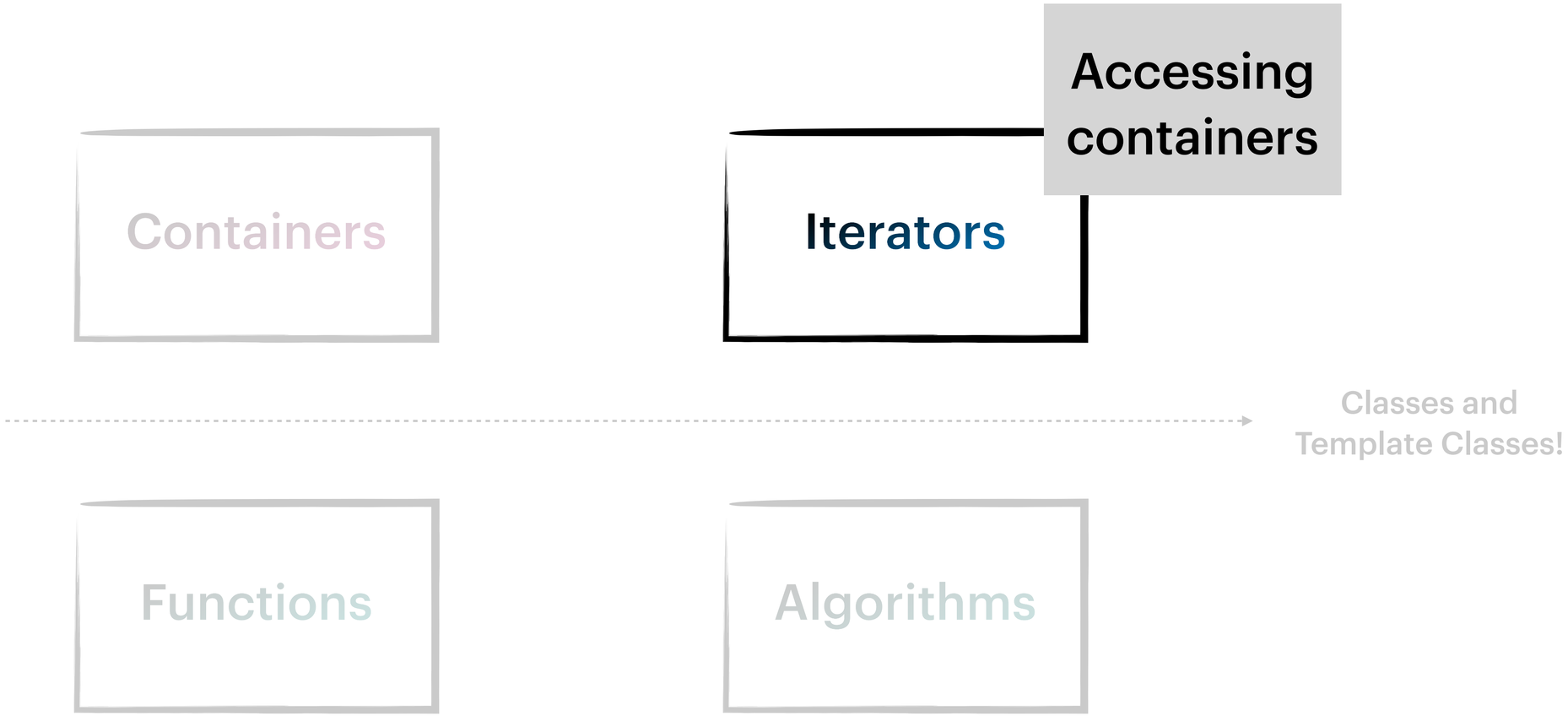
Iterators 

A way to access all containers programmatically!

What's in the STL?



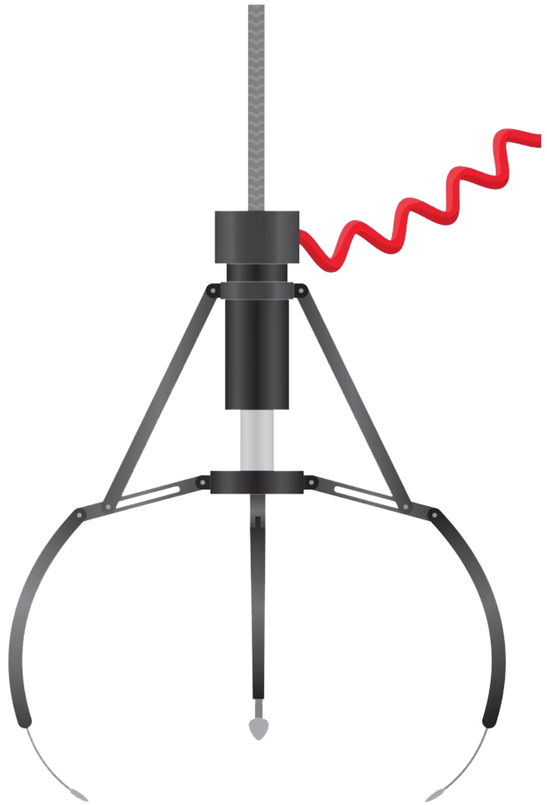
What's in the STL?



Iterations allow iterating over *any* container

whether ordered, unordered, sequence, or associative!

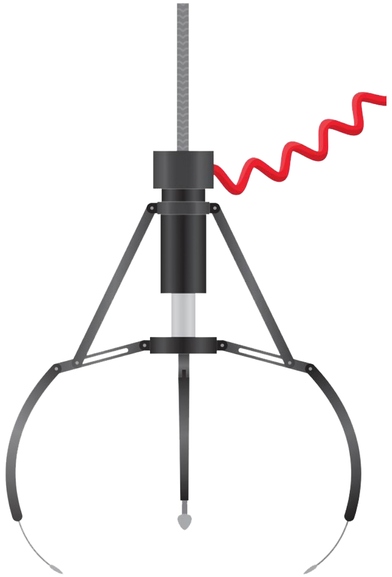
An iterator is like a "claw"!



An iterator is like a "claw"!

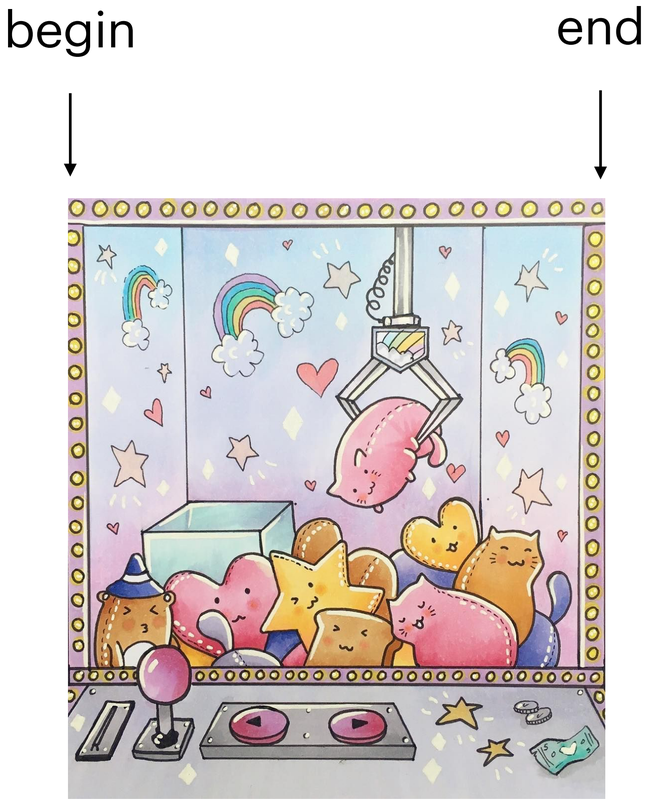
With iterators (the "claw"), we can:

- move forward or backward
  - according to some order (which order)?
- retrieve an element
- check if two claws are in the same place



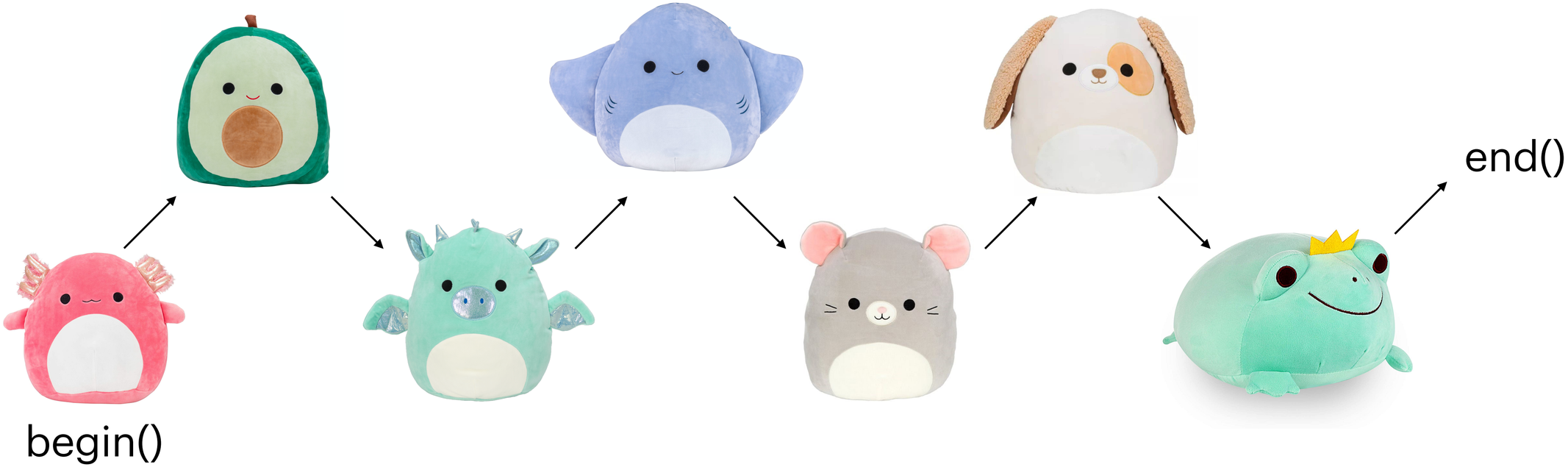
From containers (the "machine"), we can obtain:

- outer bounds (beginning and end)
- a direction/order to move in



Key idea: iterator has an "ordering" over elements

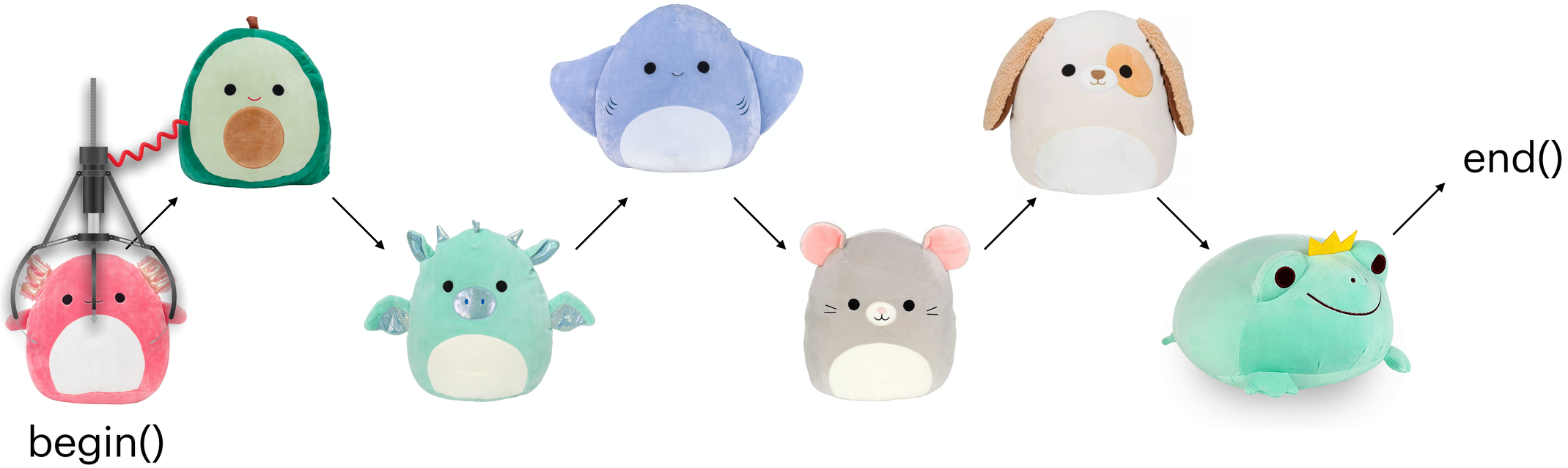
i.e. it knows what the "next" element is



Key idea: iterator has an "ordering" over elements

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

```
it = container.begin()
```



Key idea: iterator has an "ordering" over elements

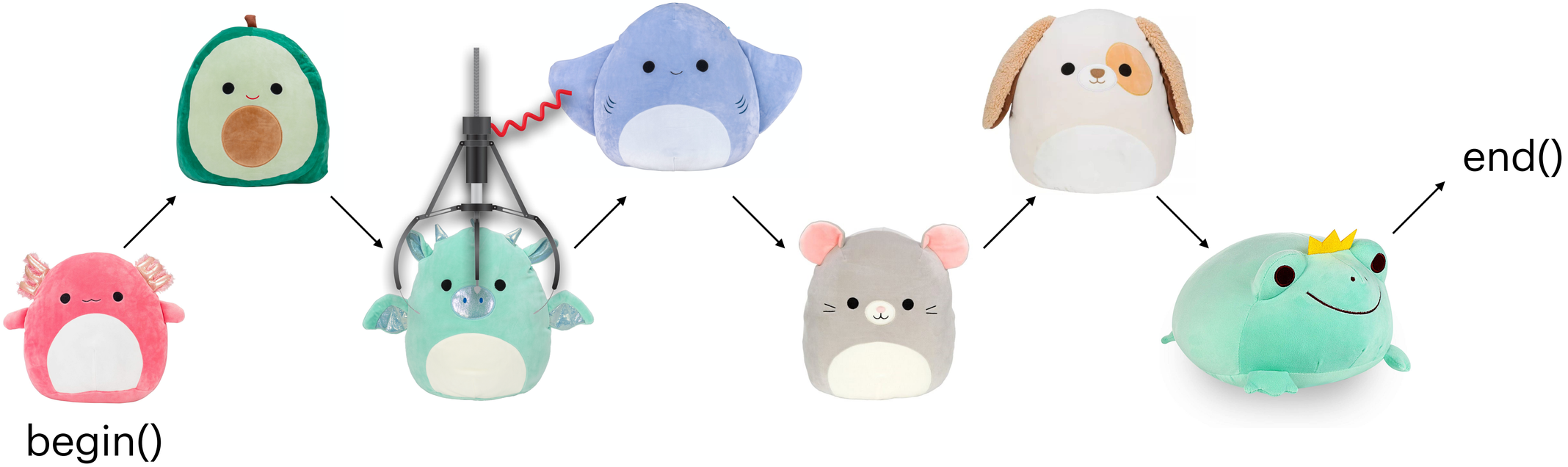
i.e. it knows what the "next" element is

```
++it;
```

Key idea: iterator has an "ordering" over elements

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

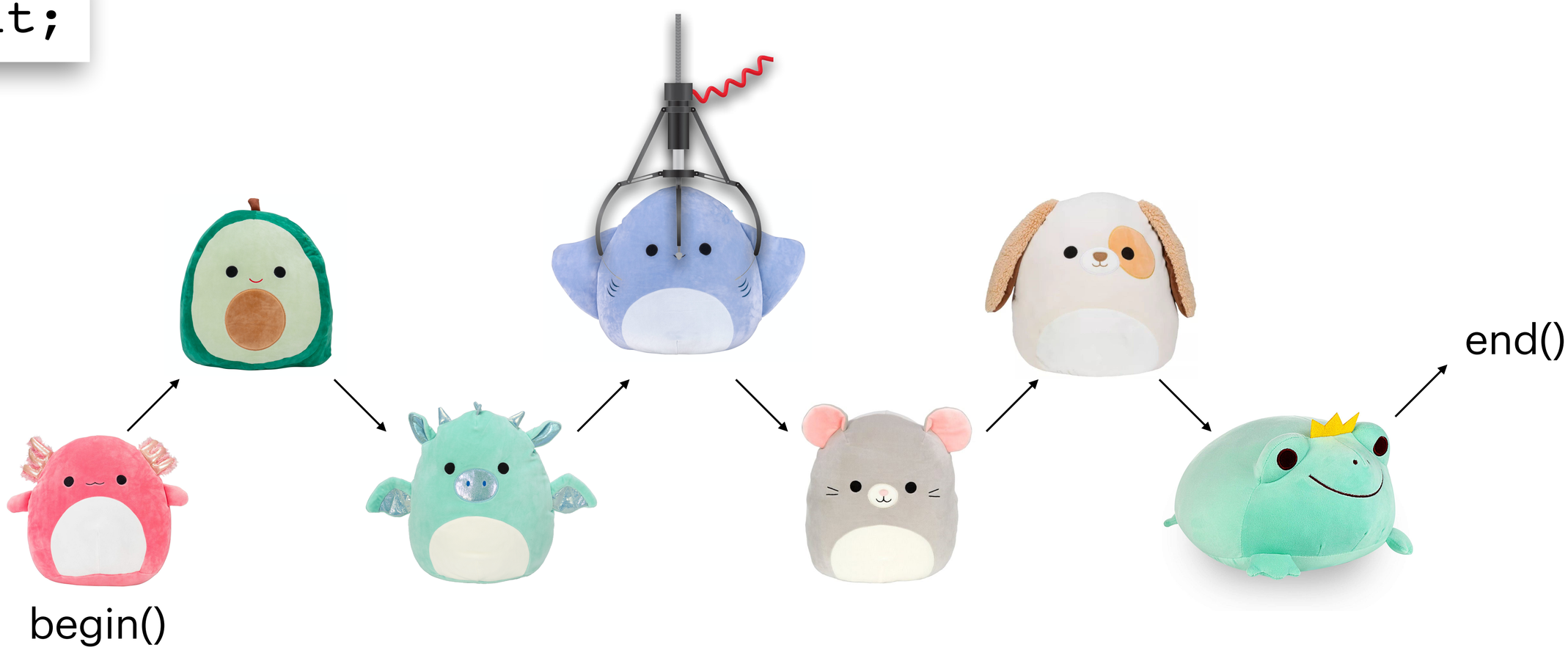
```
++it;
```



Key idea: iterator has an "ordering" over elements

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

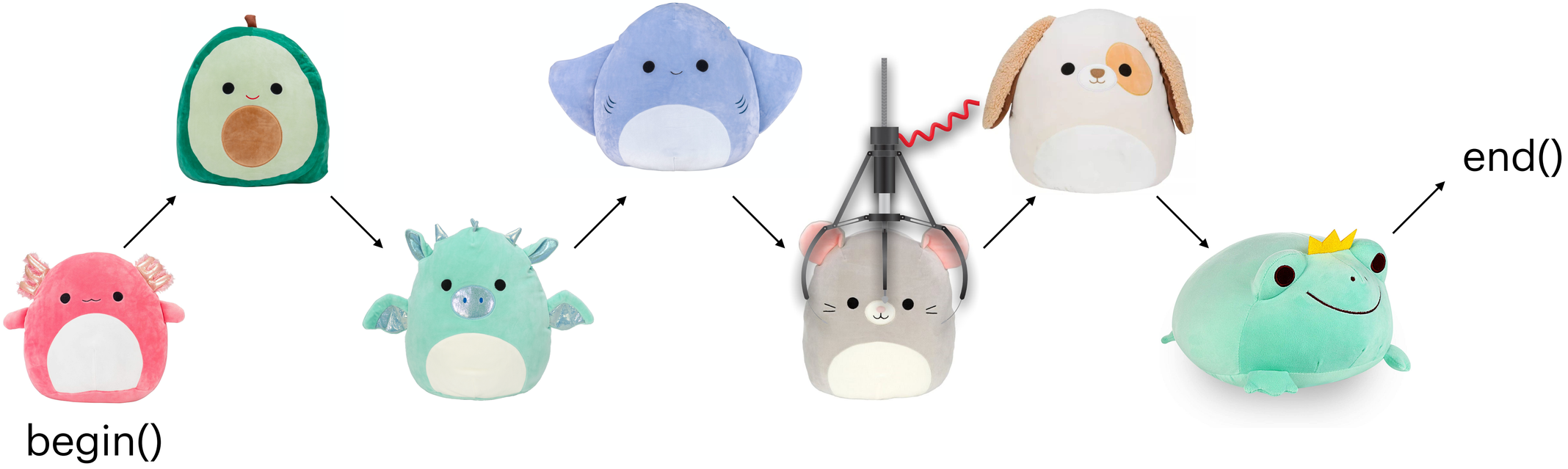
```
++it;
```



Key idea: iterator has an "ordering" over elements

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

```
++it;
```

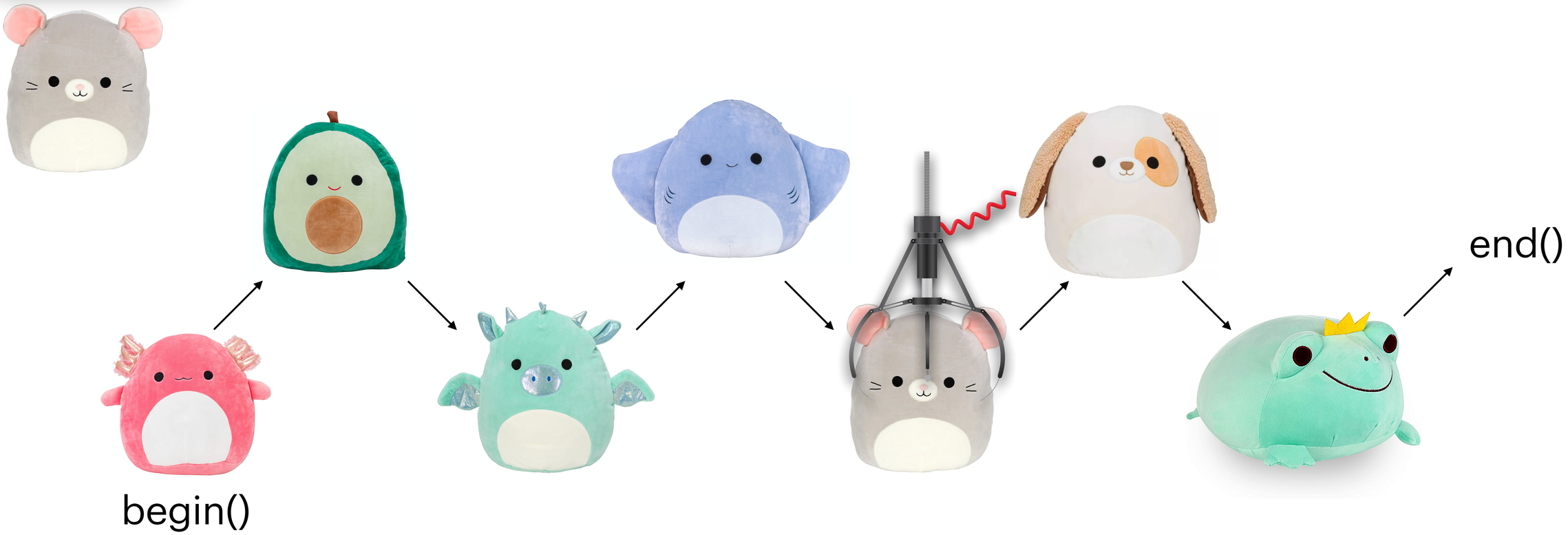


Key idea: iterator has an "ordering" over elements

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

```
*it;
```

to access the element pointed at by the iterator (dereference)



Key idea: iterator has an "ordering" over elements

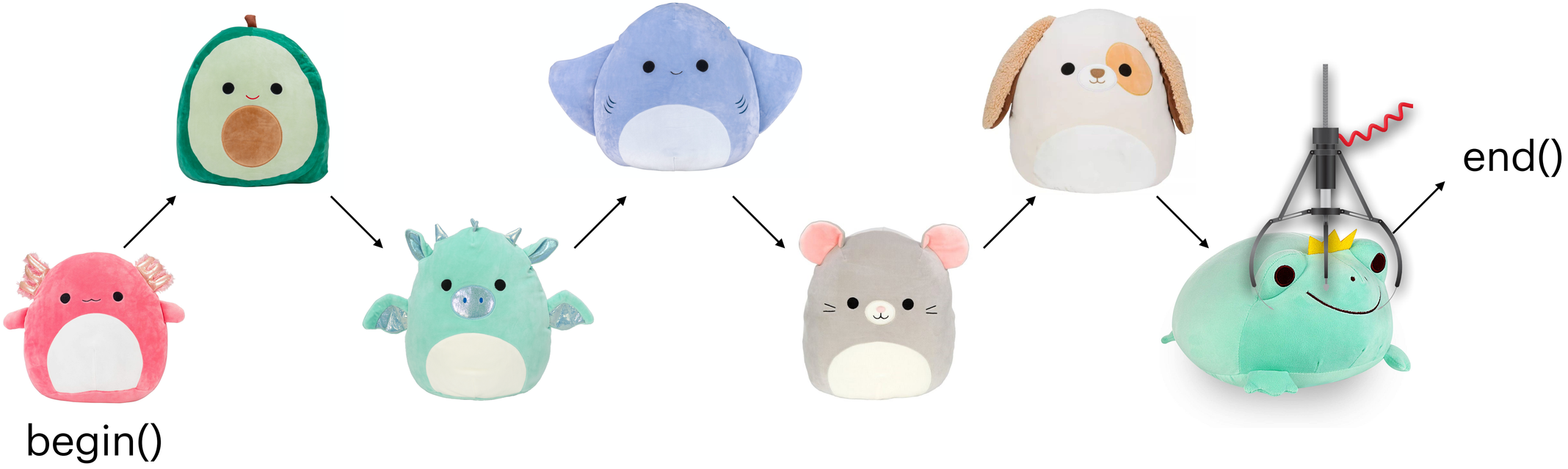
i.e. it knows what the "next" element is

```
++it;
```

Key idea: iterator has an "ordering" over elements

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

```
++it;
it == container.end() // true
```



STL Iterators

- Iterators are objects that point to elements inside containers.
- Each STL container has its own iterator, but all of these iterators exhibit a similar behavior!
- Generally, STL iterators support the following operations:

STL Iterators

- Iterators are objects that point to elements inside containers.
- Each STL container has its own iterator, but all of these iterators exhibit a similar behavior!
- Generally, STL iterators support the following operations:

```
std::set<type> s = {0, 1, 2, 3, 4};
std::set::iterator iter = s.begin();          // at 0
++iter;                                       // at 1
*iter;                                        // 1
(iter != s.end());                            // can compare iterator equality
auto second_iter = iter;                      // "copy construction"
```

STL Iterators

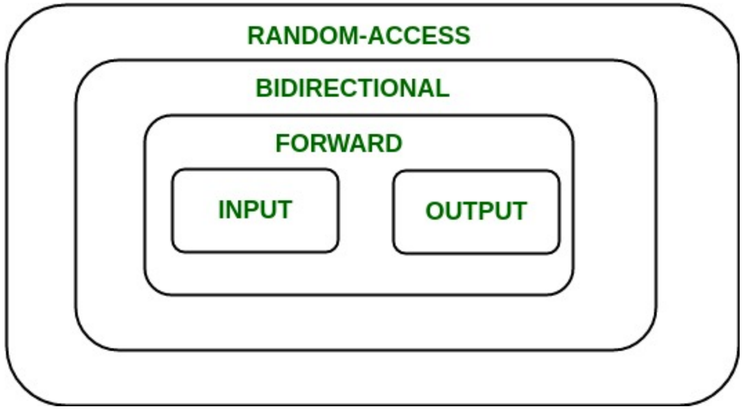
- Iterators are objects that point to elements inside containers.
- Each STL container has its own iterator, but all of these iterators exhibit a similar behavior!
- Generally, STL iterators support the following operations:

```
std::set<type> s = {0, 1, 2, 3, 4};
std::set::iterator iter = s.begin();          // at 0
++iter;                                       // at 1
*iter;                                        // 1
(iter != s.end());                            // can compare iterator equality
auto second_iter = iter;                      // "copy construction"
```

Let's get some practice looking at the docs!
https://www.cplusplus.com/reference/iterator/

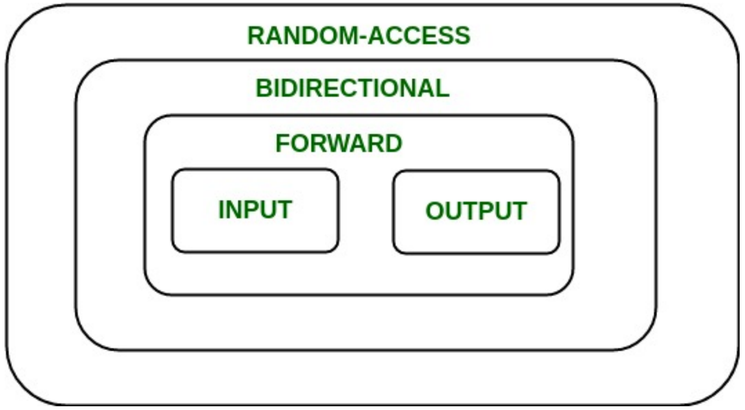
What was that diagram we saw?

- There are a few different types of iterators, since containers are different!



What was that diagram we saw?

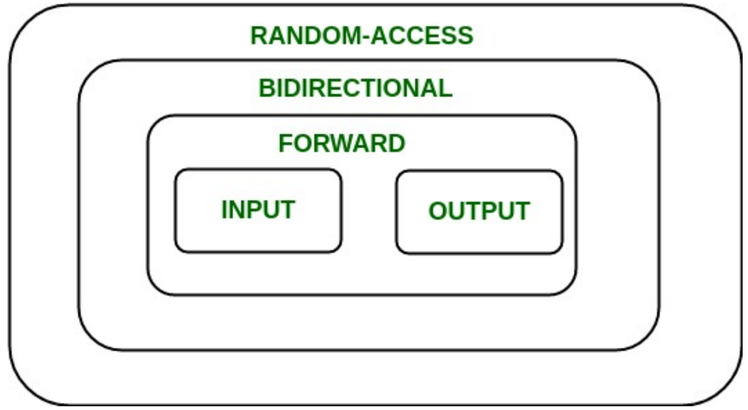
- There are a few different types of iterators, since containers are different!
- All iterators can be incremented (++)



What was that diagram we saw?

- There are a few different types of iterators, since containers are different!
- All iterators can be incremented (++)
- Input iterators can be on the RHS (right hand side) of an = sign:

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



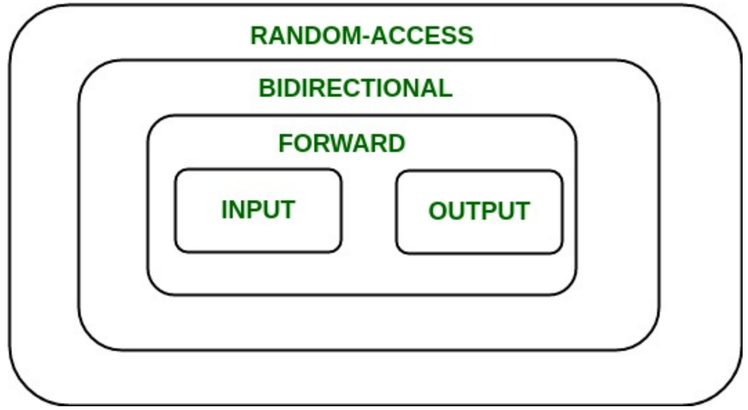
What was that diagram we saw?

- There are a few different types of iterators, since containers are different!
- All iterators can be incremented (++)
- Input iterators can be on the RHS (right hand side) of an = sign:

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

- Output iterators can be on the LHS of =:

```
*elem = value;
```



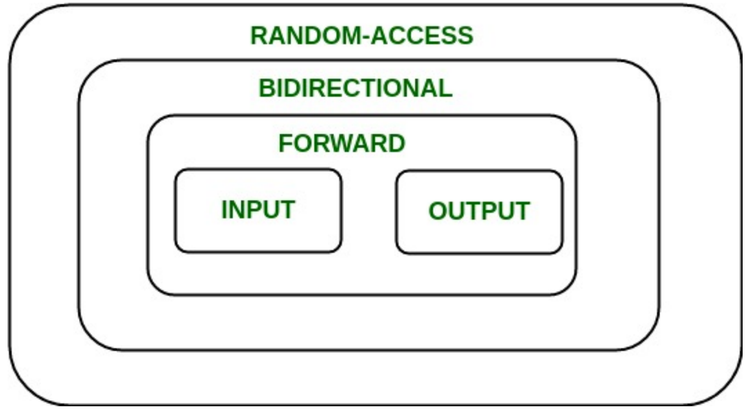
What was that diagram we saw?

- There are a few different types of iterators, since containers are different!
- All iterators can be incremented (++)
- Input iterators can be on the RHS (right hand side) of an = sign:

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

- Output iterators can be on the LHS of =:

```
*elem = value;
```

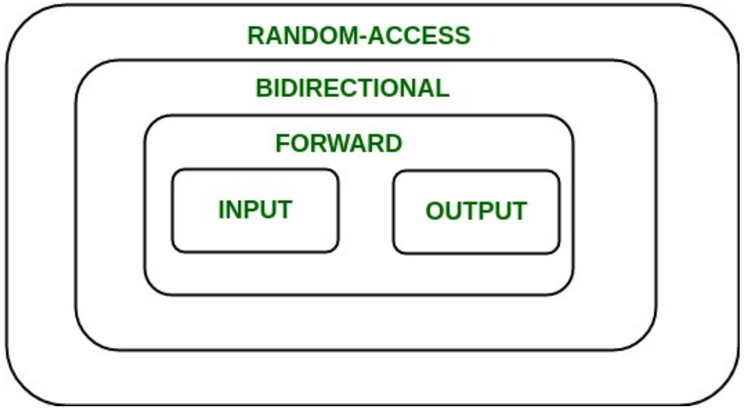


Container	Type of Iterator
Vector	Random-Access
Deque	Random-Access
List	Bidirectional
Map	Bidirectional
Set	Bidirectional
Stack	No Iterator
Queue	No Iterator
Priority Queue	No Iterator

What was that diagram we saw?

- Random access iterators support indexing by integers!

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



Container	Type of Iterator
Vector	Random-Access
Deque	Random-Access
List	Bidirectional
Map	Bidirectional
Set	Bidirectional
Stack	No Iterator
Queue	No Iterator
Priority Queue	No Iterator

a quick tip:

Why ++iter and not iter++?

a quick tip:

Why ++iter and not iter++?

Answer: *++iter returns the value after being incremented!*
iter++ returns the previous value and then increments it. (wastes just a bit of time)

Iterator Practice

Let's practice using iterators and explore their common uses!

Iterator Basics

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

Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
auto it = map.first();                          // std::map::iterator
auto map_elem = *it;                            // what about this?
```

Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
auto it = map.first();                          // std::map::iterator
auto map_elem = *it;                            // std::pair. remember that
                                                // std::map's are implemented as
                                                // std::pairs behind the scenes!
```

Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
auto iter = map.begin();                          // what is *iter?
++iter;

auto iter2 = iter;                                // what is (*iter2).second?
++iter2;                                          // now what is (*iter).first?
// ++iter: go to the next element
// *iter: retrieve what's at iter's position
// copy constructor: create another iterator pointing to the same thing
```

Iterator Basics

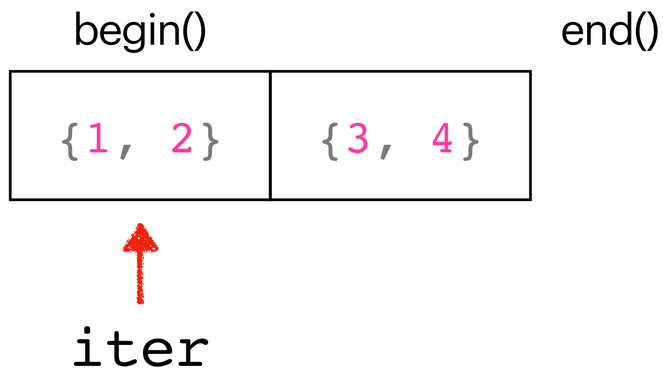
```
std::map<int, int> map {{1, 2}, {3, 4}};
auto iter = map.begin();                    // what is *iter?
++iter;

auto iter2 = iter;                          // what is (*iter2).second?
++iter2;                                    // now what is (*iter).first?
// ++iter: go to the next element
// *iter: retrieve what's at iter's position
// copy constructor: create another iterator pointing to the same thing
```

Declare a map.

Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
auto iter = map.begin();                    // what is *iter?
++iter;

auto iter2 = iter;                          // what is (*iter2).second?
++iter2;                                    // now what is (*iter).first?
// ++iter: go to the next element
// *iter: retrieve what's at iter's position
// copy constructor: create another iterator pointing to the same thing
```

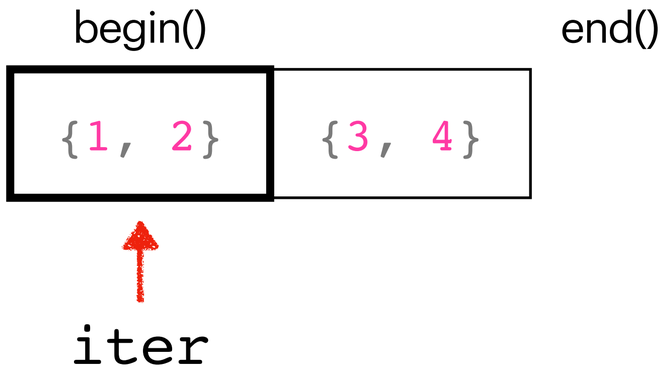
`iter` is a copy of the **`map.begin()`** iterator.



Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
auto iter = map.begin();                                    // what is *iter?
++iter;

auto iter2 = iter;                                          // what is (*iter2).second?
++iter2;                                                    // now what is (*iter).first?
// ++iter: go to the next element
// *iter: retrieve what's at iter's position
// copy constructor: create another iterator pointing to the same thing
```

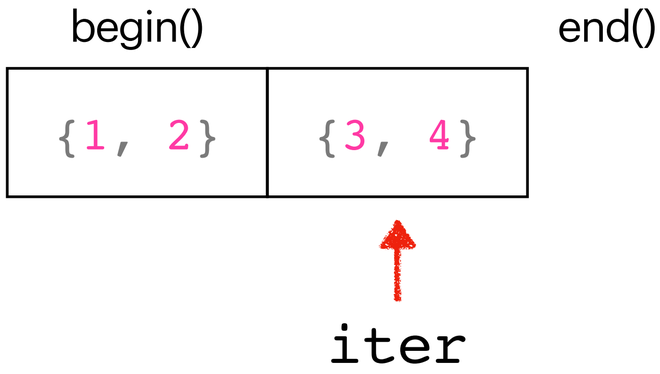
***iter** returns the
std::pair {1, 2}



Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
auto iter = map.begin();                          // what is *iter?
++iter;

auto iter2 = iter;                                // what is (*iter2).second?
++iter2;                                          // now what is (*iter).first?
// ++iter: go to the next element
// *iter: retrieve what's at iter's position
// copy constructor: create another iterator pointing to the same thing
```

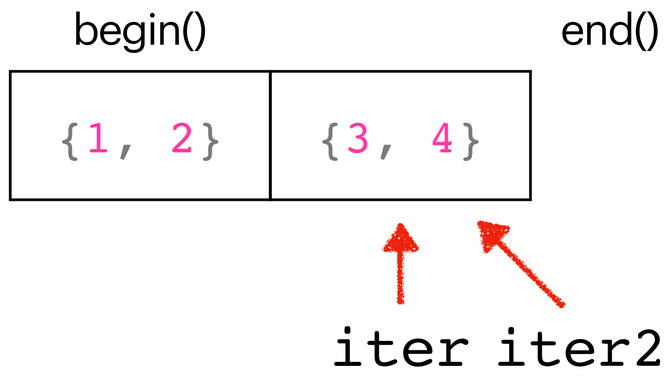
`iter` is incremented to the next element in the map.



Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
auto iter = map.begin();                          // what is *iter?
++iter;

auto iter2 = iter;                                // what is (*iter2).second?
++iter2;                                          // now what is (*iter).first?
// ++iter: go to the next element
// *iter: retrieve what's at iter's position
// copy constructor: create another iterator pointing to the same thing
```

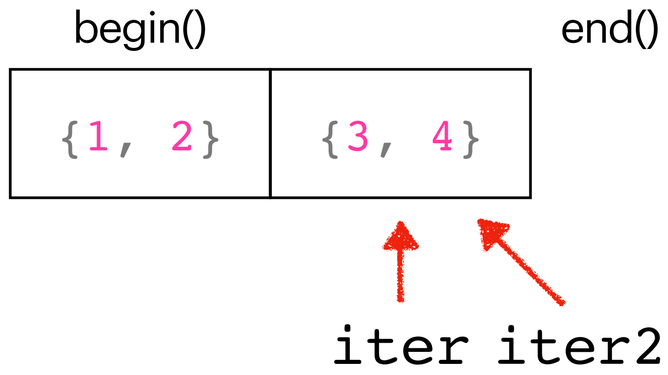
`iter2` is initialized as a completely separate copy of **`iter`** (pointing to the same element).



Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
auto iter = map.begin();                          // what is *iter?
++iter;

auto iter2 = iter;                                // what is (*iter2).second?
++iter2;                                          // now what is (*iter).first?
// ++iter: go to the next element
// *iter: retrieve what's at iter's position
// copy constructor: create another iterator pointing to the same thing
```

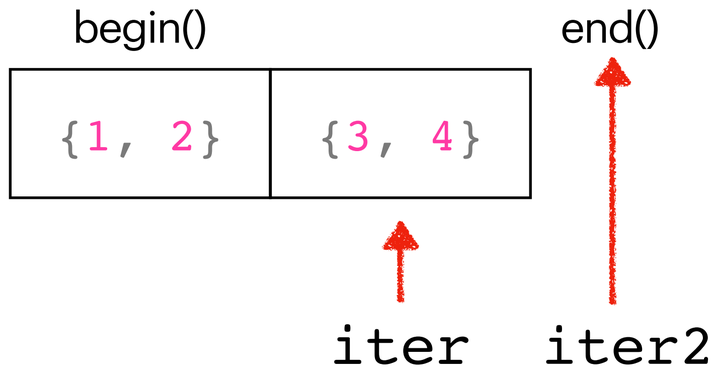
(*iter2).second returns 4.



Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
auto iter = map.begin();                          // what is *iter?
++iter;

auto iter2 = iter;                                // what is (*iter2).second?
++iter2;                                          // now what is (*iter).first?
// ++iter: go to the next element
// *iter: retrieve what's at iter's position
// copy constructor: create another iterator pointing to the same thing
```

iter2 is incremented to the next element in the map.
iter is unaffected.

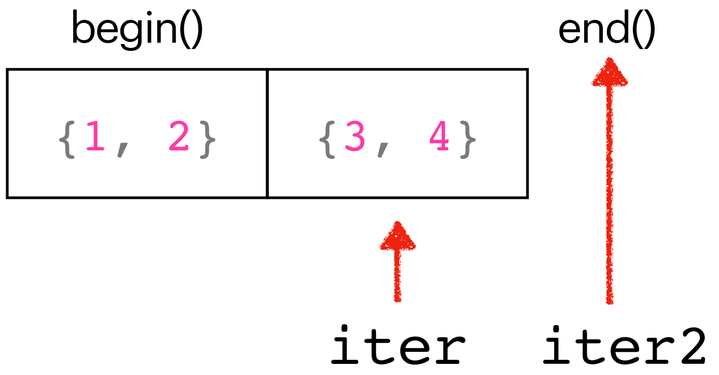


Iterator Basics

```
std::map<int, int> map {{1, 2}, {3, 4}};
auto iter = map.begin();                          // what is *iter?
++iter;

auto iter2 = iter;                                // what is (*iter2).second?
++iter2;                                          // now what is (*iter).first?
// ++iter: go to the next element
// *iter: retrieve what's at iter's position
// copy constructor: create another iterator pointing to the same thing
```

undefined behavior!

`(*map.end())` is not a valid element, so **`(*map.end()).first`** doesn't exist.



Back to looping over collections

Let's try this again!

```
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 at index; // structured binding!
  cout << key << ":" << value << ", " << endl;
}
```

Back to looping over collections

Let's try this again!

```
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 (initialization; termination condition; increment) {
  const auto& [key, value] = retrieve element at index;      // structured binding!
  cout << key << ":" << value << ", " << endl;
}
```

Back to looping over collections

Let's try this again!

```
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!

Back to looping over collections

🎉 You discovered C++'s **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, but why?

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

Pointers

One of the main strengths of C++:
accessing objects by address!

Pointers

- When variables are created, they're given an address in memory.

Pointers

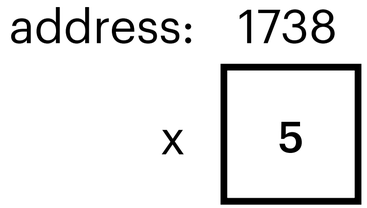
- When variables are created, they're given an address in memory.

```
int x = 5;
vector<int> vec = {0, 1, 2, 3, 4};
std::string str = "Xadia";
```

Pointers

- When variables are created, they're given an address in memory.

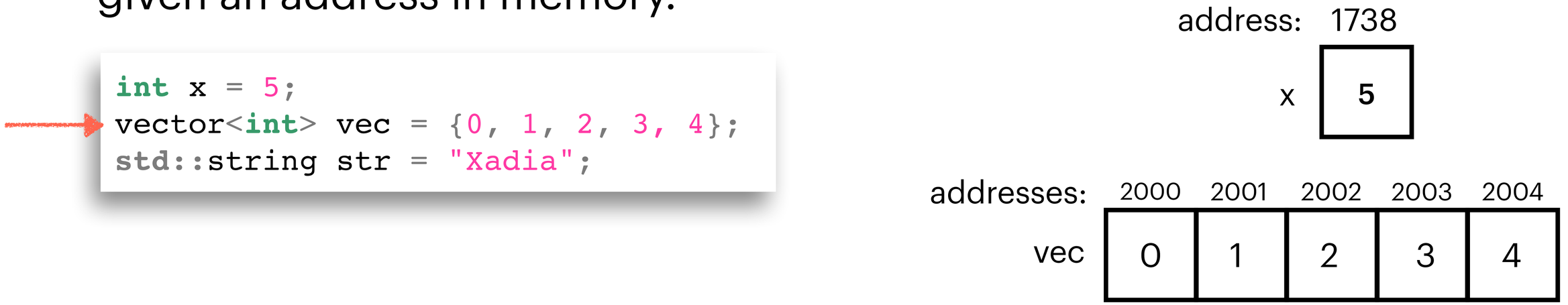
```
int x = 5;
vector<int> vec = {0, 1, 2, 3, 4};
std::string str = "Xadia";
```



Pointers

- When variables are created, they're given an address in memory.

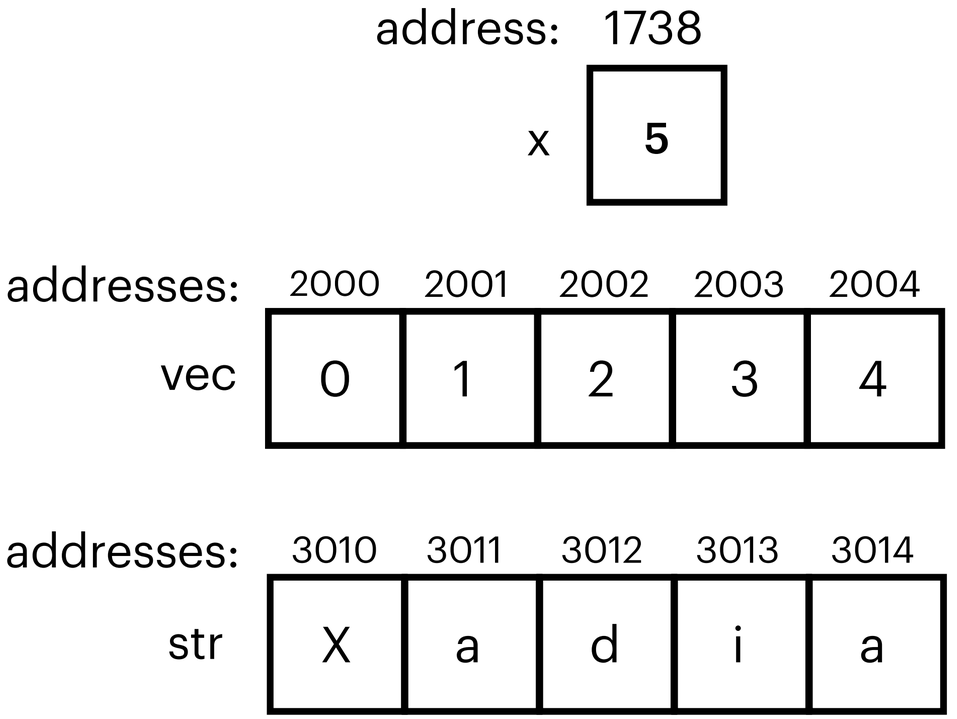
```
int x = 5;
vector<int> vec = {0, 1, 2, 3, 4};
std::string str = "Xadia";
```



Pointers

- When variables are created, they're given an address in memory.

```
int x = 5;
vector<int> vec = {0, 1, 2, 3, 4};
std::string str = "Xadia";
```

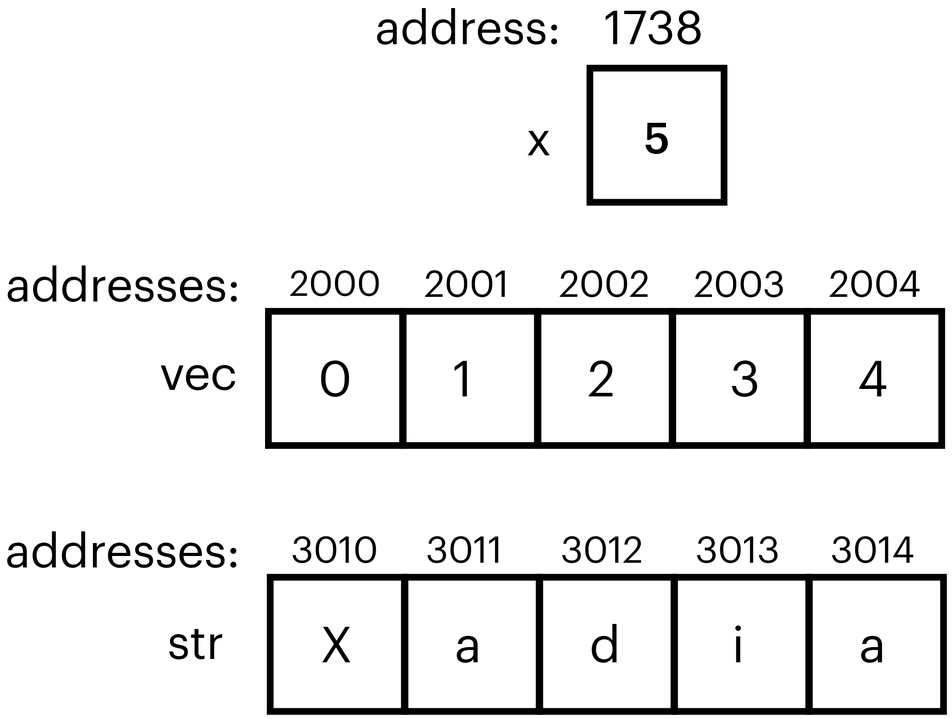


Pointers

- When variables are created, they're given an address in memory.
- Pointers are objects that store an address and type of a variable.

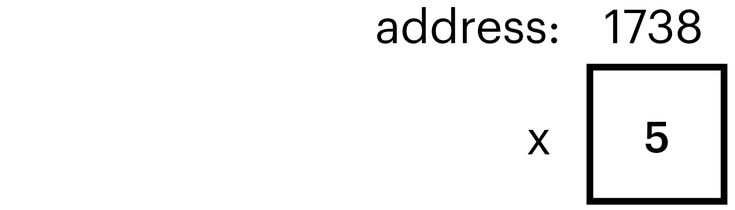
adding a "*" after a data type creates a pointer to a variable of that type.

```
int* p = &x;
int* q = &vec[0];
char* r = &str[0];
```



Pointers

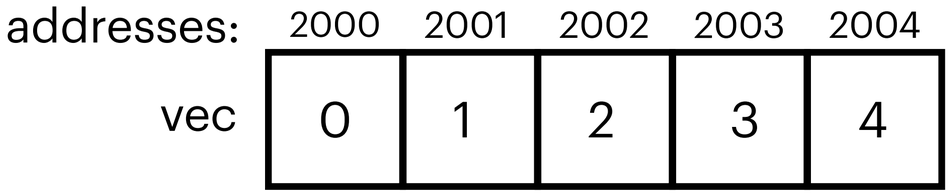
- When variables are created, they're given an address in memory.
- Pointers are objects that store an address and type of a variable.

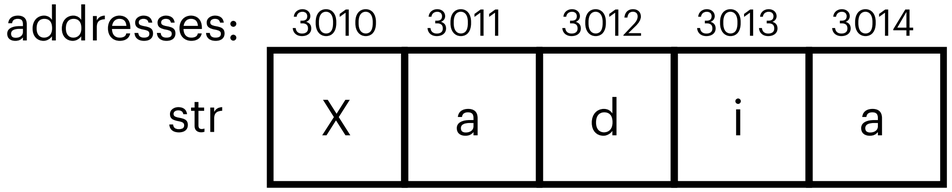


adding a "&" before a variable returns its address, just like passing ***by reference***!

```
void f(int& x) ...
```

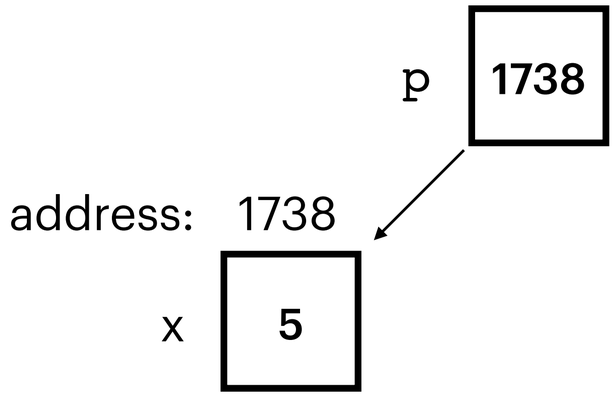
```
int* p = &x;
int* q = &vec[0];
char* r = &str[0];
```





Pointers

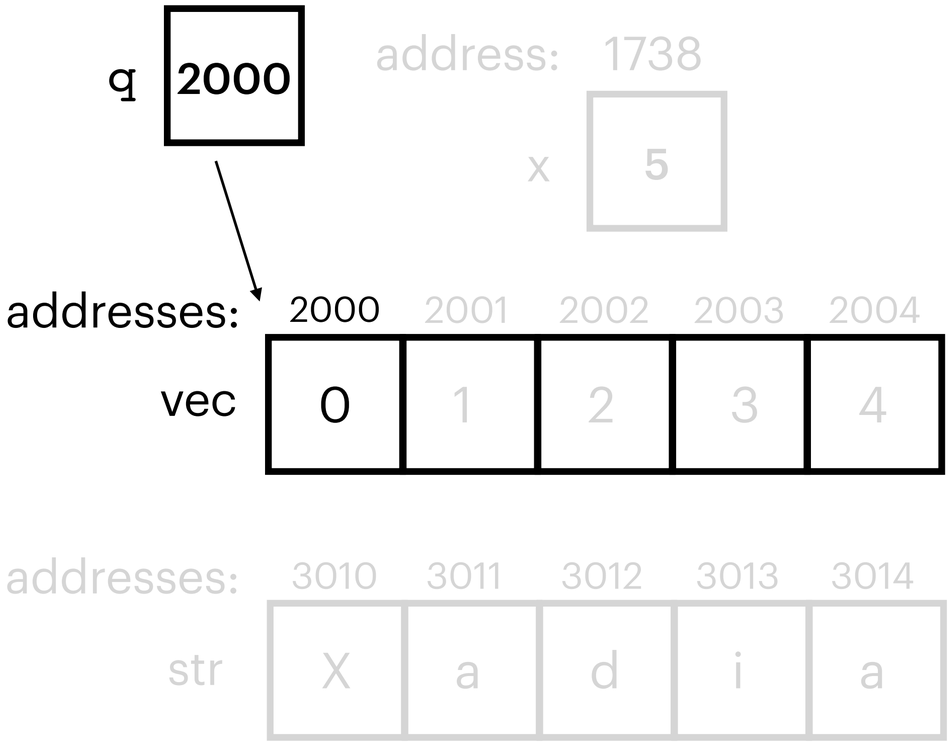
- When variables are created, they're given an address in memory.
- Pointers are objects that store an address and type of a variable.




```
int* p = &x;
int* q = &vec[0];
char* r = &str[0];
```

Pointers

- When variables are created, they're given an address in memory.
- Pointers are objects that store an address and type of a variable.

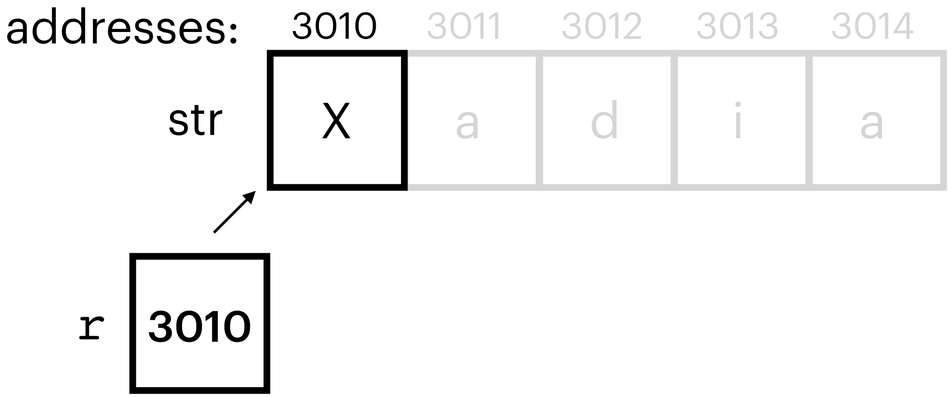



```
int* p = &x;
int* q = &vec[0];
char* r = &str[0];
```

Pointers

- When variables are created, they're given an address in memory.
- Pointers are objects that store an address and type of a variable.


```
int* p = &x;
int* q = &vec[0];
char* r = &str[0];
```



Pointers

```
int x = 5;
int* pointerToInt = &x;                                 // creates pointer to int
cout << *pointerToInt << endl;                          // 5

std::pair<int, int> pair = {1, 2};                      // creates pair
std::pair<int, int>* pointerToPair = &pair;             // creates pointer to pair
cout << (*pair).first << endl;                          // 1
cout << pair->first << endl;                            // 1
```

- To get the value of a pointer, we can *dereference* it (get the object *referenced* by the pointer)

Pointers

```
int x = 5;
int* pointerToInt = &x;                                  // creates pointer to int
cout << *pointerToInt << endl;                           // 5

std::pair<int, int> pair = {1, 2};                       // creates pair
std::pair<int, int>* pointerToPair = &pair;              // creates pointer to pair
cout << (*pair).first << endl;                           // 1
cout << pair->first << endl;                             // 1
```

- To get the value of a pointer, we can *dereference* it (get the object *referenced* by the pointer)

Pointers

```
int x = 5;
int* pointerToInt = &x;                                  // creates pointer to int
cout << *pointerToInt << endl;                           // 5

std::pair<int, int> pair = {1, 2};                       // creates pair
std::pair<int, int>* pointerToPair = &pair;              // creates pointer to pair
cout << (*pair).first << endl;                           // 1
cout << pair->first << endl;                             // 1
```

- To get the value of a pointer, we can *dereference* it (get the object *referenced* by the pointer)
- A shorthand for dereferencing a pointer and then accessing a member variable (doing `someObject.variableName`) is using the `->` operator.

Pointers vs. Iterators

- Iterators are a form of pointers!
- Pointers are more generic iterators
  - can point to any object, not just elements in a container!

Pointers vs. Iterators

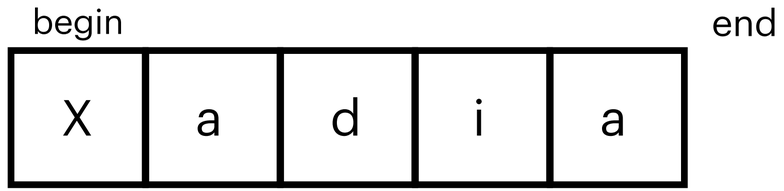
- Iterators are a form of pointers!
- Pointers are more generic iterators
  - can point to any object, not just elements in a container!

```
std::string lands = "Xadia";
// iterator
auto iter = lands.begin();

// syntax for a pointer. don't worry about
the specifics if you're in 106B! they'll be
discussed in the latter half of the course.
char* firstChar = &lands[0];
```

Pointers vs. Iterators

- Iterators are a form of pointers!
- Pointers are more generic iterators
  - can point to any object, not just elements in a container!

```
std::string lands = "Xadia";
// iterator
auto iter = lands.begin();

// syntax for a pointer. don't worry about
the specifics if you're in 106B! they'll be
discussed in the latter half of the course.
char* firstChar = &lands[0];
```



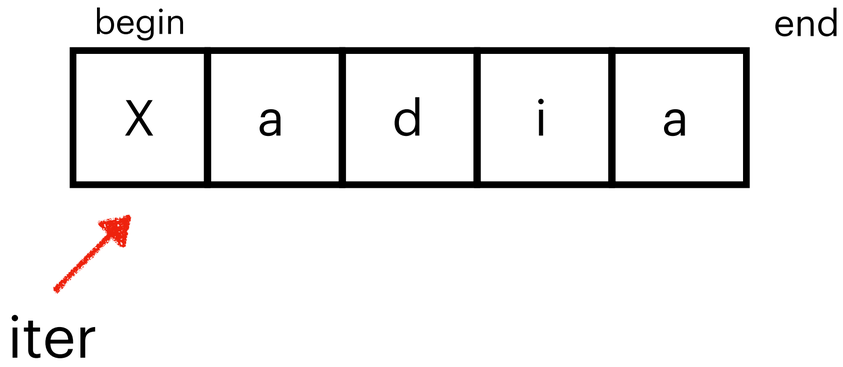
remember arrays (fixed-size lists)?

strings are char arrays.

Pointers vs. Iterators

- Iterators are a form of pointers!
- Pointers are more generic iterators
  - can point to any object, not just elements in a container!

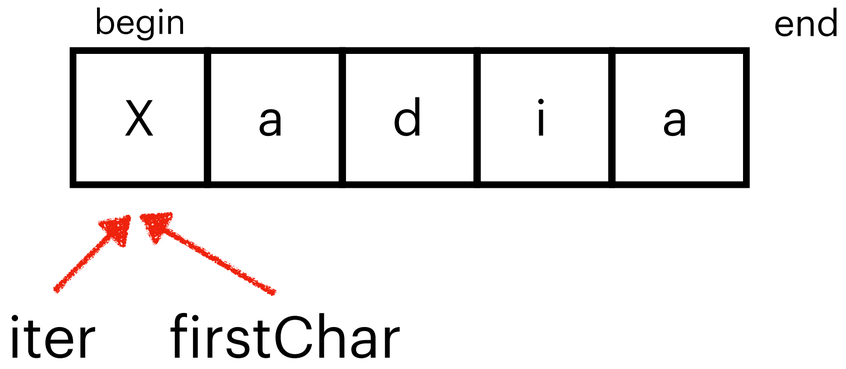
```
std::string lands = "Xadia";
// iterator
auto iter = lands.begin();

// syntax for a pointer. don't worry about
the specifics if you're in 106B! they'll be
discussed in the latter half of the course.
char* firstChar = &lands[0];
```



Pointers vs. Iterators

- Iterators are a form of pointers!
- Pointers are more generic iterators
  - can point to any object, not just elements in a container!

```
std::string lands = "Xadia";
// iterator
auto iter = lands.begin();

// syntax for a pointer. don't worry about
the specifics if you're in 106B! they'll be
discussed in the latter half of the course.
char* firstChar = &lands[0];
```



Iterators and Pointers

Let's see both in action!