Recall: Writing a **templated** `min` function

This is a **template**

T gets replaced with a specific type

```
template <typename T>
T min(T a, T b) {
  return a < b ? a : b;
}
```

Recall: explicit instantiation

Template functions cause the compiler to **generate code** for us

```
int min(int a, int b) {                 // Compiler generated
   return a < b ? a : b;                // Compiler generated
}                                       // Compiler generated

double min(double a, double b) {        // Compiler generated
   return a < b ? a : b;                // Compiler generated
}                                       // Compiler generated

min<int>(106, 107);           // Returns 106
min<double>(1.2, 3.4);        // Returns 1.2
```

Recall: Writing a templated `find` function

This find function generalizes across all iterator types!

```
template <typename It, typename T>
It find(It begin, It end, const T& value) {
   for (auto it = begin; it != end; ++it) {
      if (*it == value) return it;
   }
   return end;
}
```

Recall: Writing a templated `find` function

Our `find` function works for other vectors, or even other containers

```
std::vector<std::string> v { "seven", "kingdoms" };
auto it = find(v.begin(), v.end(), "kingdoms");
// It = vector<std::string>::iterator
// T = std::string

std::set<std::string> s { "house", "targaryen" };
auto it = find(s.begin(), s.end(), "targaryen");
// It = std::set<std::string>::iterator
// T = std::string
```

Implicit Instantiation!

Compiler deduces template types by looking at arguments

Wait… why pass in iterators to `find`?

An alternative `find` function

We could have passed the whole container to find. Why not?

```
template <typename Container, typename T>
auto find(const Container& c, const T& value) {
   for (auto it = c.begin(); it != c.end(); ++it) {
      if (*it == value) return it;
   }
   return end;
}

std::vector<std::string> v { "seven", "kingdoms" };
auto it = find(v, "kingdoms");
```

Advantage: Now the caller doesn't have to worry about begin and end!

Container = std::vector<std::string>

T = std::string

An alternative `find` function

Using iterators instead allows us to search *only part* of a container

```
std::vector<int> v { 106, 107, 106, 143, 149, 106 };

// Search for 106, skipping first and last elements
auto it = find(v.begin() + 1, v.end() - 1, 106);

// Get index of iterator using std::distance
std::cout << std::distance(v.begin(), it);
// Prints 2, not 0
```

How can we make `find` even more general!?

- Our `find` searches for the first occurrence of `value` in a container
- What if we wanted to find the first occurrence of:
  - A vowel in a `string`?
  - A prime number in a `vector<int>`?
  - A number divisible by 5 in a `set<int>`?

Definition: A predicate is a `boolean`-valued function

Predicate Examples

Unary

```
bool isVowel(char c) {
    c = toupper(c);
    return c == 'A' || c == 'E' ||
           c == 'I' || c == 'O' ||
           c == 'U';
}

bool isPrime(size_t n) {
    if (n < 2) return false;
    for (auto i = 3; i<=sqrt(n); i++)
        if (n % i == 0) return false;
    return true;
}
```

Binary

```
bool isLessThan(int x, int y) {
    return x < y;
}

bool isDivisible(int n, int d)
{
    return n % d == 0;
}
```

Using predicates

- How can we use `isVowel` to find the first vowel in a `string`?
- Or `isPrime` to find a prime number in a `vector<int>`?
- Or `isDivisible` to find a number divisible by 5?

Key Idea: We need to pass a predicate to a function

Modifying our find function

```
template <typename It>
It find(It first, It last, ???? pred) {
  for (auto it = first; it != last; ++it) {
    if (*it == value) return it;
  }
  return last;
}
```

What if we could instead pass a predicate to this function as a parameter?

Then we could replace this critical section of the code with a call to our predicate.

Modifying our `find` function

```
template <typename It>
It find(It first, It last, ???? pred) {
  for (auto it = first; it != last; ++it) {
    if (pred(*it)) return it;
  }
  return last;
}
```

What if we could instead pass a predicate to this function as a parameter?

Then we could replace this critical section of the code with a call to our predicate… like so!

Answer: Templates plus predicates

```
template <typename It, typename Pred>
It find(It first, It last, Pred pred) {
   for (auto it = first; it != last; ++it) {
      if (pred(*it)) return it;
   }
   return last;
}
```

Pred: the type of our predicate.

Compiler will figure this out for us using implicit instantiation!

pred: our predicate, passed as a parameter

Hey look! We're calling our predicate on each element. As soon as we find one that matches, we return

Answer: Templates plus predicates

```
template <typename It, typename Pred>
It find_if(It first, It last, Pred pred) {
  for (auto it = first; it != last; ++it) {
    if (pred(*it)) return it;
  }
  return last;
}
```

Pred: the type of our predicate.

Compiler will figure this out for us using implicit instantiation!

pred: our predicate, passed as a parameter

Hey look! We're calling our predicate on each element. As soon as we find one that matches, we return

Let's give this function a new name so it doesn't get confused with old one!

Using our find_if function

```
bool isVowel(char c) {
   c = ::toupper(c);
   return c == 'A' || c == 'E' || c == 'I' ||
          c == 'O' || c == 'U';
}


std::string corlys = "Lord of the Tides";
auto it = find_if(corlys.begin(), corlys.end(), isVowel);
*it = '0'; // "L0rd of the Tides"
```

You: "What type is this?"

Compiler: "Don't worry about it!"

Using our `find_if` function

```
bool isPrime(size_t n) {
   if (n < 2) return false;
   for (size_t i = 3; i <= std::sqrt(n); i+
      if (n % i == 0) return false;
   return true;
}

std::vector<int> ints = {1, 0, 6};
auto it = find_if(ints.begin(), ints.end(), isPrime);
assert(it == ints.end());
```

You: ”What type is this!!?”

Compiler: “I gottttchuuu man”

Passing functions allows us to generalize an algorithm with user-defined behaviour

Pred is a function pointer

```
find_if(corlys.begin(), corlys.end(), isVowel);
// Pred = bool(*)(char)

find_if(ints.begin(), ints.end(), isPrime);
// Pred = bool(*)(int)
```

My function returns a bool

I'm a function pointer

And I take in a single int as a parameter

As we'll see shortly, a function pointer is ***just one*** of the things we can pass to `find_if`

Function pointers generalize poorly

Consider that we want to find a number less than **N** in a vector

```
bool lessThan5(int x) { return x < 5; }
bool lessThan6(int x) { return x < 6; }
bool lessThan7(int x) { return x < 7; }

find_if(begin, end, lessThan5);
find_if(begin, end, lessThan6);
find_if(begin, end, lessThan7);
```

Function pointers generalize poorly

What if we want to find a number less than N, but we don't know what N is until runtime?

```
int n;
std::cin >> n;
find_if(begin, end, /* lessThan... Haelpp... */)
```

We can't just add another parameter

Turn to someone next to you and talk about why this wouldn't work!

```
bool isLessThan(int elem, int n) {
  return elem < n;
}
```

We can't add another parameter to pred!

```
template <typename It, typename Pred>
It find_if(It first, It last, Pred pred) {
  for (auto it = first; it != last; ++it) {
    if (pred(*it)) return it;
  }
  return last;
}
```

We only pass one parameter to **pred** here!

Introducing... lambda functions

Lambda functions are functions that capture state from an enclosing scope

```
int n;
std::cin >> n;

auto lessThanN = [n](int x) { return x < n; };

find_if(begin, end, lessThanN); // 😎 😎
```

Lambda Syntax

I don't know the type! But the compiler does.

Capture clause lets us use outside variables

Parameters

Function parameters, exactly like a normal function

```
auto lessThanN = [n](int x) {
    return x < n;
};
```

Function body

Exactly as a normal function, except only parameters and captures are in-scope

A note on captures

```
auto lambda = [capture-values](arguments) {
   return expression;
}

[x](arguments)      // captures x by value (makes a copy)
[x&](arguments)     // captures x by reference
[x, y](arguments)   // captures x, y by value
[&](arguments)      // captures everything by reference
[&, x](arguments)   // captures everything except x by reference
[=](arguments)      // captures everything by value
```

We don’t have to use captures!

Lambdas are good for making functions on the fly

```
std::string corlys = "Lord of the tides";
auto it = find_if(corlys.begin(), corlys.end(),
   [](auto c) {
     c = toupper(c);
     return c == 'A' || c == 'E' ||
            c == 'I' || c == 'O' || c == 'U’;
  });
```

How do lambdas work?

Recall: 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?

Definition: A functor is any object that defines an `operator()`

In English: an object that acts like a function

An example of a functor: `std::greater<T>`

```cpp
template <typename T>
struct std::greater {
  bool operator()(const T& a, const T& b) const {
    return a > b;
  }
};

std::greater<int> g;
g(1, 2); // false
```

Hmm.. Seems like a function

🦆🦆

Another STL functor: `std::hash<T>`

```
template <>
struct std::hash<MyType> {
  size_t operator()(const MyType& v) const {
    // Crazy, theoretically rigorous hash function
    // approved by 7 PhDs and Donald Knuth goes here
    return ...;
  }
};

MyType m;
std::hash<MyType> hash_fn;
hash_fn(m); // 125123201 (for example)
```

Aside: This syntax is called a *template specialization* for type MyType

Hint hint: This is also *one* of the ways to create a hash function for a custom type

Since a functor is an object, it can have state

Functors can have state!

```cpp
struct my_functor {
    bool operator()(int a) const {
        return a * value;
    }

    int value;
};

my_functor f;
f.value = 5;
f(10); // 50
```

Oooh such state

🥺 🥺

When you use a lambda, a functor type is generated

This code...

```
int n = 10;
auto lessThanN = [n](int x) { return x < n; };
find_if(begin, end, lessThanN);
```

...is equivalent to this code!

```
class __lambda_6_18
{
public:
    bool operator()(int x) const { return x < n; }
    __lambda_6_18(int& _n) : n{_n} {}
private:
    int n;
};

int n = 10;
auto lessThanN = __lambda_6_18{ n };
find_if(begin, end, lessThanN);
```

Random name that only the compiler will see!

Recall: functor call operator

Class constructor

Our captures became fields in the class!

Capturing variable n from outer scope by passing to constructor

If you are curious about this stuff, check out https://cppinsights.io/!

You've seen this kind of thing before…

```cpp
std::vector<int> v {1,2,3};
for (const int& e : v)
{
    // ...
}
```

```cpp
auto begin = v.begin();
auto end = v.end();
for (auto it = begin; it != end; ++it)
{
    // ...
}
```

It’s the same ordeal! Syntactic sugar

```
int n = 10;
auto lessThanN = [n](int x)
{ return x < n; };
find_if(begin, end, lessThanN);
```

```
class __lambda_6_18
{
public:
    bool operator()(int x) const
    { return x < n; }
    __lambda_6_18(int& _n) : n{_n}
{}
private:
    int n;
};

int n = 10;
auto lessThanN = __lambda_6_18{n};
find_if(begin, end, lessThanN);
```

Functions & Lambdas Recap

- Use functions/lambdas to pass around **behaviour** as variables
- Aside: `std::function` is an overarching type for functions/lambdas
  - Any functor/lambda/function pointer can be cast to it
  - It is a bit slower
  - I usually use auto/templates and don't worry about the types!

```
std::function<bool(int, int)> less = std::less<int>{};
std::function<bool(char)> vowel = isVowel;
std::function<int(int)> twice = [](int x) { return x * 2; };
```

Recall: 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?

Huh... that looks familiar

std::find, std::find_if, std::find_if_not

Defined in header `<algorithm>`

```
template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T& value );
```
(constexpr since C++20) (until C++26)

```
template< class InputIt, class T = typename std::iterator_traits
                                      <InputIt>::value_type >
constexpr InputIt find( InputIt first, InputIt last, const T& value );
```
(1) (since C++26)

```
template< class ExecutionPolicy, class ForwardIt, class T >
ForwardIt find( ExecutionPolicy&& policy,
                ForwardIt first, ForwardIt last, const T& value );
```
(since C++17) (until C++26)

```
template< class ExecutionPolicy,
          class ForwardIt, class T = typename std::iterator_traits
                                        <ForwardIt>::value_type >
ForwardIt find( ExecutionPolicy&& policy,
                ForwardIt first, ForwardIt last, const T& value );
```
(2) (since C++26)

```
template< class InputIt, class UnaryPred >
InputIt find_if( InputIt first, InputIt last, UnaryPred p );
```
(3) (constexpr since C++20)

```
template< class ExecutionPolicy, class ForwardIt, class UnaryPred >
ForwardIt find_if( ExecutionPolicy&& policy,
                   ForwardIt first, ForwardIt last, UnaryPred p );
```
(4) (since C++17)

<algorithm> is a collection of template functions

```
std::count_if(InputIt first, InputIt last, UnaryPred p);
```

How many elements in [first, last] match predicate p?

```
std::sort(RandomIt first, RandomIt last, Compare comp);
```

Sorts the elements in [first, last) according to comparison comp

```
std::max_element(ForwardIt first, ForwardIt last, Compare comp);
```

Finds the maximum element in [first, last] according to comparison comp

<algorithm> functions operate on iterators

```
std::copy_if(InputIt r1, InputIt r2, OutputIt o, UnaryPred p);
```

Copy the only elements in [r1, r2) into o which meet predicate p

```
std::transform(ForwardIt1 r1, ForwardIt1 r2, ForwardIt2 o, UnaryOp op);
```

Apply op to each element in [r1, r2), writing a new sequence into o

```
std::unique_copy(InputIt i1, InputIt i2, OutputIt o, BinaryPred p);
```

Remove consecutive duplicates from [r1, r2), writing new sequence into o

There are a lot of algorithms...

all_of	copy	merge	random_shuffle	is_sorted
any_of	copy_n	inplace_merge	shuffle	is_sorted_until
none_of	copy_if	includes	push_heap	nth_element
for_each	copy_backward	set_union	pop_heap	min
find	move	set_intersection	make_heap	max
find_if	move_backward	set_difference	sort_heap	minmax
find_if_not	swap	set_symmetric_difference	is_heap	min_element
find_end	swap_ranges	remove	is_heap_until	max_element
find_first_of	iter_swap	remove_if	is_partitioned	minmax_element
adjacent_find	transform	remove_copy	partition	lexicographical_compare
count	replace	remove_copy_if	stable_partition	next_permutation
count_if	replace_if	unique	partition_copy	prev_permutation
mismatch	replace_copy	unique_copy	partition_point	
equal	replace_copy_if	reverse	sort	
is_permutation	fill	reverse_copy	stable_sort	
search	fill_n	rotate	partial_sort	
search_n	generate	rotate_copy	partial_sort_copy	

Things you can do with the STL

binary search • heap building • min/max
lexicographical comparisons • merge • set union
• set difference • set intersection • partition • sort
*n*th sorted element • shuffle • selective removal •
selective copy • for-each • random sample

all in their most general form!

Ranges and Views

Ranges are a new version of the STL

Definition: A range is anything with a begin and end

What’s a range?

std::vector<T>

std::unordered_set<K,V>

Your own custom type with a begin and end!

std::map<K, V>

std::set<K>

Recall: why did we pass iterators to `find`?

It allows us to find in a subrange! But most of the time, we don't need to.

```
int main() {
  std::vector<char> v = {'a', 'b', 'c', 'd', 'e'};
  auto it = std::find(v.begin(), v.end(), 'c');
}
```

Do we really care about iterators here? I just wanted to search the entire container!

Range algorithms operate on ranges

STD ranges provides new versions of `<algorithm>` for ranges

```
int main() {
  std::vector<char> v = {'a', 'b', 'c', 'd', 'e’};
  auto it = std::ranges::find(v, 'c');
}
```

Look! I can pass v here because it is a range!

Range algorithms operate on ranges

We can still work with iterators if we need to

```
int main() {
   std::vector<char> v = {'a', 'b', 'c', 'd', 'e'};

   // Search from 'b' to 'd'
   auto first = v.begin() + 1;
   auto last = v.end() - 1;
   auto it = std::ranges::find(first, last, 'c');
}
```

Ranges: The STL v2

- There are range equivalents of most of the STL `<algorithm>` library
- These are very new! C++20/23/26 and beyond!

```
ranges::find_last           (C++23)
ranges::find_last_if        (C++23)
ranges::find_last_if_not    (C++23)
ranges::find_end            (C++20)
ranges::find_first_of       (C++20)
ranges::adjacent_find       (C++20)
ranges::search              (C++20)
ranges::search_n            (C++20)
ranges::contains            (C++23)
ranges::contains_subrange   (C++23)
ranges::starts_with         (C++23)
ranges::ends_with           (C++23)
```

```
ranges::copy                (C++20)
ranges::copy_if             (C++20)
ranges::copy_n              (C++20)
ranges::copy_backward       (C++20)
ranges::move                (C++20)
ranges::move_backward       (C++20)
ranges::fill                (C++20)
ranges::fill_n              (C++20)
ranges::transform           (C++20)
ranges::generate            (C++20)
ranges::generate_n          (C++20)
```

```
ranges::remove              (C++20)
ranges::remove_if           (C++20)
ranges::remove_copy         (C++20)
ranges::remove_copy_if      (C++20)
ranges::replace             (C++20)
ranges::replace_if          (C++20)
ranges::replace_copy        (C++20)
ranges::replace_copy_if     (C++20)
ranges::swap_ranges         (C++20)
ranges::reverse             (C++20)
ranges::reverse_copy        (C++20)
ranges::rotate              (C++20)
ranges::rotate_copy         (C++20)
ranges::shuffle             (C++20)
```

Range algorithms are constrained

That just means they make use of the new STL concepts! Remember them?

```
template<class T>
concept range = requires(T& t) { ranges::begin(t); ranges::end (t); };

template<class T>
concept input_range =
   ranges::range<T> && std::input_iterator<ranges::iterator_t<T>>;

template<ranges::input_range R, class T, class Proj = std::identity>
borrowed_iterator_t<R> find( R&& r, const T& value, Proj proj = {} );
```

A range has a begin and end! :)

An input range is a range using an input iterator

I've cut out some of the code here, but notice that ranges find uses **concepts**!!

Ranges Recap

- Ranges use concepts! Better error messages, what’s not to like?
- We can pass entire containers

Views: a way to compose algorithms

Definition: A view is a range that *lazily* adapts another range

Filter and transform in the old STL

This code is a bit awkward in the current STL

```
std::vector<char> v = {'a', 'b', 'c', 'd', 'e'};

// Filter -- Get only the vowels
std::vector<char> f;
std::copy_if(v.begin(), v.end(), std::back_inserter(f), isVowel);

// Transform -- Convert to uppercase
std::vector<char> t;
std::transform(f.begin(), f.end(), std::back_inserter(t), toupper);

// { 'A', 'E' }
```

Filter and transform with **views**!

A **view** is a range that lazily transforms its underlying range, one element at a time

```
std::vector<char> letters = {'a', 'b', 'c', 'd', 'e'};

auto f = std::ranges::views::filter(letters, isVowel);
auto t = std::ranges::views::transform(f, toupper);

auto vowelUpper = std::ranges::to<std::vector<char>>(t);
```

Views are composable

```
auto f = std::ranges::views::filter(letters, isVowel);
// f is a view! It takes an underlying range letters
// and yields a new range with only vowels!

auto t = std::ranges::views::transform(f, toupper);
// t is a view! It takes an underlying range f
// and yields a new range with uppercase chars!

auto vowelUpper = std::ranges::to<std::vector<char>>(t);
// Here we materialize the view into a vector!
// Nothing actually happens until this line!
```

We can chain views together use operator|

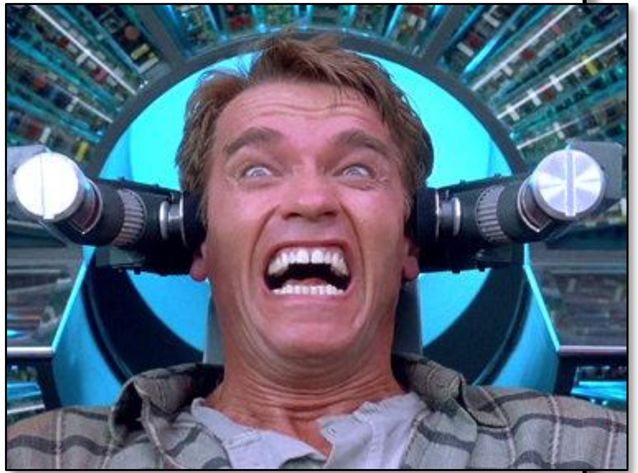
```
std::vector<char> letters = {'a','b','c','d','e'};
std::vector<char> upperVowel = letters
  | std::ranges::views::filter(isVowel)
  | std::ranges::views::transform(toupper)
  | std::ranges::to<std::vector<char>>();

// upperVowel = { 'A', 'E' }
```

Remember: range algorithms are eager

`std::ranges` are a reskin of the old STL algorithms

```
// This actually sorts vec, RIGHT NOWWW!!!!
std::ranges::sort(v);
```

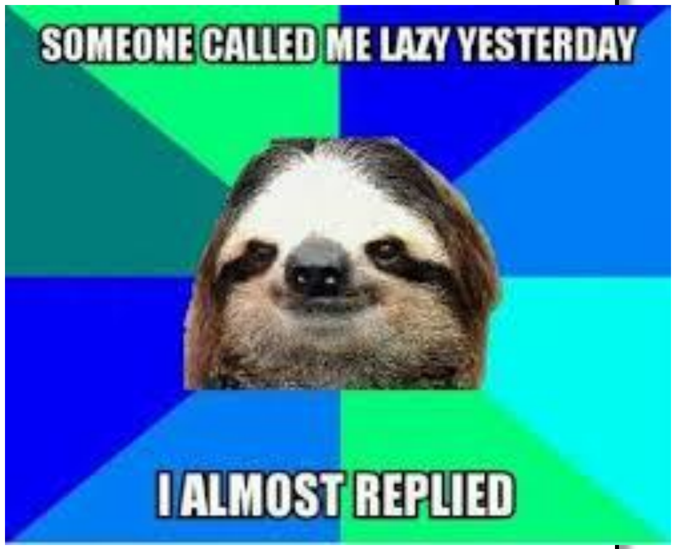


Remember: views are lazy

`std::ranges::views` are a lazy way of composing algorithms

```
auto view = letters
    | std::ranges::views::filter(isVowel)
    | std::ranges::views::transform(toupper)

std::vector<char> upperVowel =
    std::ranges::to<std::vector<char>>(view);
```



Pro tip: Views are like Python generators

This code in C++ works *exactly the same* as this Python code

```cpp
auto view = letters
    | std::ranges::views::filter(isVowel)
    | std::ranges::views::transform(toupper);
auto upperVowel = std::ranges::to<std::vector<char>>(view);
```

```python
view = (l for l in letters if isVowel(l))    # Lazy evaluation
view = (l.upper() for l in view)             # Lazy evaluation
upperVowel = list(view)
```

Ranges and view **recap**

- Why you might like ranges/views?
  - ✅ Worry less about iterators
  - ✅ Constrained algorithms mean better error messages
  - ✅ Super readable, functional syntax
- Why you might dislike ranges/views?
  - ❌ They are extremely new, not fully feature complete yet
  - ❌ Lack of compiler support
  - ❌ Loss of performance compared to hand-coded version
    - ℹ️ For more info, see [The Terrible Problem of Incrementing a Smart Iterator]()

Soundex: C++26?

Once views are fully implemented, our Soundex code might look like this

```
namespace rng = std::ranges;
namespace rv = std::ranges::views;

auto ch = *rng::find_if(s, isalpha);                          // Get first letter
auto sx = s  | rv::filter(isalpha)                          // Discard non-letters
             | rv::transform(soundexEncode)                 // Encode letters
             | rv::unique                                   // Remove duplicates
             | rv::filter(notZero)                          // Remove zeros
             | rv::concat("0000")                           // Ensure length >= 4
             | rv::drop(1)                                  // Skip first digit
             | rv::take(3)                                  // Take next three
             | rng::to<std::string>();                      // Convert to string
return toupper(ch) + v;
```