

Lecture 11:

Functions & Lambdas

Preston Seay & Emily Ai

CS106L, Fall 2026

Today's Agenda

- Functions and Lambdas
 - How can we represent functions as variables in C++?
- Algorithms
 - Tackling a popular algorithm with modern C++
- Ranges and Views
 - A new, functional approach to C++ algorithms

Recall: Template Functions

- Turn to a partner and discuss:
 - What's one thing you remember from Tuesday's lecture on function templates?

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;
}
```

Writing a **templated** find function

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

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

Writing a **templated** find function

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

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

Implicit Instantiation!

Compiler deduces
template types by
looking at arguments

**We defined our `find` function
in a `general` way!**

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

- Instead of **find** searching for **value** in a container...
- What if we could ask arbitrary questions?
 - A vowel in a **string**?
 - A prime number in a **vector<int>**?
 - A number divisible by 5 in a **set<int>**?
- More generally, what if we could perform arbitrary operations?

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 wouldPrestonApproveOf(Trail t)  
{  
    return t.isWaterfall();  
}
```



Binary

```
bool isDivisible(int n, int d) {  
    return n % d == 0;  
}  
bool isLongerThan(Trail x, Trail y) {  
    return x.len > y.len;  
}
```

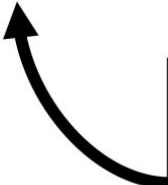
Using predicates

- How can we use `isVowel` to find the first vowel in a `string`?
- Or `wouldPrestonApproveOf` to find a trail with a waterfall in a `vector<Trail>`?
- 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, typename T>
It find(It first, It last, const T& value) {
    for (auto it = first; it != last; ++it) {
        if (*it == value) return it;
    }
    return last;
}
```



This condition worked for finding a specific value, but it's too specific. How can we modify it to handle a general condition?

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?

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;
}
```

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

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

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;
}
```

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

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

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;
}
```

Wait... what's the type of this predicate?

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!



**Let's Practice w/
Slide 1!**

What questions do you have?



bjarne_about_to_raise_hand

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 flower = "rose";  
auto it = find_if(flower.begin(), flower.end(), isVowel);  
*it = 'i'; // "rise"
```

You: "What type
is this?"

Compiler: "Don't
worry about it!"



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

Functions 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 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... Helpppp... */)
```



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!

We want to give our function **extra state...**

...without introducing another parameter

Introducing... **lambda functions**

Lambda functions are functions that can capture state from an enclosing scope

```
int n;  
std::cin >> n;  
  
auto lessThanN = [n] (int x) { return x < n; };  
  
find_if(begin, end, lessThanN); // 😎 😎
```



**Let's Practice w/
Slide 2!**

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



**Let's Practice w/
Slide 3!**

What questions do you have?



bjarne_about_to_raise_hand

How do lambdas work?

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>`

```
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



Since a functor is an **object, it can have **state****

Functors can have state!

```
struct my_functor {  
    int operator()(int a) const {  
        return a * value;  
    }  
  
    int value;  
};  
  
my_functor f;  
f.value = 5;  
f(10); // returns?
```

Functors can have state!

```
struct my_functor {  
    int operator()(int a) const {  
        return a * value;  
    }  
  
    int value;  
};  
  
my_functor f;  
f.value = 5;  
f(10); // 50
```



**Let's Practice w/
Slide 4!**



Time for a dark secret...



**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

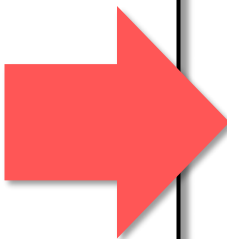
Our captures became
fields in the class!

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...

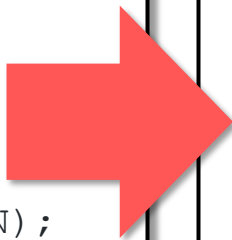
```
std::vector<int> v {1,2,3};  
for (const int& e : v)  
{  
    // ...  
}
```



```
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);
```

What questions do you have?



bjarne_about_to_raise_hand

Where do we use functions & lambdas?

Algorithms

Huh... that looks familiar

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

Defined in header <algorithm>

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

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

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

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

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

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

<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 that match predicate p

```
std::transform(InputIt r1, InputIt r2, OutputIt 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...

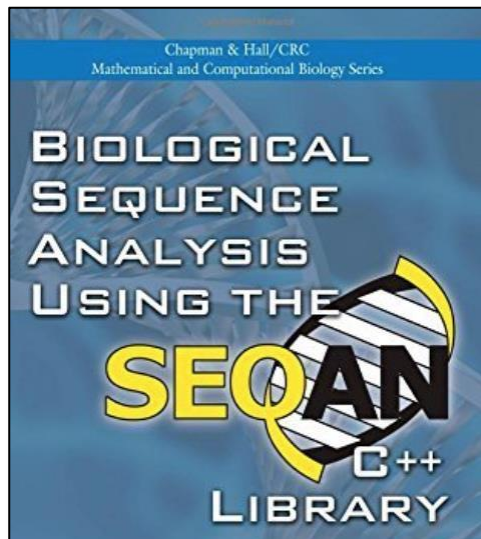
all_of	copy	merge	random_shuffle	is_sorted
any_of	copy_n	inplace_merge	shuffle	is_sorted_until
none_of	copy_if	includes	push_heap	nth_element
for_each	copy_backward	set_union	pop_heap	min
find	move	set_intersection	make_heap	max
find_if	move_backward	set_difference	sort_heap	minmax
find_if_not	swap	set_symmetric_difference	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!

<algorithm> lets us inspect and transform data

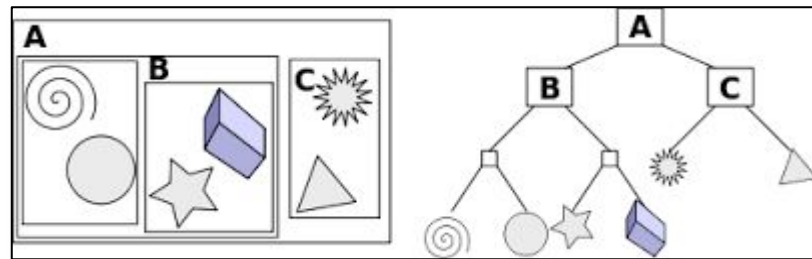
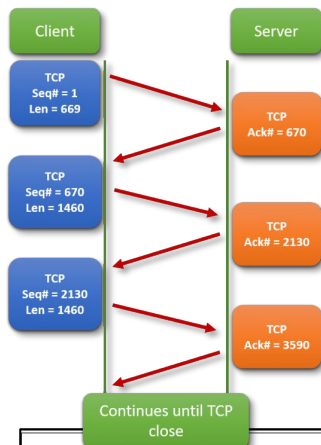


\$457⁹²

You Earn: 200 pts [Learn more](#)

\$3.99 delivery **September 30 - October 4.**

[Details](#)

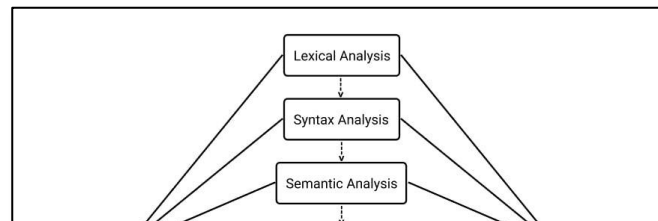


Huffman Coding

Example: "COMPRESSION_IS_COOL"

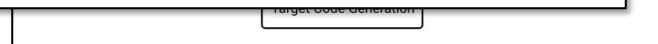
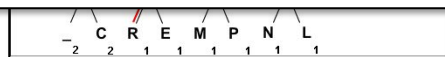
To compute the bit pattern for each datum, we go up the tree and note down a "1" (true) if we take the branch to the left and a "0" (false) if we take the branch to the right, respectively.

—: 0000
C: 0001



In what language is LLVM written?

All of the LLVM tools and libraries are written in C++ with extensive use of the STL.



What questions do you have?



bjarne_about_to_raise_hand

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?

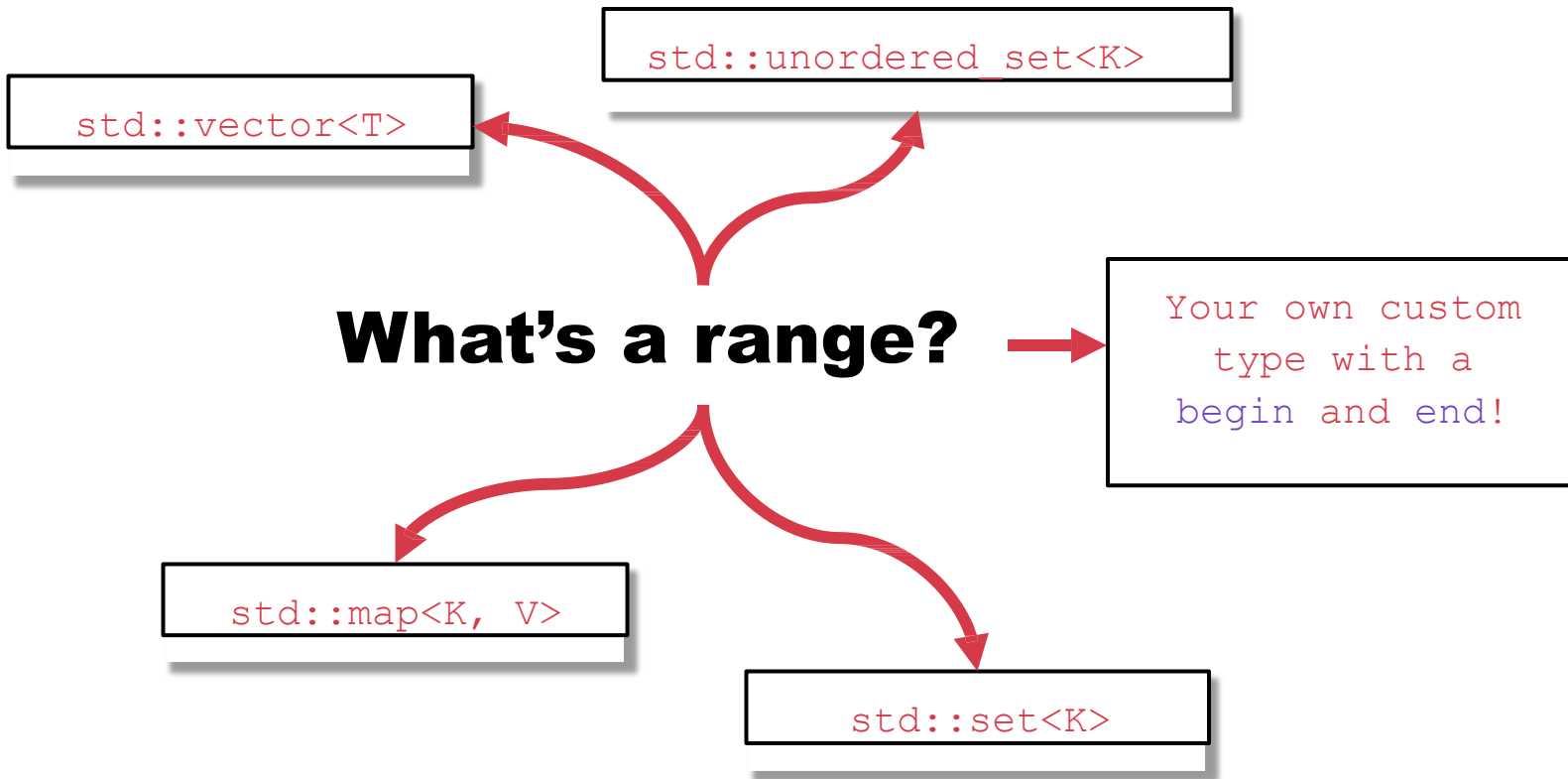
`std::vector<T>`

`std::unordered_set<K>`

Your own custom
type with a
begin and end!

`std::map<K, V>`

`std::set<K>`



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!

<code>ranges::find_last</code>	(C++23)
<code>ranges::find_last_if</code>	(C++23)
<code>ranges::find_last_if_not</code>	(C++23)

<code>ranges::find_end</code>	(C++20)
-------------------------------	---------

<code>ranges::find_first_of</code>	(C++20)
------------------------------------	---------

<code>ranges::adjacent_find</code>	(C++20)
------------------------------------	---------

<code>ranges::search</code>	(C++20)
-----------------------------	---------

<code>ranges::search_n</code>	(C++20)
-------------------------------	---------

<code>ranges::contains</code>	(C++23)
<code>ranges::contains_subrange</code>	(C++23)

<code>ranges::starts_with</code>	(C++23)
----------------------------------	---------

<code>ranges::ends_with</code>	(C++23)
--------------------------------	---------

<code>ranges::copy</code>	(C++20)
<code>ranges::copy_if</code>	(C++20)

<code>ranges::copy_n</code>	(C++20)
-----------------------------	---------

<code>ranges::copy_backward</code>	(C++20)
------------------------------------	---------

<code>ranges::move</code>	(C++20)
---------------------------	---------

<code>ranges::move_backward</code>	(C++20)
------------------------------------	---------

<code>ranges::fill</code>	(C++20)
---------------------------	---------

<code>ranges::fill_n</code>	(C++20)
-----------------------------	---------

<code>ranges::transform</code>	(C++20)
--------------------------------	---------

<code>ranges::generate</code>	(C++20)
-------------------------------	---------

<code>ranges::generate_n</code>	(C++20)
---------------------------------	---------

<code>ranges::remove</code>	(C++20)
<code>ranges::remove_if</code>	(C++20)

<code>ranges::remove_copy</code>	(C++20)
<code>ranges::remove_copy_if</code>	(C++20)

<code>ranges::replace</code>	(C++20)
<code>ranges::replace_if</code>	(C++20)

<code>ranges::replace_copy</code>	(C++20)
<code>ranges::replace_copy_if</code>	(C++20)

<code>ranges::swap_ranges</code>	(C++20)
----------------------------------	---------

<code>ranges::reverse</code>	(C++20)
------------------------------	---------

<code>ranges::reverse_copy</code>	(C++20)
-----------------------------------	---------

<code>ranges::rotate</code>	(C++20)
-----------------------------	---------

<code>ranges::rotate_copy</code>	(C++20)
----------------------------------	---------

<code>ranges::shuffle</code>	(C++20)
------------------------------	---------

What questions do you have?



bjarne_about_to_raise_hand

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!  
// elements aren't processed into results until the view  
is used!
```

We can chain views together use **operator |**

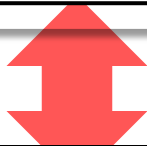
```
std::vector<char> letters = {'a', 'b', 'c', 'd', 'e'};
std::vector<char> upperVowel = letters
    | std::ranges::views::filter(isVowel)
    | std::ranges::views::transform(toupper)
    | std::ranges::to<std::vector<char>>();

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

Pro tip: **Views** are like Python **generators**

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

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



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



**Let's Practice w/
Slide 5!**

What questions do you have?



bjarne_about_to_raise_hand

Recap

What We Covered

- Predicates
 - Functions that return booleans
- Lambdas
 - Anonymous functions that can capture outside variables
- Functors
 - Objects that act like functions using `operator()`
- STL Algorithms
 - Built-in functions for searching, sorting, and transforming data
- Ranges & Views
 - Simplify working with containers and support lazy operations

Next Time: Operator Overloading