

**Welcome back! Link to Attendance Form ↓**



[bit.ly/a24-lec10](https://bit.ly/a24-lec10)

# Recall: Template Functions

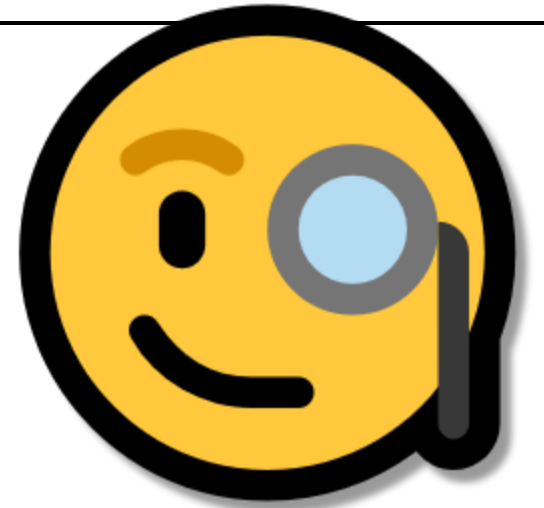
- Turn to a partner and discuss:
- What's one thing you remember from Tuesday's lecture on template functions?

# Recall: Writing a min function

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

```
double min(double a, double b) {  
    return a < b ? a : b;  
}
```

```
std::string min(std::string a, std::string b) {  
    return a < b ? a : b;  
}
```



# 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;
}
```

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

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

**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
```

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

# 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>`?

# **Lecture 10:**

# **Functions and Lambdas**

CS106L, Autumn 2024

# Today's Agenda

- Functions and Lambdas
  - How can we represent functions as variables in C++?
- Algorithms
  - Revisiting an old algorithm you may have seen before in modern C++
- Ranges and Views
  - A brand new (C++26), functional approach to C++ algorithms

# What questions do you have?



bjarne\_about\_to\_raise\_hand

# **Functions and Lambdas**



**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;  
}
```

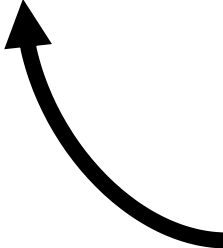
# Using predicates

- 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

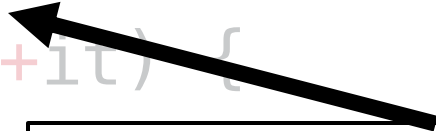
```
template <typename It, typename T>
It find(It first, It last, const T& value) {
    for (auto it = first; it != last; ++it) {
        if (*it == value) return it;
    }
    return last;
}
```



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

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

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

# 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 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  
gotttttchuuu man"



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

**Aside: Seriously though, what is the type of `Pred`?**



# 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... */)
```



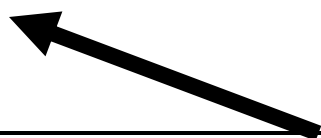
# We can't just add another parameter

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!

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

...without introducing another parameter

# 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';  
    });
```

# What questions do you have?



bjarne\_about\_to\_raise\_hand

**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>`

```
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****

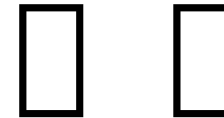
# Functors can have state!

```
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



**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

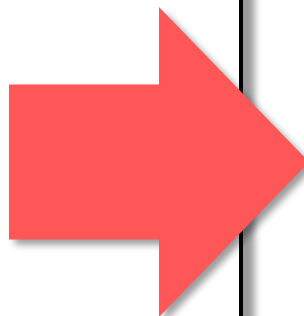
Our captures became  
fields in the class!

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...

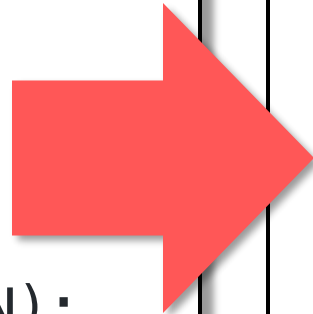
```
std::vector<int> v {1,2,3};  
for (const int& e : v)  
{  
    // ...  
}
```



```
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);
```



# 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; };
```

# What questions do you have?



bjarne\_about\_to\_raise\_hand

**Where do we use functions & lambdas?**

# Algorithms

# 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>`

<code>template&lt; class InputIt, class T &gt;</code>		(constexpr since C++20)
<code>InputIt find( InputIt first, InputIt last, const T&amp; value );</code>		(until C++26)
<code>template&lt; class InputIt, class T = typename std::iterator_traits</code>	(1)	
<code>&lt;InputIt&gt;::value_type &gt;</code>		(since C++26)
<code>constexpr InputIt find( InputIt first, InputIt last, const T&amp; value );</code>		
<code>template&lt; class ExecutionPolicy, class ForwardIt, class T &gt;</code>		(since C++17)
<code>ForwardIt find( ExecutionPolicy&amp;&amp; policy,</code>		(until C++26)
<code>ForwardIt first, ForwardIt last, const T&amp; value );</code>		
<code>template&lt; class ExecutionPolicy,</code>	(2)	
<code>class ForwardIt, class T = typename std::iterator_traits</code>		(since C++26)
<code>&lt;ForwardIt&gt;::value_type &gt;</code>		
<code>ForwardIt find( ExecutionPolicy&amp;&amp; policy,</code>		
<code>ForwardIt first, ForwardIt last, const T&amp; value );</code>		
<code>template&lt; class InputIt, class UnaryPred &gt;</code>	(3)	(constexpr since C++20)
<code>InputIt find_if( InputIt first, InputIt last, UnaryPred p );</code>		
<code>template&lt; class ExecutionPolicy, class ForwardIt, class UnaryPred &gt;</code>	(4)	(since C++17)
<code>ForwardIt find_if( ExecutionPolicy&amp;&amp; policy,</code>		
<code>ForwardIt first, ForwardIt last, UnaryPred p );</code>		

# **<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...

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

# 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

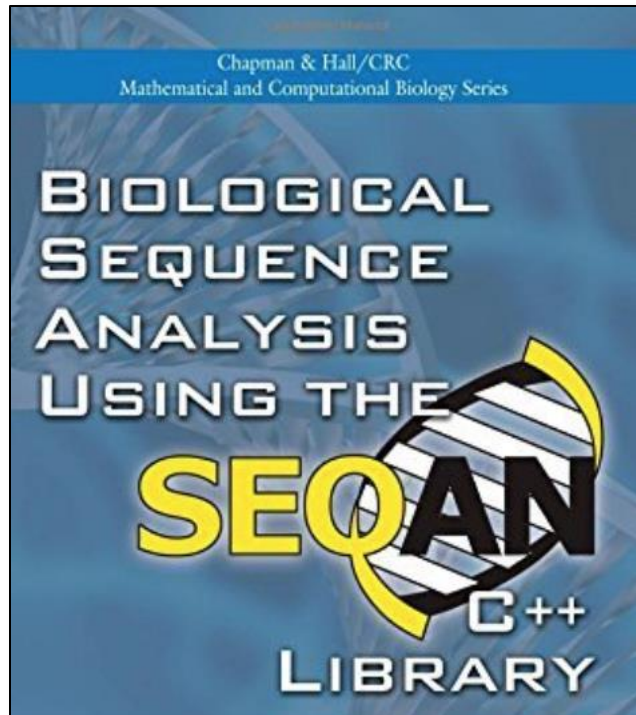
*all in their most general form!*

# What questions do you have?



bjarne\_about\_to\_raise\_hand

# <algorithm> lets us inspect and transform data

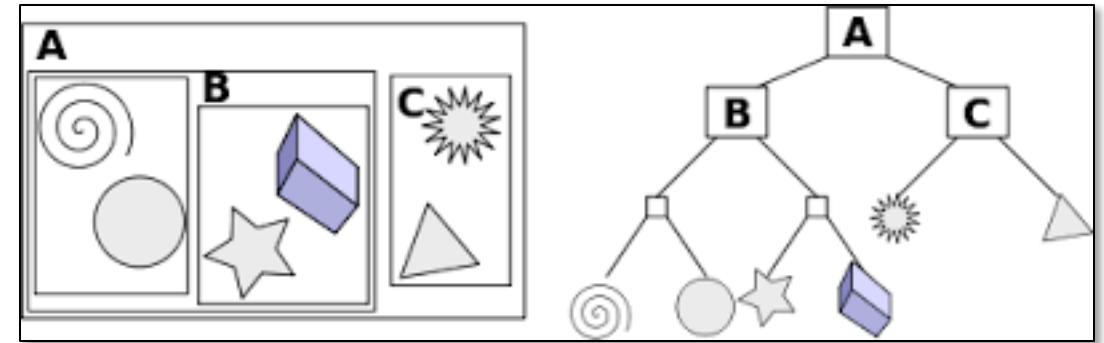
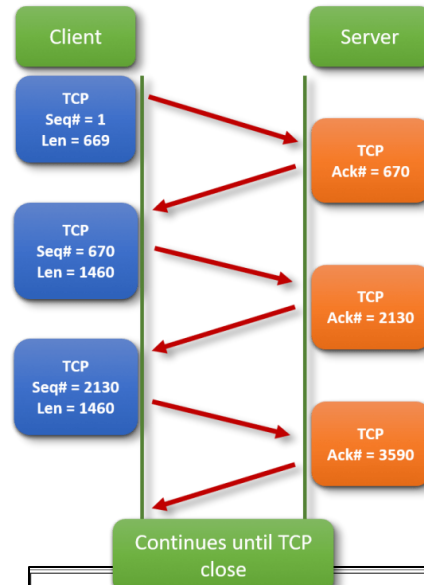


\$457<sup>92</sup>

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

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

[Details](#)

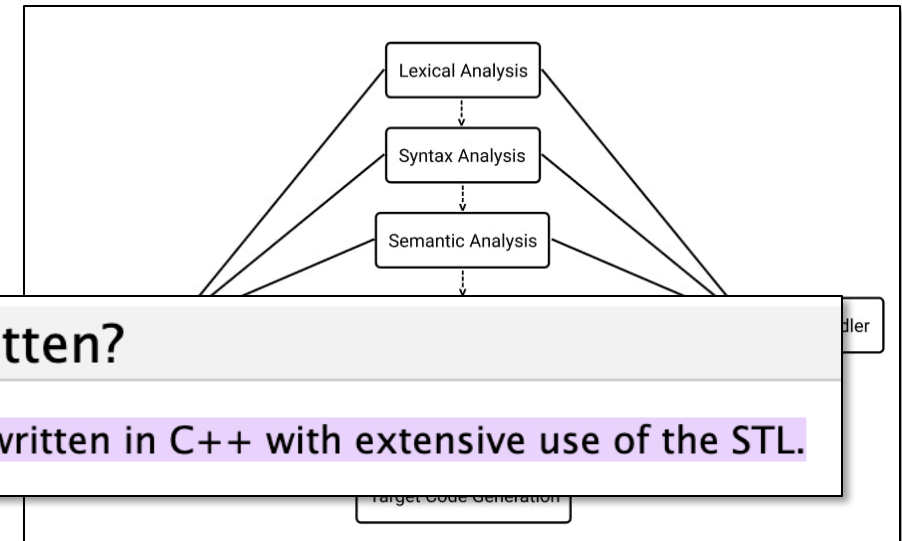


### Huffman Coding

**Example:** "COMPRESSION\_IS\_COOL"

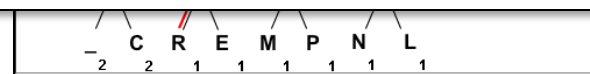
To compute the bit pattern for each datum, we go up the tree and note down a "1" (true) if we take the branch to the *left* and a "0" (false) if we take the branch to the *right*, respectively.

... : 0000  
C : 0001



In what language is LLVM written?

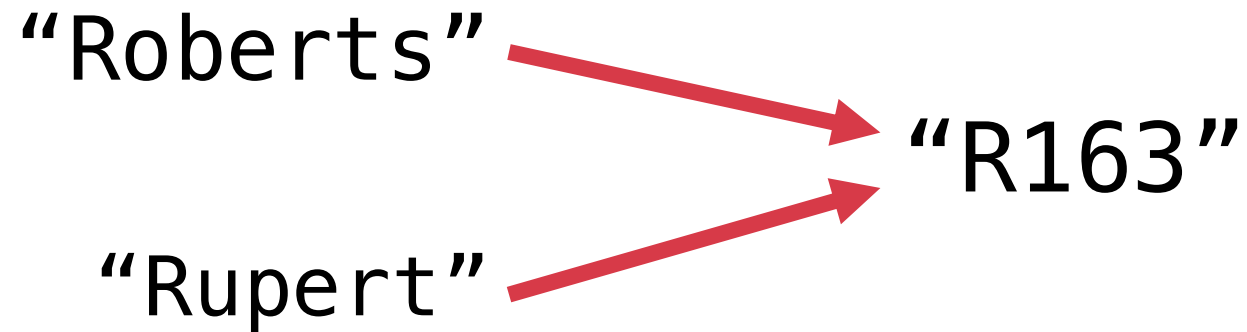
All of the LLVM tools and libraries are written in C++ with extensive use of the STL.



**Let's write an algorithm using the STL!**

**Soundex!!**  

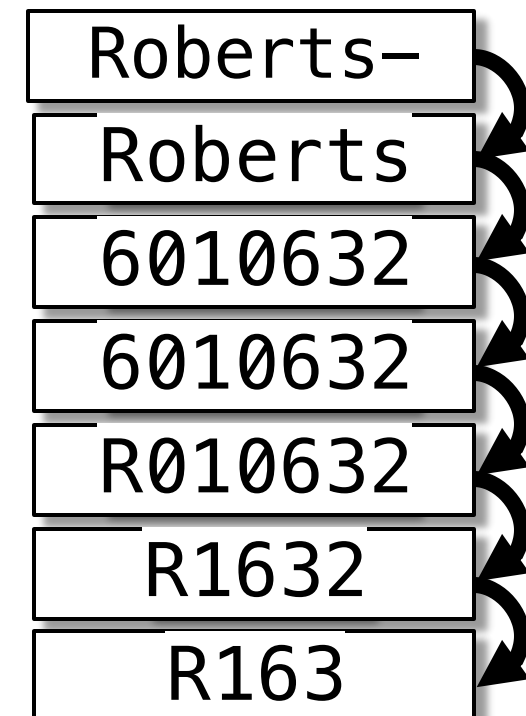
**Goal: produce a **phonetic encoding** for names**



# How do we implement **soundex**?

1. Given a string **s**, extract **the letters** from **s**
2. Replace each letter with its **soundex encoding**
3. Coalesce **adjacent duplicates** (222025 becomes 2025)
4. Replace **first digit** with the uppercase first letter of **s**
5. Discard any **zeros** from the code
6. Make the code exactly **length 4** (truncate or zero-pad)

Digit represents the letters	
0	A E I O U H W Y
1	B F P V
2	C G J K Q S X Z
3	D T
4	L
5	M N
6	R





**Let's implement Soundex with the STL!**

# What questions do you have?



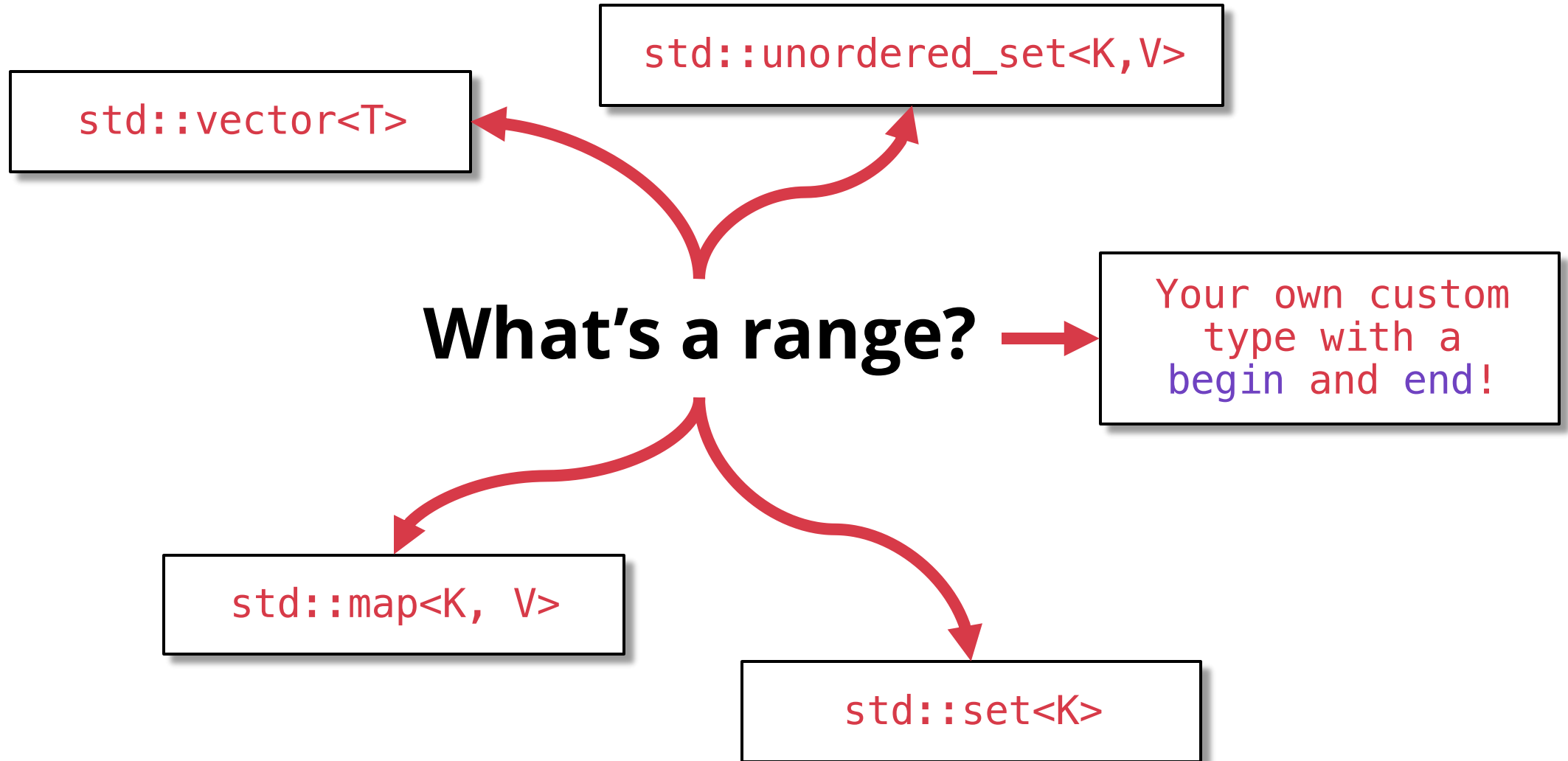
bjarne\_about\_to\_raise\_hand

**Can we make our Soundex more readable?**

# **Ranges and Views**

**Ranges are a new version of the STL**

**Definition:** A range is anything with a **begin** and **end**



# 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!

<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)
------------------------------	---------

# 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); };
```

A range has a begin and end! :)

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

An input range is a range using an input iterator

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

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

# What questions do you have?



bjarne\_about\_to\_raise\_hand

# Ranges Recap

- Ranges use concepts! Better error messages, what's not to like?
- We can pass entire containers
- ***Okay... is that it?*** 👁️👁️

**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

```
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)
```










# What questions do you have?



bjarne\_about\_to\_raise\_hand

**What would Soundex with *views* look like?**

# 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
  -  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;
```