




Welcome back! Link to Attendance Form ↓



Pop Quiz: Containers

- Which type(s) lets you insert at the back and front equally efficiently?
- Which type(s) requires a comparison operator on the element type?
 - What type(s) can we use to get around this?
- Which is *usually* faster: `unordered_set` or `set`? Why?

Pop Quiz: Containers (Answers)

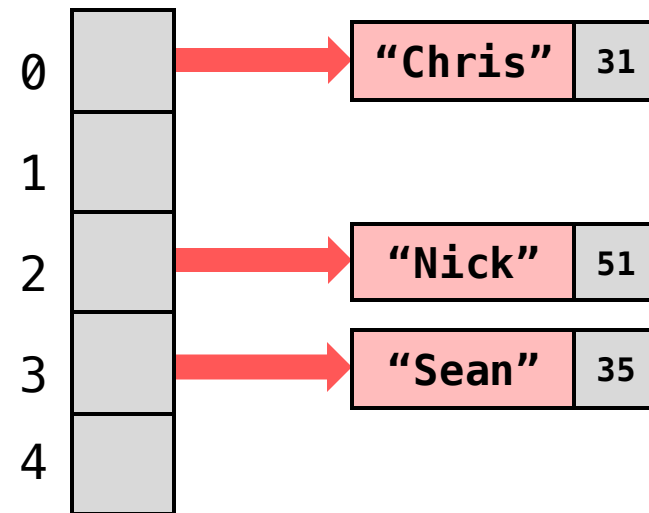
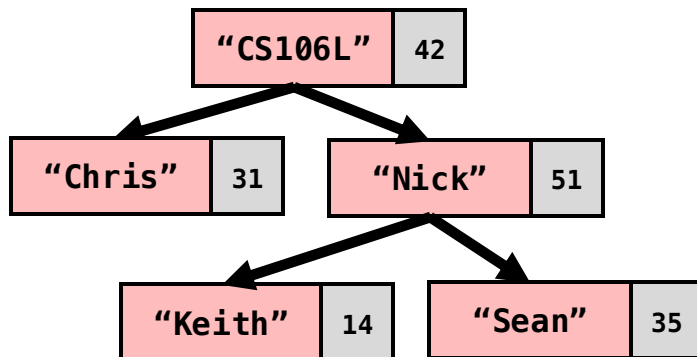
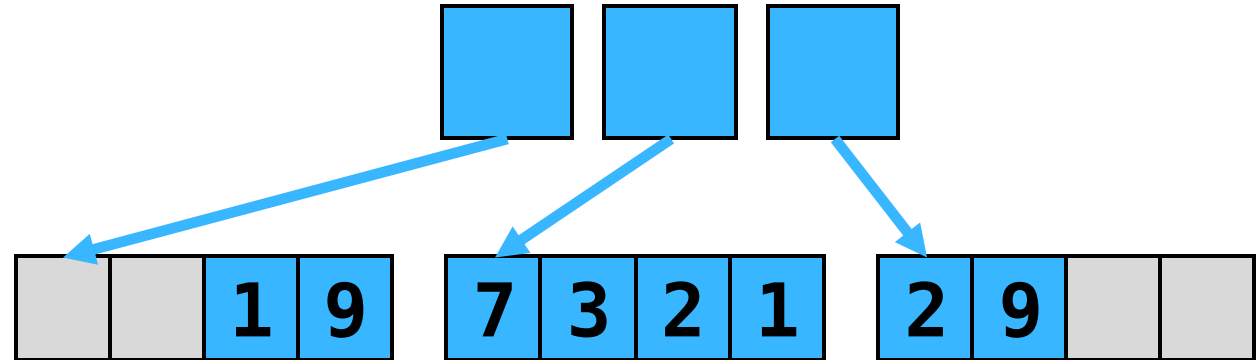
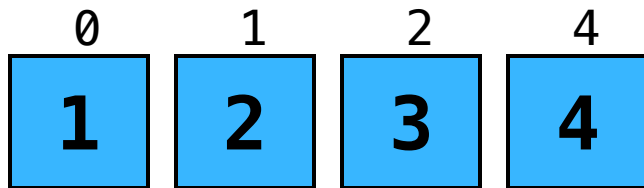
- Which type(s) lets you insert at the back and front equally efficiently?
 -  `std::deque`
- Which type(s) requires a comparison operator on the element type?
 -  `std::map`, `std::set`
- Which is *usually* faster: `unordered_set` or `set`? Why?
 -  `std::unordered_set` (Hashing + small load factor)!

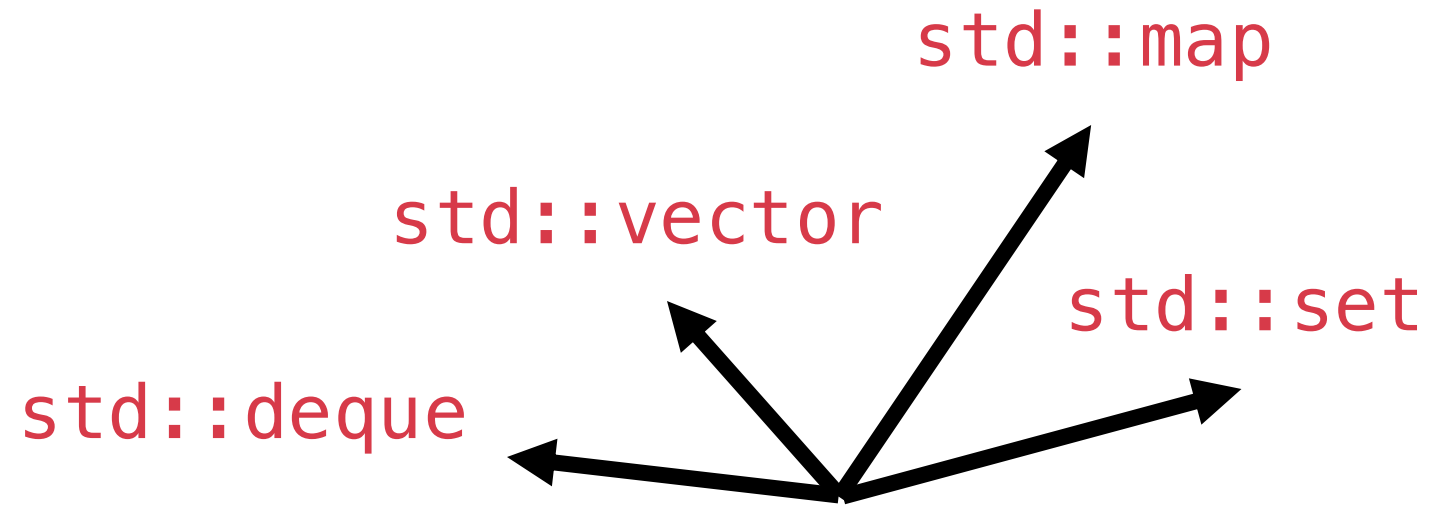
What questions do you have?



bjarne_about_to_raise_hand

Last Time: Containers



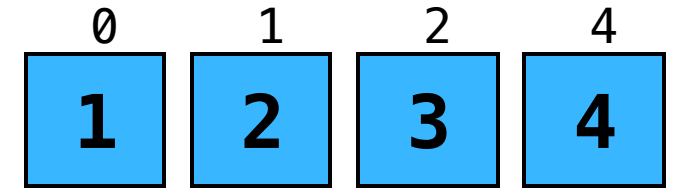


```
for (const auto& elem : container)
```

How does this work?

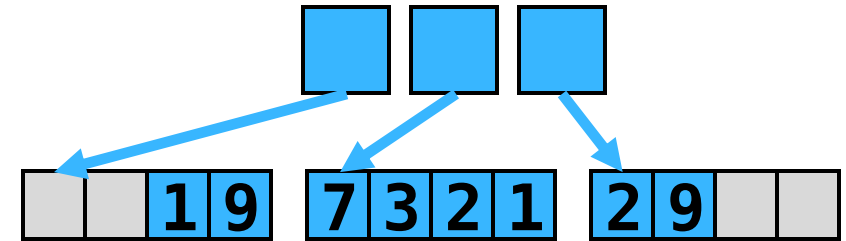
For-each loops... huh?

```
std::vector<int> v { 1, 2, 3, 4 };  
  
for (const auto& elem : v) {  
    std::cout << elem << std::endl;  
}
```



For-each loops... huh?

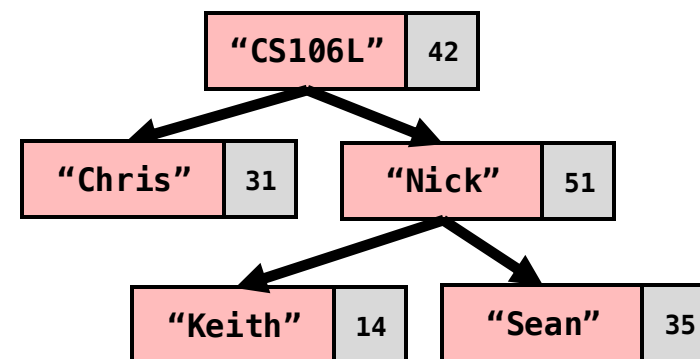
```
std::deque<int> d {  
    1, 9, 7, 3,  
    2, 1, 2, 9  
};  
  
for (const auto& elem : d) {  
    std::cout << elem << std::endl;  
}
```



For-each loops... huh?

```
std::map<std::string, int> m {  
    { "Chris", 31 }, { "CS106L", 42 },  
    { "Keith", 14 }, { "Nick", 51 },  
    { "Sean", 35 },  
};
```

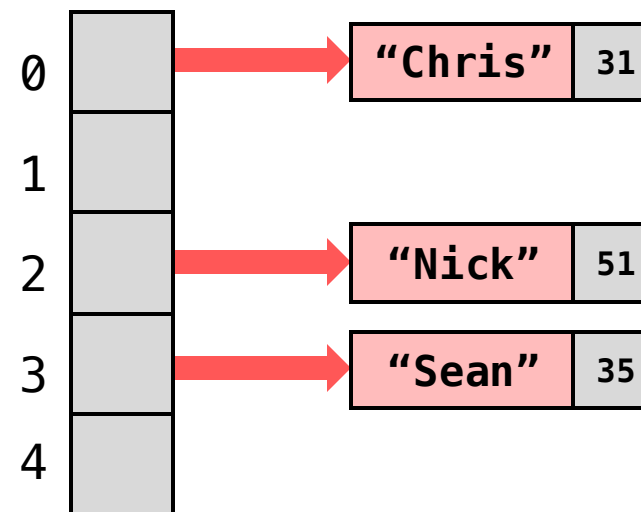
```
for (const auto& pair : m) {  
    std::cout << pair.first << " ";  
    std::cout << pair.second;  
}
```



For-each loops... huh?

```
std::unordered_map<string, int> m
{
    { "Chris", 31 }, { "Nick", 51 },
    { "Sean", 35 },
};

for (const auto& pair : m) {
    std::cout << pair.first << " ";
    std::cout << pair.second;
}
```



```
for (const auto& elem : container)
```



How does this work?

Lecture 6: Iterators

CS106L, Winter 2025

The Standard Template Library (STL)

Containers

How do we store groups of things?

Iterators

How do we traverse containers?

Functors

How can we represent functions as objects?

Algorithms

How do we transform and modify containers in a generic way?

Today's Agenda

- Iterator Basics
 - What even is an iterator?
- Iterator Types
 - Iterators are organized by their properties
- Pointers and Memory
 - What is a pointer? What is memory?

What questions do you have?



bjarne_about_to_raise_hand

Iterator Basics

Question: How do we iterate?

```
std::vector<int> v {1,2,3,4};  
for (size_t i = 0; i < v.size(); i++) {  
    const auto& elem = v[i];  
}  
  
for (var-init; condition; increment) {  
    const auto& elem = /* grab element */;  
}
```

Question: How do we iterate?

`for (auto e : s)`
is not allowed
...for now



```
for (var-init; condition; increment)
    const auto& elem = /* grab element */;
}
```

```
std::set<int> s {1,2,3,4};
for (uhhh; ummm; what?) {
    const auto& elem = /* haelp 🙄🙄 */;
}
```

**We need something to track where we are
in a container... sort of like an index**

Introducing *iterators* 😎😎

C++ iterators are like a “claw” in a claw machine

The claw can:

1. Grab a toy
2. Move forward
3. Check if we're done

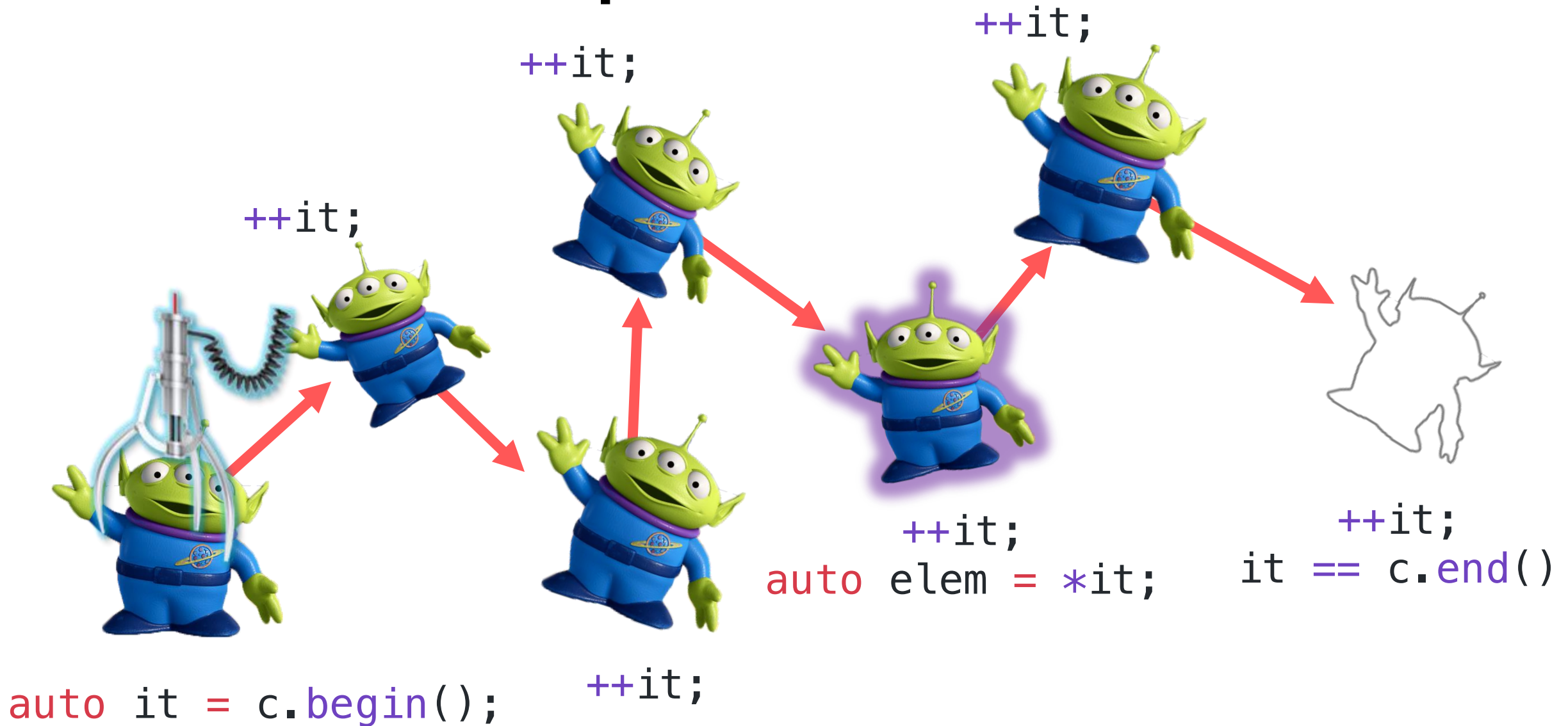


The machine can:

1. Tell us where to start
2. Tell us when to stop



C++ Iterators Example



Containers and iterators *work together* to allow iteration

Container Interface

`container.begin()`

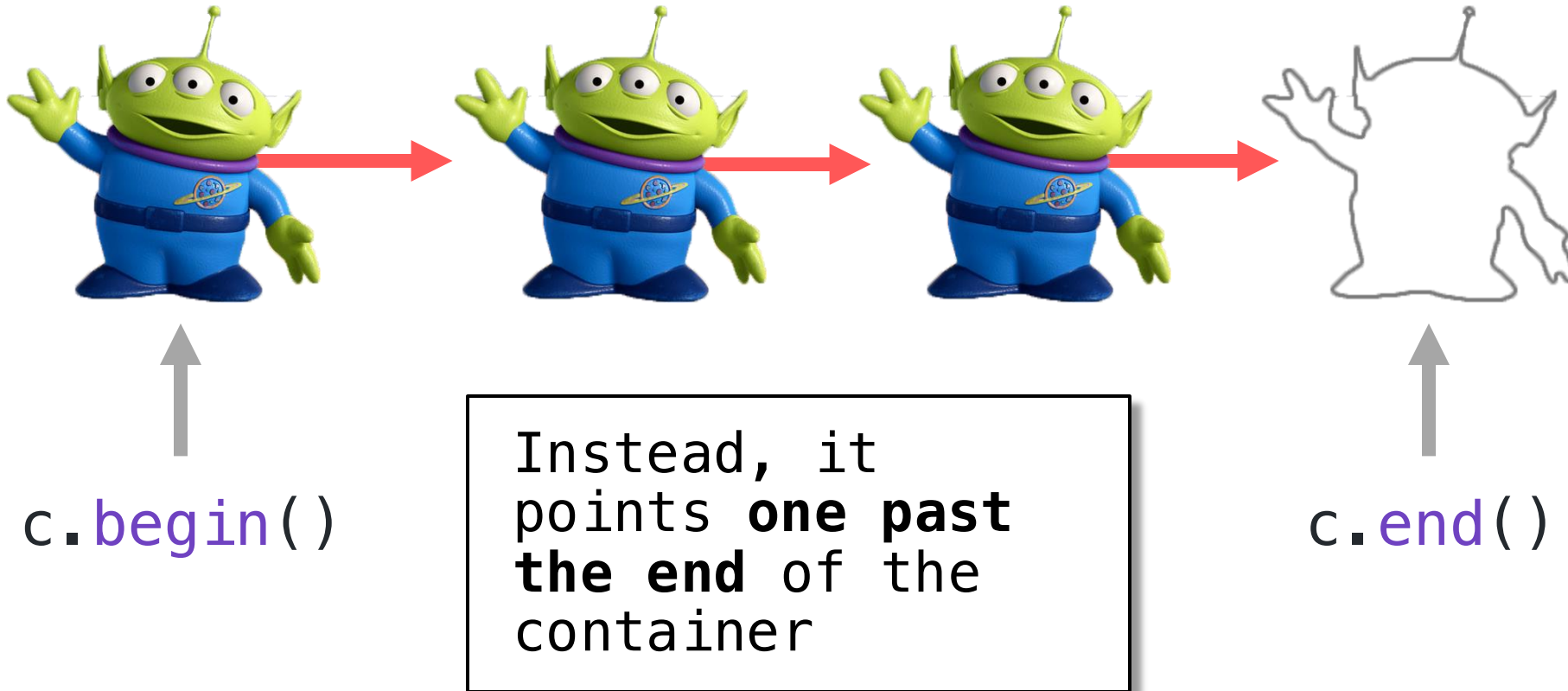
Gets an iterator to
the **first element**
of the container
(assuming non-empty)

`container.end()`

Gets a **past-the-end**
iterator

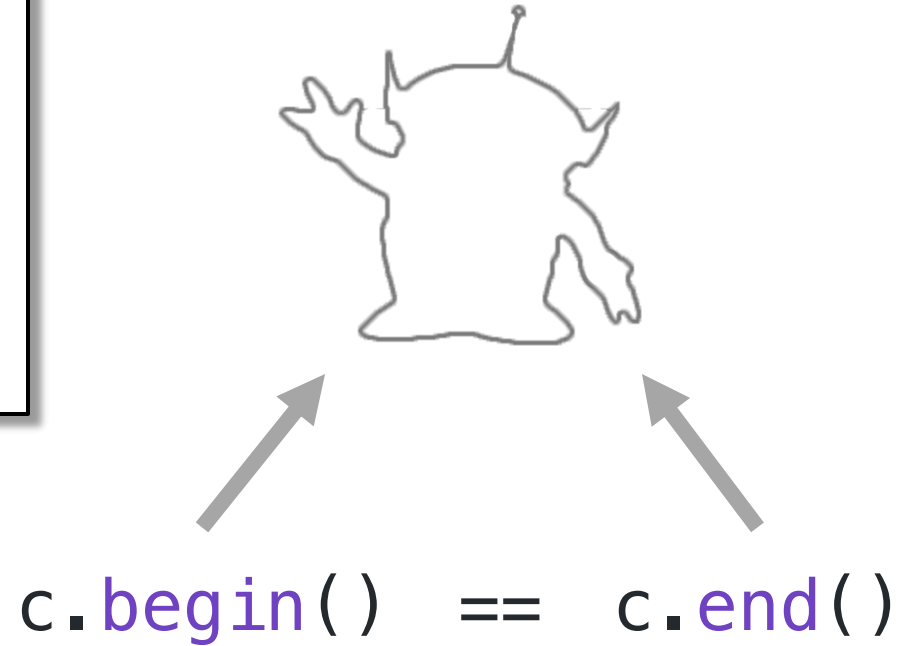
That is, an iterator to
one element **after** the
end of the container

`end()` never points to an element!



`end()` never points to an element!

If `c` is empty,
then `begin()` and
`end()` are equal!



Iterator Interface

```
// Copy construction  
auto it = c.begin();
```

```
// Increment iterator forward  
++it;
```

```
// Dereference iterator -- undefined if it == end()  
auto elem = *it;
```

```
// Equality: are we in the same spot?  
if (it == c.end()) ...
```

We have an answer now!

```
std::set<int> s {1,2,3,4};
```

```
for (var-init; condition; increment) {  
    const auto& elem = /* grab element */;  
}
```

```
for (  
    ;  
    const auto& elem = /* grab element */;  
)
```

`for (auto e : s)`
is not allowed
...for now



We have an answer now!

```
std::set<int> s {1,2,3,4};
```

```
for (var-init; condition; increment) {  
    const auto& elem = /* grab element */;  
}
```

```
for (auto it = s.begin();           ;      ) {  
    const auto& elem = /* grab element */;  
}
```

for (auto e : s)
is not allowed
...for now



We have an answer now!

```
std::set<int> s {1,2,3,4};
```

```
for (var-init; condition; increment) {  
    const auto& elem = /* grab element */;  
}
```

```
for (auto it = s.begin(); it != s.end();      ) {  
    const auto& elem = /* grab element */;  
}
```

`for (auto e : s)`
is not allowed
...for now



We have an answer now!

```
std::set<int> s {1,2,3,4};
```

```
for (var-init; condition; increment) {  
    const auto& elem = /* grab element */;  
}
```

```
for (auto it = s.begin(); it != s.end(); ++it) {  
    const auto& elem = /* grab element */;  
}
```

for (auto e : s)
is not allowed
...for now



We have an answer now!

```
std::set<int> s {1,2,3,4};
```

```
for (var-init; condition; increment) {  
    const auto& elem = /* grab element */;  
}
```

```
for (auto it = s.begin(); it != s.end(); ++it) {  
    const auto& elem = *it;  
}
```

for (auto e : s)
is not allowed
...for now



When you write...

```
for (auto elem : s)
{
    std::cout << elem;
}
```

It's actually this:

```
auto b = s.begin();
auto e = s.end();

for (auto it = b; it != e; ++it)
{
    auto elem = *it;
    std::cout << elem;
}
```

What questions do you have?



bjarne_about_to_raise_hand

We have an answer now!

```
std::set<int> s {1,2,3,4};
```

```
for (var-init; condition; increment) {  
    const auto& elem = /* grab element */;  
}
```

```
for (auto it = s.begin(); it != s.end(); ++it) {  
    const auto& elem = *it;  
}
```

for (auto e : s)
is not allowed
...for now



Guess we're done here!



We have an answer now!

```
std::set<int> s {1,2,3,4};
```

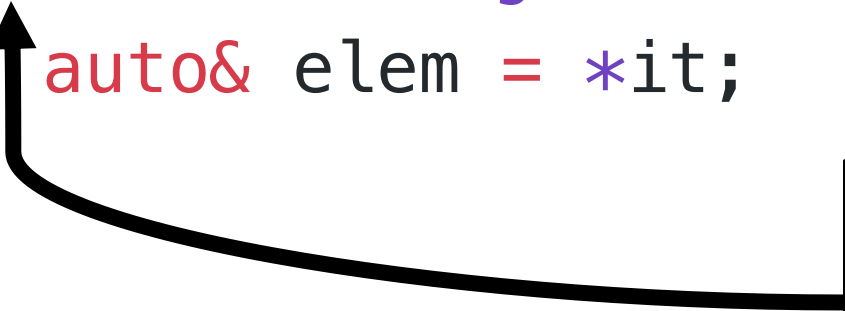
```
for (var-init; condition; increment) {  
    const auto& elem = /* grab element */;  
}
```

```
for (auto it = s.begin(); it != s.end(); ++it) {  
    const auto& elem = *it;  
}
```

for (auto e : s)
is not allowed
...for now



What type is
this?



What are the types?

Using **auto** avoids spelling out long iterator types

```
std::map<int, int> m { {1, 2}, {3, 4}, {5, 6}};  
auto it = m.begin();  
auto elem = *it;           // {1, 2}
```

```
std::map<int, int> m { {1, 2}, {3, 4}, {5, 6}};  
std::map<int, int>::iterator it = m.begin();  
std::pair<int, int> elem = *it;
```

Remember: **using** makes a type alias

```
// Inside <map> header
template <typename K, typename V>
class std::map {
    using iterator = /* some iterator type */;
};
```

```
// Outside <map> header (e.g. main.cpp)
std::map<int, int>::iterator it = m.begin();
```



Iterator types are really long, so we like to use **auto** with iterators

Aside: Why do we use `++it` instead of `it++`?

++it avoids making an unnecessary copy

```
// Prefix ++it  
// Increments it and returns a reference to same object  
Iterator& operator++(int);
```

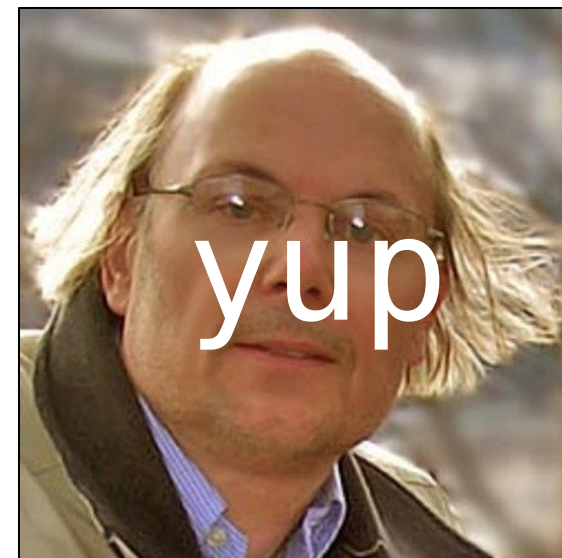
```
// Postfix it++  
// Increments it and returns a copy of the old value  
Iterator operator++();
```

Remember: an iterator can be fully-fledged object, so it's often more expensive to copy than, say, an **int**

Does it actually make a difference?

Bjarne's Thoughts

“



`++i` is sometimes faster than, and is never slower than, `i++`. ... So if you're writing `i++` as a statement rather than as part of a larger expression, why not just write `++i` instead? You never lose anything, and you sometimes gain something.

[source]

What questions do you have?

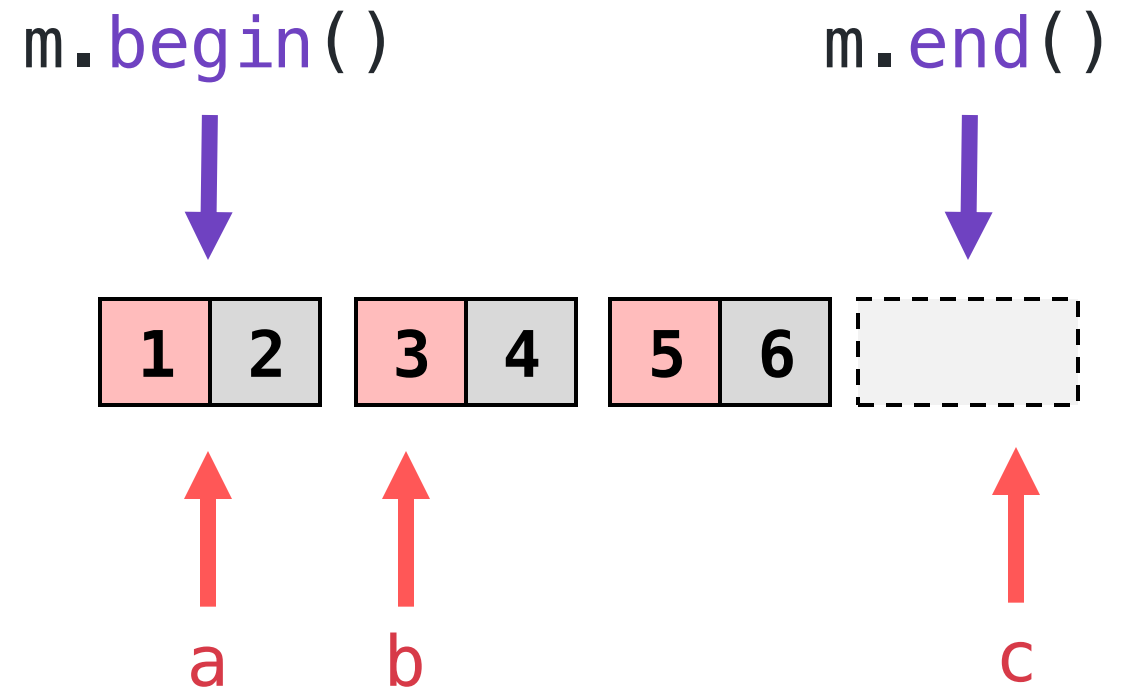


bjarne_about_to_raise_hand

Your Turn

Trace this code with a partner to find out where each iterator points

```
std::map<int, int> m {  
    {1, 2}, {3, 4}, {5, 6}  
};  
auto a = m.begin();  
++a;  
auto b = a;  
++a;  
auto c = ++a;
```



Announcements

Apply to Section Lead!

- Section leading is one of the most rewarding things we've done at Stanford – it's how we're here!
- PLEASE, ask us questions about it :)
- App is due **Thursday, January 30th** or if you're currently enrolled in CS106B, **February 15th**
- [Apply here!](#)



Assignment #1 & OH

- **Assignment 1 is now due Monday at midnight!**
 - We know you are busy and may still be getting the hang of C++!
 - This assignment is a little longer than the others, so take the weekend!
- **Office hours**
 - Jacob's OH will be tomorrow (**1/24**) at **1:30pm outside Turing**

What questions do you have?



bjarne_about_to_raise_hand

Iterator Types

Not all iterators are made equal

All iterators provide these four operations

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

```
++it;
```

```
*it;
```

```
it == c.end()
```

But most provide even more

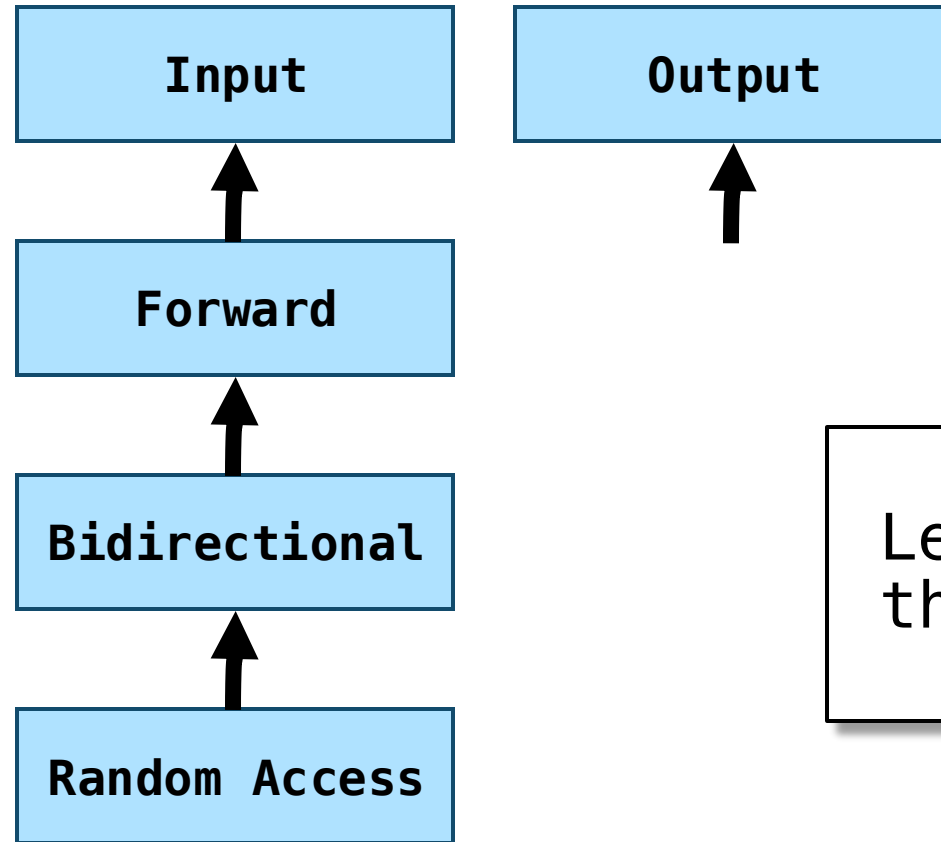
```
--it; // Move backwards
```

```
*it = elem; // Modify
```

```
it += n; // Rand. access
```

```
it1 < it2 // Is before?
```

Iterator types determine their functionality

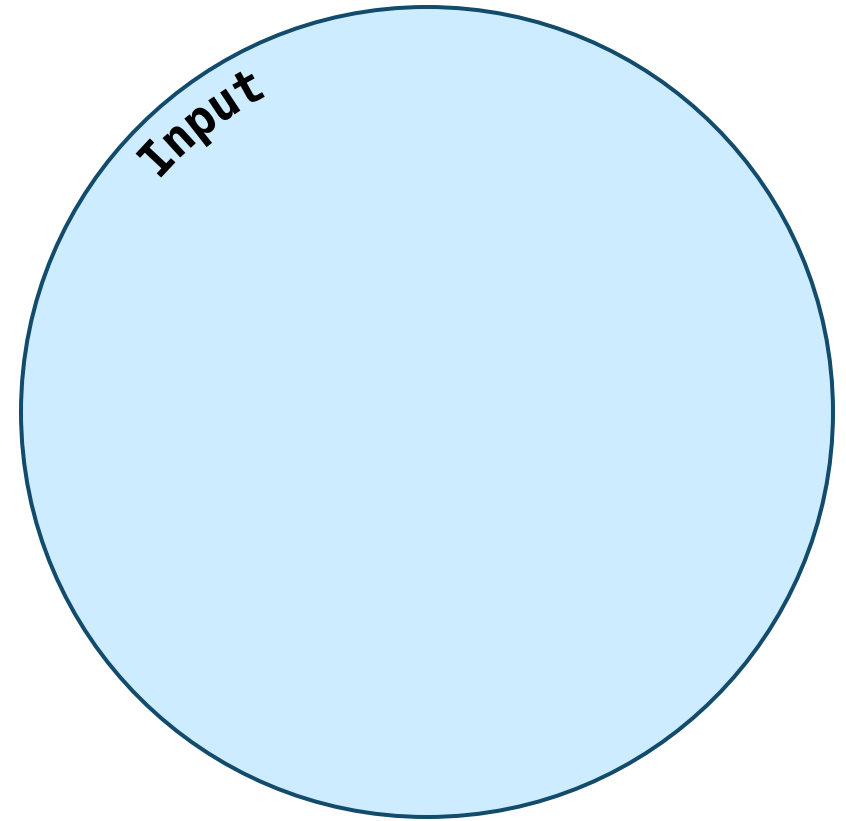


Let's unpack
this!

Input Iterators

- Most basic kind of iterator
- Allows us to read elements

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



**Vivid Venn Diagram of
Vexing Iterators**

Input Iterators: operator->

If the element is a struct, we can access its members with ->

```
struct Bibble {  
    int zarf;  
};
```

Bibble, v.
“To eat and/or drink noisily”

```
std::vector<Bibble> v {...};  
auto it = v.begin();  
int m = (*it).zarf;  
int m = it->zarf;
```

// Exactly the same as prev!



Input Iterators

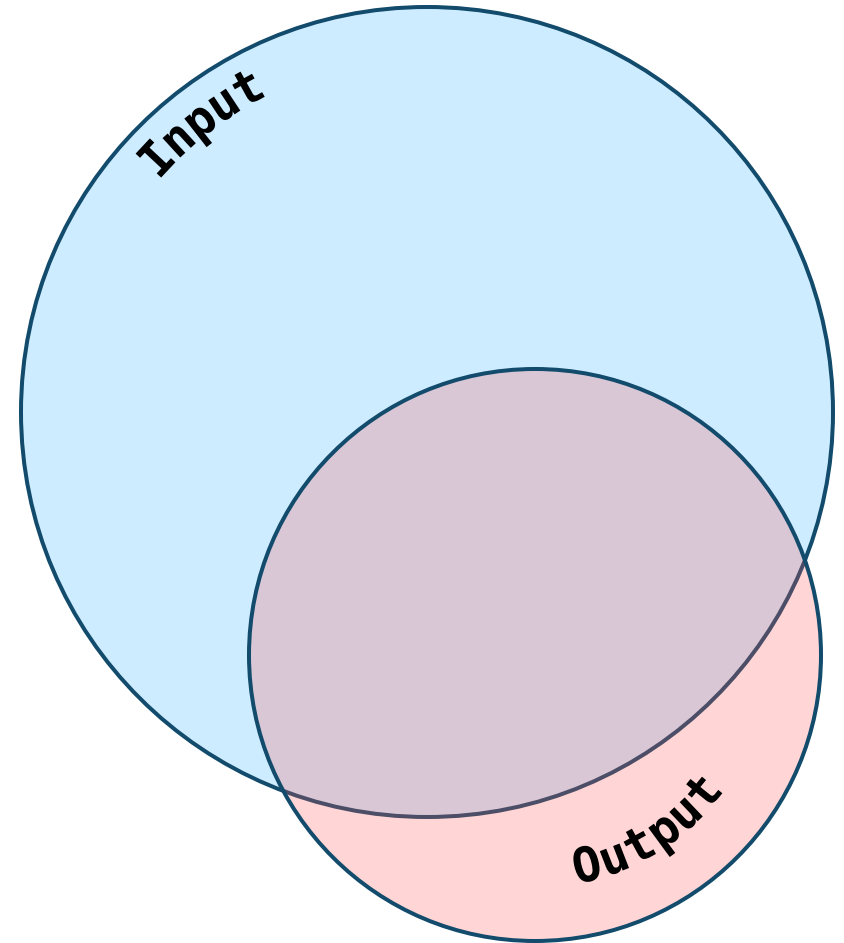
- Most basic kind of iterator
- Allows us to read elements

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

Output Iterator

Allows us to write elements

```
*it = elem;
```



**Vivid Venn Diagram of
Vexing Iterators**

Forward Iterator

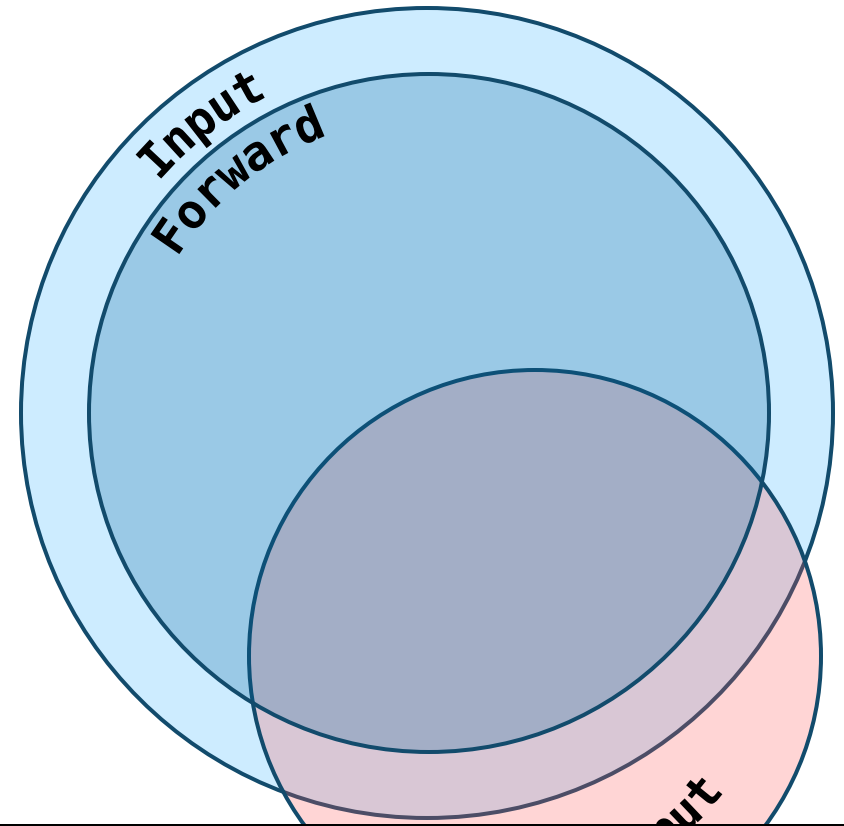
- An input iterator that allows us to make multiple passes
- All STL container iterators fall here

Multi-pass guarantee

`it1 == it2`



`++it1 == ++it2`



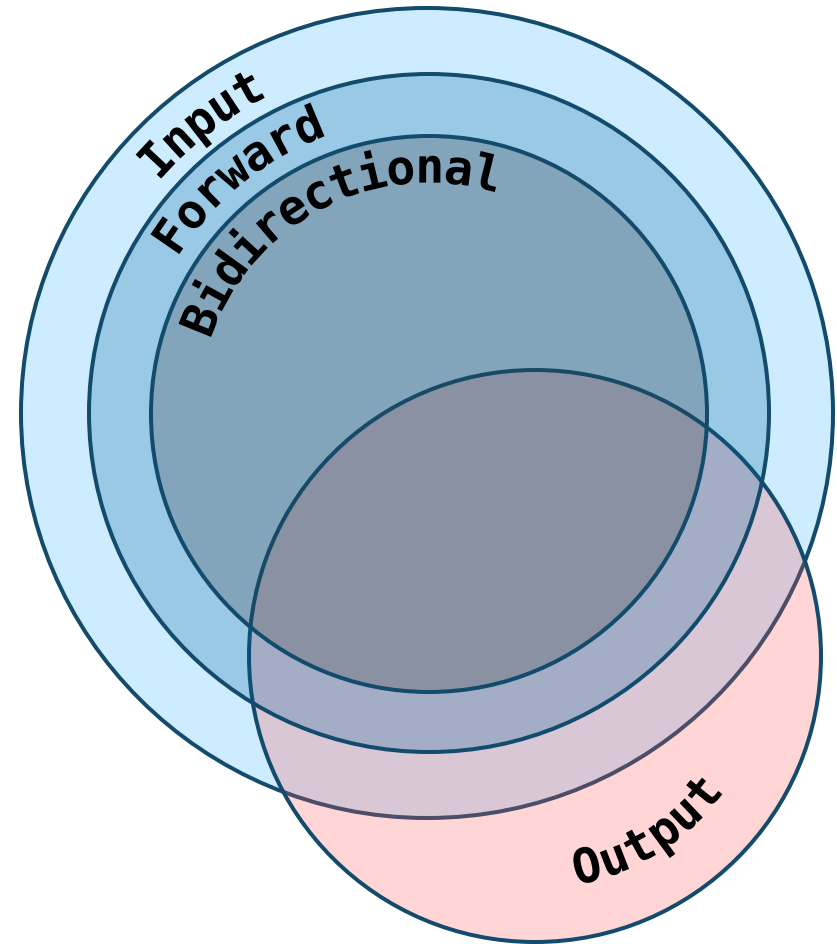
What kind of data structure might not want a multi-pass iterator?

Streams!!!

Bidirectional Iterators

- Allows us to move forwards *and* backwards
- `std::map`, `std::set`

```
auto it = m.end();  
  
// Get last element  
--it;  
auto& elem = *it;
```



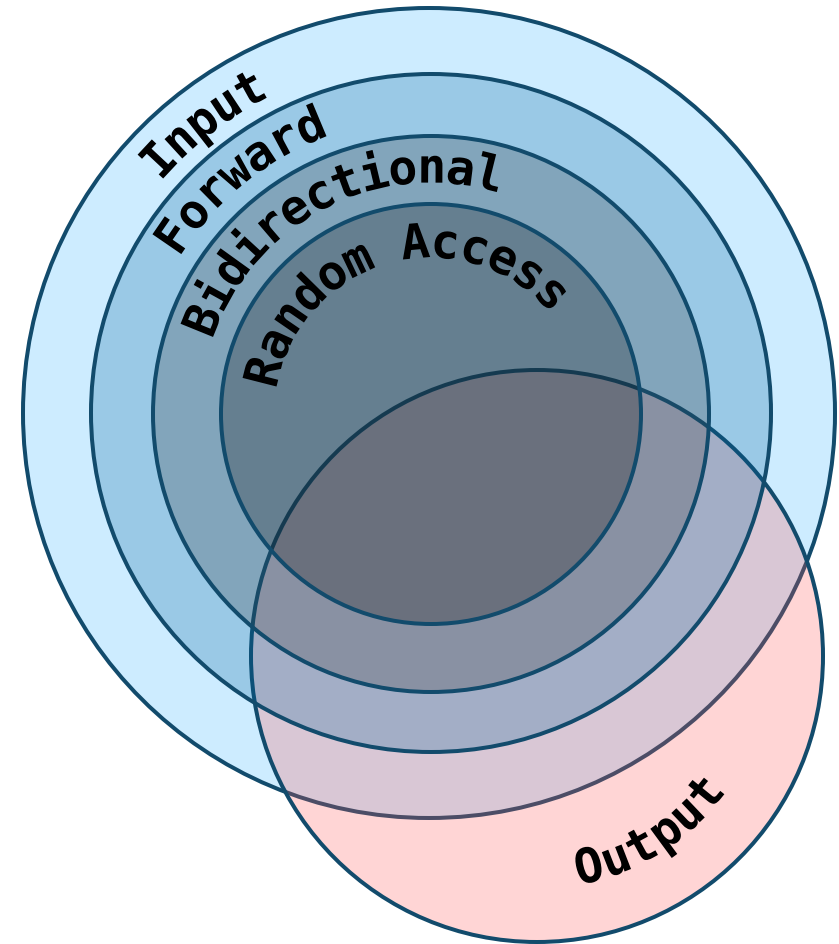
Vivid Venn Diagram of
Vexing Iterators

Random Access Iterators

- Allows us to quickly skip forward and backward
- `std::vector`, `std::deque`

```
auto it2 = it + 5; // 5 ahead
auto it3 = it2 - 2; // 2 back

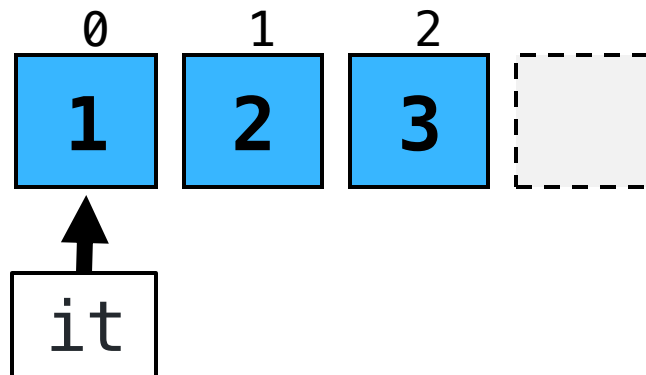
// Get 3rd element
auto& second = *(it + 2);
auto& second = it[2];
```



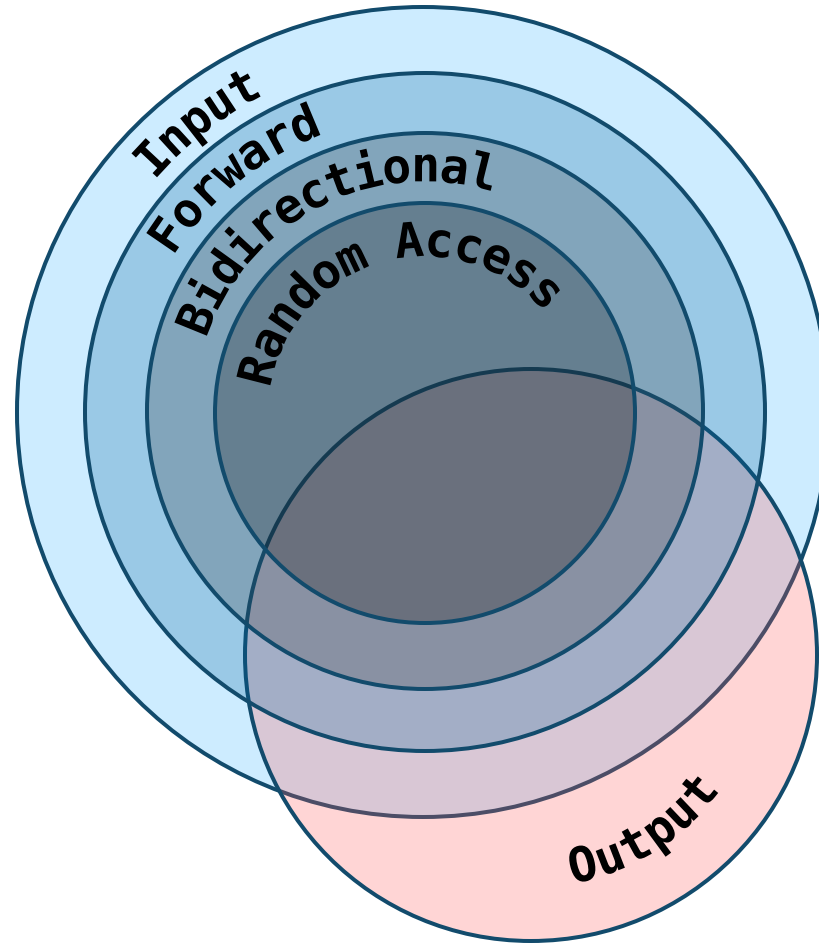
Vivid Venn Diagram of
Vexing Iterators

Be careful not to go out of bounds

```
std::vector<int> v { 1, 2, 3 };  
auto it = v.begin();  
it += 3;  
int& elem = *it; // Undefined behaviour
```



STL Iterator Types



Why does it matter?

Why does it matter?

As we'll soon see, some algorithms require a certain iterator type!

```
std::vector<int> vec{1,5,3,4};  
std::sort(vec.begin(), vec.end());  
// ✓ begin/end are random access  
  
std::unordered_set<int> set {1,5,3,4};  
std::sort(set.begin(), set.end());  
// ✗ begin/end are bidirectional
```

Why have multiple iterator types?

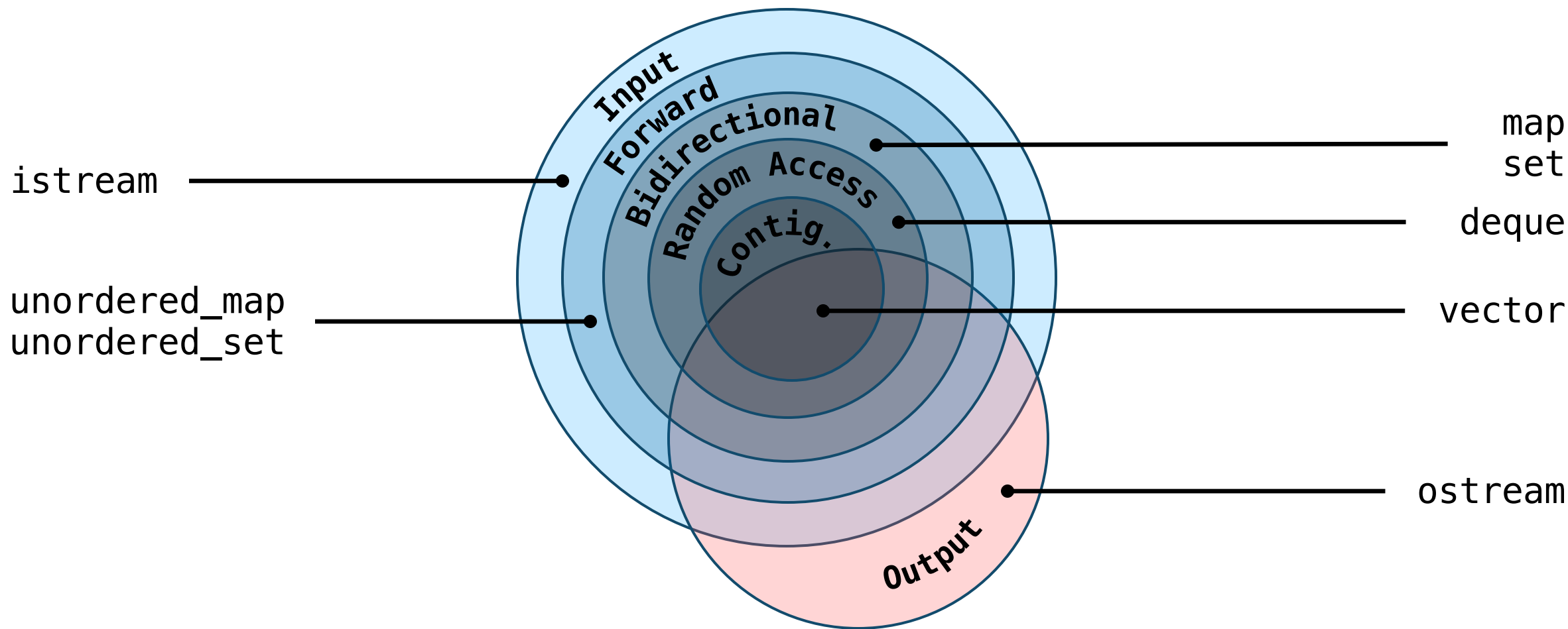
- **Goal:** provide a uniform abstraction over all containers
- **Caveat:** the way that a container is implemented affects how you iterate through it
 - Skipping ahead 5 steps (random access) is a lot easier/faster when you have a sequence container (**vector**, **deque**) than associative (**map**, **set**)
 - C++ generally avoids providing you with slow methods by design, so that's why you can't do random access on a **map::iterator**

What questions do you have?



bjarne_about_to_raise_hand

STL Iterator Types



Pointers and Memory

An iterator points to a container element

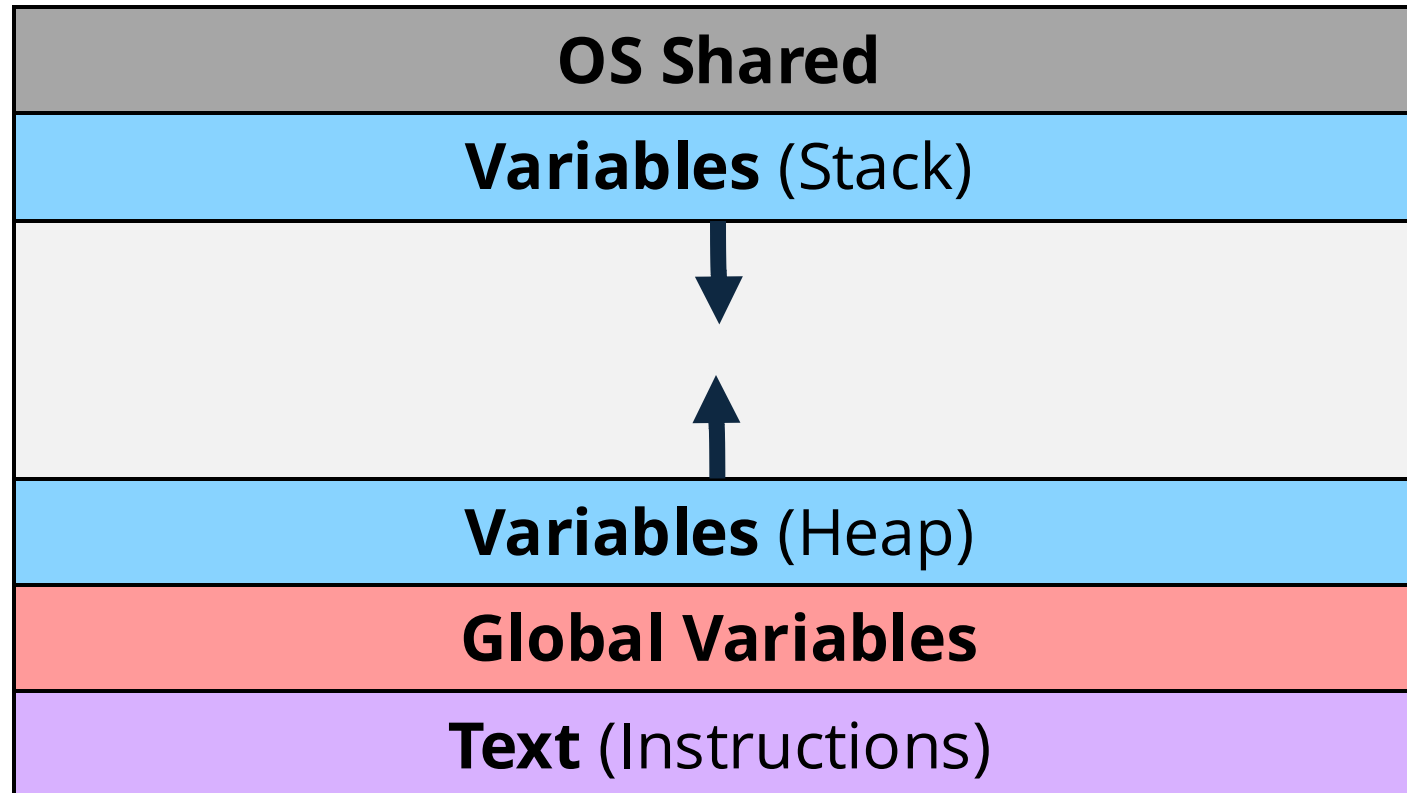
A pointer points to any object

Memory Basics



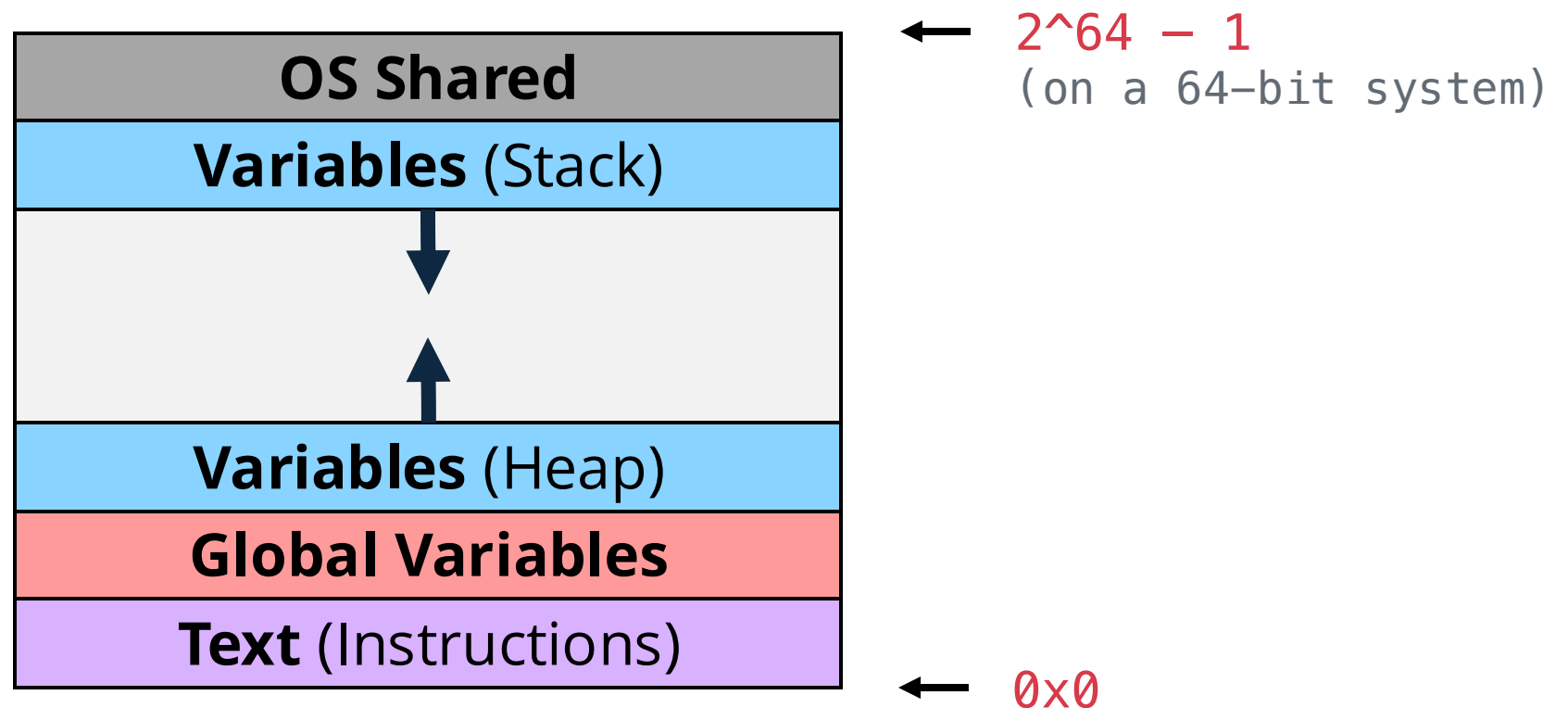
Memory Basics

- Every variable lives somewhere in memory
- All the places something could live form the **address space**



Memory Basics

- Memory is usually byte-addressable, with each byte numbered from 0
- 1 byte = 8 bits



Memory Basics

- The **address** of an object is the location of its lowest byte
- For example, an integer always uses 32 bits = 4 bytes

```
int x = 106; // 32 bits
```

0x10 is the
address of x

x's memory

00000000	00000000	00000000	01101010
----------	----------	----------	----------

0x10

0x11

0x12

0x13

What questions do you have?



bjarne_about_to_raise_hand

How do we get the address of a variable in C++?

Pointers!   

A pointer is the **address** of a variable

```
int x = 106;  
int* px = &x;
```

`int*` means `px`
is a pointer
to an `int`

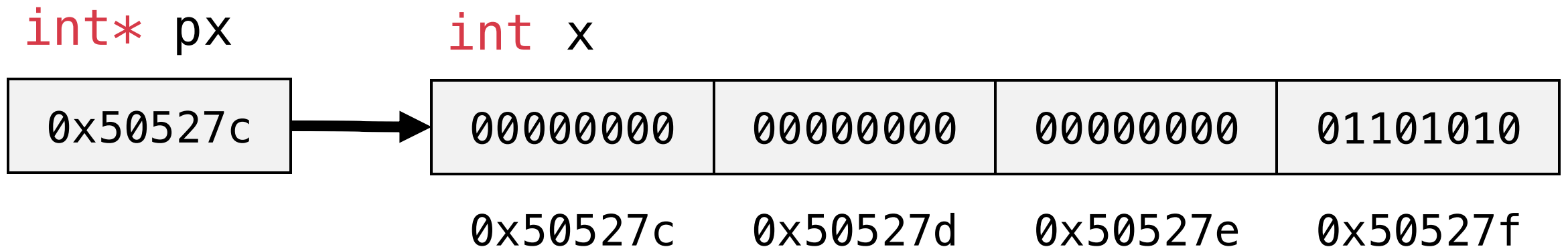
`&` is the
address of
operator

```
std::cout << x << std::endl;    // 106  
std::cout << *px << std::endl;  // 106  
std::cout << px << std::endl;   // 0x50527c
```

MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME
A FEW POINTERS?



A pointer is just a number!



What questions do you have?



bjarne_about_to_raise_hand

`int* px`

`int x`



We can have pointers to all kinds of things!

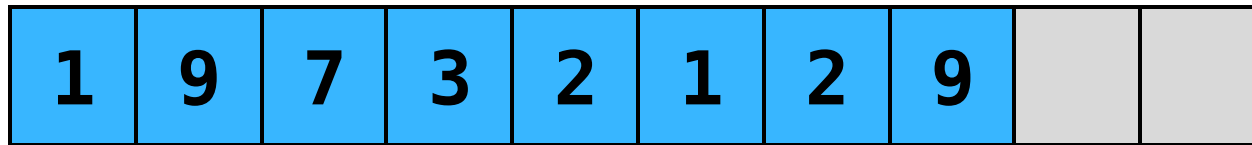
```
int x = 106;  
int* px = &x;
```

```
StanfordID id { "jtrb" };  
StanfordID* p = &id;  
auto name = p->name;
```

```
std::vector<int> v;  
std::vector<int>* p = &v;
```

```
std::vector<int> v {  
    1, 2, 3, 4, 5  
};  
int* arr = &v[0];
```

Recall: a vector is a contiguous array

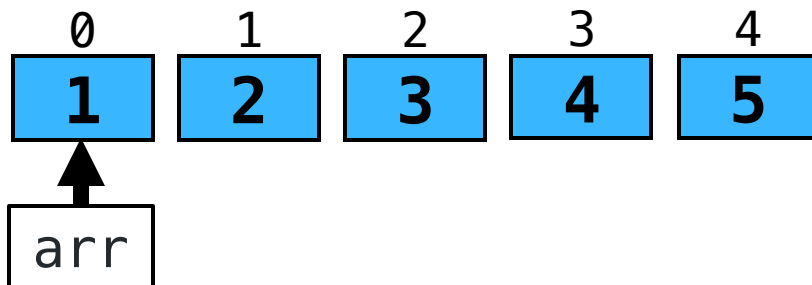


A **vector** is a single chunk of memory

Array pointer

```
std::vector<int> v {1,2,3,4,5};
```

```
int* arr = &v[0];    std::cout << *arr << " ";  
arr += 1;            std::cout << *arr << " ";  
++arr;               std::cout << *arr << " ";  
arr += 2;            std::cout << *arr << " ";  
if (arr == &v[4])   std::cout << "At last index";
```



Output:

1 2 3 5 At last index

Notice anything?

```
std::vector<int> v {1,2,3,4,5};  
  
int* arr = &v[0];           // Copy construction  
arr += 1;                   // Random access  
++arr;                      // Move pointer forward  
arr += 2;                   // Random access  
if (arr == &v[4])          // Pointer comparison
```

We could do the same thing with iterators!

```
auto it = v.begin();    std::cout << *it << " ";
it += 1;               std::cout << *it << " ";
++it;                 std::cout << *it << " ";
it += 2;              std::cout << *it << " ";
if (it == --v.end())  std::cout << "At last element";
```

Iterators have a similar interface to **pointers**

Recall: **iterator** is a type alias

```
template <typename T>
class vector {
    using iterator = /* some iterator type */;;

    // Implementation details...
};
```

T* is the backing type for **vector<T>::iterator**

```
template <typename T>
class vector {
    using iterator = T*;

    // Implementation details...
};
```

In the real STL implementation, the actual type is not **T***.
But for all intents and purposes, you can think of it this way.

What questions do you have?



bjarne_about_to_raise_hand

Recap

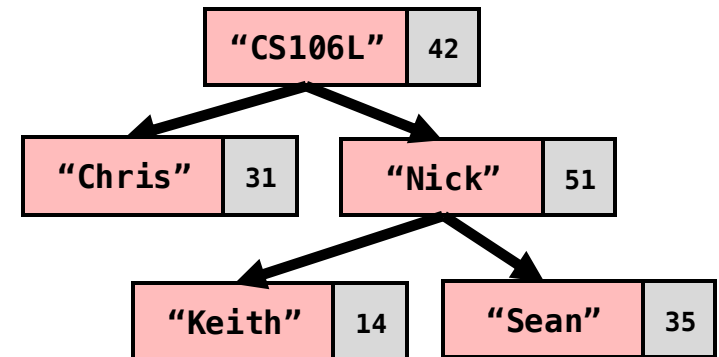
What we covered

- Iterator Basics
 - An iterator allows us to step forward through a container
- Iterator Types
 - Input, Output, Forward, Bidirectional, Random Access
- Pointers and Memory
 - A pointer points to an arbitrary C++ object in memory
 - Pointers and iterators have the same interface

So how do we implement other iterators?

```
template <typename K, typename V>
class map {
    using iterator = ????????;

    // Implementation details...
};
```



Classes

We'll learn about them next time