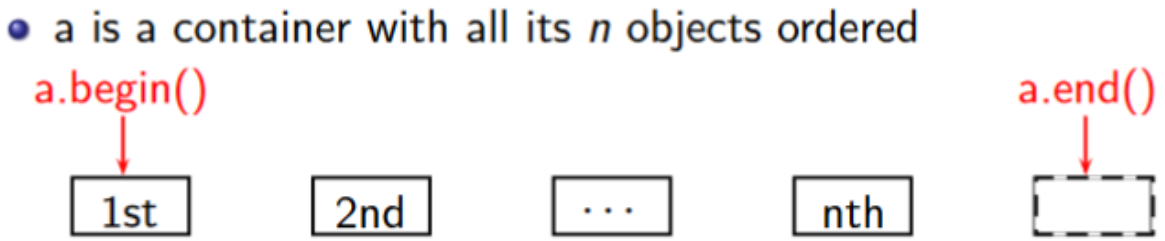
COMP6771
Advanced C++ Programming

2.3 Standard Iterators

What is an Iterator?

- Iterators abstract the concept of a **P**ointer
  - i.e., reference semantics with the same or a subset of the operations as a **p**ointer.
- Iterators are types that abstract container data as a linear **sequence** of objects.
- Iterators allow us to connect a wide range of containers with a wide range of algorithms via a common interface.

- `a.begin()`: abstractly "points" to the first element
- `a.end()`: abstractly "points" to one past the last element
- `a.end()` is not invalid itself, but it is illegal to dereference it.
- If `iter` is an iterator to the k-th element, then:
  - `*iter` is the k-th element.
  - `++iter` abstractly points to the (k + 1)-st element




```
#include <iostream>
#include <vector>

int main() {
  std::vector<int> nums = {1, 2, 3};
  // could use std::vector<std::string>::iterator instead of auto
  for (auto iter = nums.begin(); iter != nums.end(); ++iter) {
    std::cout << *iter << "\n";
  }
}
```

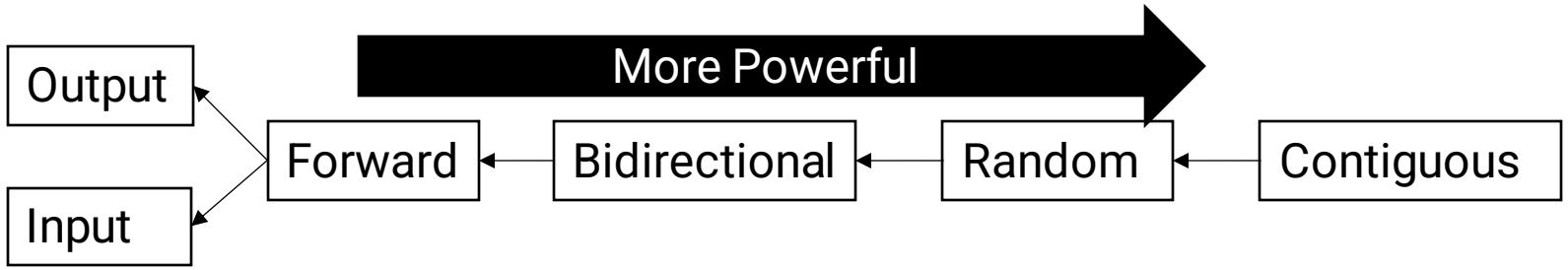
Constant Iterator vs. `const_iterator`

- Remember pointers can be top-level or bottom-level `const`.
- A **constant iterator** is a *top-level* *`const`* iterator. The iterator variable cannot be reassigned.
- A `const_iterator` is a bottom-level `const` iterator. The element pointed to by the iterator cannot be modified.
- Just like a pointer, an iterator can both be a constant iterator and a `const_iterator` simultaneously.
  - Not very useful, however.

```
#include <vector>

auto n = std::vector<int>{1, 2, 3};

// std::vector<int>::iterator
auto it = n.begin();

// const std::vector<int>::iterator;
const auto constant_it = n.begin();

// std::vector<int>::const_iterator
auto const_it = n.cbegin();

// const std::vector<int>::const_iterator
const auto constant_const_it = n.cbegin();
```

Iterator Categories

Operation	Output	Input	Forward	Bidirectional	Random Access	Contiguous
Read		=*p	=*p	=*p	=*p	=*p
Access		->	->	->	->, []	->, []
Write	*p=		*p=	*p=	*p=, p[n]=	*p=, p[n]=
Advance	++	++	++	++, --	++, --, +, -, +=, -=	++, --, +, -, +=, -=
Compare		==, !=	==, !=	==, !=	==, !=, <, >, <=, >=	==, !=, <, >, <=, >=
Array-like?	Potentially	Potentially	Potentially	Potentially	Potentially	Guaranteed



Stream Iterators

- iostreams (`std::ifstream`, `std::ofstream`, `std::cout`, `std::cin`, etc.) do not define their own iterators.
- Can use std::istream_iterator or std::ostream_iterator to make one.
- Need to specify in the <> what the type of data to find is.
  - Delimits by default on whitespace.
  - Delimiter customisable.

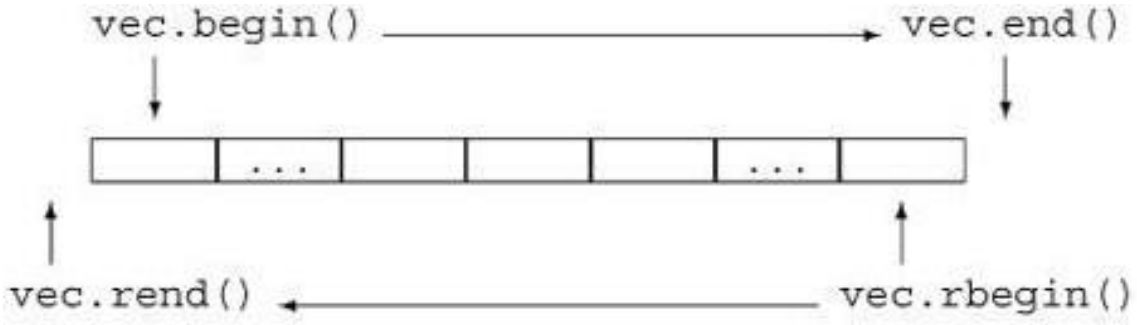
```
auto in = std::ifstream{“ass1.sol”};
auto begin = std::istream_iterator<int>{in};

// below is equivalent to EOF for stream iters
auto end = std::istream_iterator<int>{};

std::cout << *begin++ << “\n”; // read an int

++begin; // skip the next int
for(; begin != end; ++begin) // read the rest
        std::cout << *begin << std::endl;
```

Iterator Adaptors

- The `<iterator>` header offers many convenience functions to adapt iterators for use in common problems.
- We will look at the most common ones:
  - Reverse iterators
  - Back-inserter iterators
  - Insert iterators

Reverse Iterators

- Reverse iterators allow us to iterate backwards through a container.
- All standard containers provide methods to get a reverse iterator.
- Later, we will see how to make our own reverse iterators extremely easily.




```
std::vector<int> v = {3, 6, 9};

for (auto r = v.rbegin(); r != v.rend(); ++r) {
        std::cout << *r << std::endl;
} // prints 9, 6, 3
```

Back Inserter Iterators

- Gives you an output iterator for a container that supports `push_back()`.
- Writing to the output iterator causes the underlying container to push back the new element.

```
#include <iterator>


std::vector<int> nums;
auto it = std::back_inserter(nums);
*it = 42;
*it = 6771;


for (int i : nums) {
        std::cout << i << std::endl;
} // prints 42, 6771
```

Insert Iterators

- Gives you an output iterator for a container that supports `insert()`.
- Writing to the output iterator causes the underlying container to insert the new element.
- Need to give the container as well as an iterator into the container to start inserting from.

```
#include <iterator>

std::vector<int> v;
auto it = std::inserter(v, v.begin());
*it = 42;           // calls v.insert()
*it = 6771;         // calls v.insert()

for (int i : v) {
        std::cout << i << std::endl;
} // prints 42, 6771
```

Feedback (stop recording)