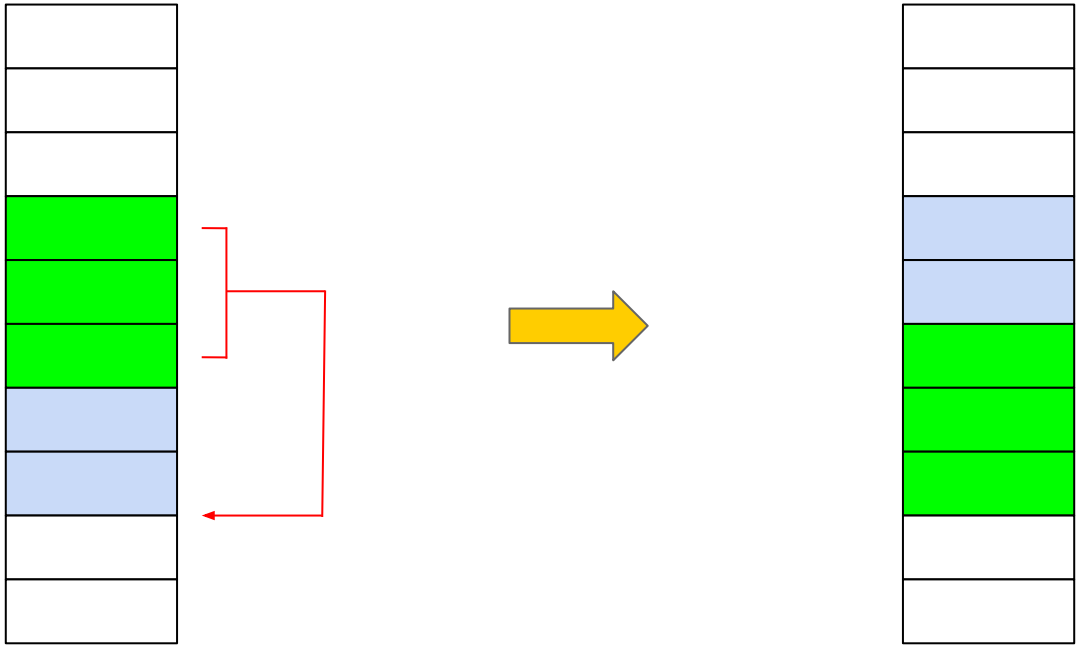
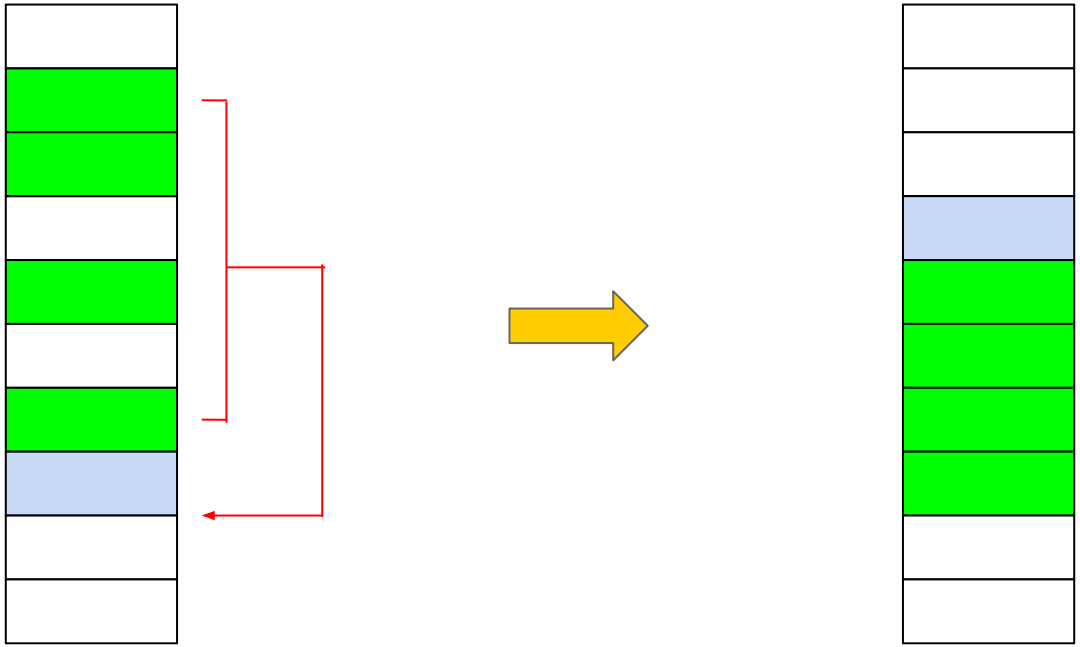
How to make your code better with C++ algorithms



How would you solve this?



And now this?

Hello!

I am Mathieu Ropert

I'm a C++ developer at Paradox Development Studio where I make Europa Universalis.

You can reach me at:

 mro@puchiko.net

 @MatRopert

 https://mropert.github.io

C++ Approach to Algorithms

- Raw loops on collections should be avoided
- Bug prone
- Lack expressiveness
- Lead to code duplication

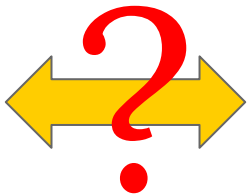
C++ Approach to Algorithms

- Based on the work of Alex Stepanov
- Generic programming
- No ties to the containers they are applied to
- Follow C++ general design principles

Container vs Algorithms

Containers

- `vector`
- `map`
- `list`
- `unordered_map`

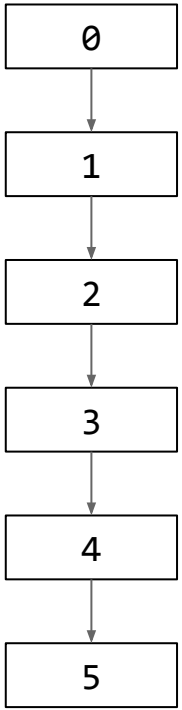


Algorithms

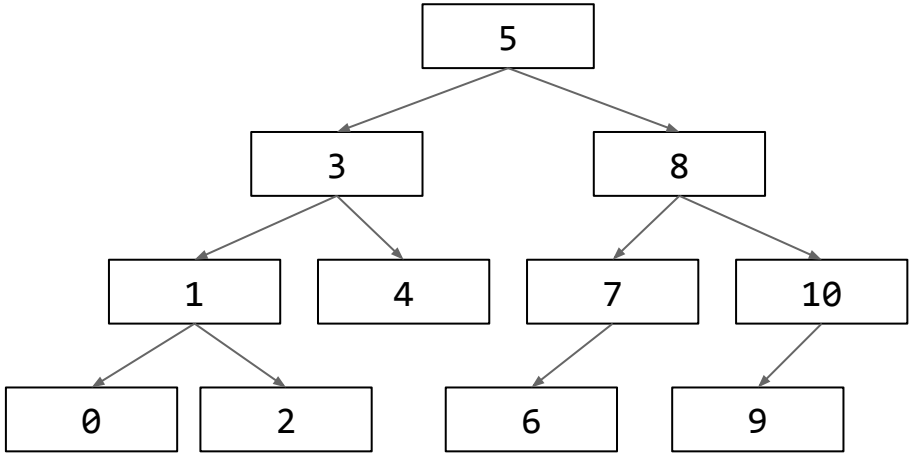
- `any_of`
- `sort`
- `transform`
- `remove_if`

Iterators

vector

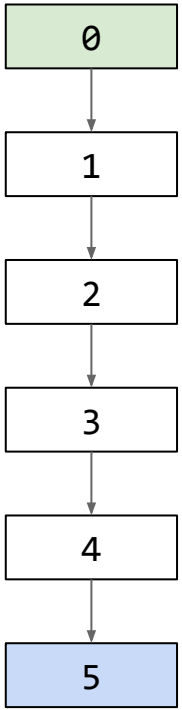


list

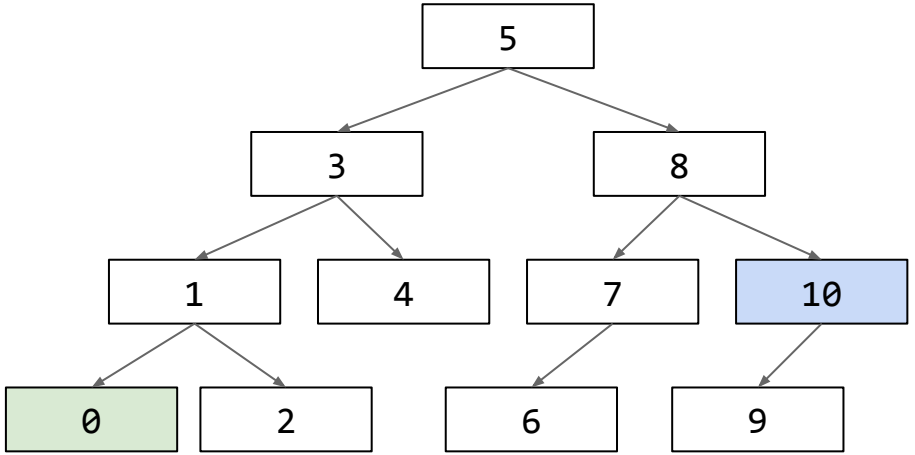


map

vector

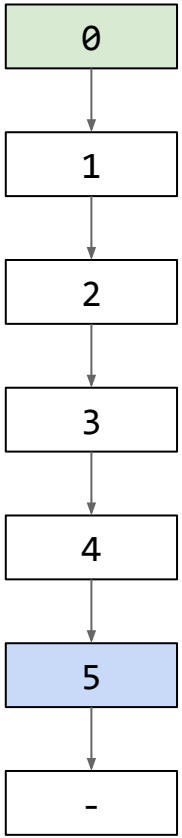


list

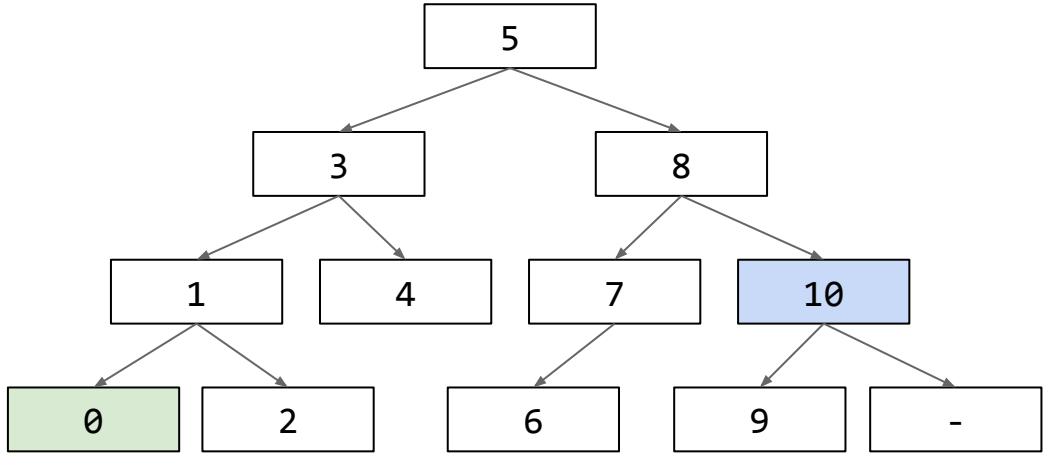


map

vector

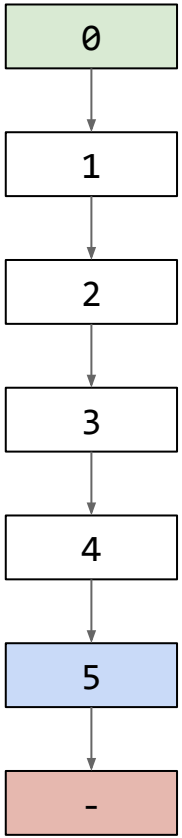


list

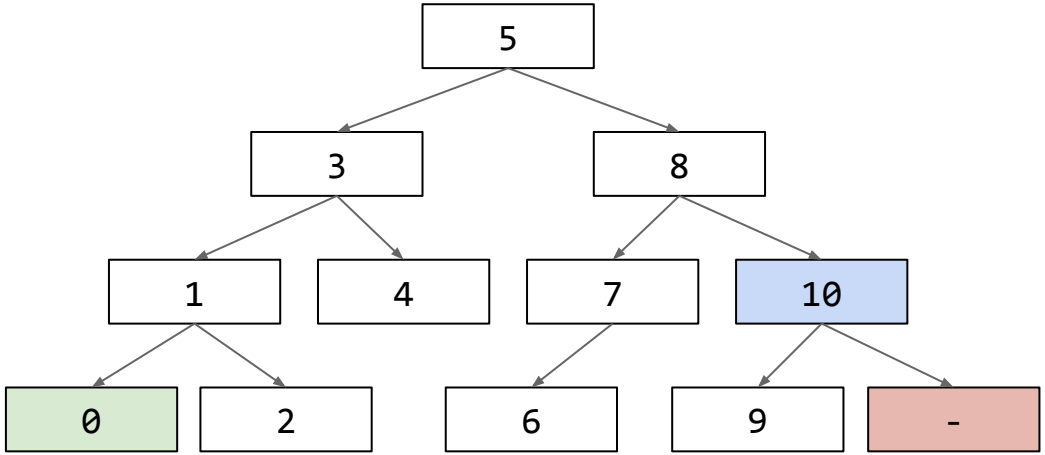


map

vector



list



map

C++ Iterators

- Can point to any element in a collection...
- ... or to an imaginary element that would be one past the last
- In math language, an iterator range is `[b,e[`
  - ... or `[b,e)` in weird english notation

Iterators Concepts

- Can be derefenced to get the pointed element
  - Unless they are past the end
- Can be incremented to point to the next one
- Further classified by subtypes with additional properties

Iterators Concepts

- `Input` iterators can be de-referenced to read from pointed value
- `Output` iterators can be de-reference to write to the pointed value
- `Mutable` iterators provide both

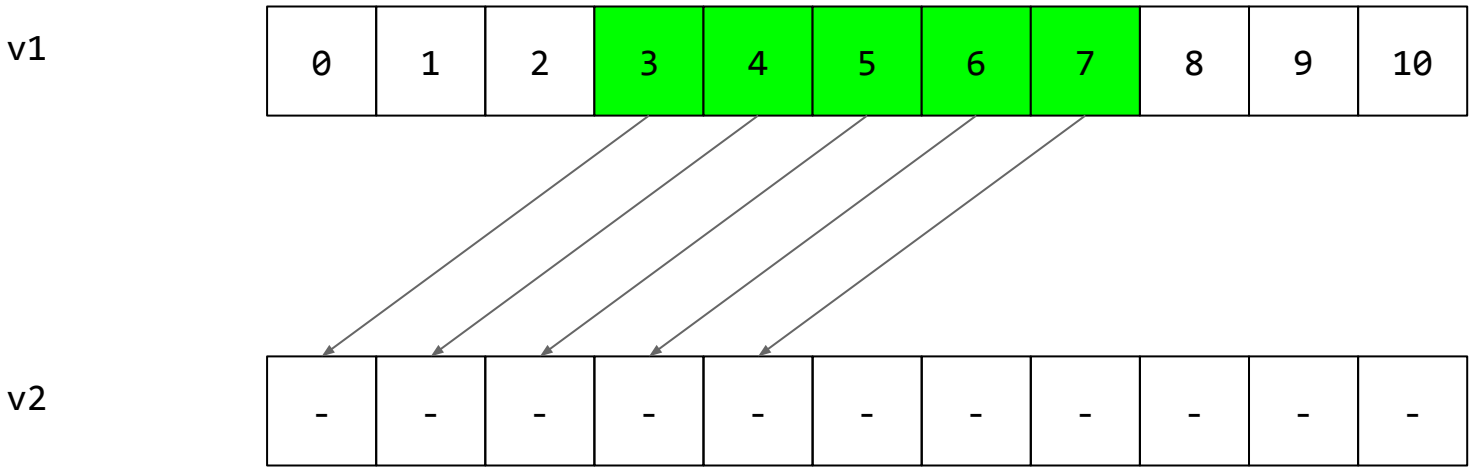
Iterators Concepts

- `Forward` iterators can traversed multiple times (incrementing does not “consume” data)
- `Bidirectional` iterators can be decremented to return to a previous element
- `RandomAccess` iterators can be incremented by an arbitrary amount in constant time

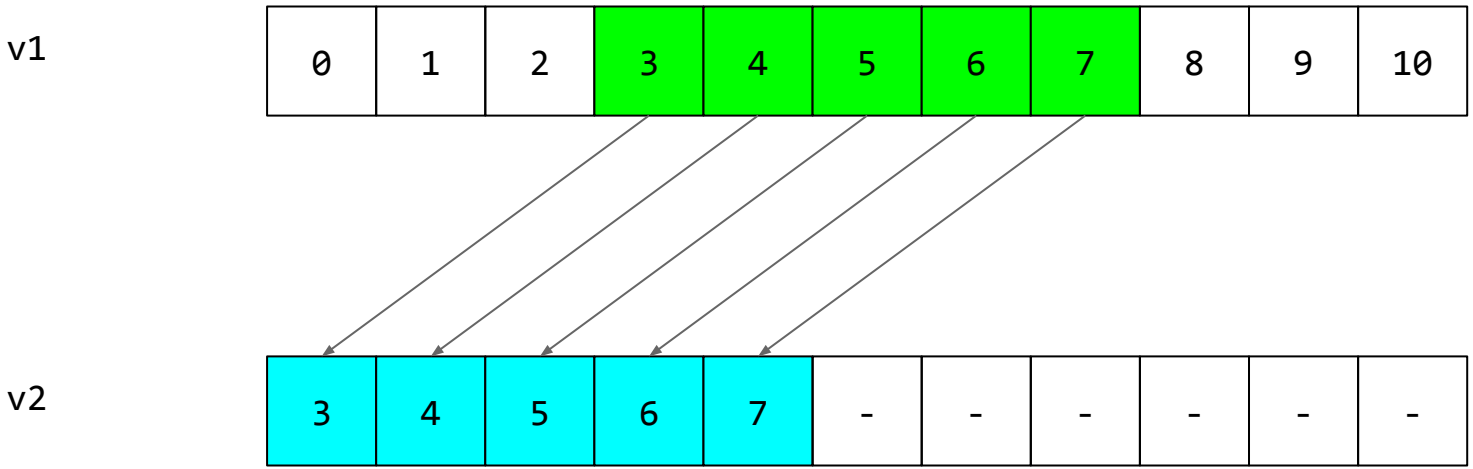
Iterators Concepts

- `Input` -> Message queue
- `Forward` -> Single linked list
- `Bidirectional` -> Double linked list
- `RandomAccess` -> Array

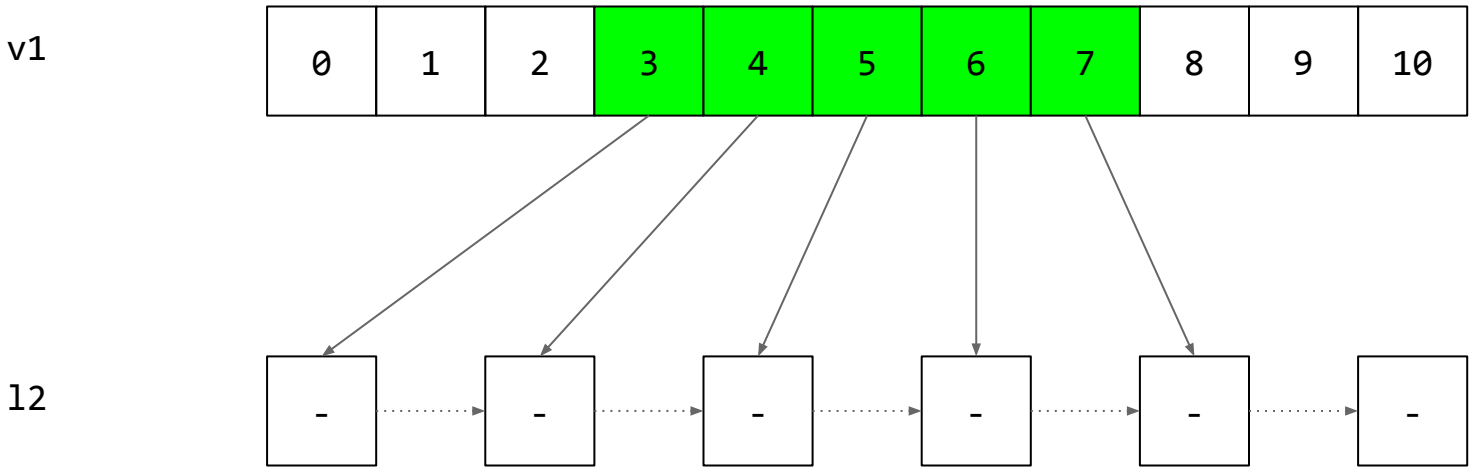
Modifying Sequence Algorithms



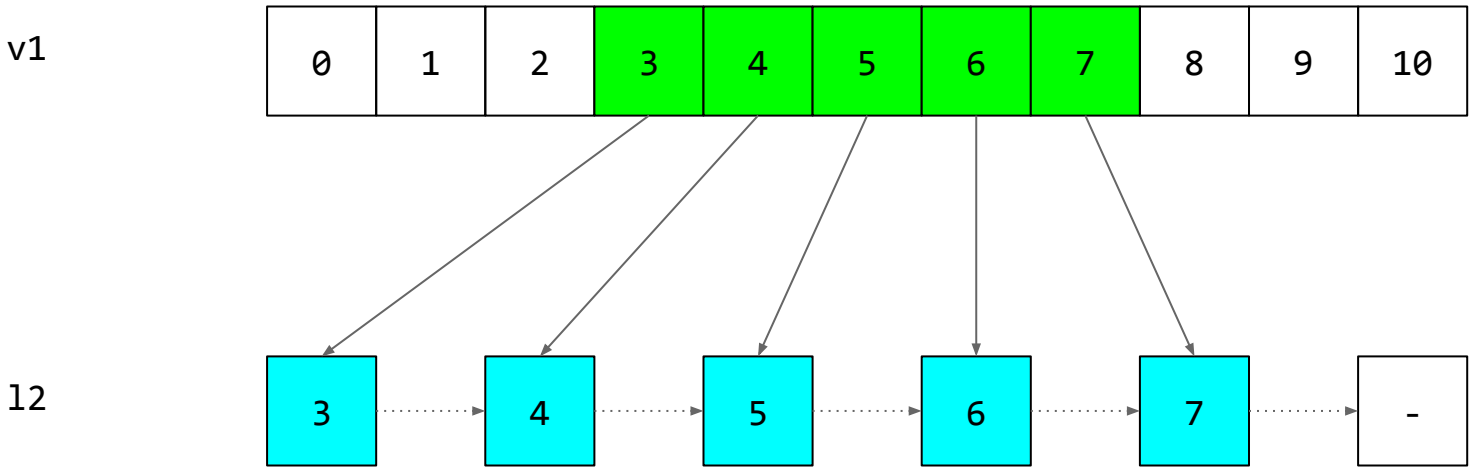
std::copy



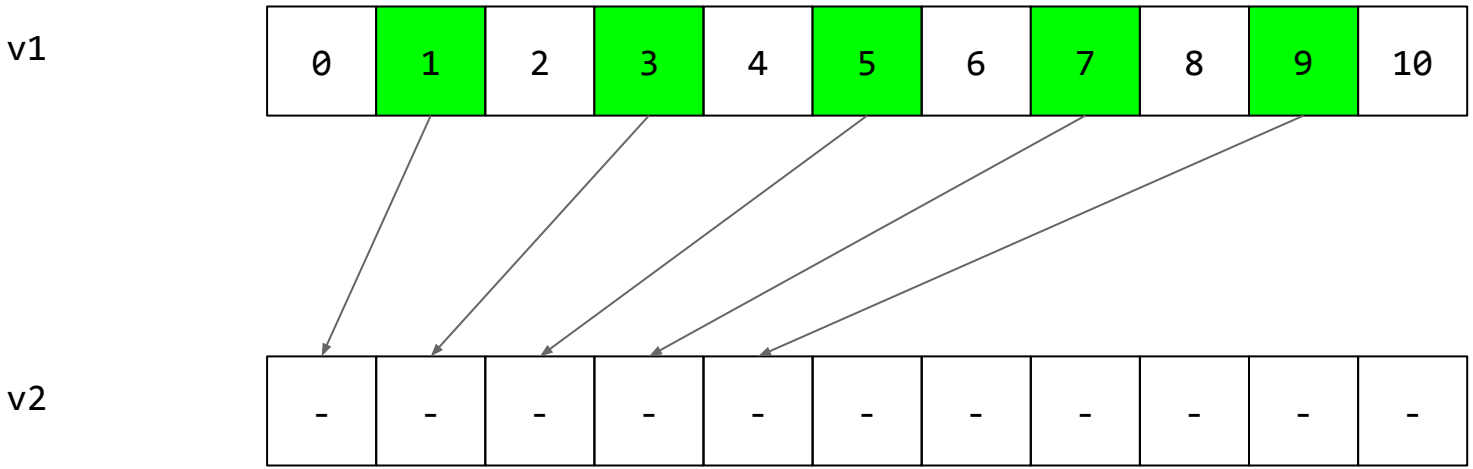
std::copy(v1.begin()+3, v1.begin()+8, v2.begin())



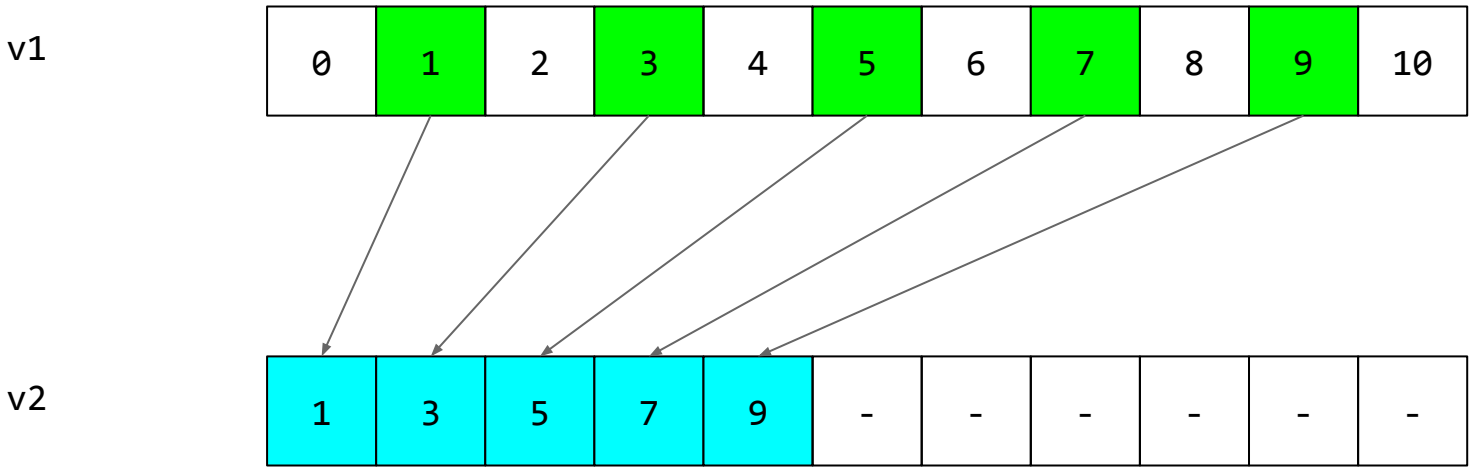
std::copy



std::copy(v1.begin()+3, v1.begin()+8, l2.begin())



std::copy_if




```
std::copy_if(v1.begin(), v1.end(), v2.begin(),
[](int v) { return v % 2 == 1; } )
```

Copy is not insert

- `std::copy()` only copies the items from collection A to collection B
- Undefined behaviour if destination is smaller than source
- Need another construct to insert

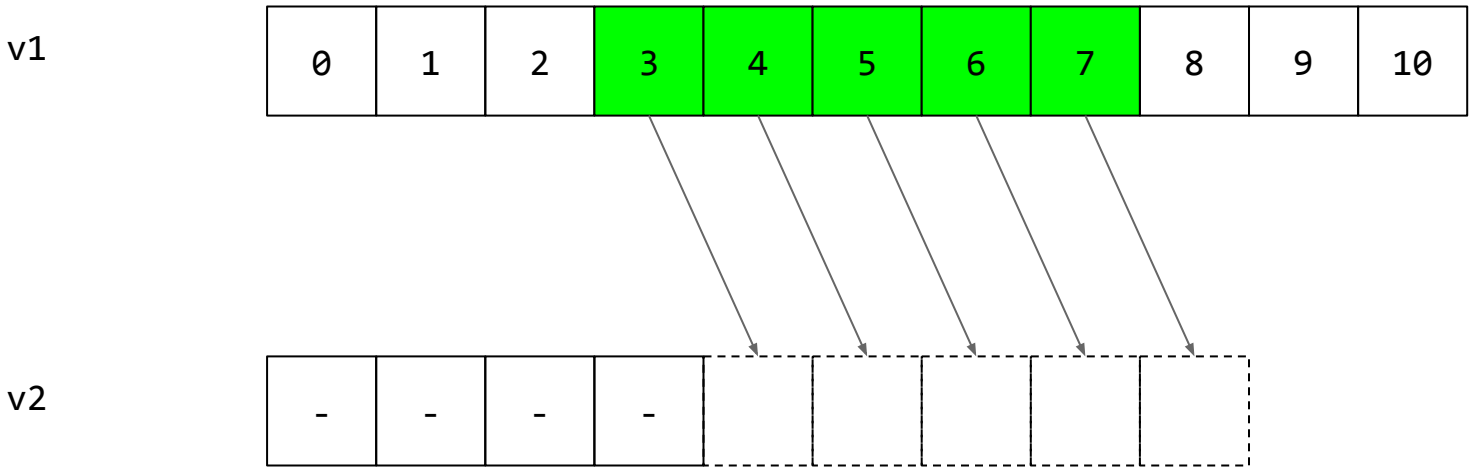
Iterator adaptors

- Iterators are concepts, it's perfectly fine to have them do more than read/write a collection in sequence

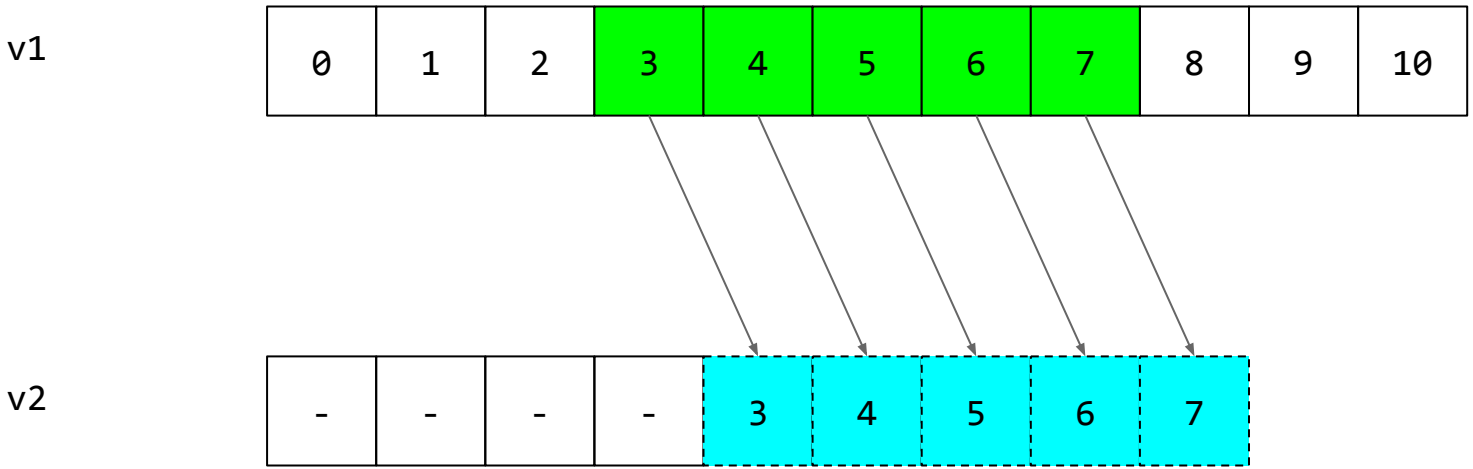
- By using the adaptor design pattern, we can make them do whatever we want and even compose them

Common iterator adaptors

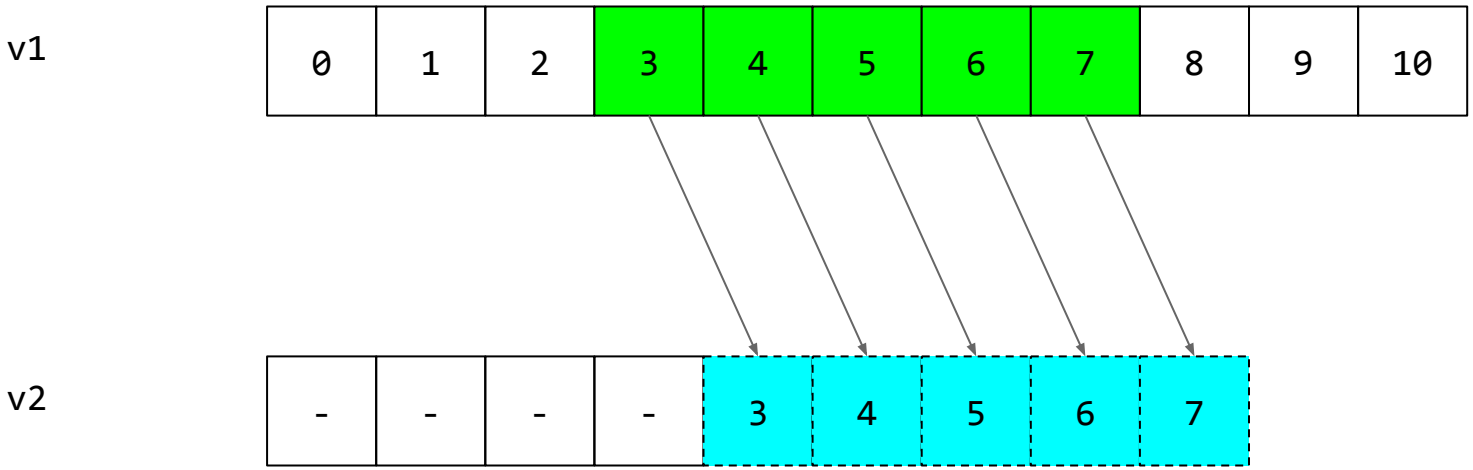
- `std::back_inserter( c )`
  - Output iterator that calls `c.push_back()` on write
- `std::inserter( c, it )`
  - Output iterator that calls `c.insert(it)` on write
- `std::reverse_iterator<T>`
  - I/O iterator that iterates backwards on increment



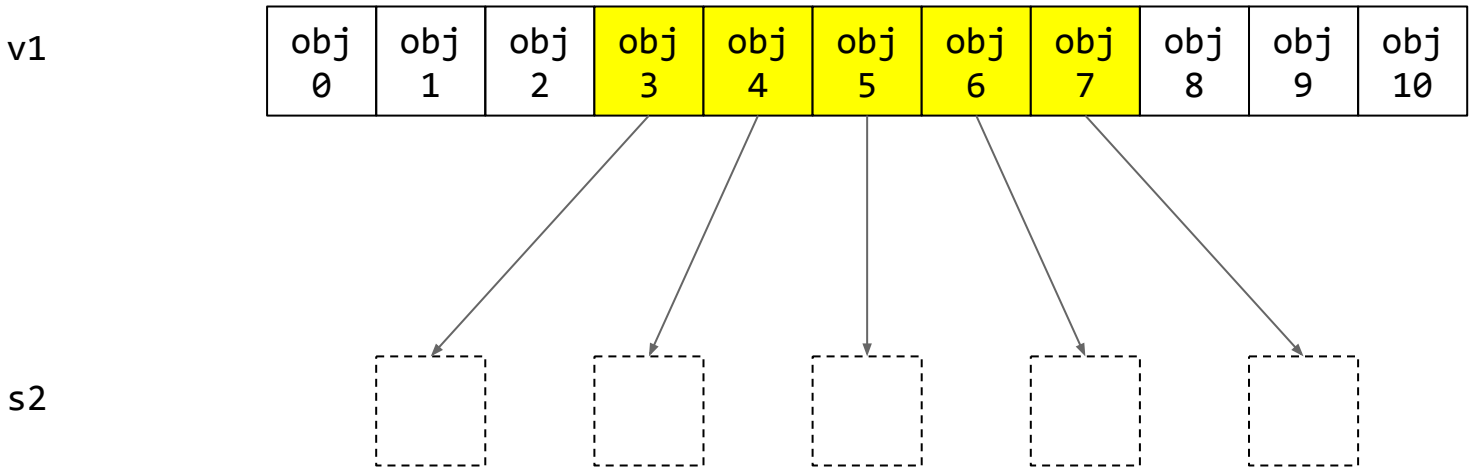
std::copy




```
std::copy( v1.begin()+3, v1.begin()+8,
           std::back_inserter( v2 ) )
```

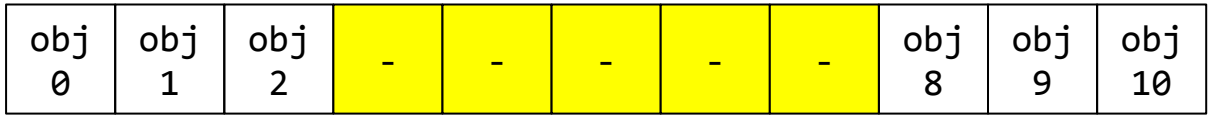



```
std::copy_n( v1.begin()+3, 5,
             std::back_inserter( v2 ) )
```

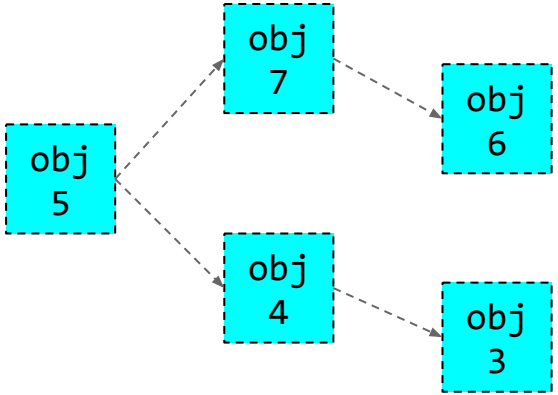


Move?

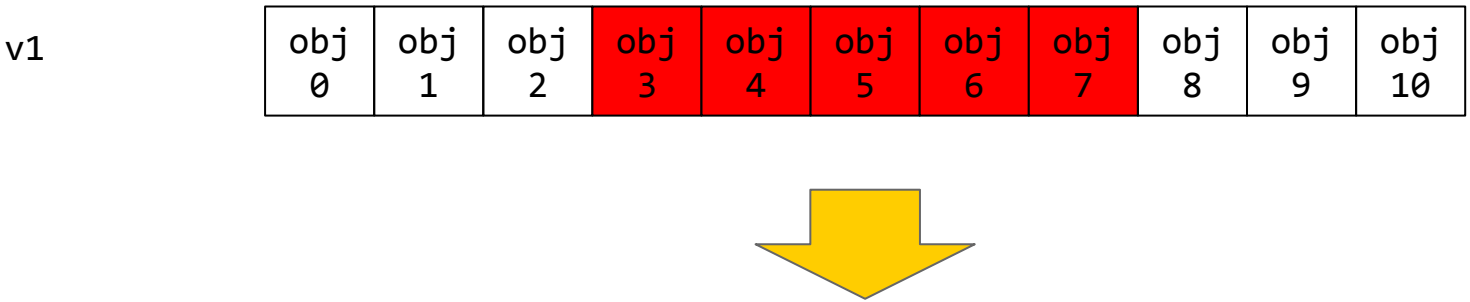
v1



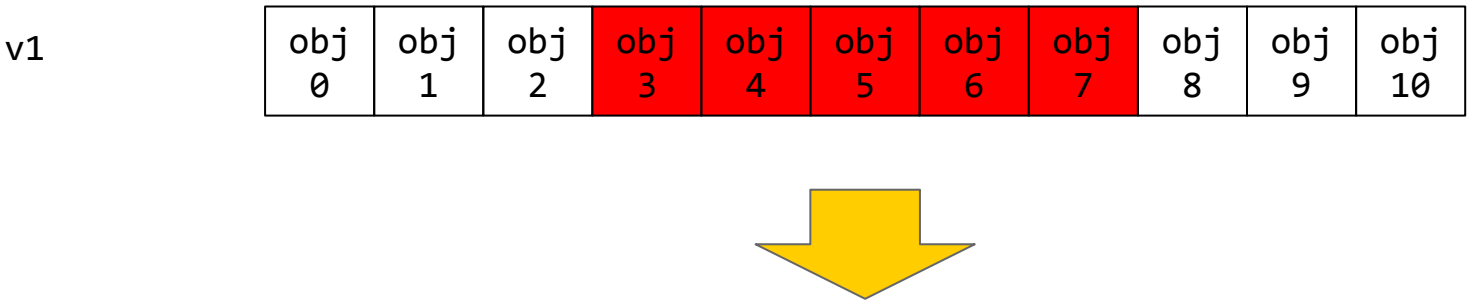
s2



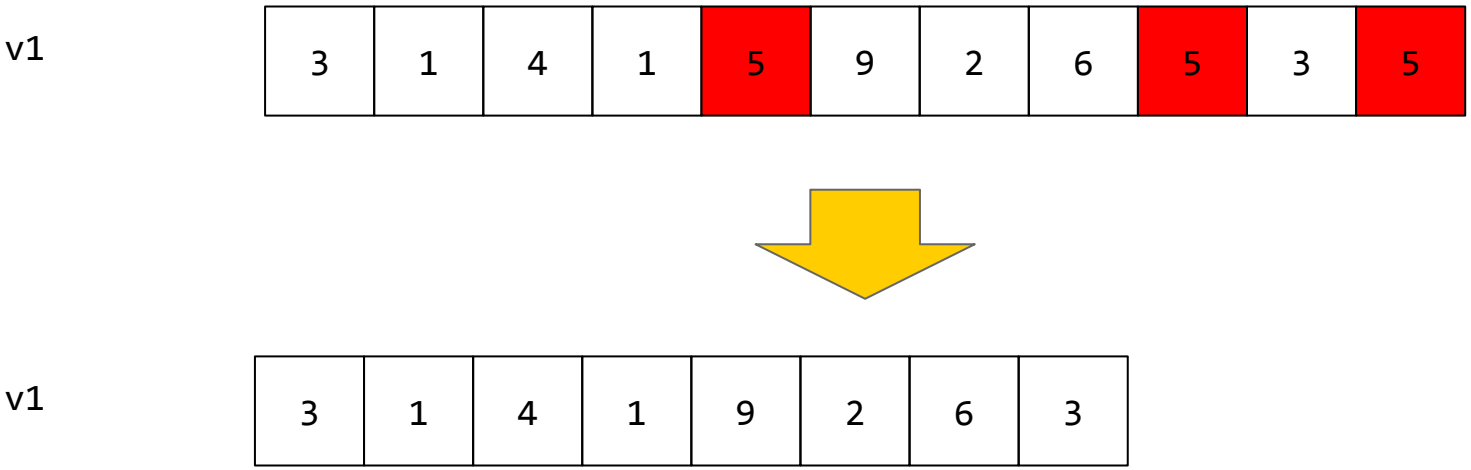

```
std::move( v1.begin()+3, v1.begin()+8,
           std::inserter( s2, s2.begin() ) )
```


v1
obj 0
obj 1
obj 2
obj 3
obj 4
obj 5
obj 6
obj 7
obj 8
obj 9
obj 10
v1
obj 0
obj 1
obj 2
obj 8
obj 9
obj 10

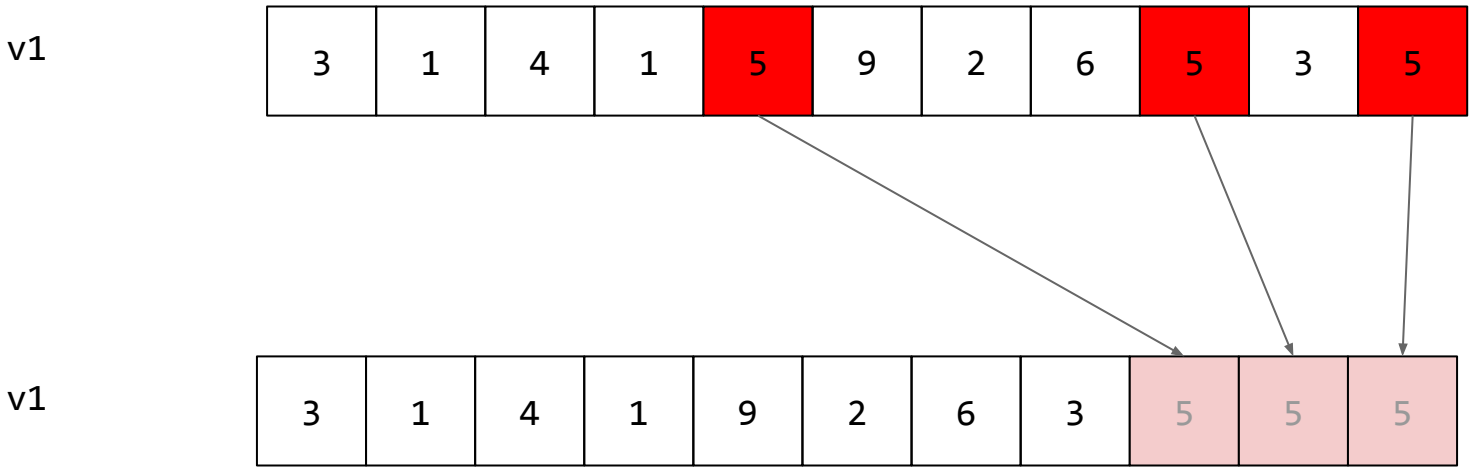
Remove?



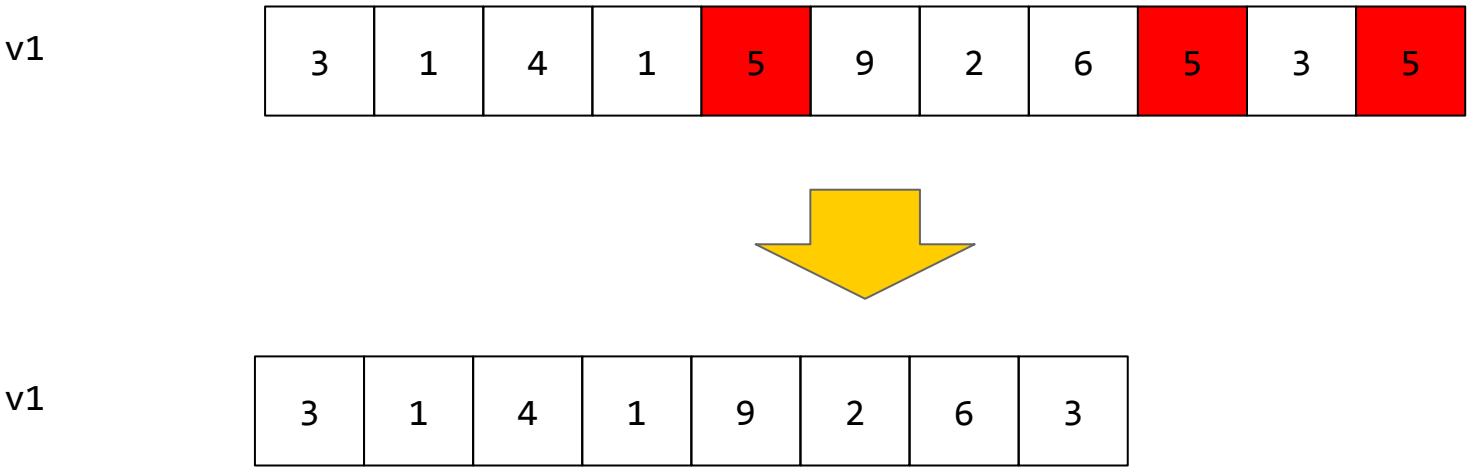
v1.erase(v1.begin()+3, v1.begin()+8)



std::remove



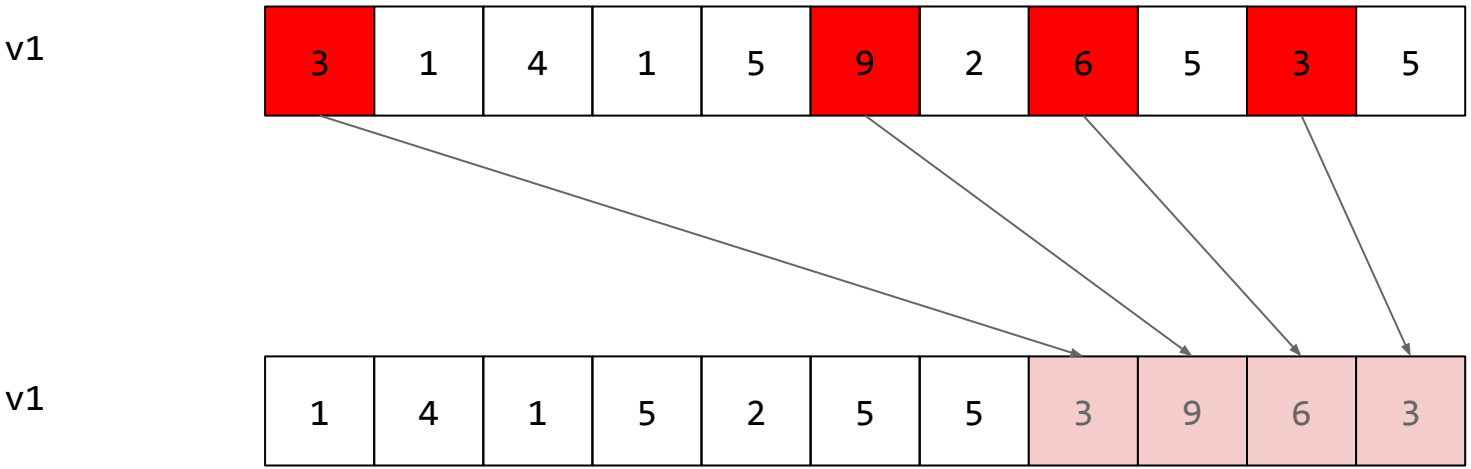
std::remove(v1.begin(), v1.end(), 5)



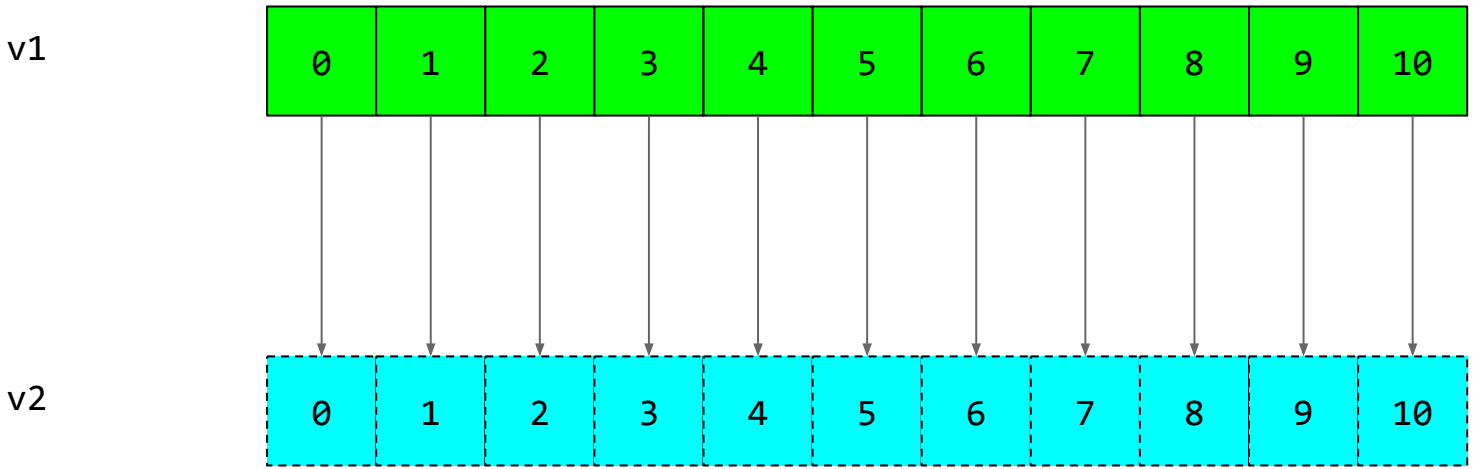

```
v1.erase ( std::remove( v1.begin(), v1.end(), 5 ),
v1.end() )
```

Remove does not erase!

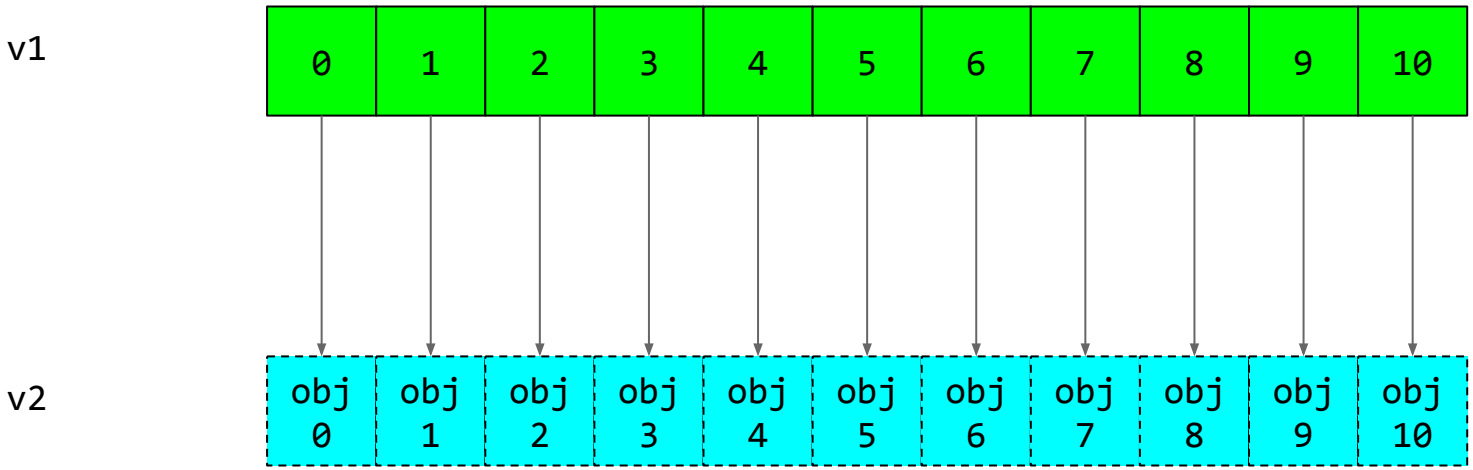
- `std::remove()` only moves matching values to the end, returning the “new” end
- Combine with call to `.erase()` on container
- Cannot work on sorted containers
  - Use `c.erase( key )` instead



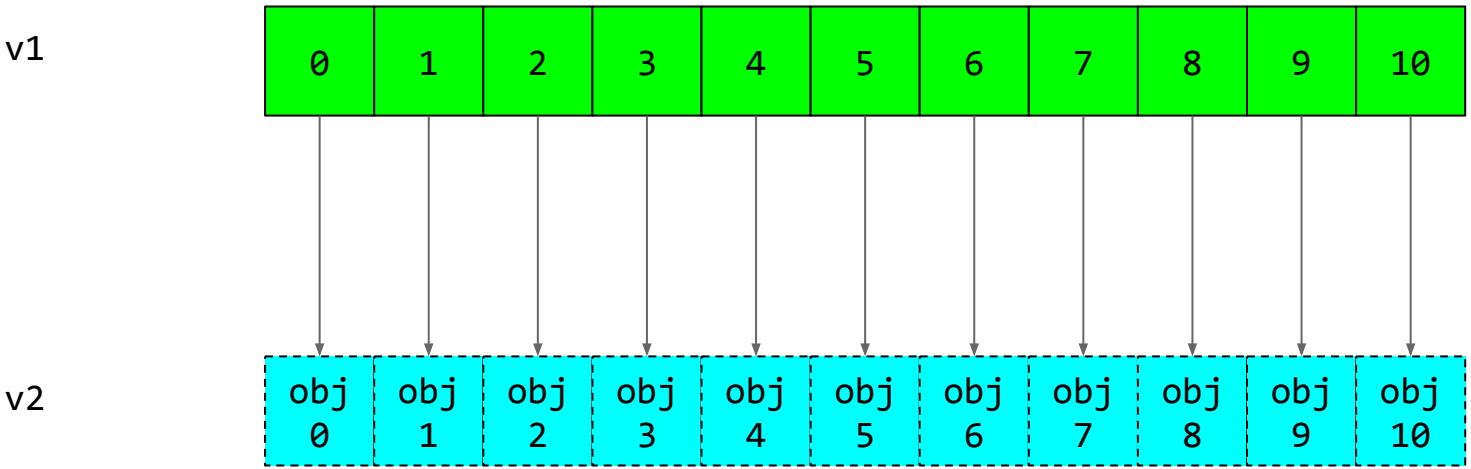

```
std::remove_if( v1.begin(), v1.end(),
                [](int v) { return v % 3 == 0; } )
```




```
std::copy( v1.begin(), v1.end(),
           std::back_inserter( v2 ) )
```

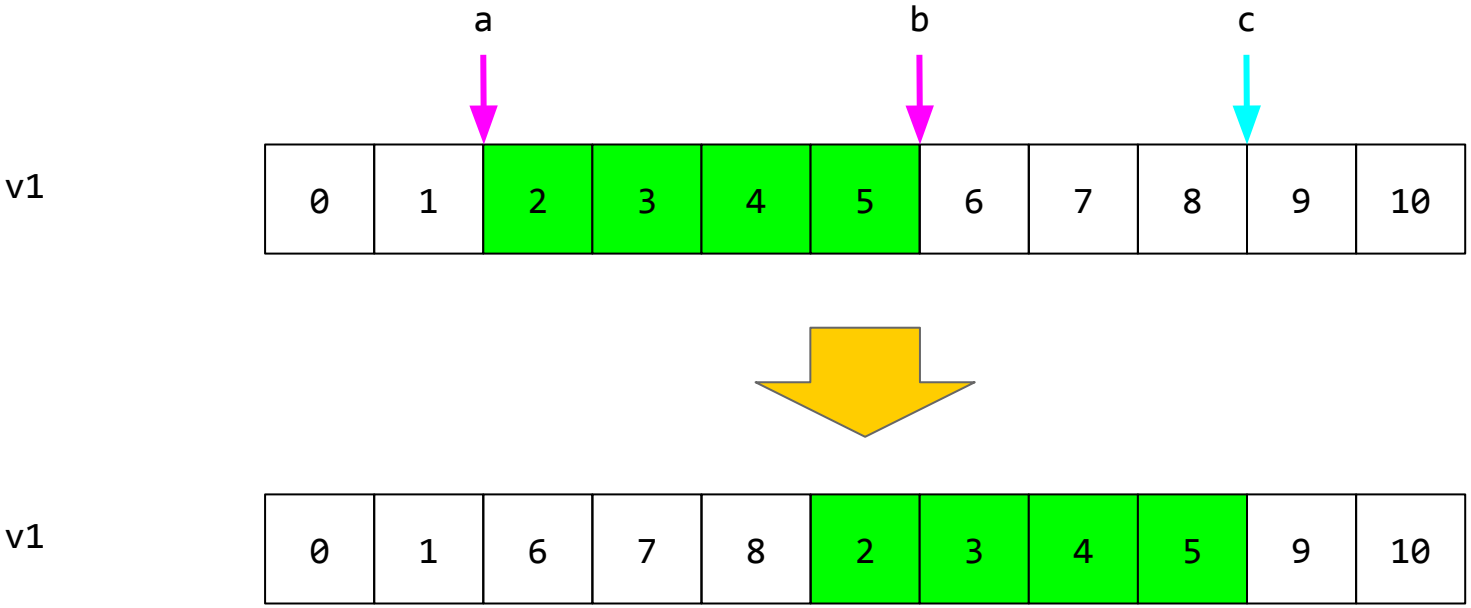

v1
0
1
2
3
4
5
6
7
8
9
10
v2
obj 0
obj 1
obj 2
obj 3
obj 4
obj 5
obj 6
obj 7
obj 8
obj 9
obj 10

f (int) -> Obj



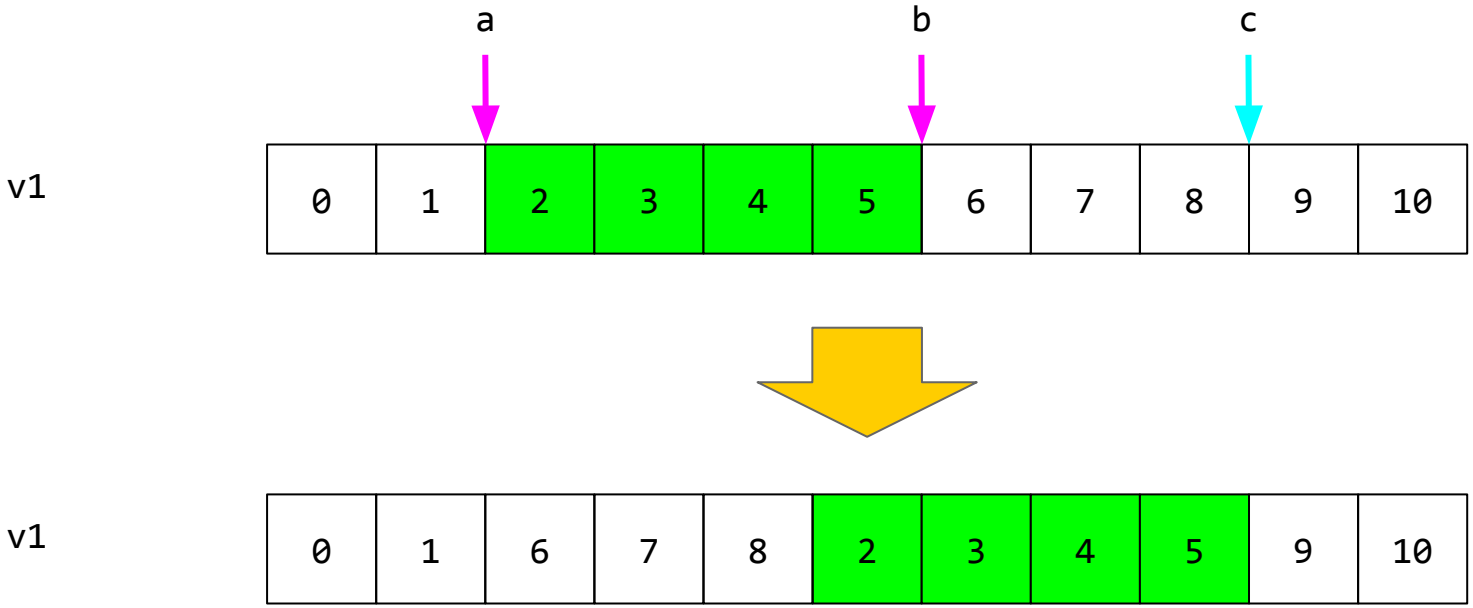

```
std::transform( v1.begin(), v1.end(),
        std::back_insert( v2 ),
    []( int v ) { return Obj( v ); } )
```

Wait, you still haven't explained the multiple selection thing from 42 slides ago!

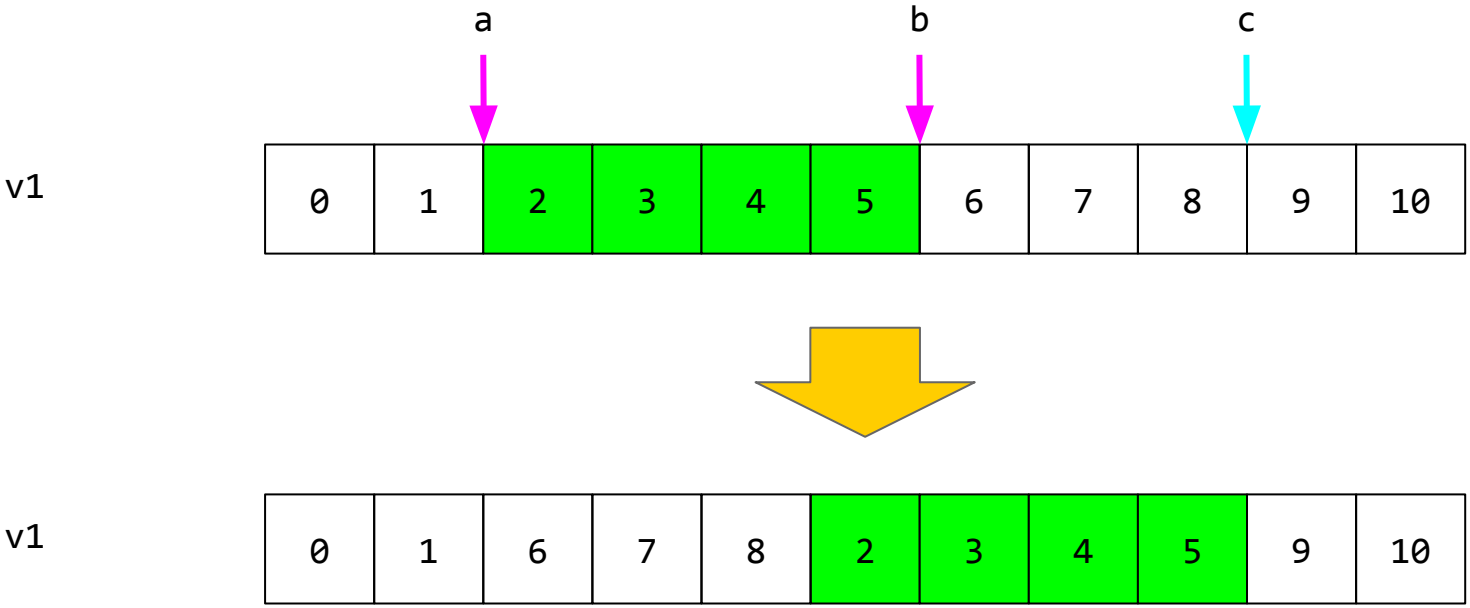


How?!

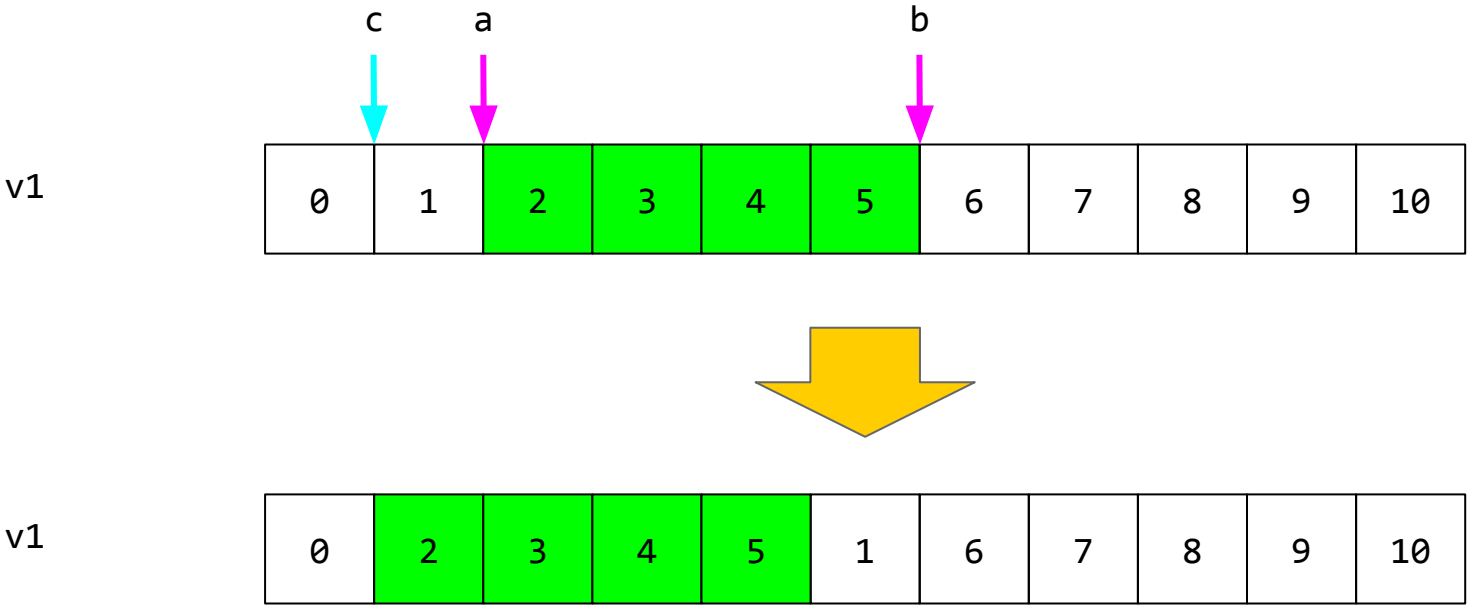

IT'S MAGIC. I AIN'T GOTTA EXPLAIN SHIT.



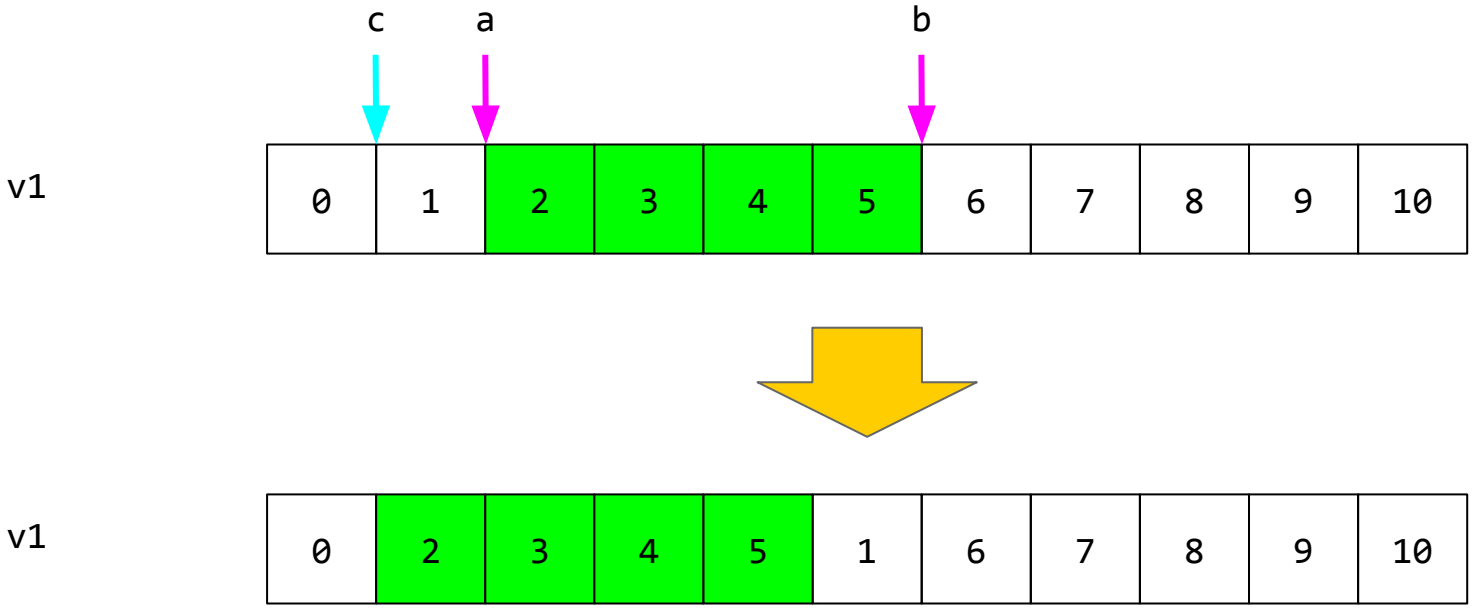
Swap [a, b[and [b,c[



std::rotate(a, b, c)



What if c comes first?



std::rotate(c, a, b)

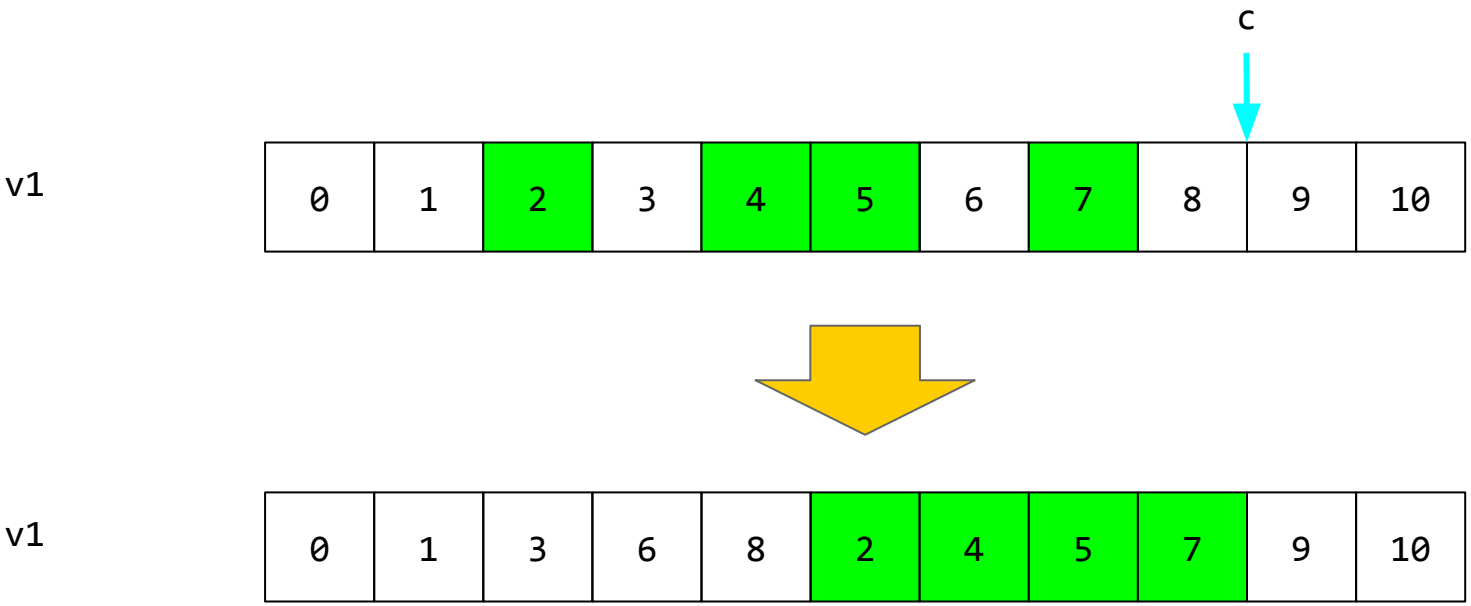
std::rotate()

- Magic!
- Swap ranges [a, b[and [b, c[
- Arguments must come in order from left to right

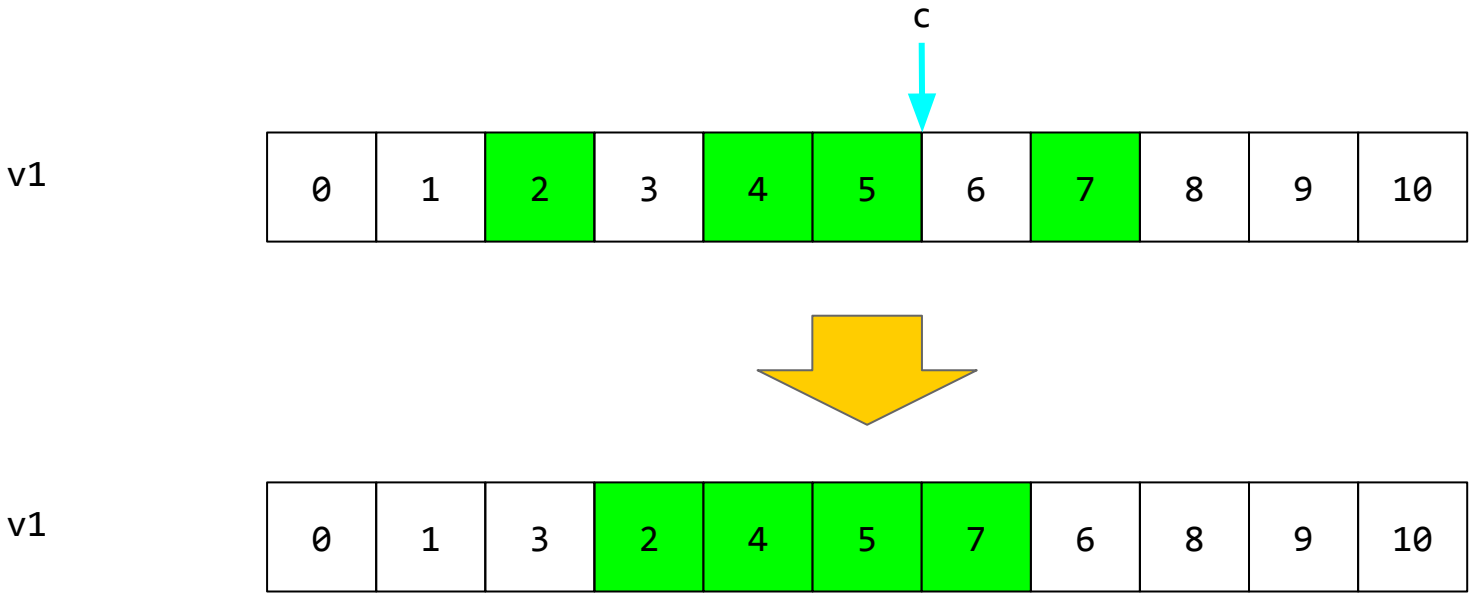
std::rotate()

```
template<typename It>
auto DragDrop( It first, It last, It target )
{
    if ( first < target )
        return { std::rotate( first, last, target),
                 target };
    return { target,
             std::rotate( target, first, last ) };
}
```

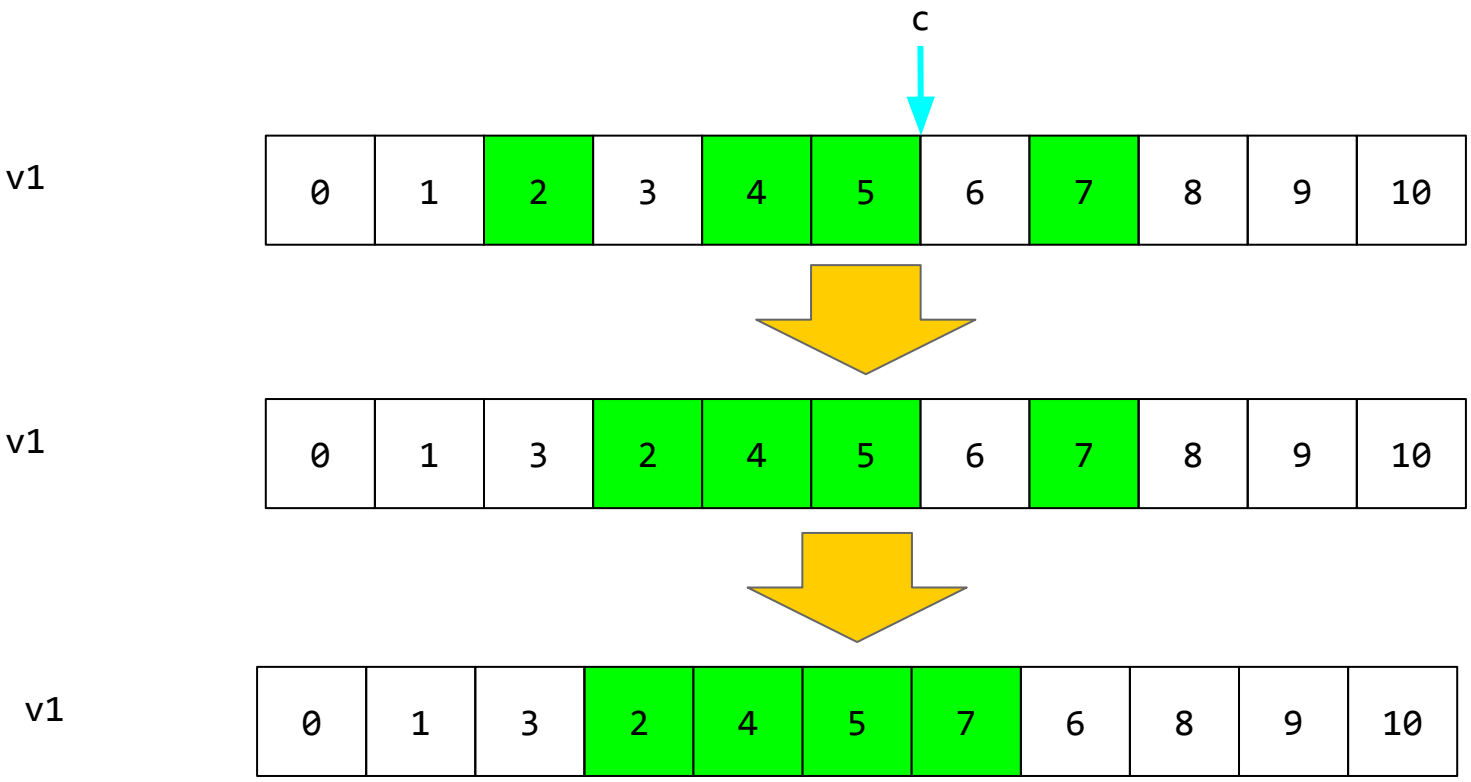
Ok fine, but disjointed selection can't be that easy!



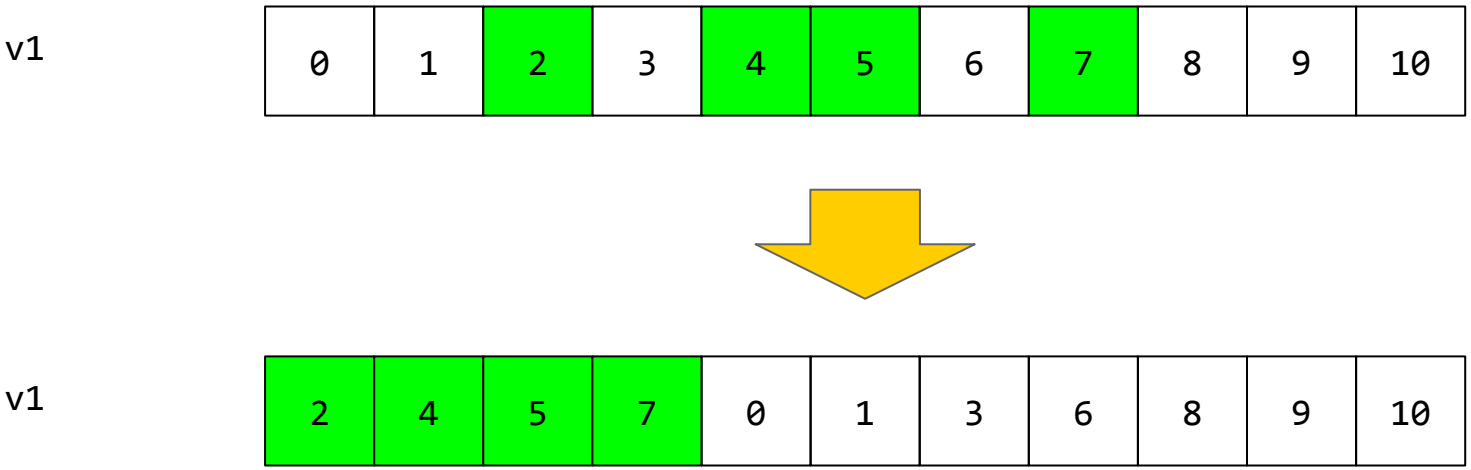
How?!



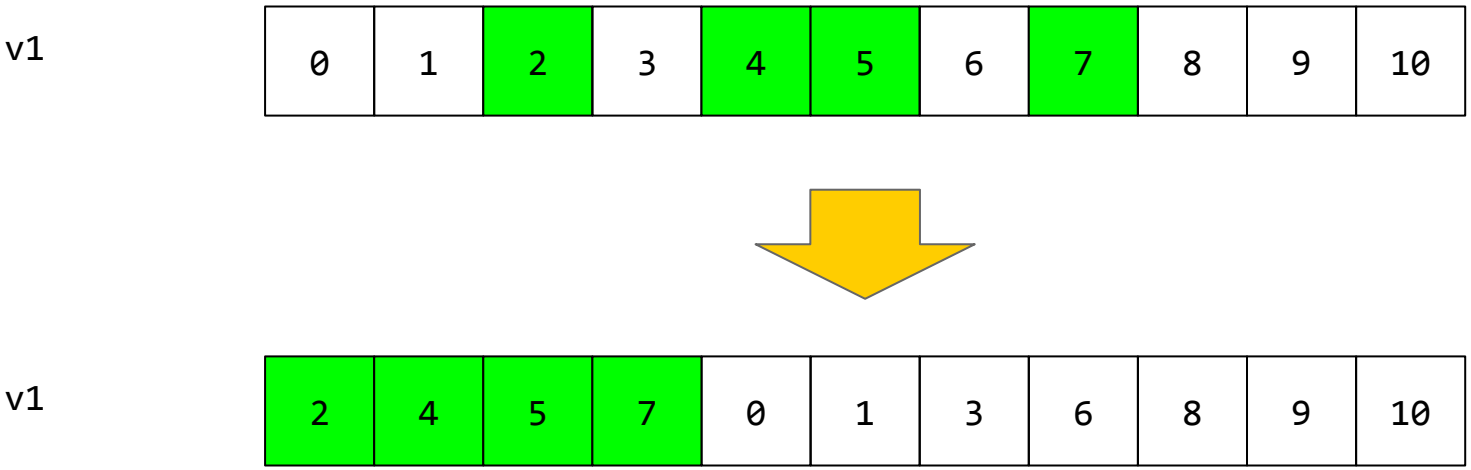
Where is your god now?



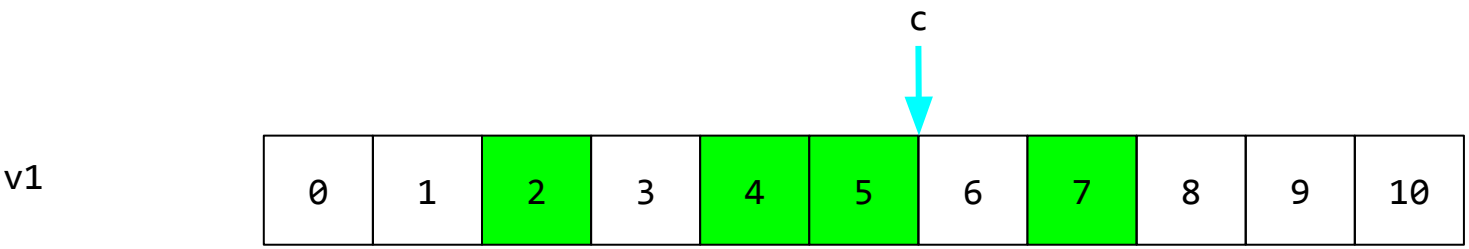
Magic?!

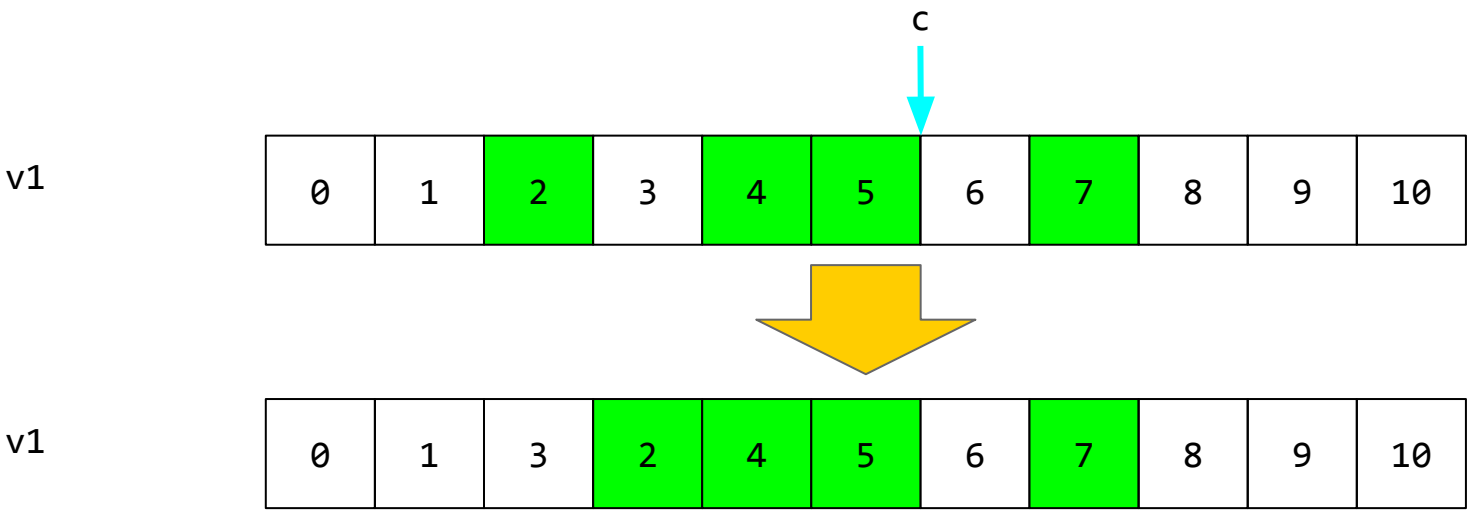


Looks familiar?

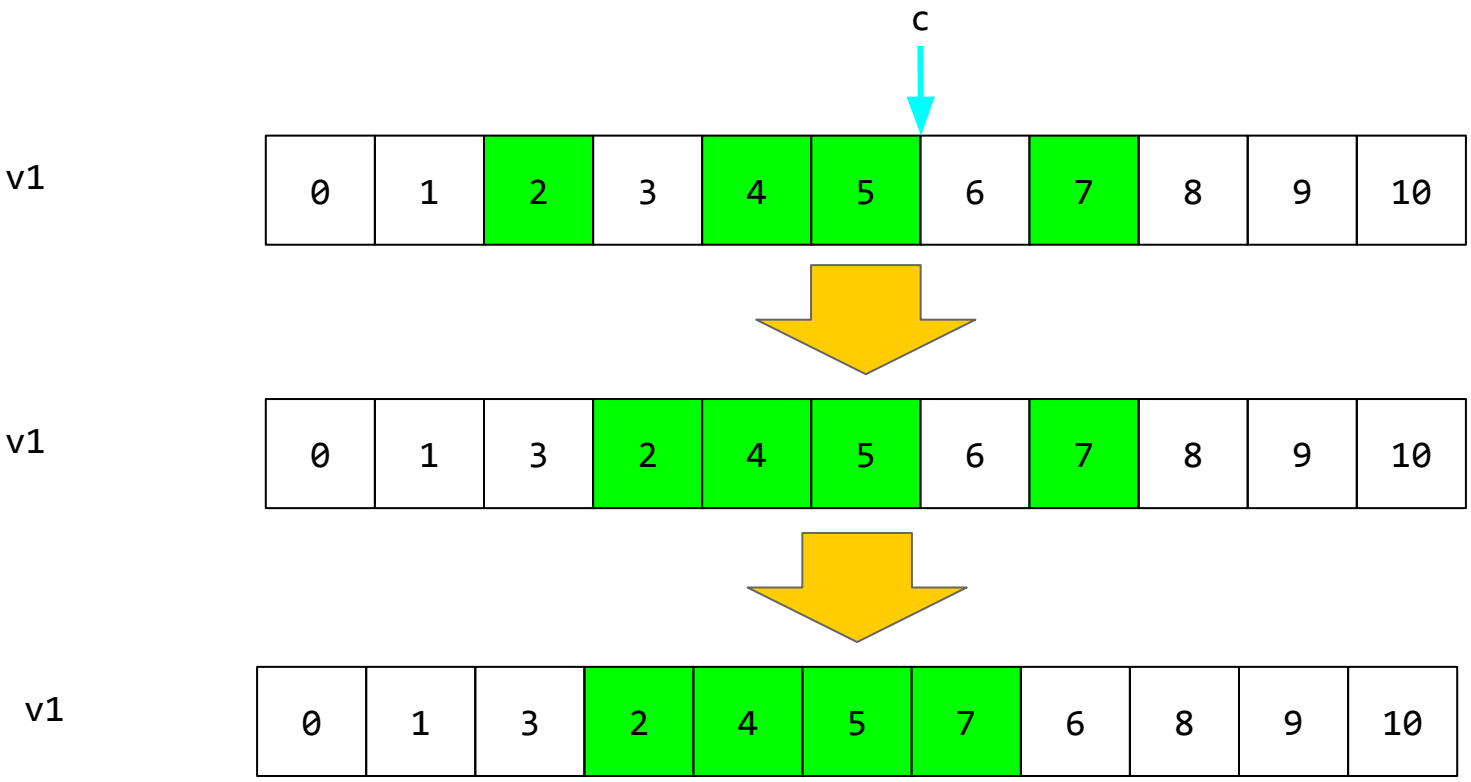


std::stable_partition(v1.begin(), v1.end(),
IsSelected)


c
v1
0
1
2
3
4
5
6
7
8
9
10



std::stable_partition(v1.begin(), c, std::not1(p))



std::stable_partition(c, v1.end(), IsSelected)

std::stable_partition()

```
template<typename It, typename Pr>
auto Gather( It first, It last, It target, Pr p )
{
    return { std::stable_partition( first, target,
                std::not1( p )),
             std::stable_partition( target, last, p )
             };
}
```

Sorted ranges, equivalence and equality

Equality

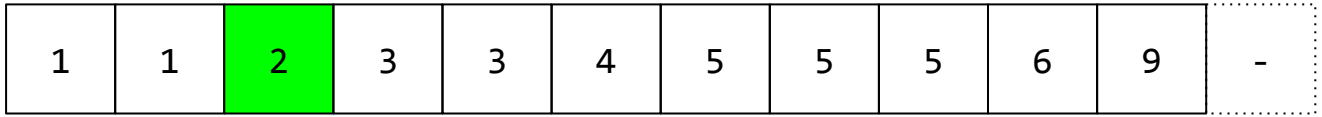
- Values are equal (duh!)
- Comparisons with `operator==`
- Synonymous with algorithms without sorting requirement

Equivalence

- Values are equivalent in terms of ordering
  - `!( a < b ) && !( b < a )`
- Comparisons with `operator<`
- Synonymous with algorithms that only work on sorted ranges

v1

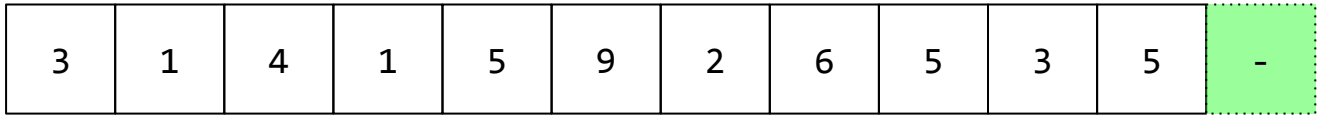
v2



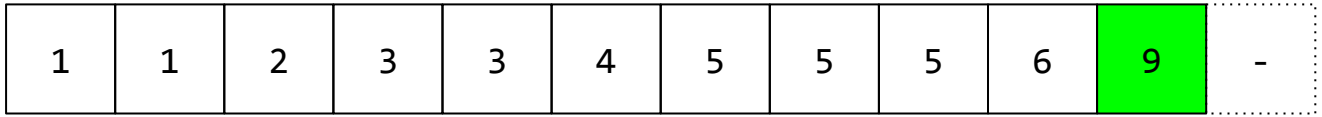
std::find(v1.begin, v1.end(), 2)

std::lower_bound(v2.begin(), v2.end(), 2)

v1



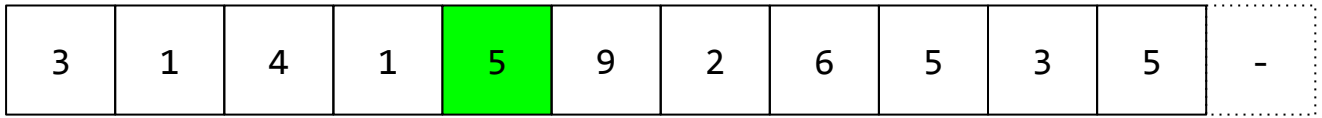
v2



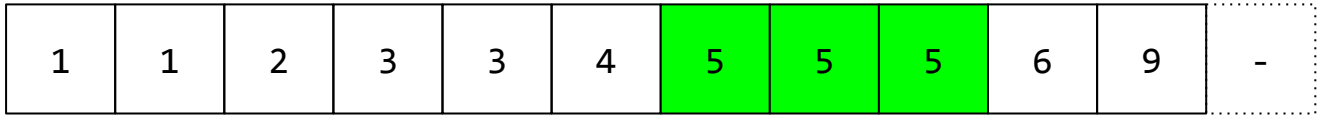
std::find(v1.begin, v1.end(), 7)

std::lower_bound(v2.begin(), v2.end(), 9)

v1



v2

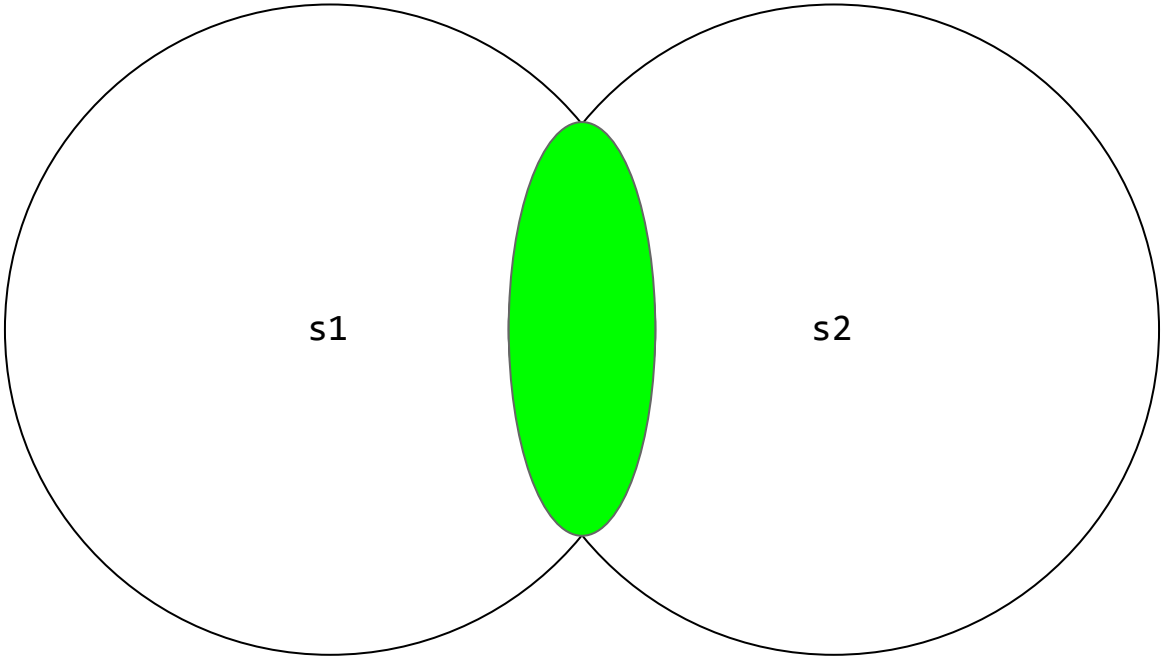


std::find(v1.begin, v1.end(), 5)

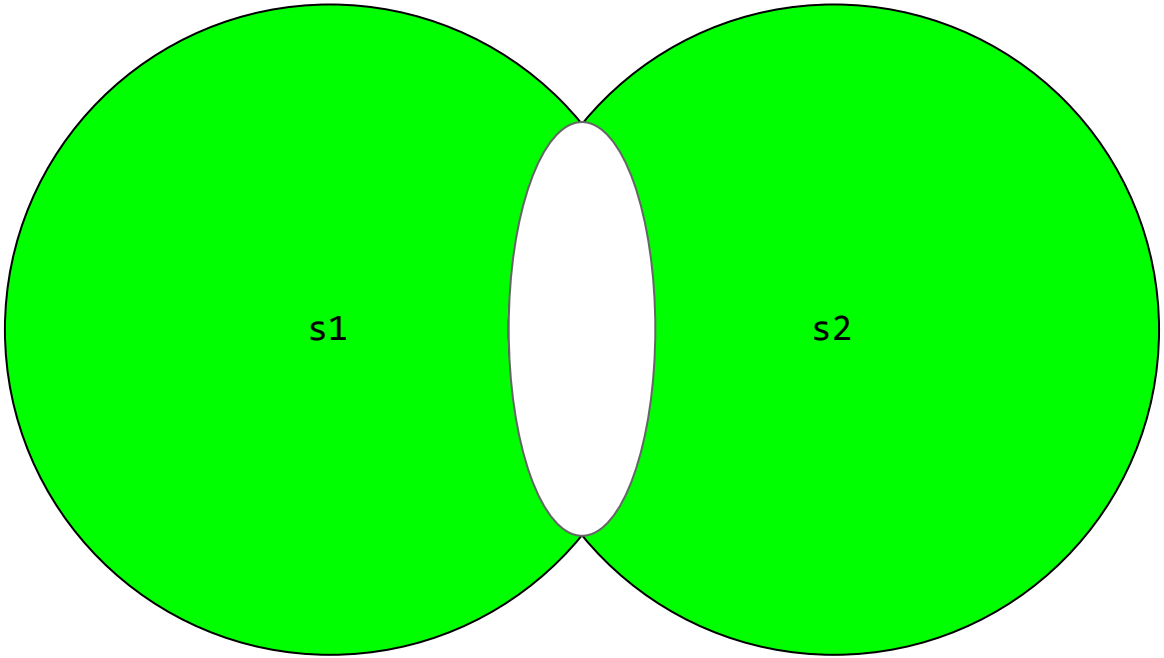
std::equal_range(v2.begin(), v2.end(), 5)

Sorted range benefits

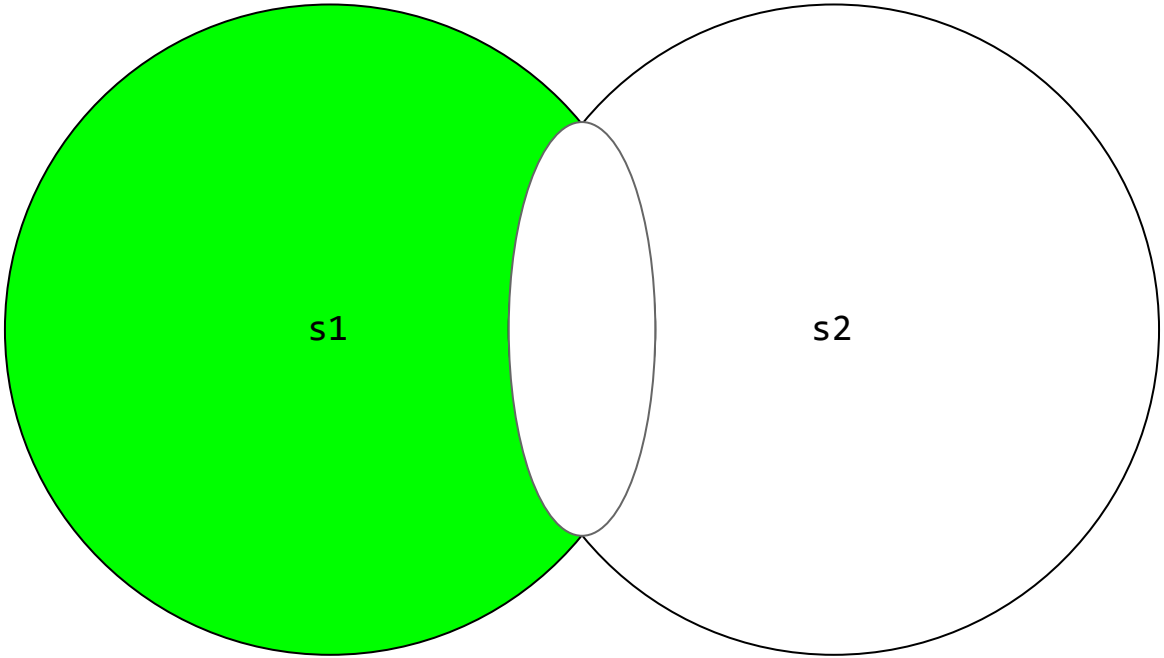
- Offers a selection of logarithmic algorithms
- Available for sorted ranges (from sorted containers or obtained with sort algorithms)
- Search-like algorithms underperform if range is not `RandomAccess`



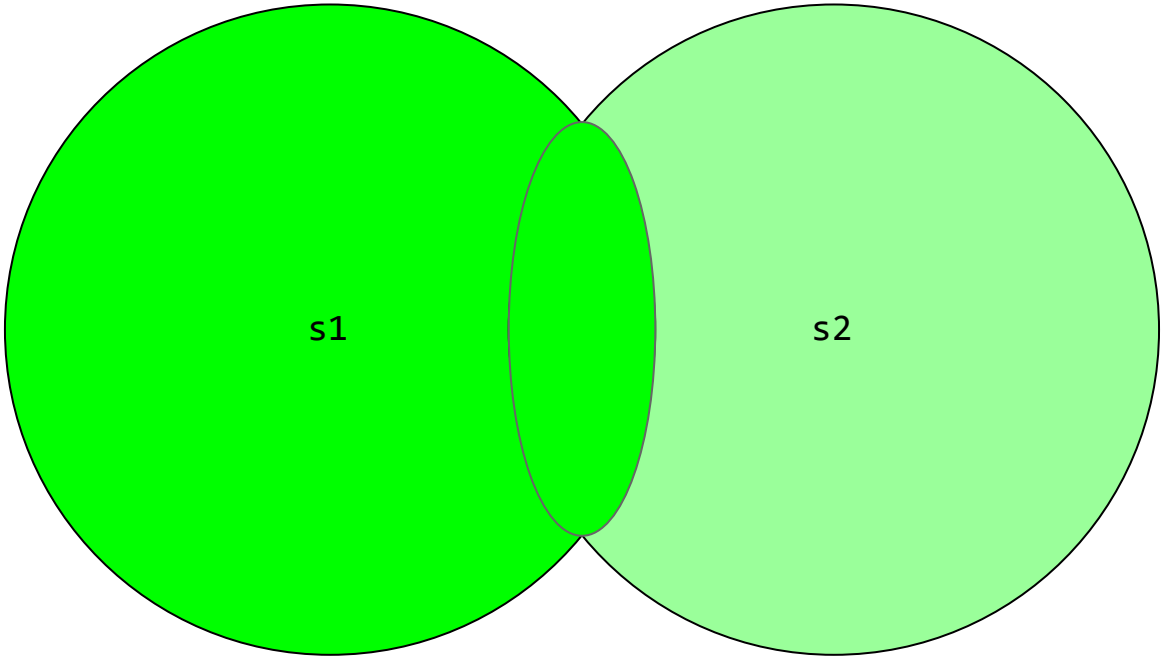
std::set_intersection()



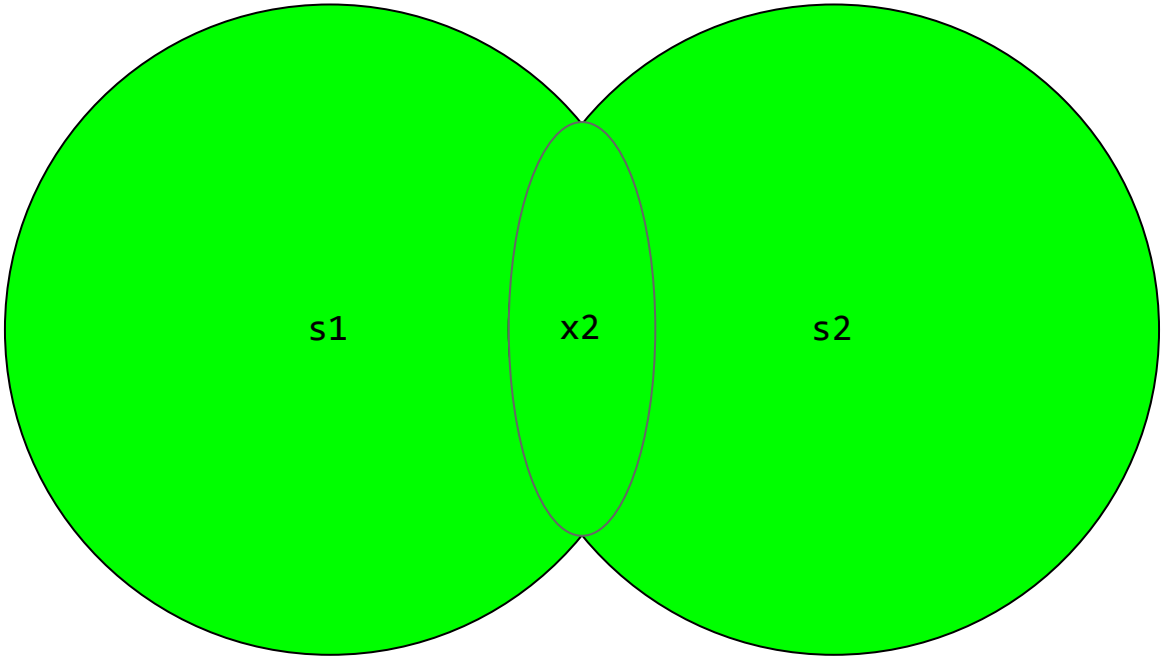
std::set_symmetric_difference()



std::set_difference()



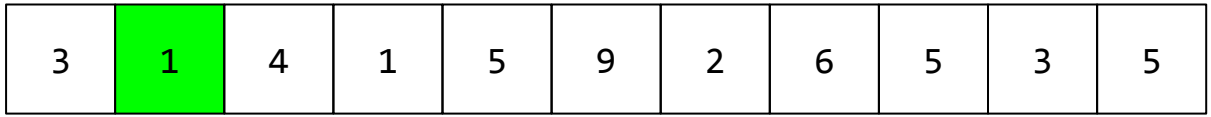
std::set_union()



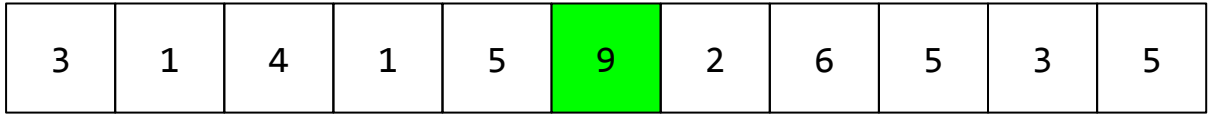
std::merge()

Other non-modifying algorithms

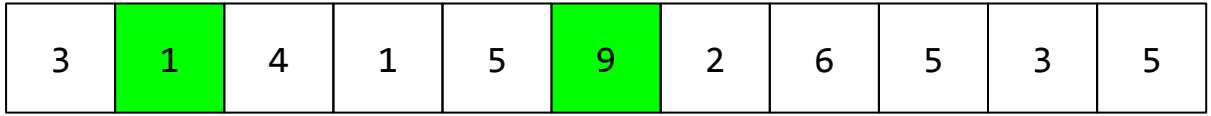
v1



v1



v1




```
std::min_element( v1.begin(), v1.end() )
std::max_element( v1.begin(), v1.end() )
std::minmax_element( v1.begin(), v1.end() )
```

Compare less functions

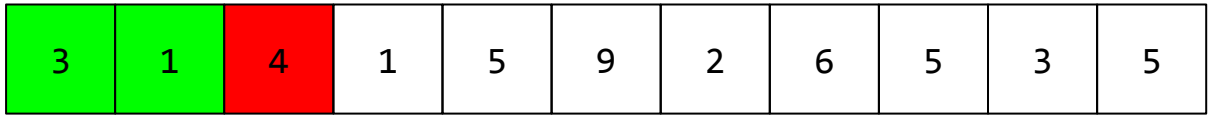
- Can overload default `operator<` with user-defined function

- ```
  bool LessThan(
      const auto& v1,
      const auto& v2 )
  ```

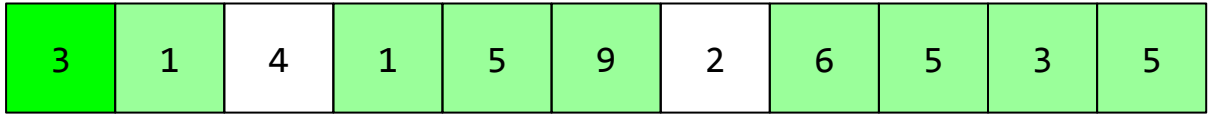
- Must match sort order!

- `binary_search`
- `equal_range`
- `lower_bound`
- `max_element`
- `merge`
- `min_element`
- `minmax_element`
- `set_*`
- `upper_bound`
  ```

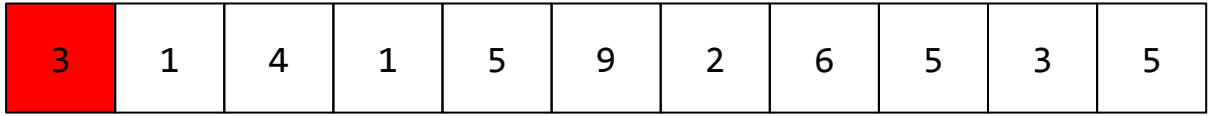
v1



v1



v1




```
std::all_of( v1.begin(), v1.end(), IsOdd )
std::any_of( v1.begin(), v1.end(), IsOdd )
std::none_of( v1.begin(), v1.end(), IsOdd )
```

Predicates

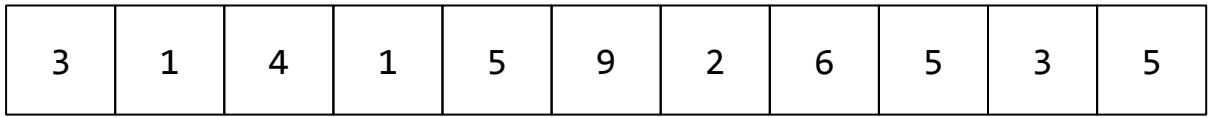
- Test algorithms take a user-defined predicate

```
bool Predicate(
    const auto& v )
```

- Prefer lambdas or inline functions

- `all/any/none_of`
- `copy_if`
- `count_if`
- `find_if`
- `remove_if`
- `replace_if`
- `partition`
- `stable_partition`

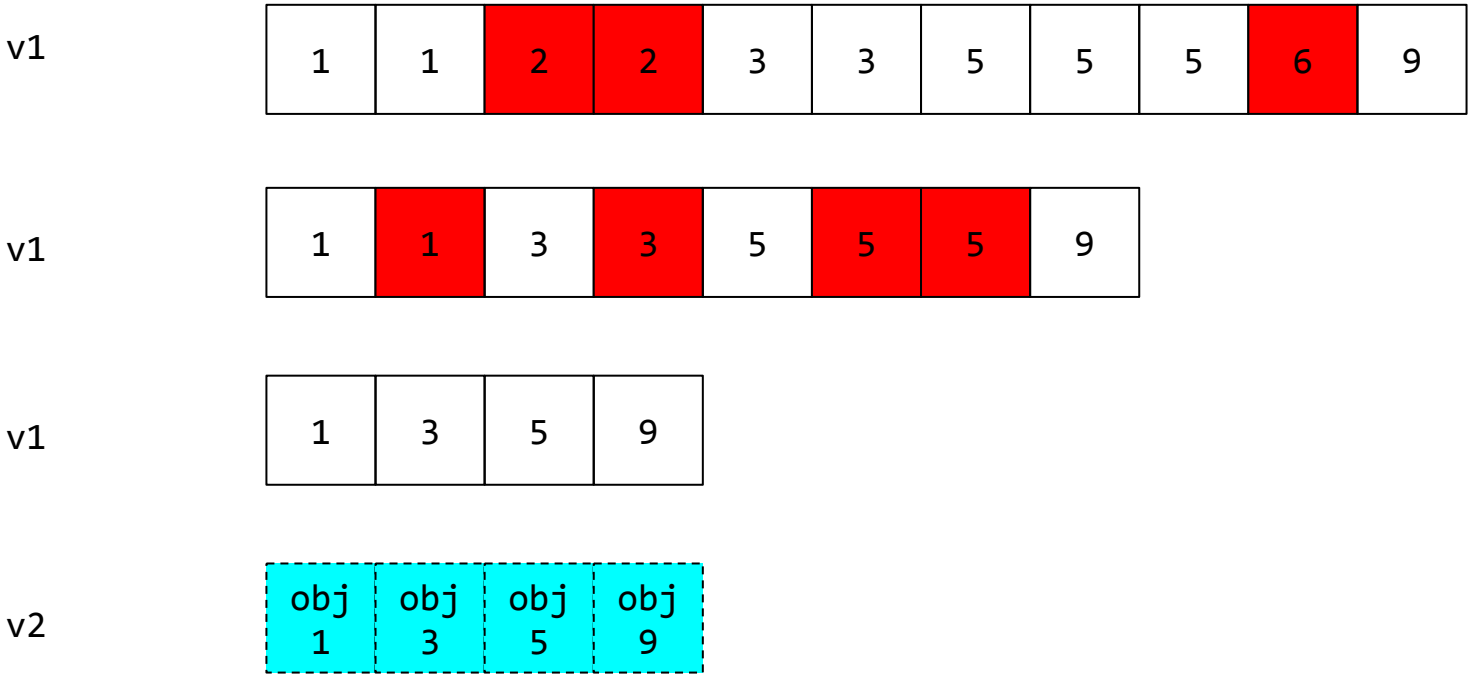
v1



$\Sigma = 44$

std::accumulate(v1.begin(), v1.end(), 0)

What the future holds


v1
1
1
2
2
3
3
5
5
5
6
9
v1
1
1
3
3
5
5
5
9
v1
1
3
5
9
v2
obj 1
obj 3
obj 5
obj 9

v1

```
v1 | ranges::view::filter( IsOdd )
```

```
v1 | ranges::view::filter( IsOdd )
   | unique{}
```

```
v1 | ranges::view::filter( IsOdd )
   | unique{}
   | transform ( ToObject )
```

```
copy(
      v1 | ranges::view::filter( IsOdd )
         | unique{}
         | transform ( ToObject ),
      std::back_inserter( v2 ) );
```

Ranges are super magic

- Ranges: iterators & algorithms on steroids
- Composable to infinity
- Negligible overhead (lazy)

Ranges are super magic

- First part coming in C++20
- Rest probably in C++23
- C++14 proof of concept available in rangev3 library: https://github.com/ericniebler/range-v3


FEDERAL
GALAXY
TOP NEWS
ENLIST
EXIT
MOBILE
INFANTRY
WOULD YOU LIKE TO KNOW MORE?

A few talks

- Sean Parent – Programming Conversations Lecture 5 part 1
- Jonathan Boccara – 105 STL Algorithms in Less Than an Hour
- Eric Niebler – Ranges for Standard Library

Thanks!

Any questions ?

You can reach me at

- mro@puchiko.net
- @MatRopert
- @mropert
- https://mropert.github.io