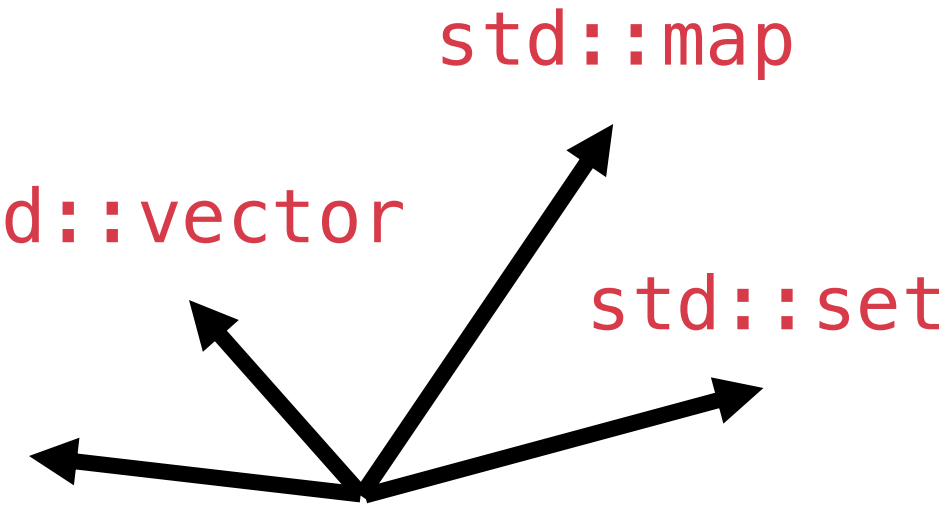
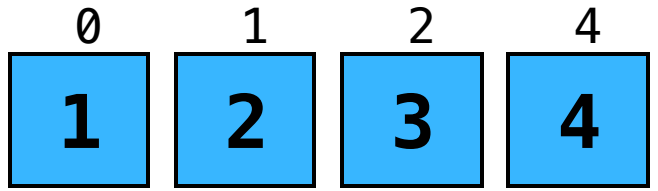



```
for (const auto& elem : container)
```

How does this work?

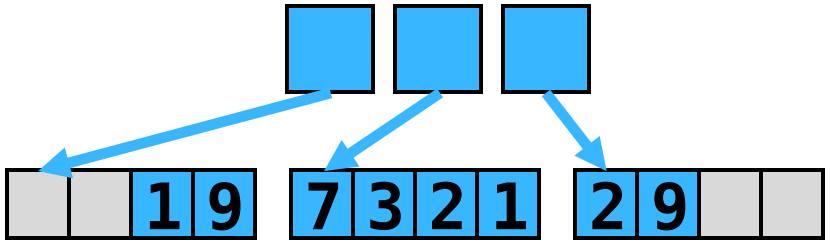
For-each loops… huh?

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

for (const auto& elem : v) {
   std::cout << elem << std::endl;
}
```



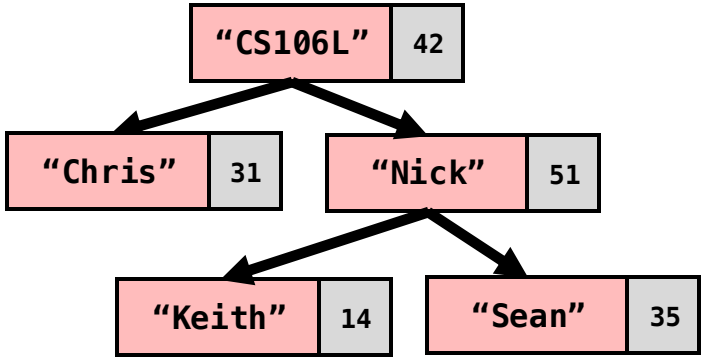
For-each loops… huh?

```
std::deque<int> d {
   1, 9, 7, 3,
   2, 1, 2, 9
};

for (const auto& elem : d) {
   std::cout << elem << std::endl;
}
```



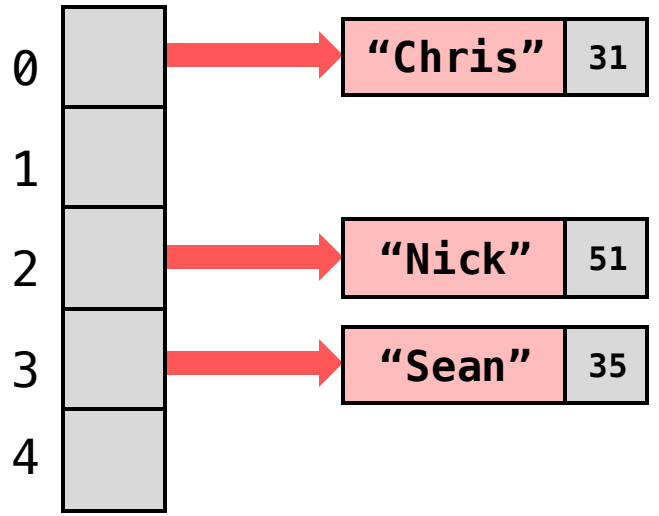
For-each loops… huh?

```
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;
}
```

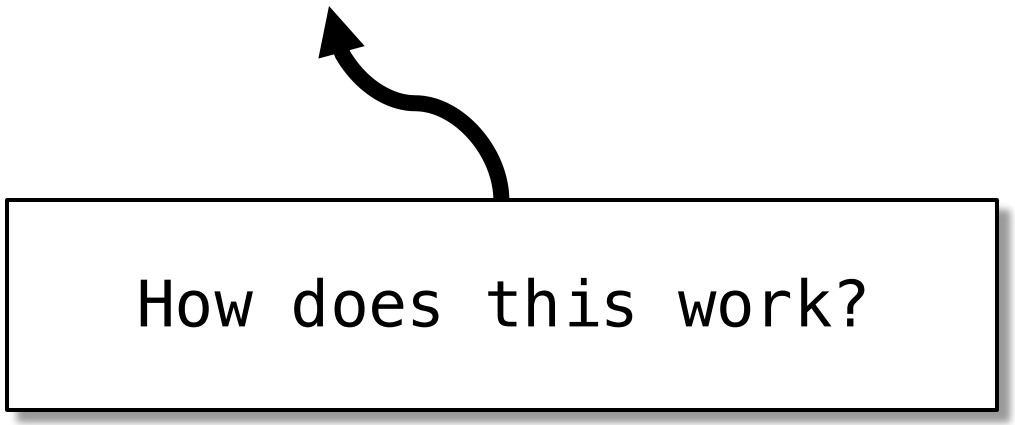


For-each loops… huh?

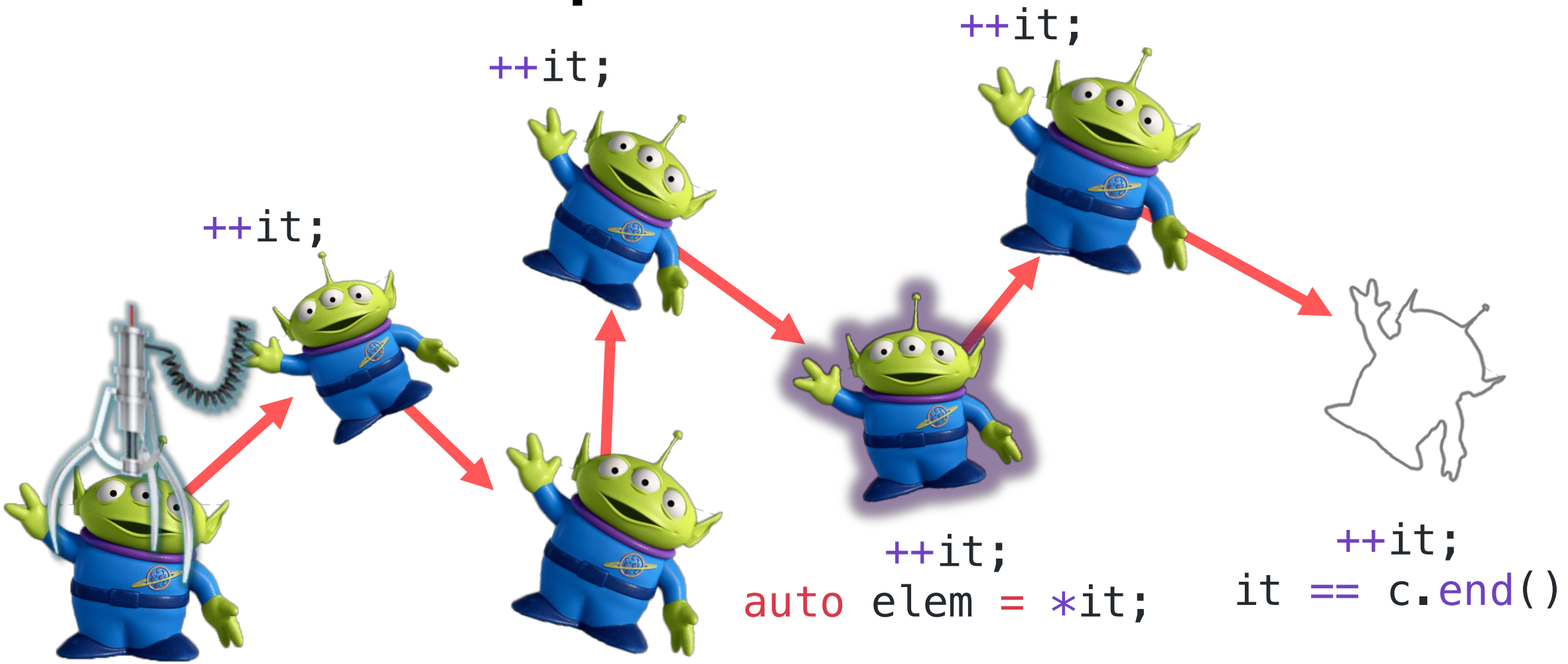
```
std::unordered_map<string, int> m
{
   { "Chris", 31 }, { "Nick", 51 },
   { "Sean", 35 },
};

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



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



C++ Iterators Example



Container Interface

```
container.begin()
```

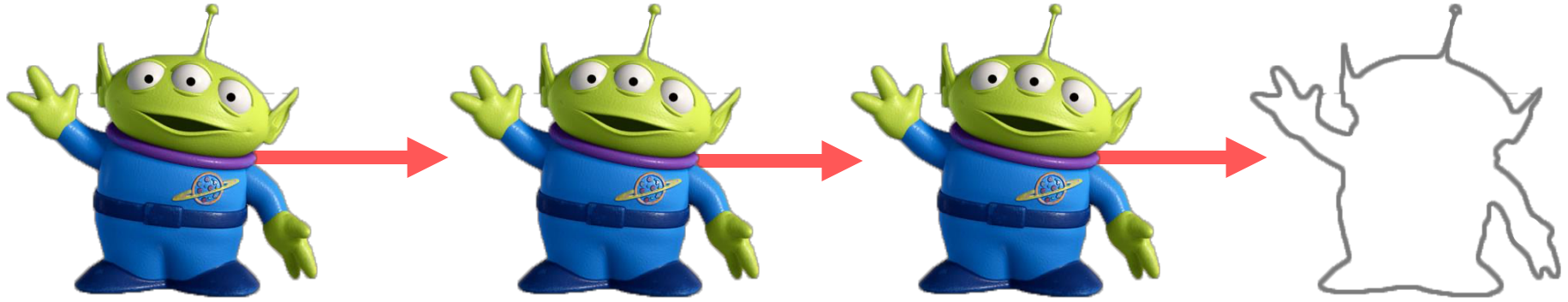
Gets an iterator to
the **first element**
of the container
(assuming non-empty)

```
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!



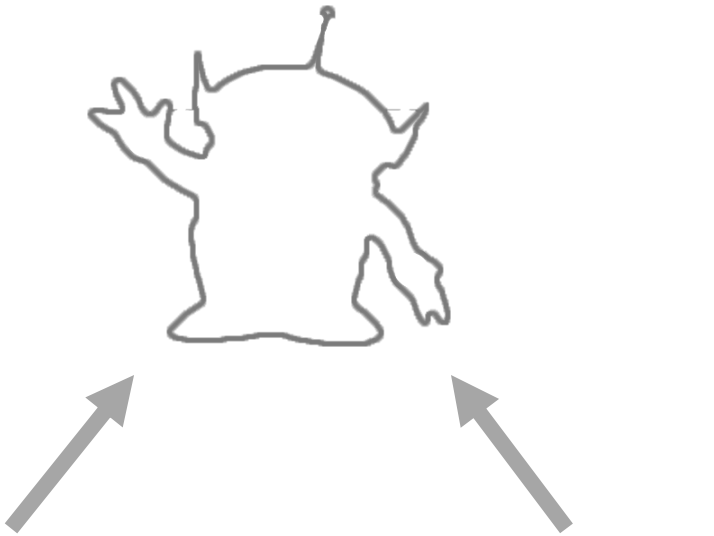
`c.begin()`

Instead, it points **one past the end** of the container

`c.end()`

`end()` never points to an element!

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



```
c.begin()  ==  c.end()
```

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 (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 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();
```

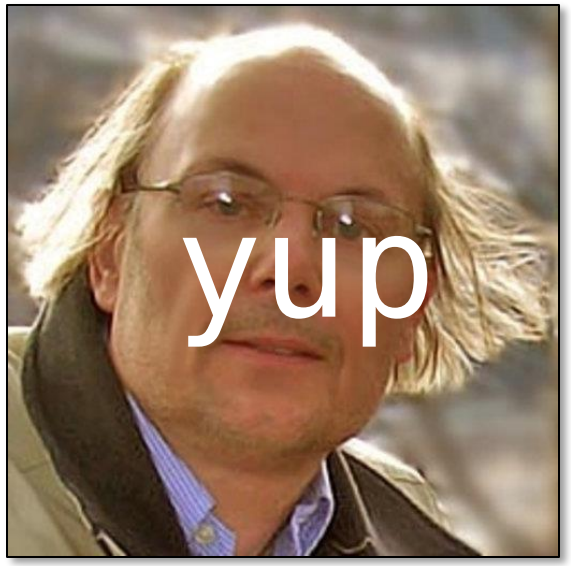
Aside: Why do we use `++it` instead of `it++`?

++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++();
```

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]

Iterator Types

All iterators provide these four operations

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

```
++it;
```

```
*it;
```

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

But most provide even more

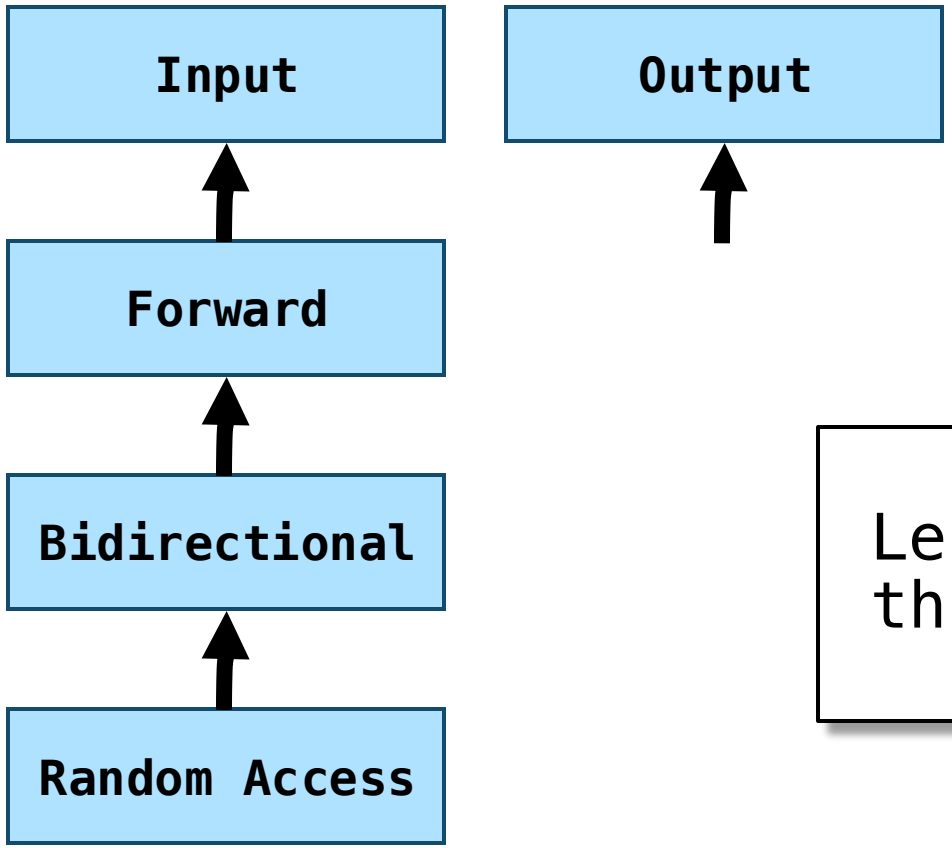
```
--it; // Move backwards
```

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

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

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

Iterator types determine their functionality

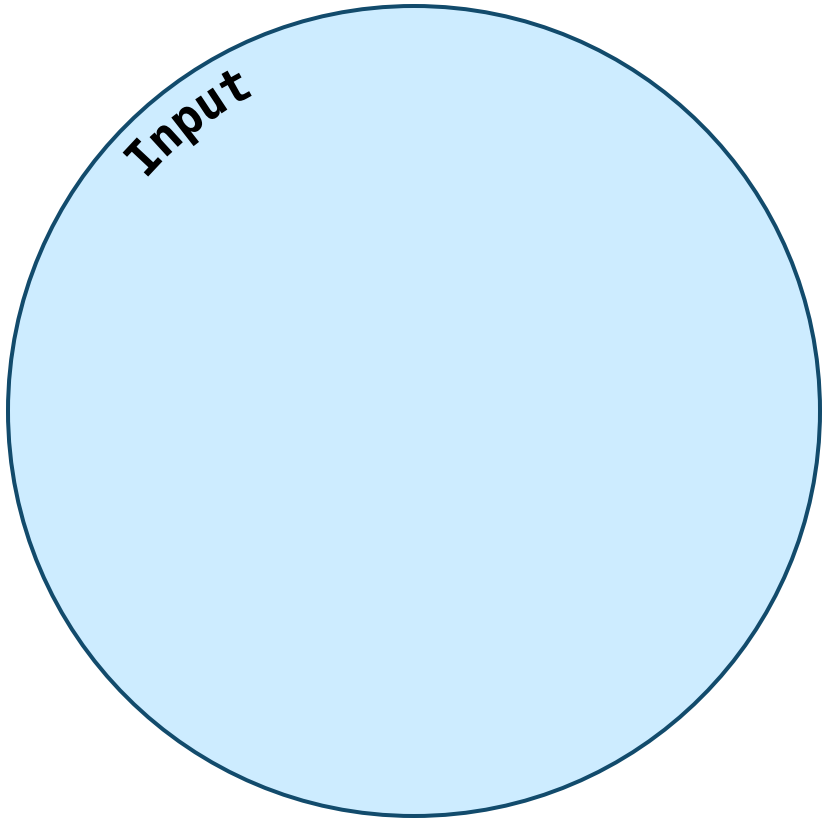


Let’s unpack this!

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 ->

Bibble, v.
"To eat and/or drink noisily"

```
struct Bibble {
   int zarf;
};

std::vector<Bibble> v {...};
auto it = v.begin();
int m = (*it).zarf;
int m = it->zarf;          // Exactly the same as prev!
```

Input Iterators

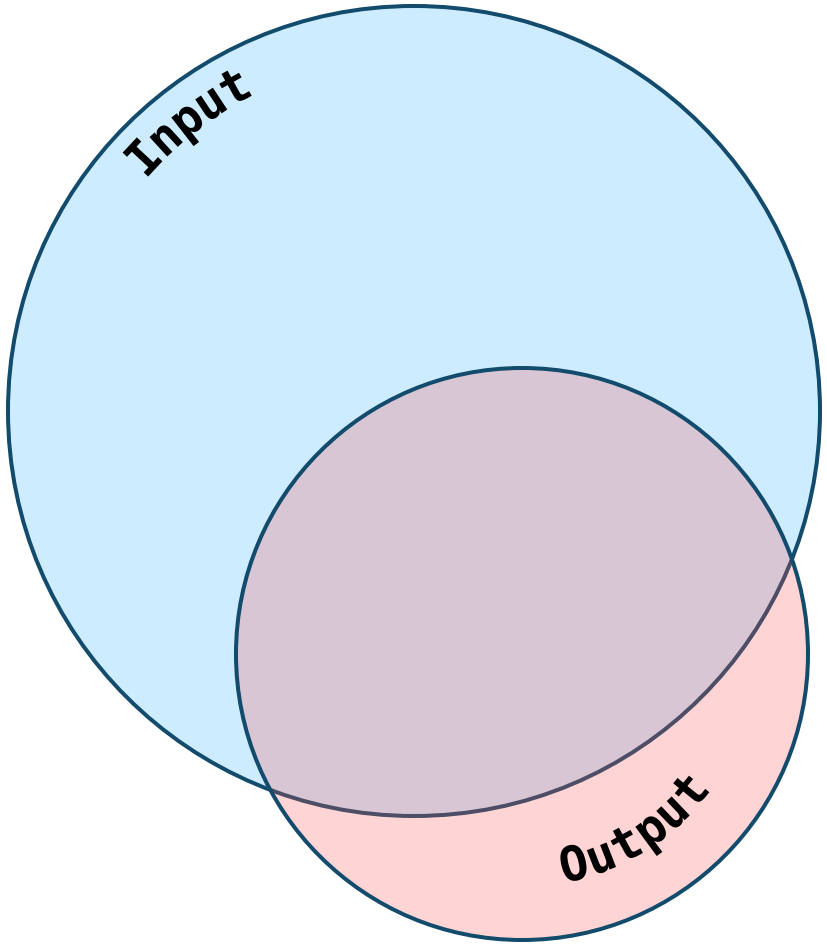
- Most basic kind of iterator
- Allows us to read elements

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

Output Iterator

Allows us to write elements

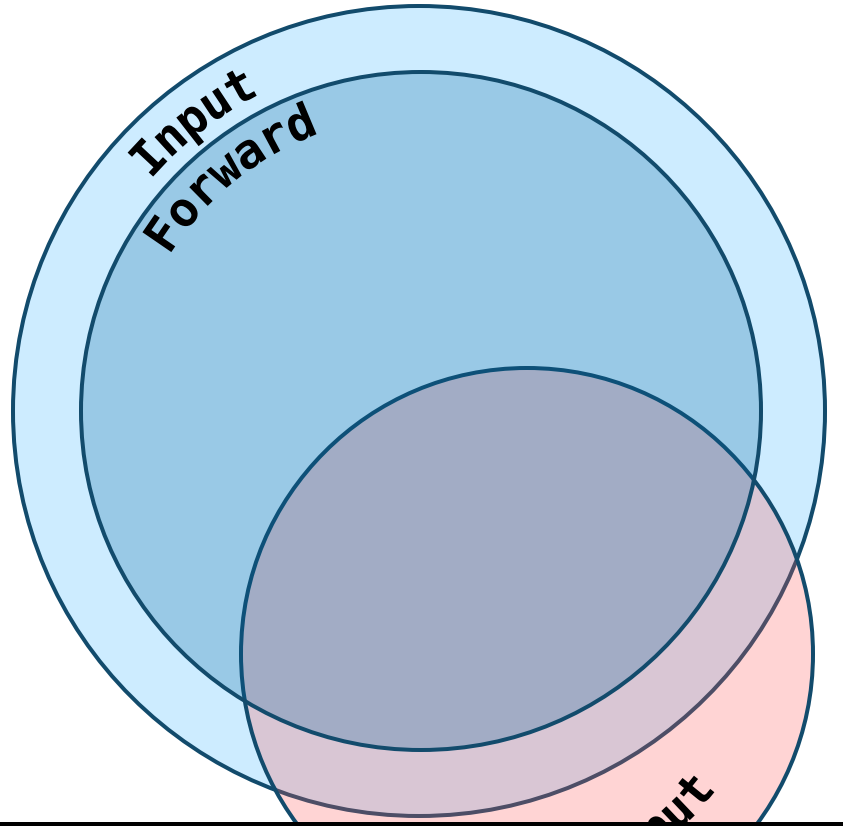
```
*it = elem;
```



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
```

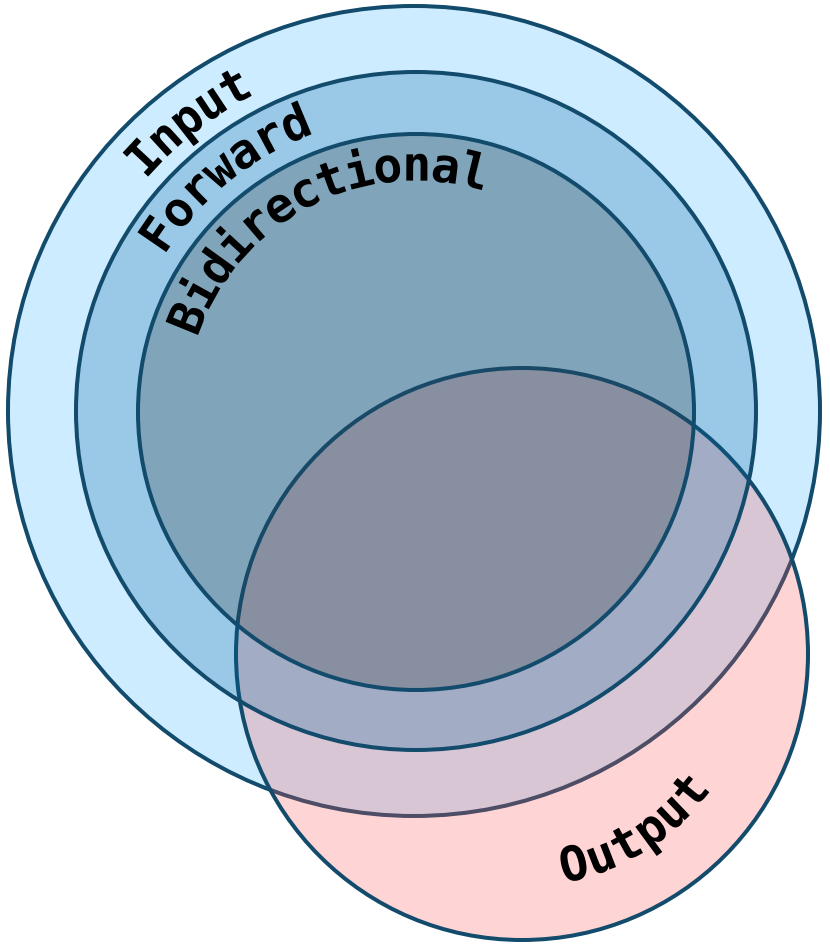
What kind of data structure might not want a multi-pass iterator?

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;
```

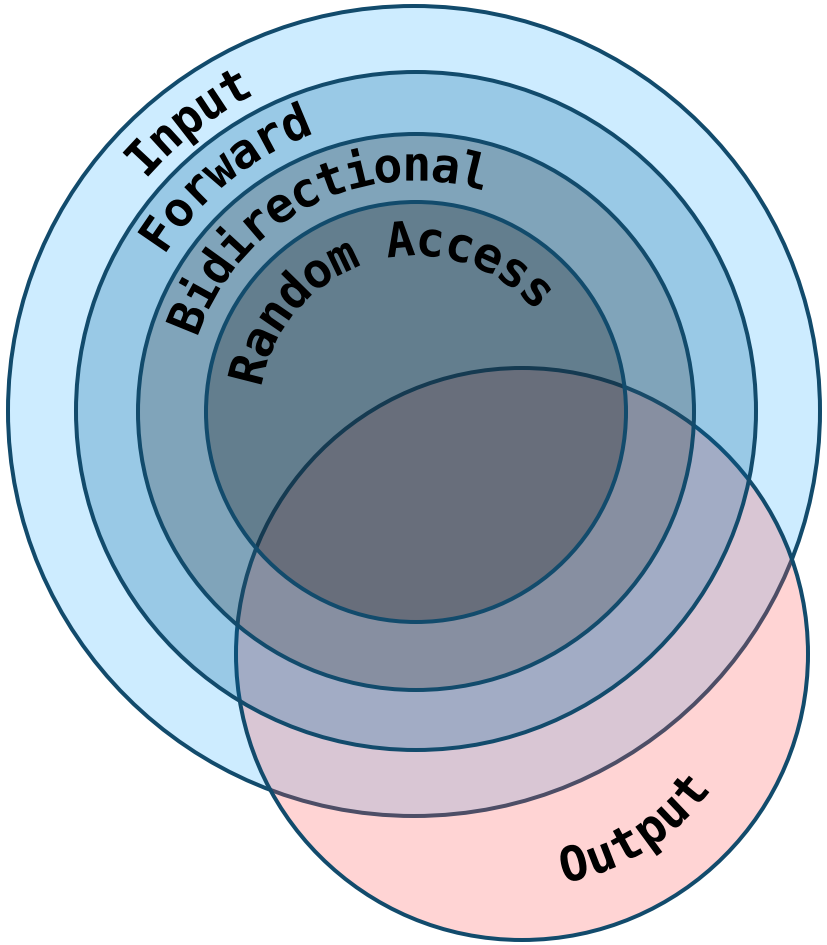


Vivid Venn Diagram of Vexing Iterators

Random Access Iterators

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

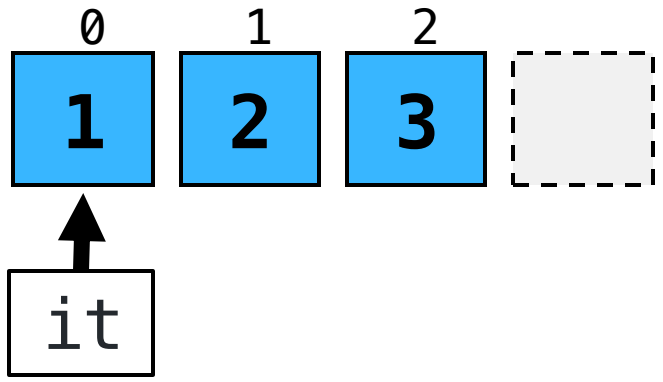
```
auto it2 = it + 5; // 5 ahead
auto it3 = it2 - 2; // 2 back

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



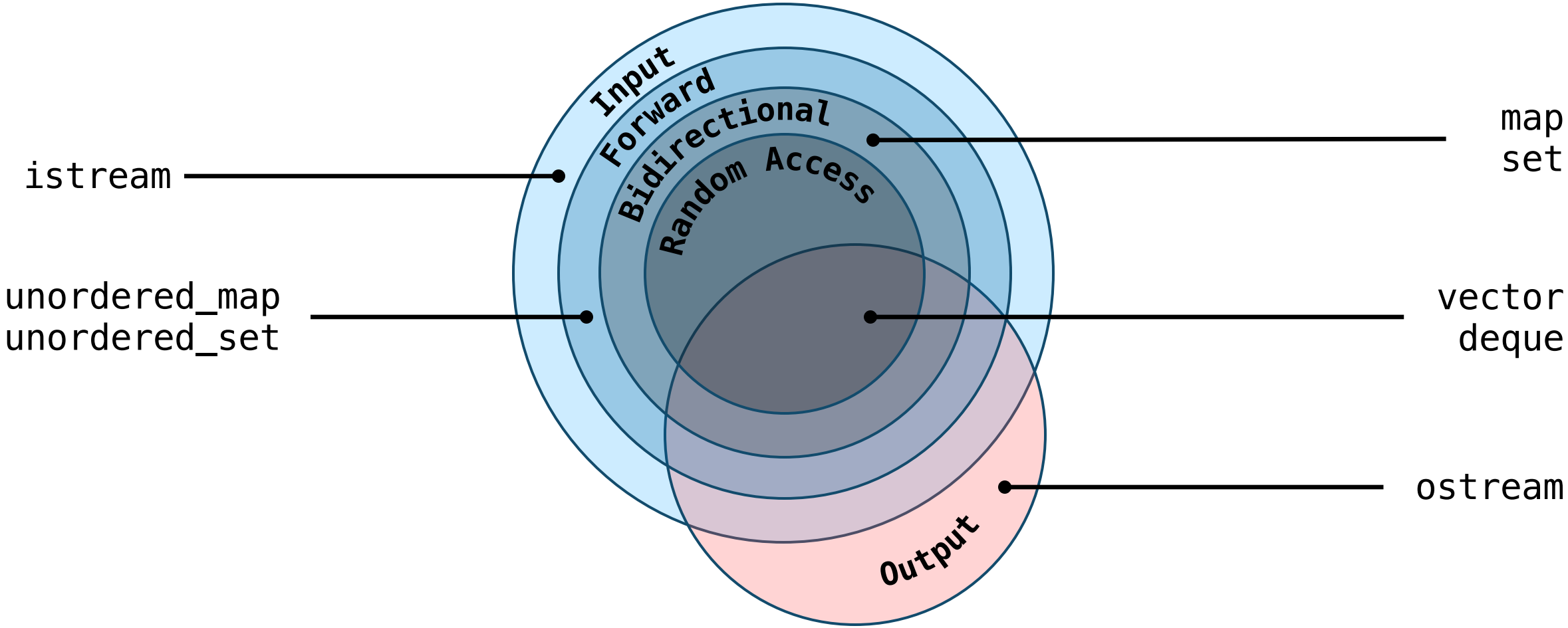
Vivid Venn Diagram of Vexing Iterators

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

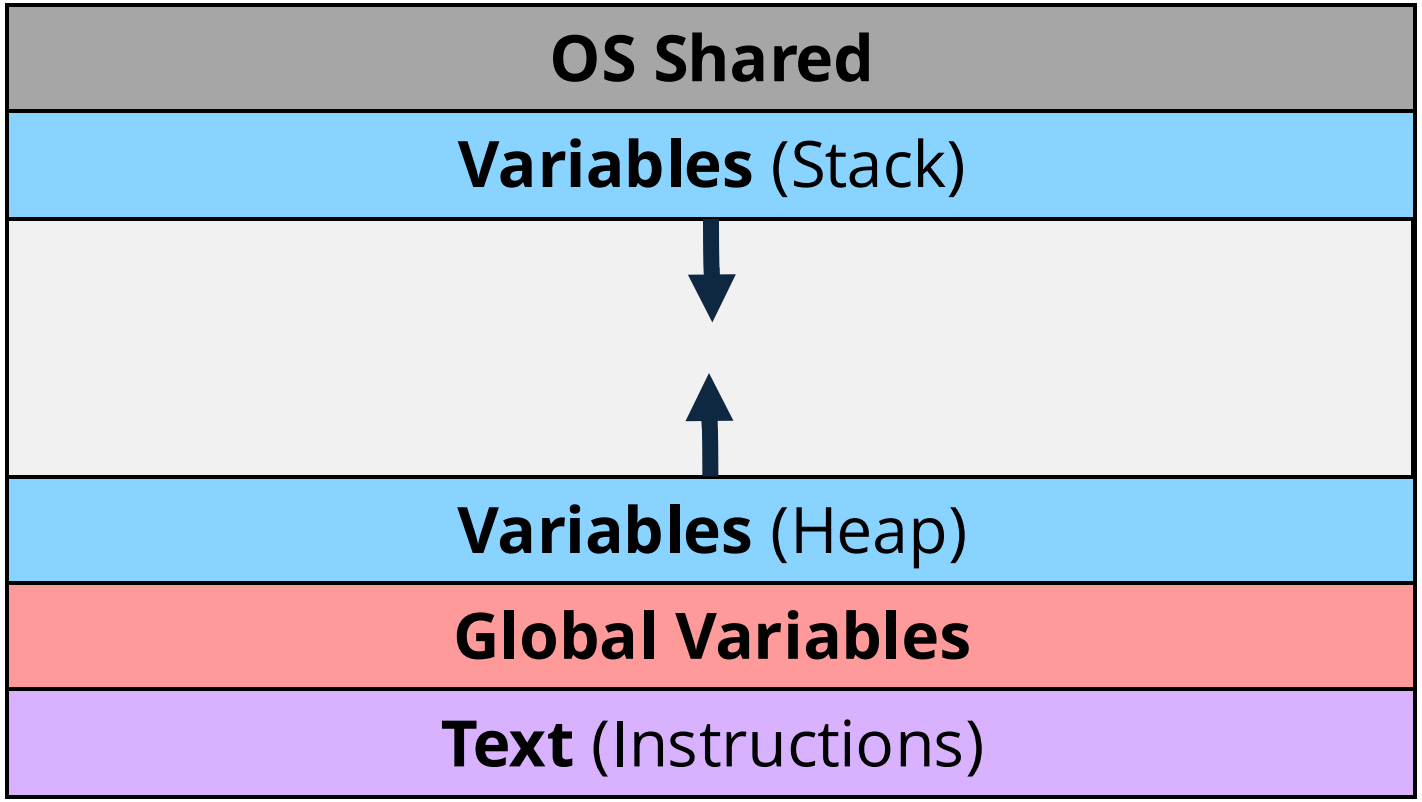


An iterator points to a container element

A pointer points to any object

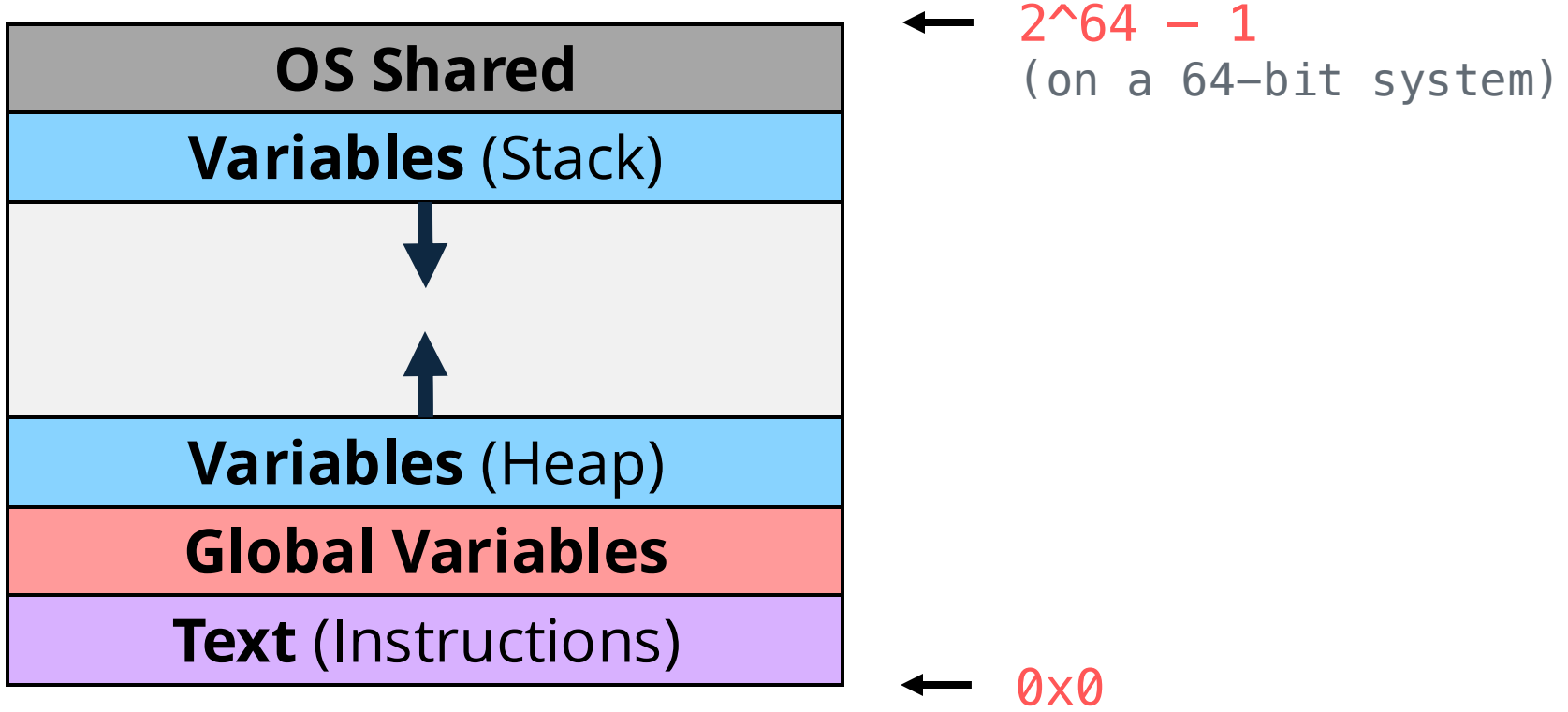
Memory Basics

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



Memory Basics

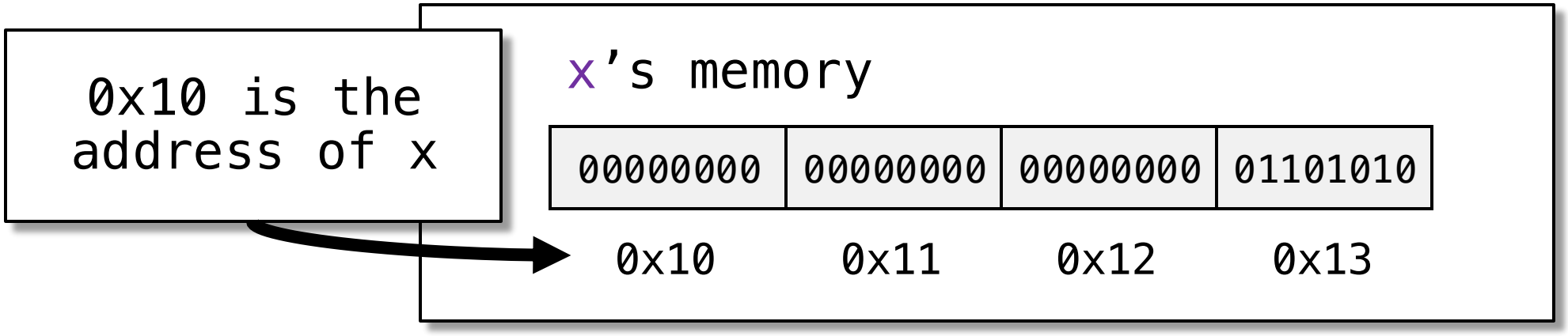
- Memory is usually byte-addressable, with each byte numbered from 0
- 1 byte = 8 bits



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
```



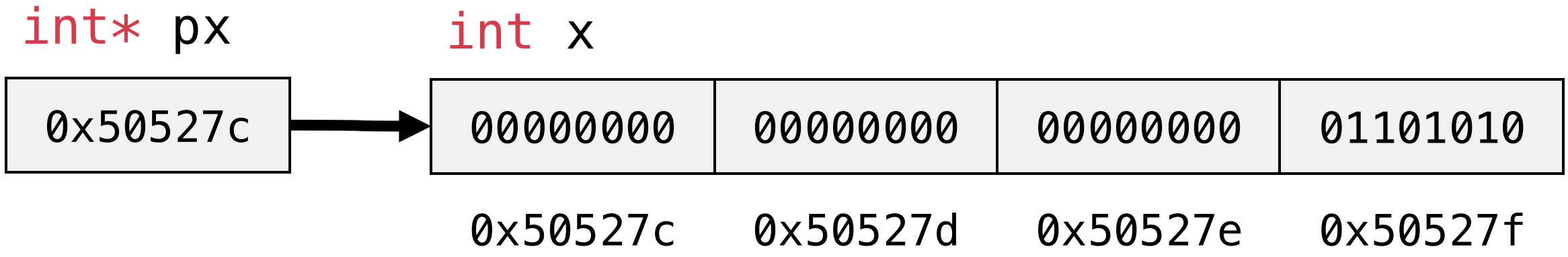
A pointer is the address of a variable

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

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

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

`&` is the address of operator

A pointer is just a number!



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;
```

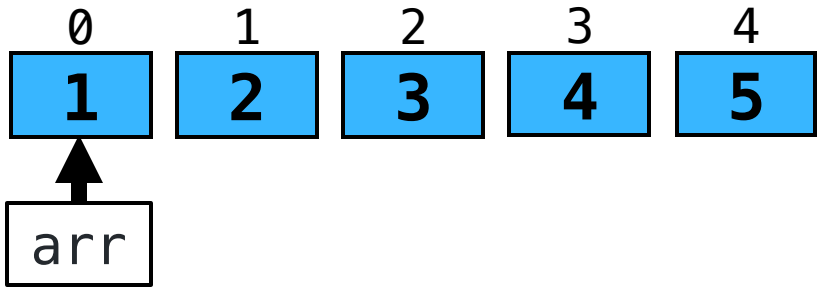
```
std::vector<int> v;
std::vector<int>* p = &v;
```

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

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
```

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();
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

`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.

So how do we implement other iterators?

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

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