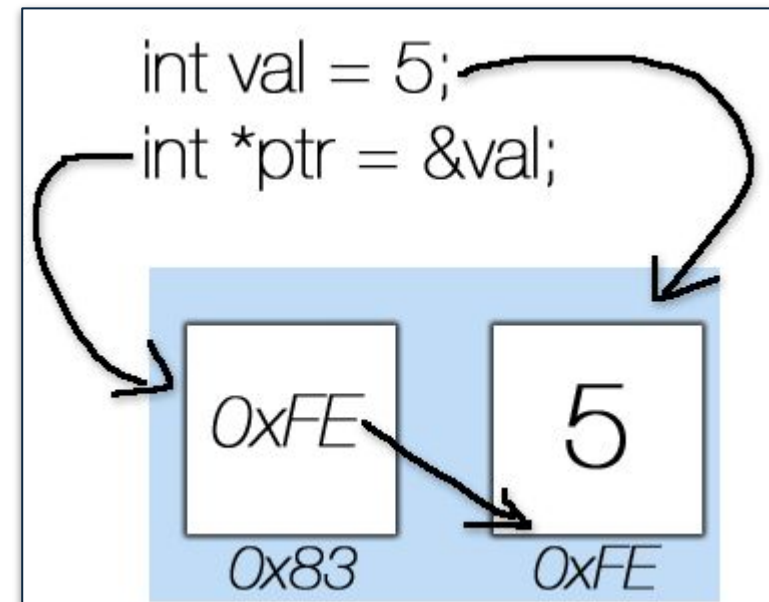
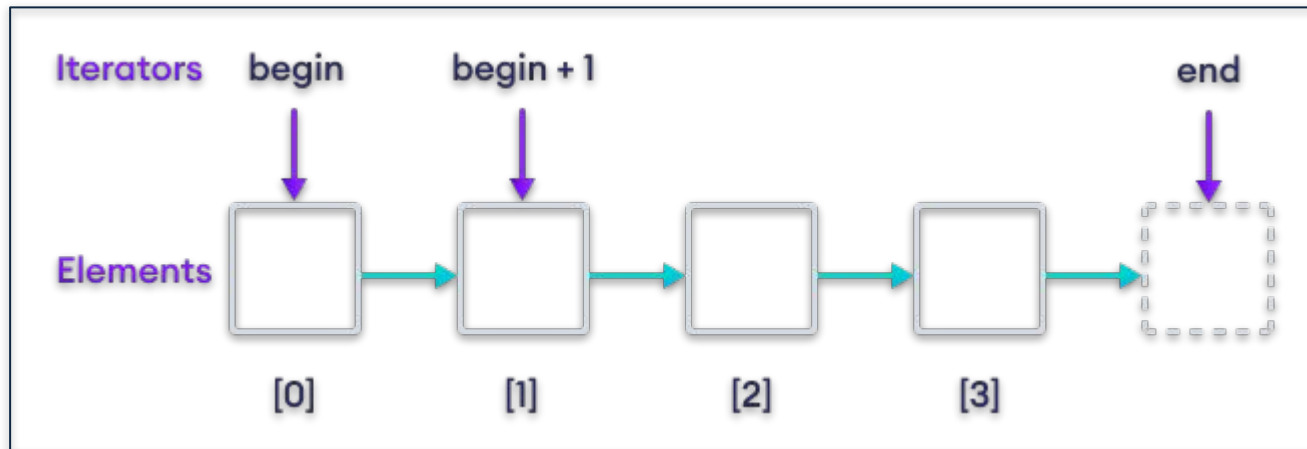


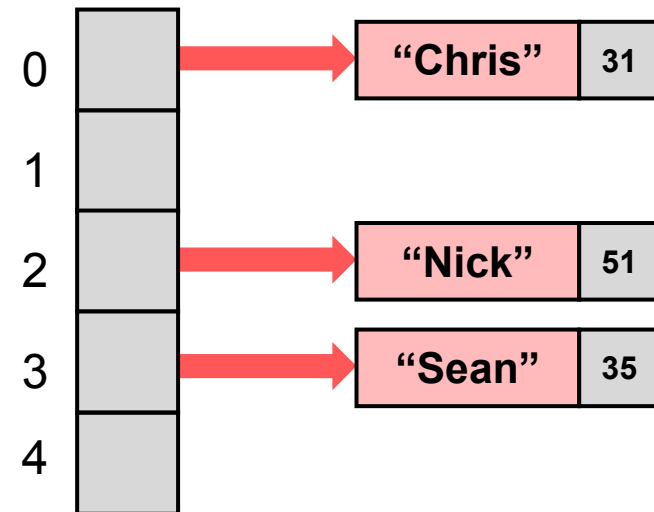
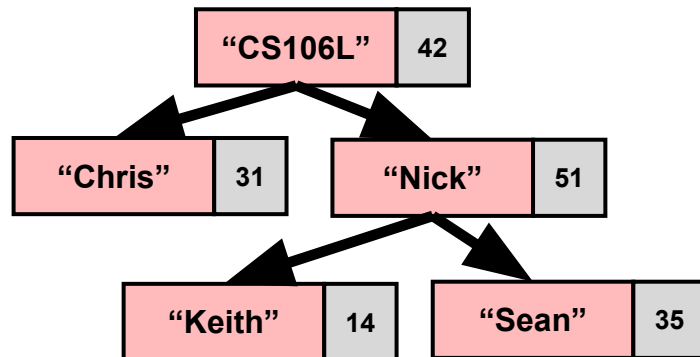
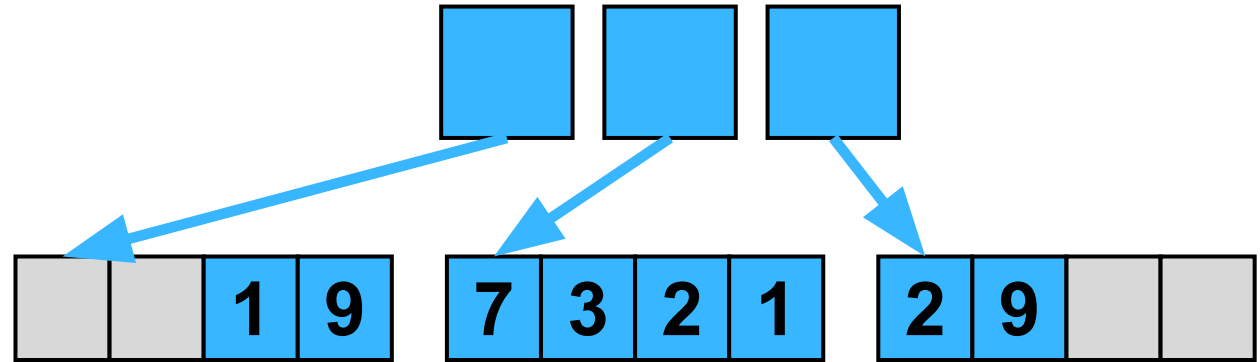
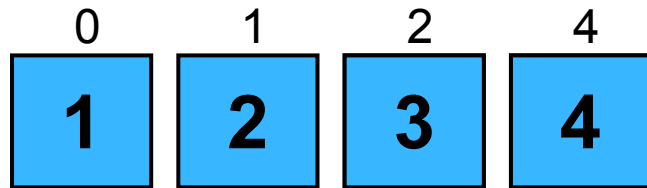
Lecture 6: Iterators and Pointers

Stanford CS106L, Fall 2026

Preston Seay & Emily Ai



Last Time: Containers



Pop Quiz: Containers

- Which container lets you insert at the back and front equally efficiently?
- Which container(s) organize their elements with trees?
- Which is *usually* faster: `unordered_set` or `set`? Why?

Pop Quiz: Containers

- Which container lets you insert at the back and front equally efficiently?
 -  `std::deque`
- Which container(s) organize their elements with trees?
- Which is *usually* faster: `unordered_set` or `set`? Why?

Pop Quiz: Containers

- Which container lets you insert at the back and front equally efficiently?
 -  `std::deque`
- Which container(s) organize their elements with trees?
 -  `std::map`, `std::set`
- Which is *usually* faster: `unordered_set` or `set`? Why?

Pop Quiz: Containers

- Which container lets you insert at the back and front equally efficiently?
 -  **std::deque**
- Which container(s) organize their elements with trees?
 -  **std::map, std::set**
- Which is *usually* faster: **unordered_set** or **set**? Why?
 -  **std::unordered_set** (Hashing + small load factor)!

Lecture 6: Iterators

CS106L, Fall 2026

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?

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];  
    std::cout << elem;  
}  
  
for (var-init; condition; increment) {  
    const auto& elem = /* grab element */;  
    /* do something with elem */  
}
```

Question: How do we iterate?

```
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 = /* idkkk */;  
}
```

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



**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

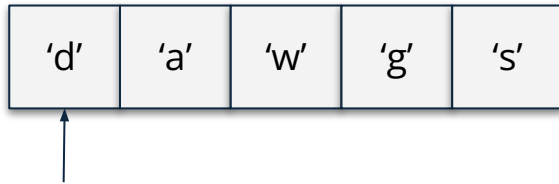


Containers and iterators *work together* to allow iteration

Container Interface

container.**begin()**

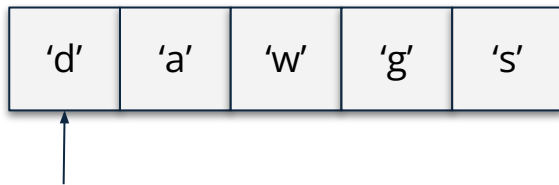
Gets an iterator to
the **first element**
of the container
(assuming non-empty)



Container Interface

container.**begin**()

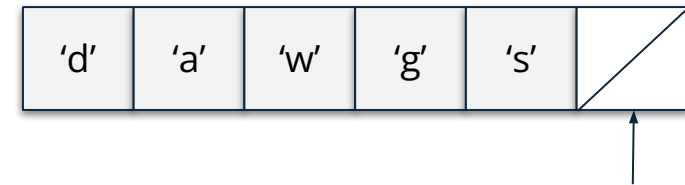
Gets an iterator to
the **first element**
of the container
(assuming non-empty)



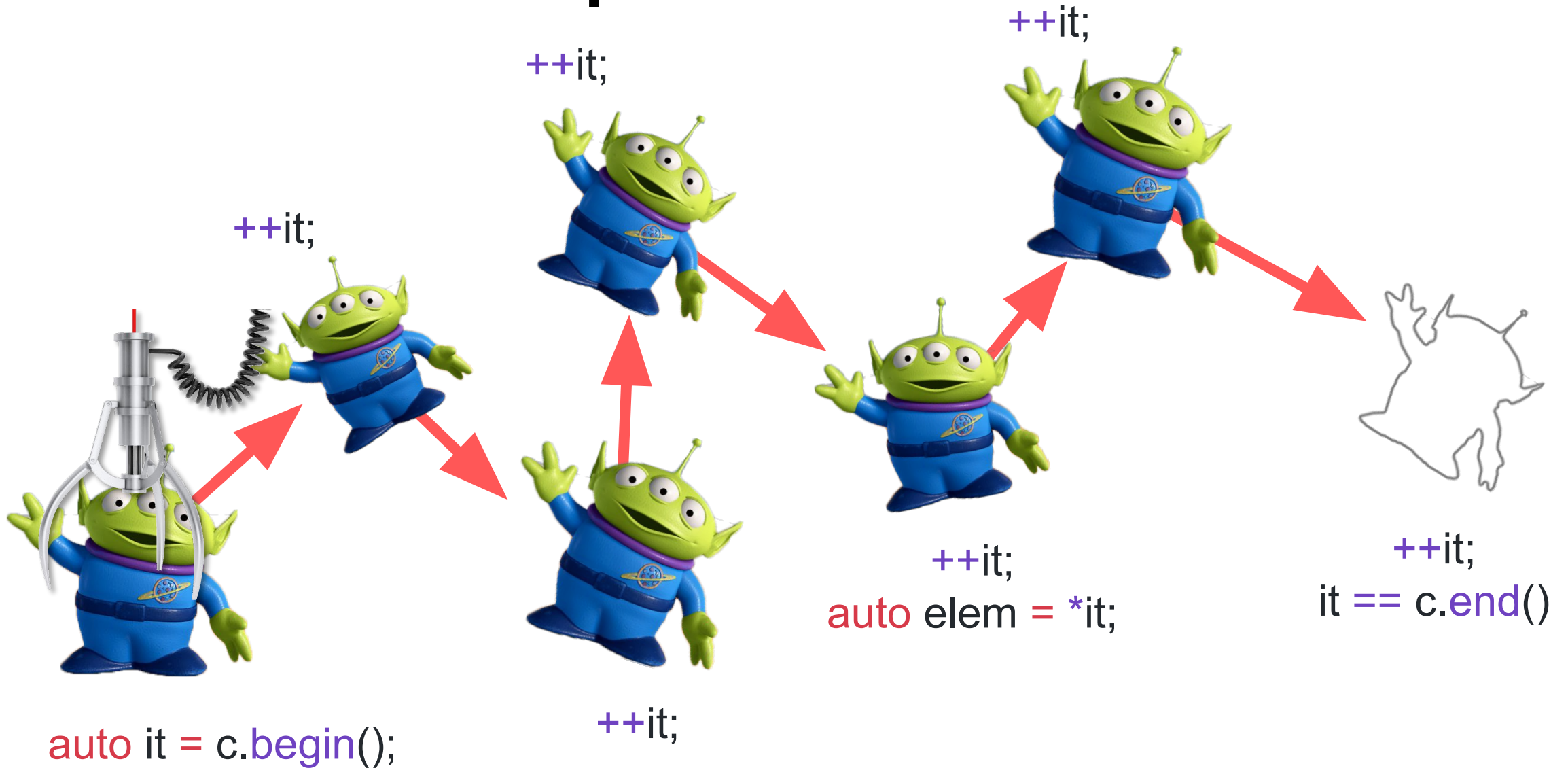
container.**end**()

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

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



C++ Iterators Example



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

++it avoids making an unnecessary copy

```
// Prefix form - ++it
```

```
// Increments it and returns a reference to same object
```

```
Iterator& operator++();
```

```
// Postfix form - it++
```

```
// Increments it and returns a copy of the old value
```

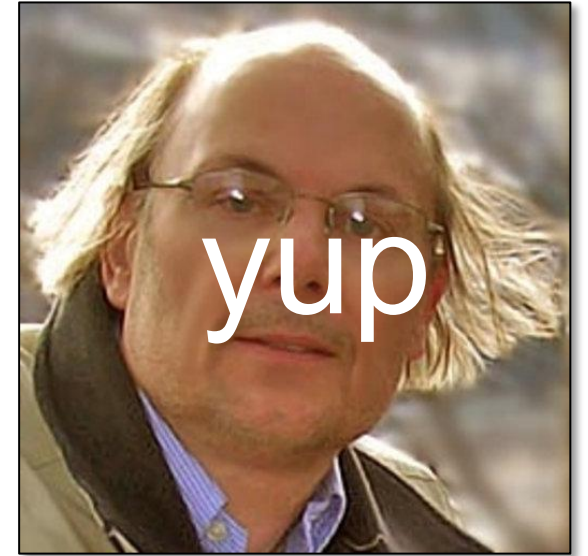
```
Iterator operator++(int);
```

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

Does it actually make a difference?

Bjarne's Thoughts

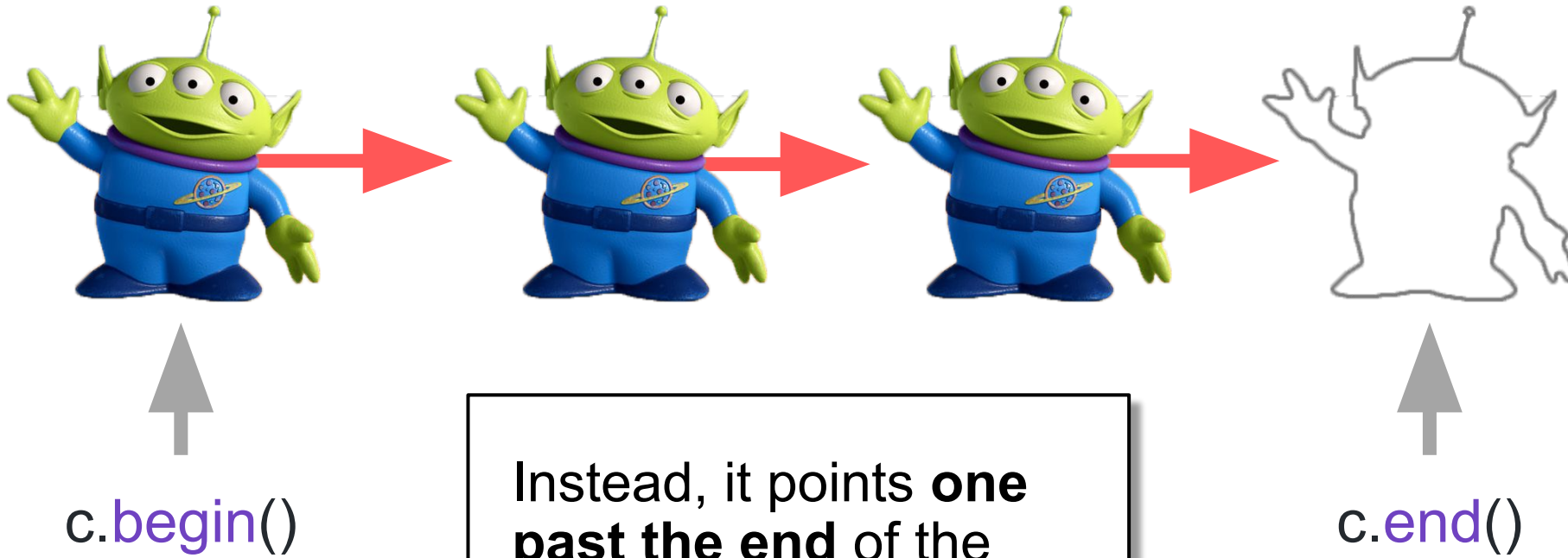
“



`++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.

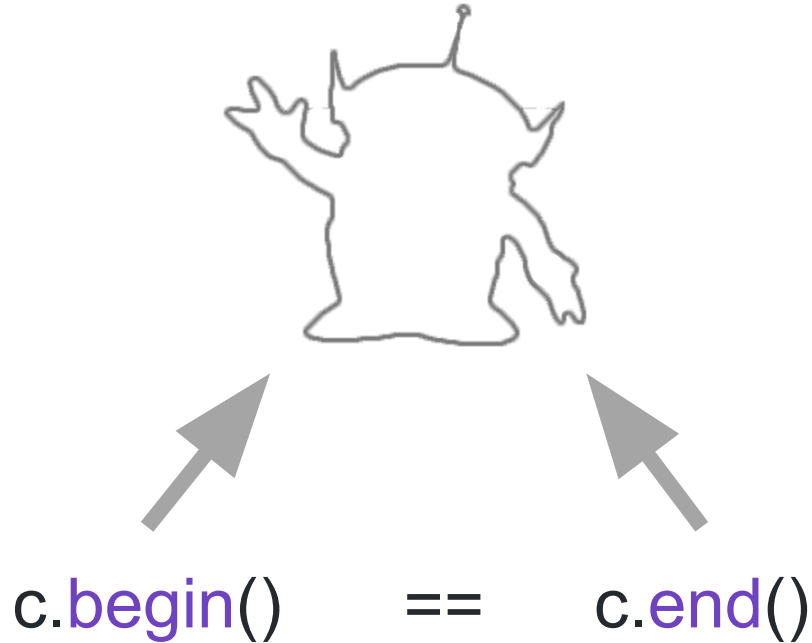
[source]

`end()` never points to an element!



`end()` never points to an element!

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



**Let's practice w/
slide #1**

Iterator Interface

```
// Direct initialization
```

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

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



We have an answer now!

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

```
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 (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 (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;
}
```

Iterator Interface

```
// Direct initialization
```

```
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()) ...
```

**Let's practice w/
slide #2**

What questions do you have?



bjarne_about_to_raise_hand

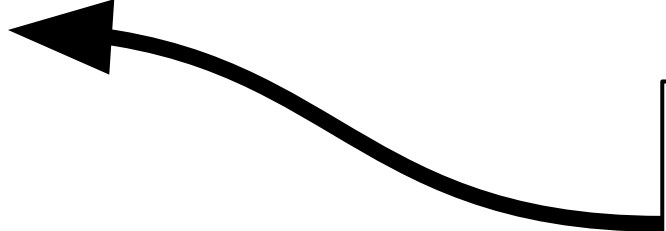
Guess we're done here!



But wait...

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

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



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;
```

What questions do you have?

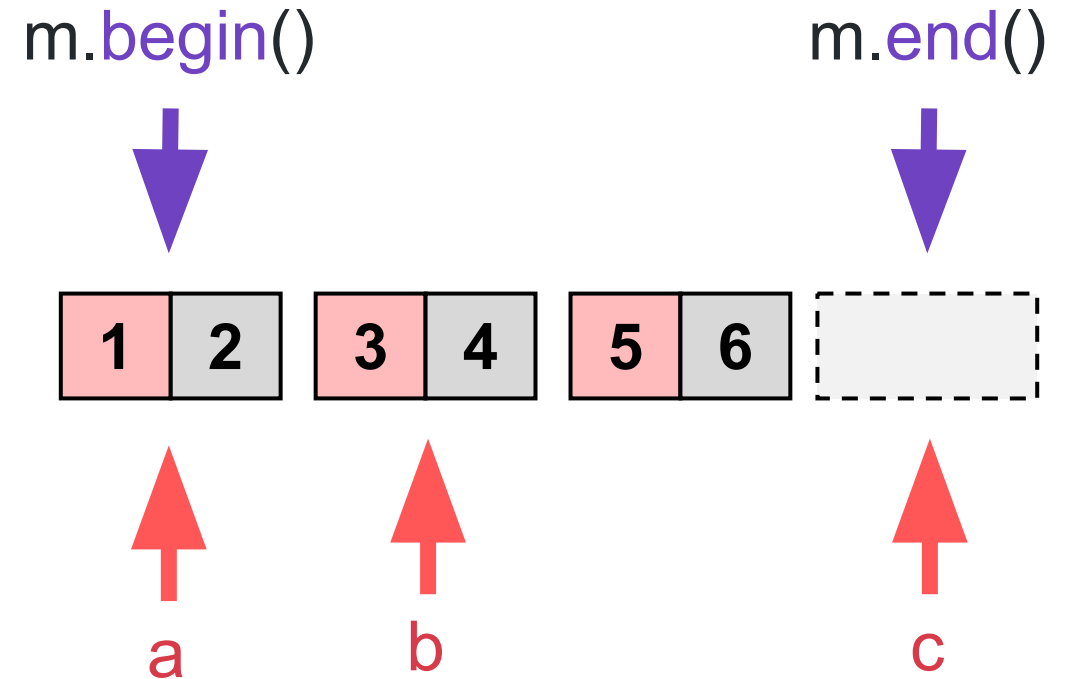


bjarne_about_to_raise_hand

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;
```



**Let's practice w/
slide #3**

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

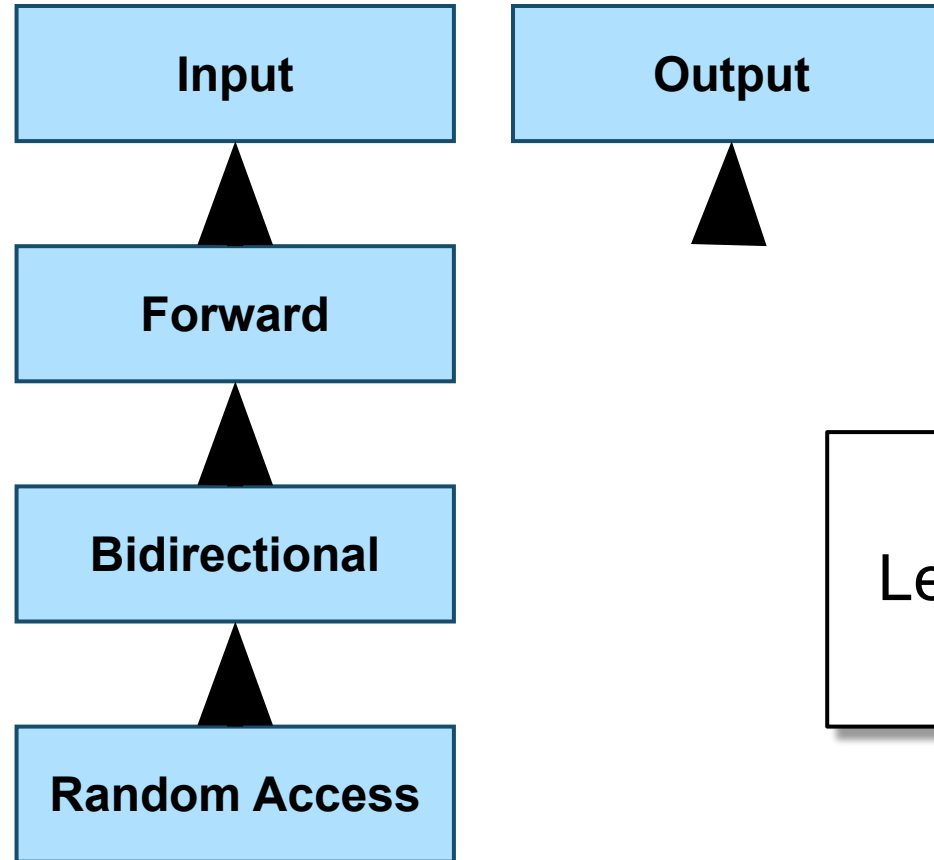
```
--it; // Move backwards
```

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

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

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

Iterator types determine their functionality

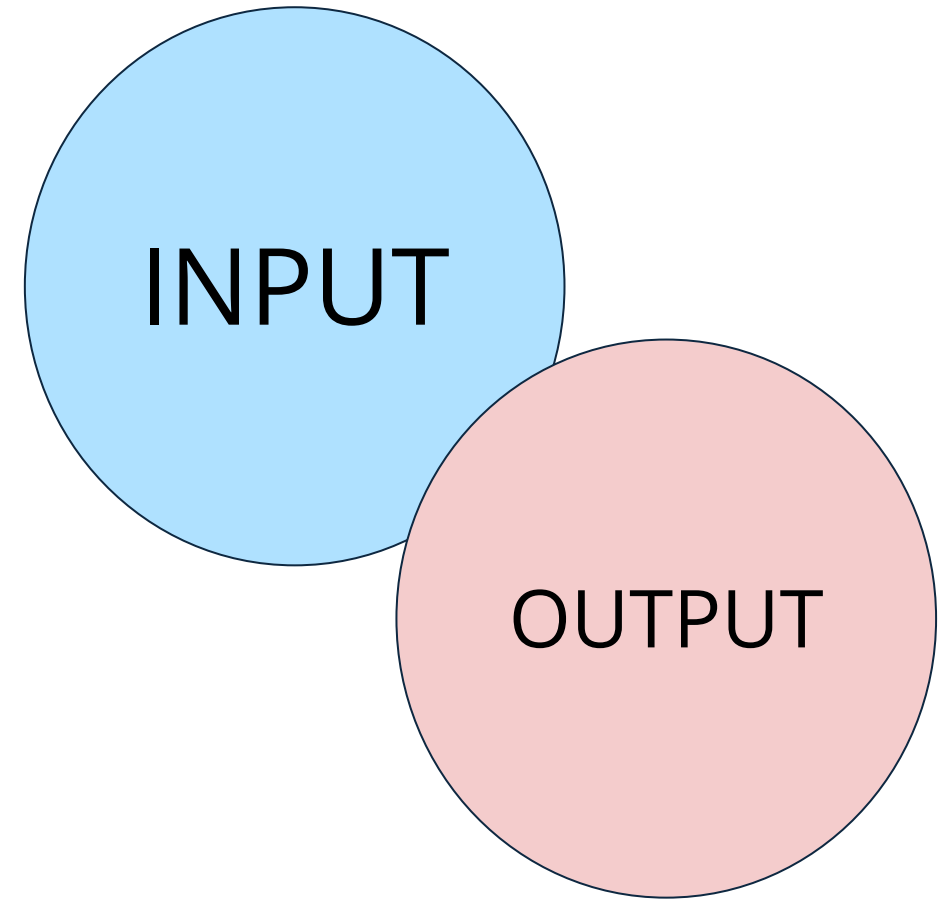


Let's unpack this!

Input Iterators

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

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



**Vivid Venn Diagram of
Vexing Iterators**

Input Iterators

```
#include <iterator>
#include <iostream>

int main() {
    std::istream_iterator<int> it(std::cin); // points at first int
    std::istream_iterator<int> end;          // default = end of stream

    while (it != end) {
        std::cout << *it << " "; // read current value
        ++it;                     // consume it, advance to next
    }

    return 0;
}
```

Aside: operator->

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

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

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



// Exactly the same as prev!

Input Iterators

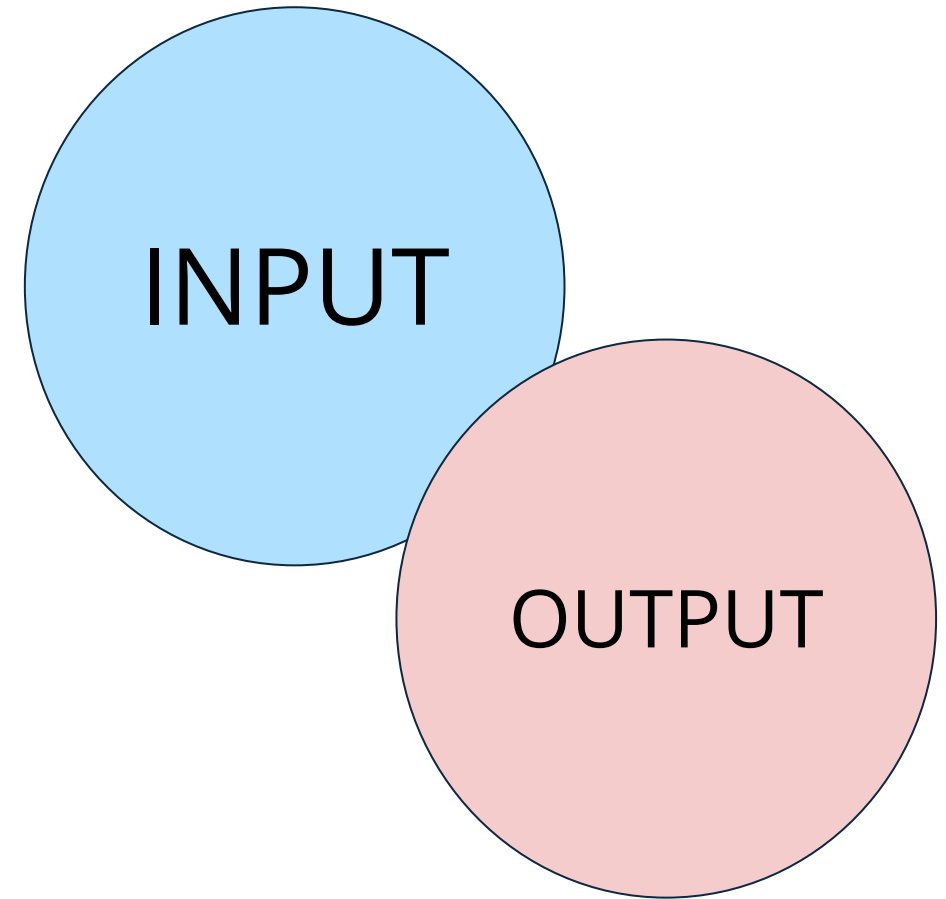
- Most basic kind of iterator
- Allows us to read elements

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

Output Iterator

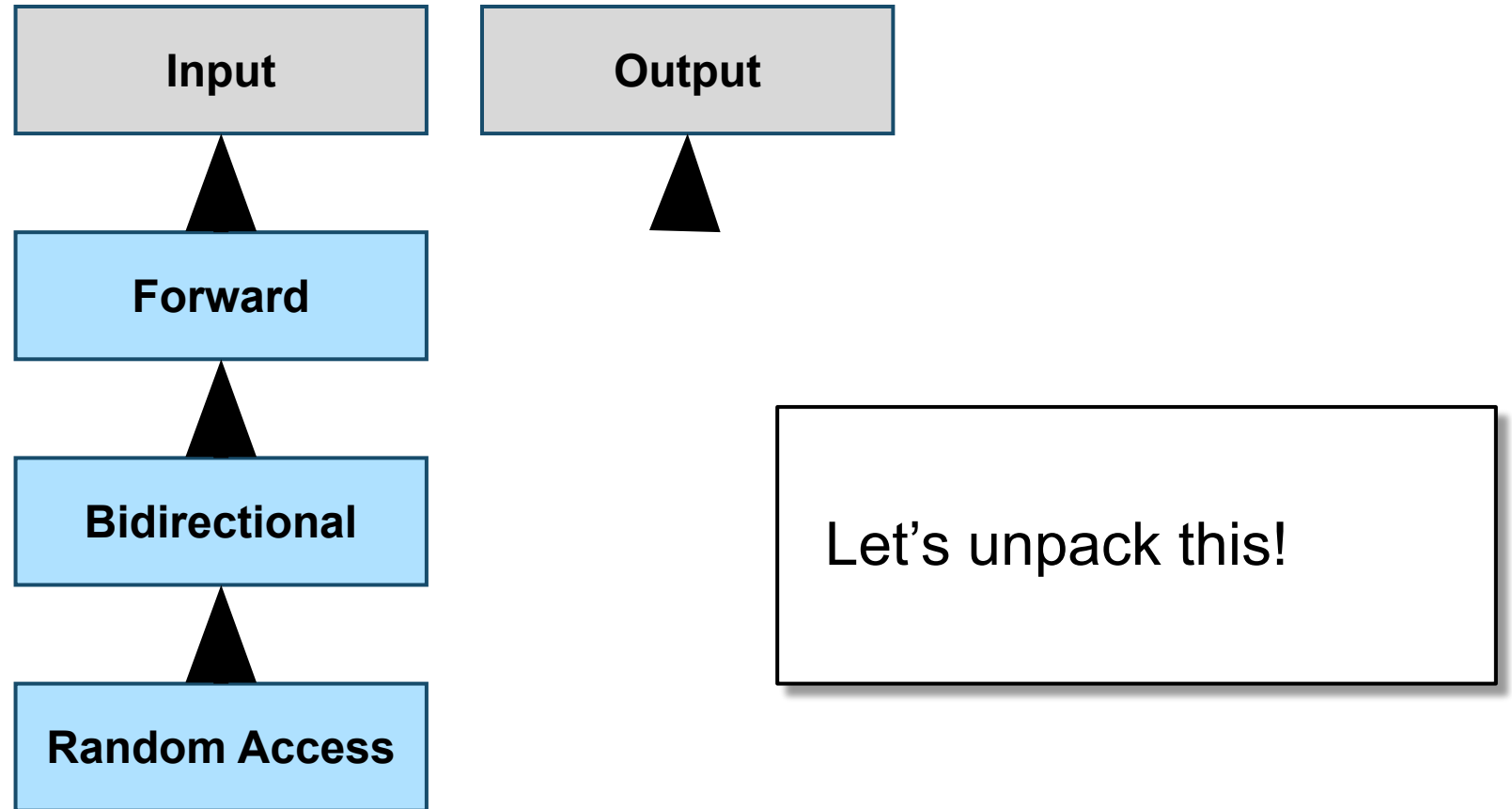
Allows us to write elements

```
*it = another_elem;
```



**Vivid Venn Diagram of
Vexing Iterators**

Iterator types determine their functionality



Forward Iterator

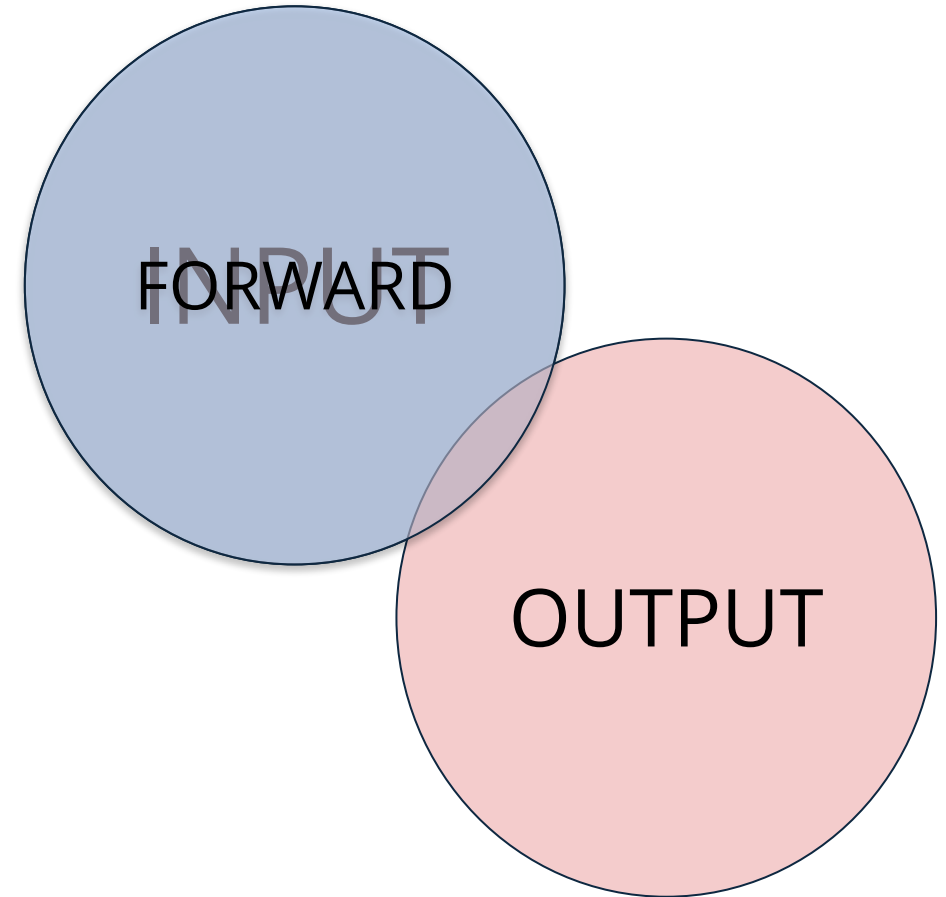
- An input iterator that allows us to make multiple passes
- All STL container iterators fall here

Multi-pass guarantee

```
auto it1 = v.begin()
```



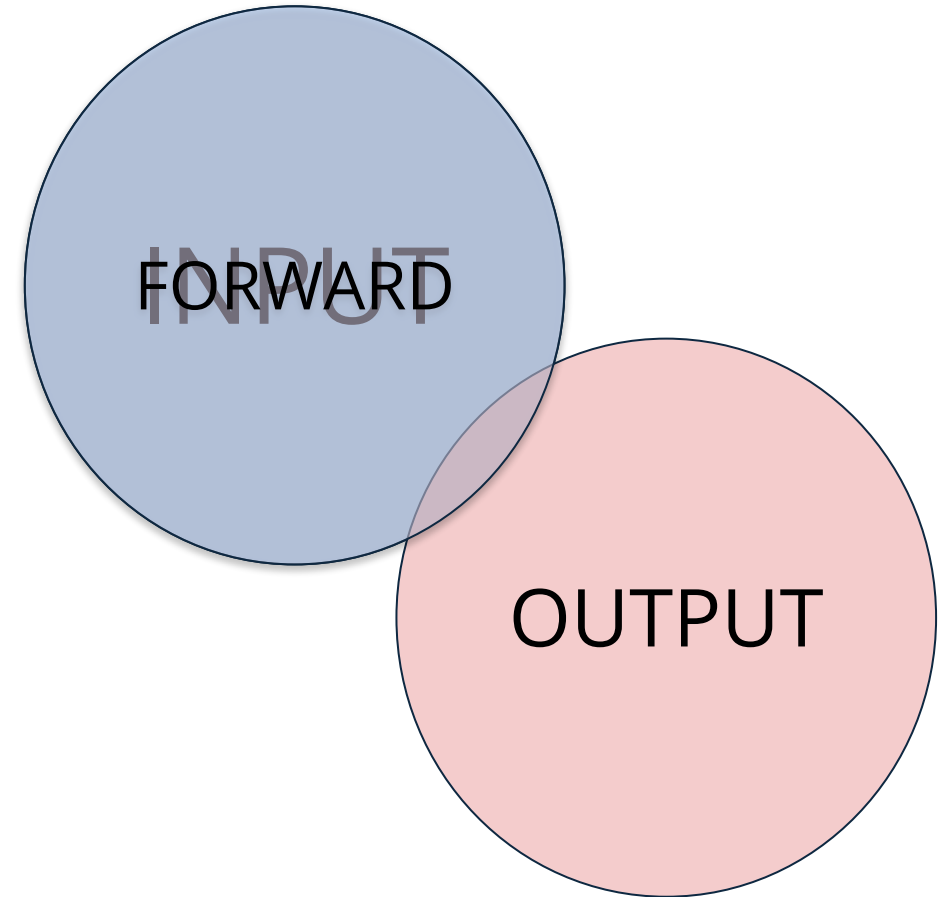
```
auto it2 = it1
```



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

```
auto it1 = v.begin()
```



```
auto it2 = it1
```

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

Streams!!!

Forward Iterator

```
#include <iterator>
#include <iostream>

int main() {
    // istream_iterator - input only
    auto it1 = std::istream_iterator<int>(std::cin);
    auto it2 = it1;    // "copy" but both are looking at the same stream

    ++it1;            // reads from stream, advances stream position
    ++it2;            // stream already moved! it2 is now in an undefined state

    return 0;
}
```

Bidirectional Iterators

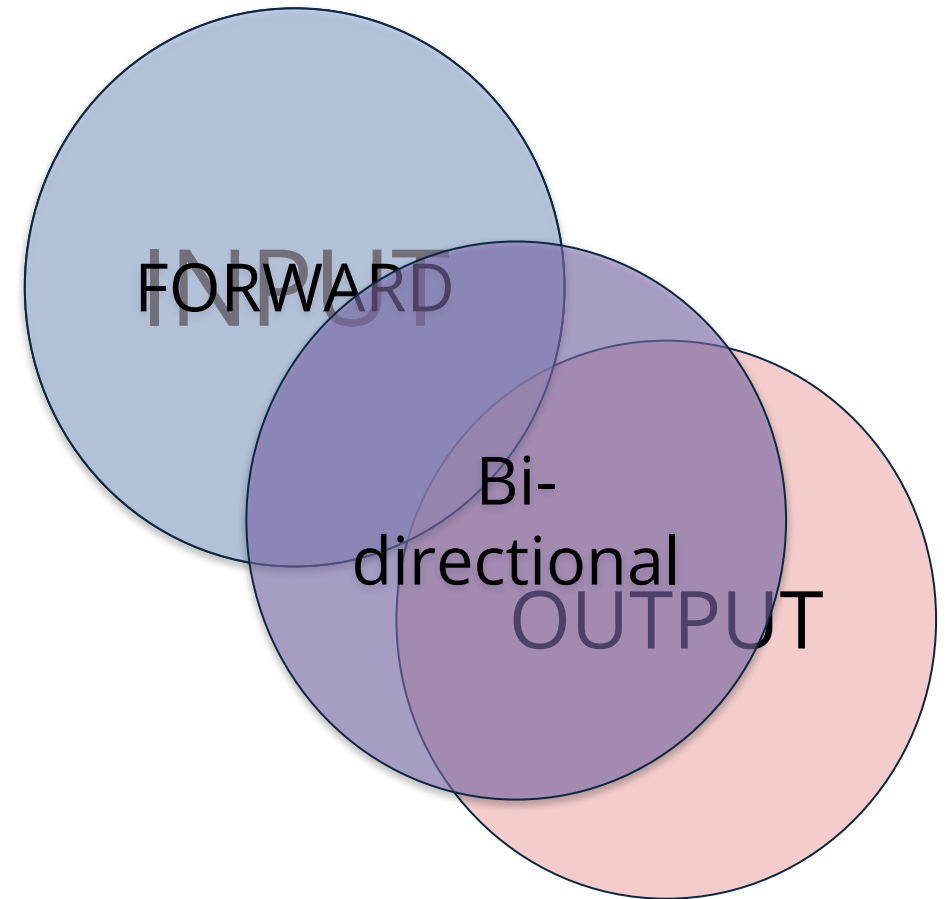
- Allows us to move forwards *and* backwards
- `std::map`, `std::set`

```
auto it = m.end();
```

```
// Get last element
```

```
--it;
```

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



**Vivid Venn Diagram of
Vexing Iterators**

Bidirectional Iterator

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> m = {
        {"apple", 1},
        {"banana", 2},
        {"cherry", 3}
    };

    auto it = m.end();    // end() points ONE PAST the last element
    --it;                 // step back to the actual last element
    auto& elem = *it;     // now dereference safely

    return 0;
}
```

Random Access Iterators

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

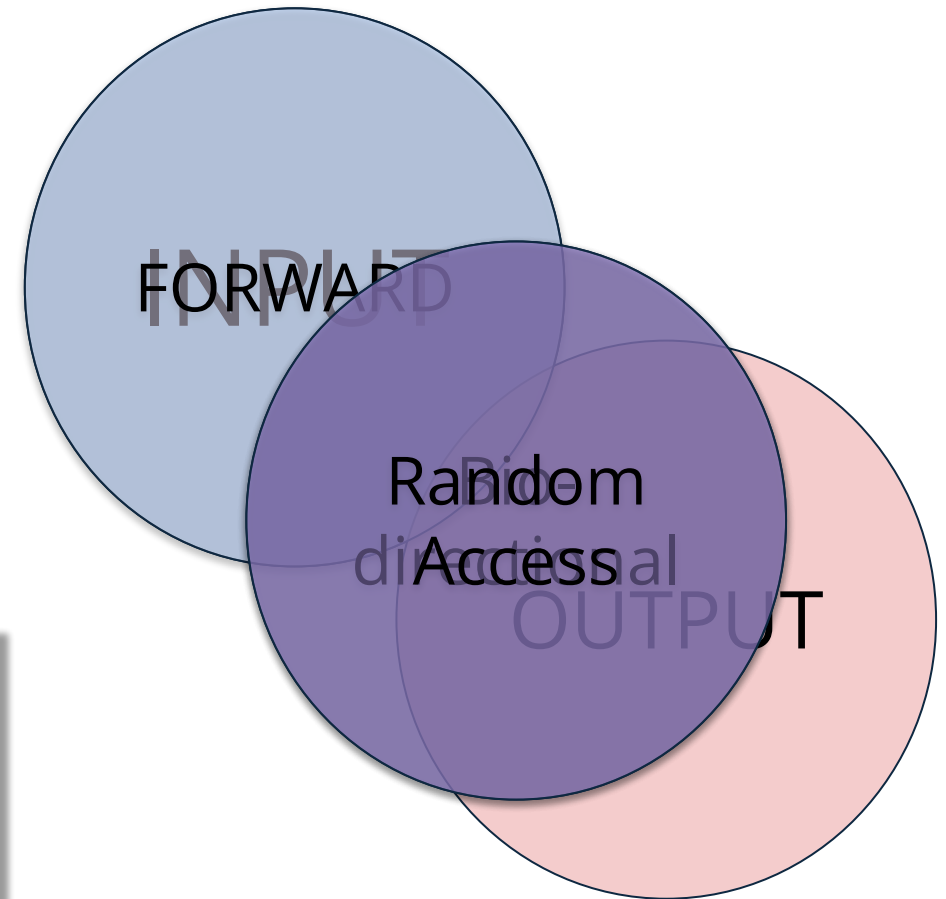
```
auto it2 = it + 5; // 5 ahead
```

```
auto it3 = it2 - 2; // 2 back
```

```
// Get 3rd element
```

```
auto& second = *(it + 2);
```

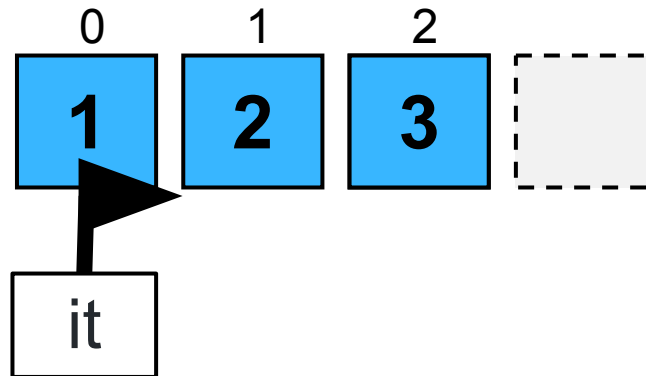
```
auto& second = it[2];
```



Vivid Venn Diagram of
Vexing Iterators

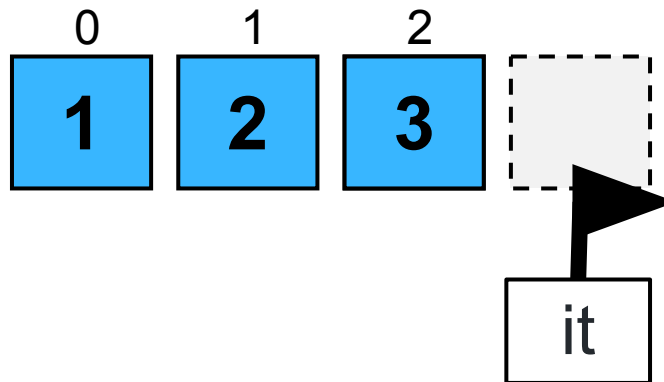
Be careful not to go out of bounds

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



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
```



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::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**

**Let's practice w/
slide #4**

What questions do you have?



bjarne_about_to_raise_hand

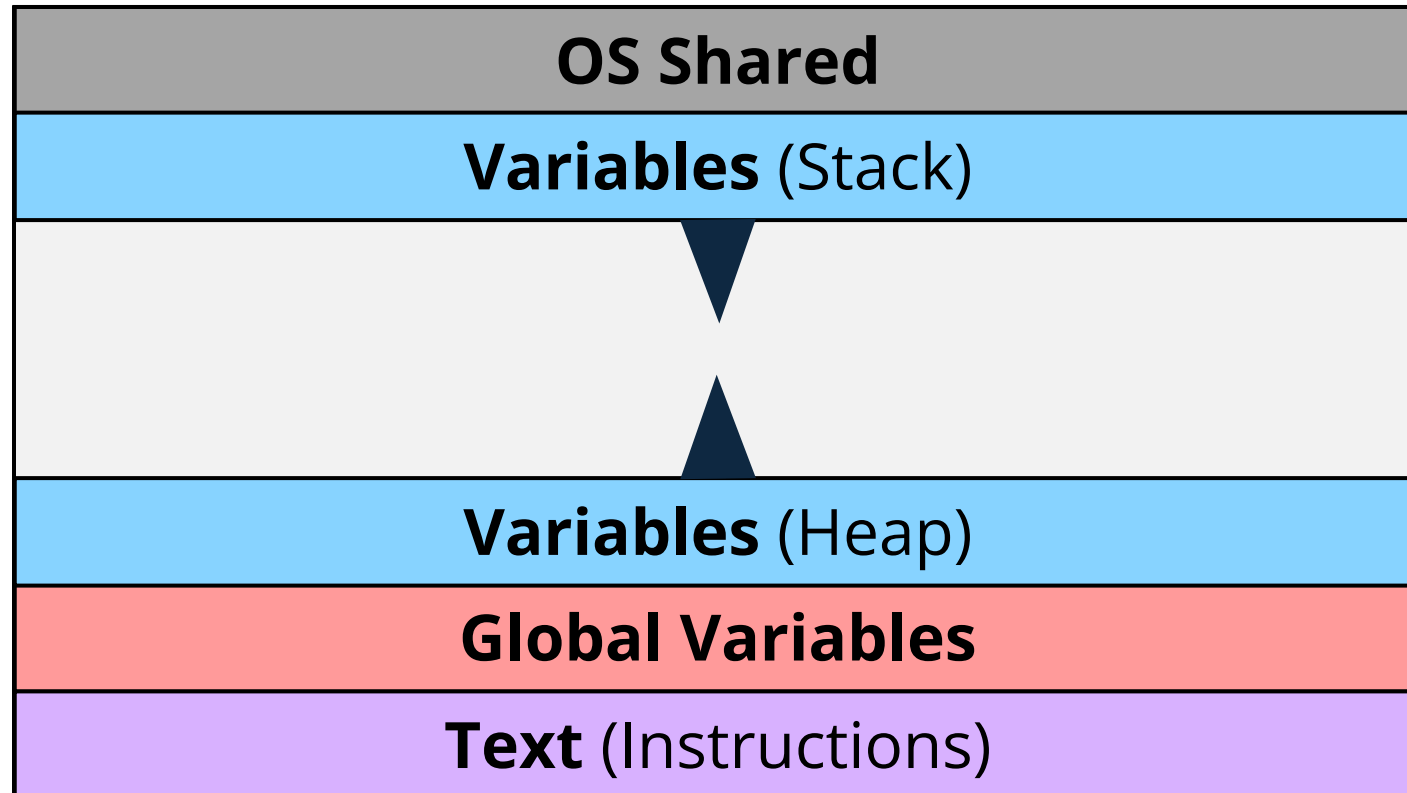
Pointers and Memory

An iterator points to a container element

A pointer points to any object

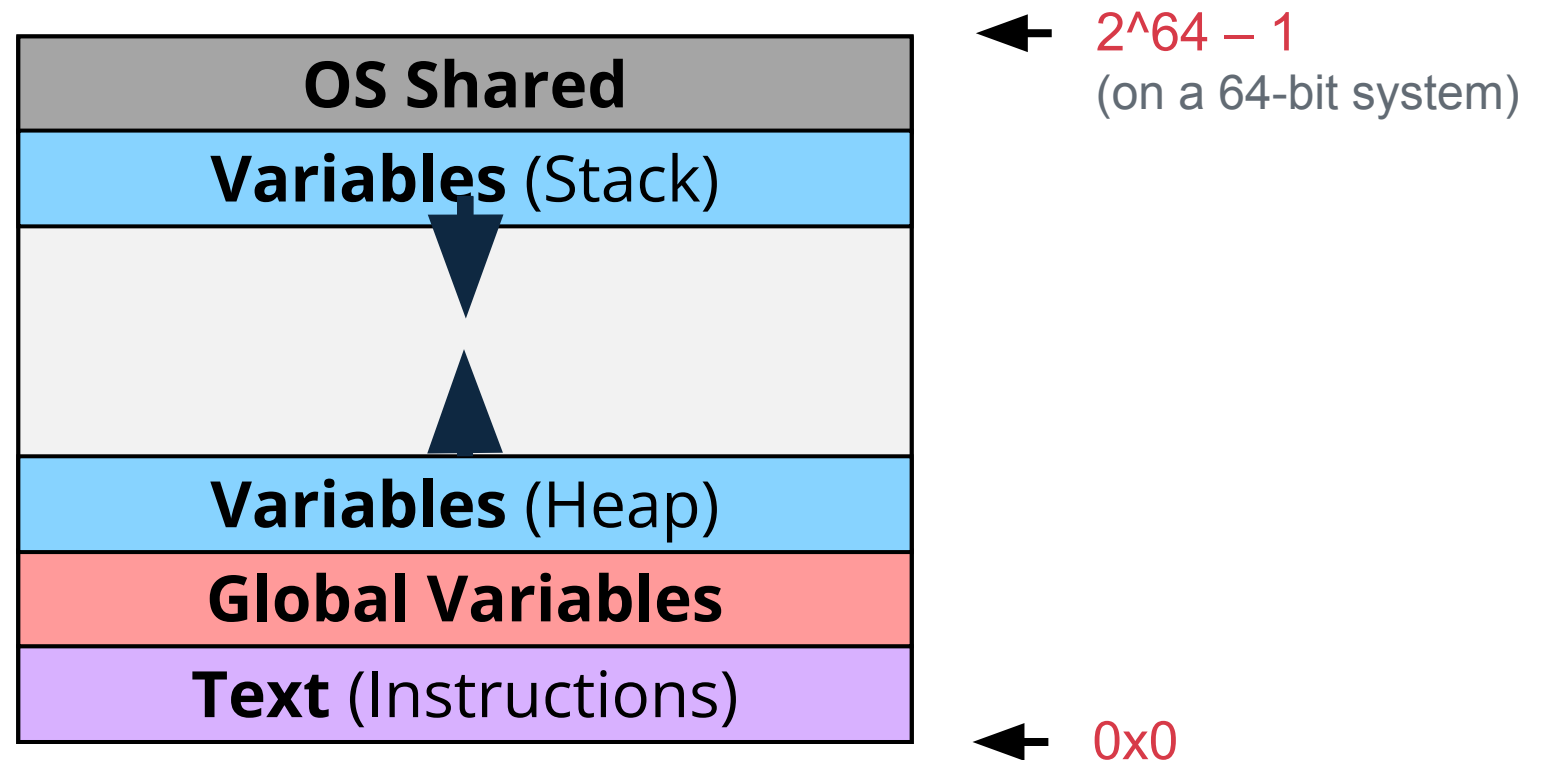
Memory Basics

- 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

*Note: This demonstration uses Big Endian. Little Endian is more common.

What questions do you have?

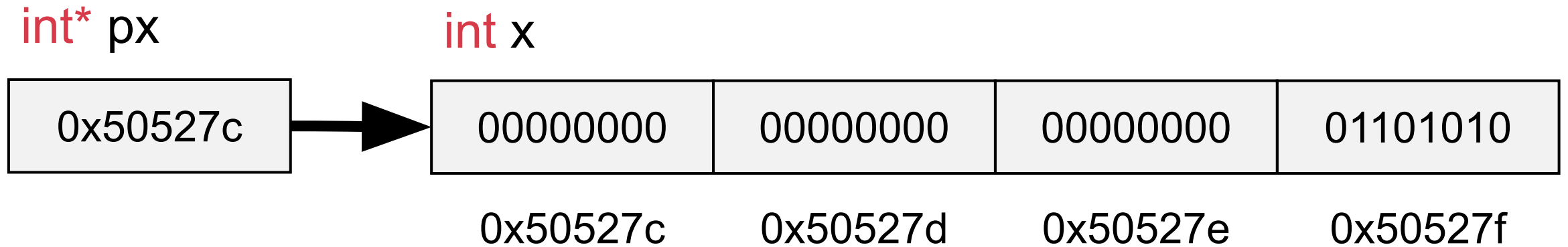


bjarne_about_to_raise_hand

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

Pointers!   

A pointer is just a number!



*Note: This demonstration uses Big Endian. Little Endian is more common.

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
```

What questions do you have?



bjarne_about_to_raise_hand

We can have pointers to all kinds of things!

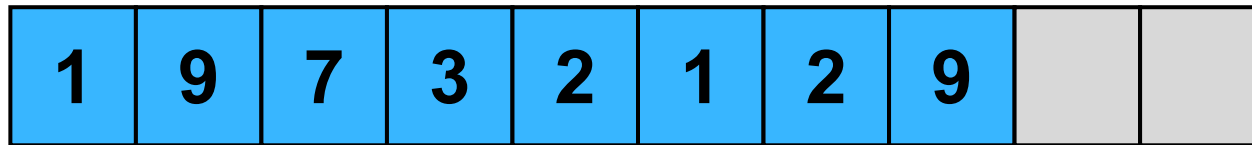
```
int x = 106;  
int* px = &x;
```

```
StanfordID id { "xai" };  
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];
```

```
arr += 1;
```

```
++arr;
```

```
arr += 2;
```

```
if (arr == &v[4])
```

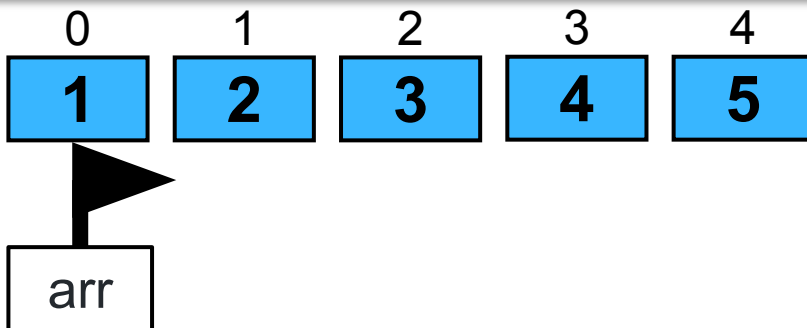
```
std::cout << *arr << " ";
```

```
std::cout << *arr << " ";
```

```
std::cout << *arr << " ";
```

```
std::cout << *arr << " ";
```

```
std::cout << "At last index";
```



Output:

1 2 3 5 At last index

Array pointer

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

```
int* arr = &v[0];
```

```
arr += 1;
```

```
++arr;
```

```
arr += 2;
```

```
if (arr == &v[4])
```

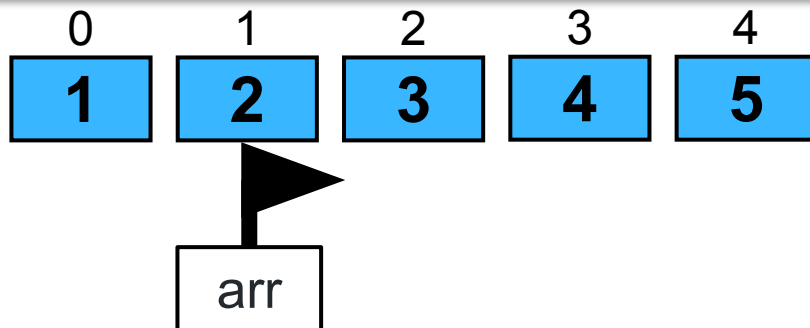
```
std::cout << *arr << " ";
```

```
std::cout << *arr << " ";
```

```
std::cout << *arr << " ";
```

```
std::cout << *arr << " ";
```

```
std::cout << "At last index";
```



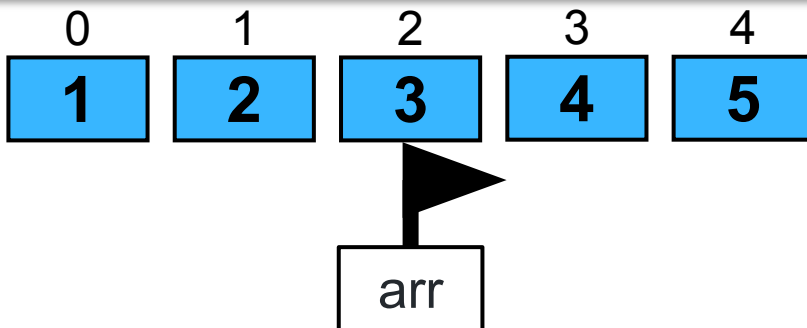
Output:

1 2 3 5 At last index

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

Array pointer

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

```
int* arr = &v[0];
```

```
arr += 1;
```

```
++arr;
```

```
arr += 2;
```

```
if (arr == &v[4])
```

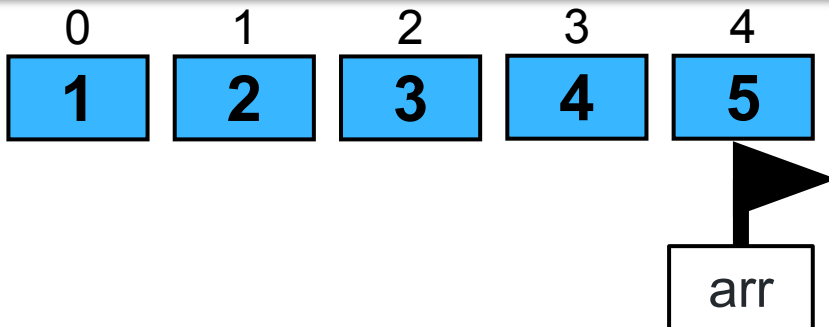
```
std::cout << *arr << " ";
```

```
std::cout << *arr << " ";
```

```
std::cout << *arr << " ";
```

```
std::cout << *arr << " ";
```

```
std::cout << "At last index";
```



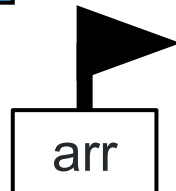
Output:

1 2 3 5 At last index

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];
```

```
arr += 1;
```

```
++arr;
```

```
arr += 2;
```

```
if (arr == &v[4])
```

// Initialization

// Random access

// Move pointer forward

// Random access

// Pointer comparison

We could do the same thing with iterators!

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

```
it += 1;
```

```
++it;
```

```
it += 2;
```

```
if (it == --v.end())
```

```
std::cout << *it << " ";
```

```
std::cout << *it << " ";
```

```
std::cout << *it << " ";
```

```
std::cout << *it << " ";
```

```
std::cout << "At last element";
```

Iterators have a similar interface to **pointers**

**Let's practice w/
slide #5**

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

Classes

We'll learn about them next time

See you all on Tuesday!! :)

Have a great weekend :D