Welcome back! Link to Attendance Form ↓

Recall: Template Functions

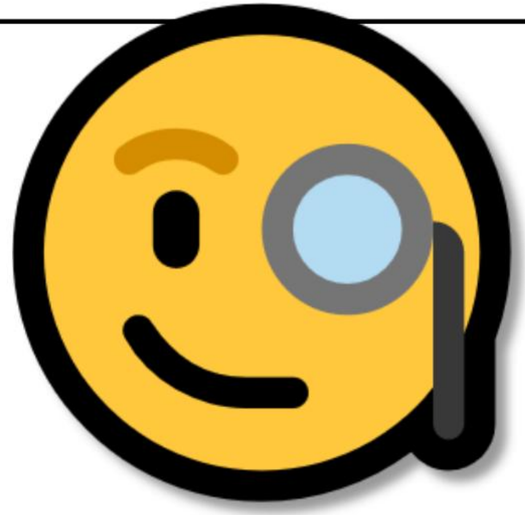
- Turn to a partner and discuss:
- What's one thing you remember from Tuesday's lecture on template functions?

Recall: Writing a `min` function

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

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

```
template <typename T>
T min(T a, T 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

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




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

Recall: Implicit instantiation is kind of like `auto`

```
int m = min(106, 107);
```

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

Recall: Writing a templated `find` function

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

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

Recall: Writing a templated `find` function

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

```
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

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) {




}
```

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");
```

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

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
```

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

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

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`?

Lecture 10: Functions and Lambdas

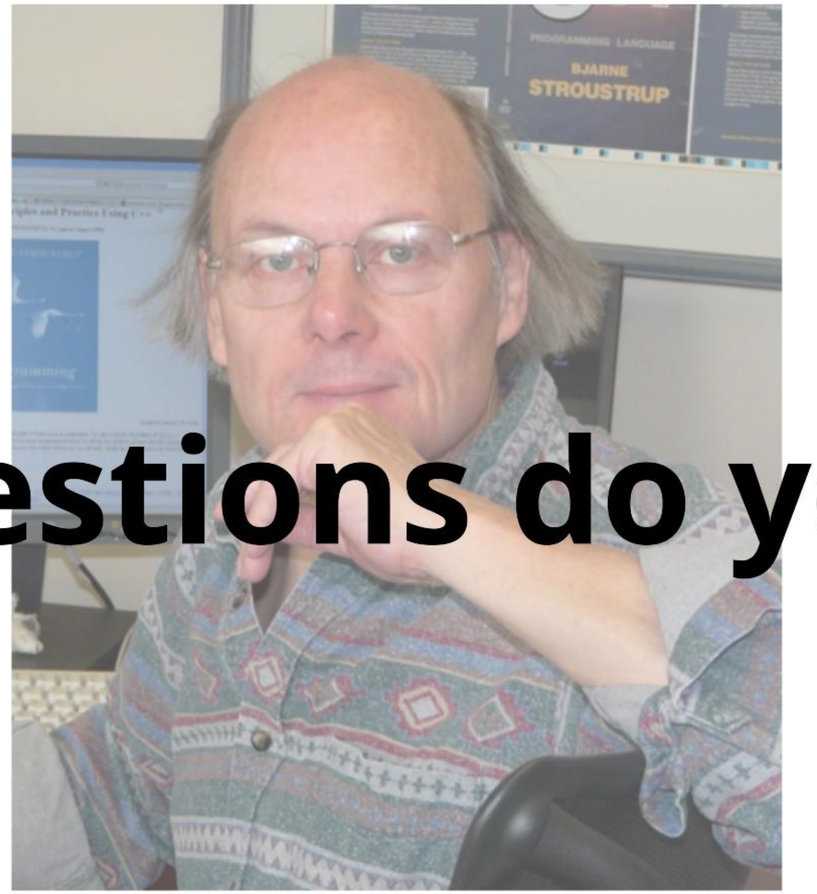
CS106L, Winter 2025

Today’s Agenda

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) {



}
```

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

Using predicates

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?

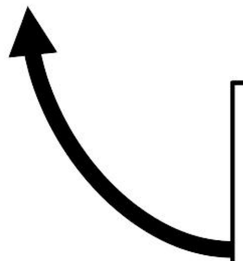
Key Idea: We need to pass a predicate to a function

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;
}
```

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.

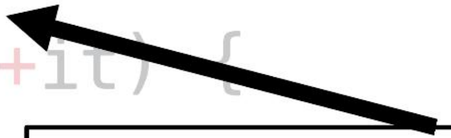
How can we modify it to handle a general condition?

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;
}
```

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;
}
```

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;
}
```

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!

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

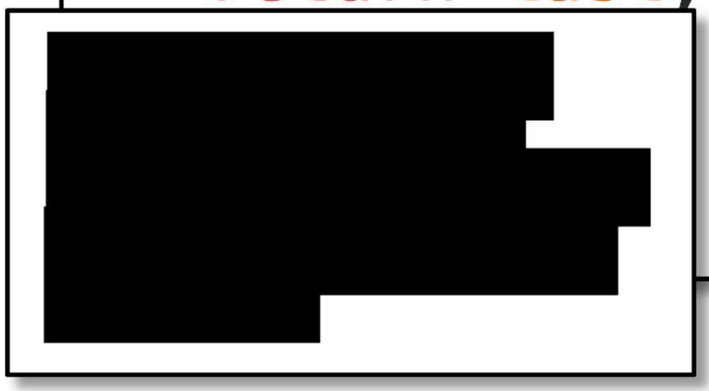
```
template <typename It, typename Pred>
It find_if(It first, It last, Pred pred) {
  for (auto it = first; it != last; ++it) {
    if (pred(*it)) return it;
  }
  return last;
}
```

Pred: the type of our predicate.

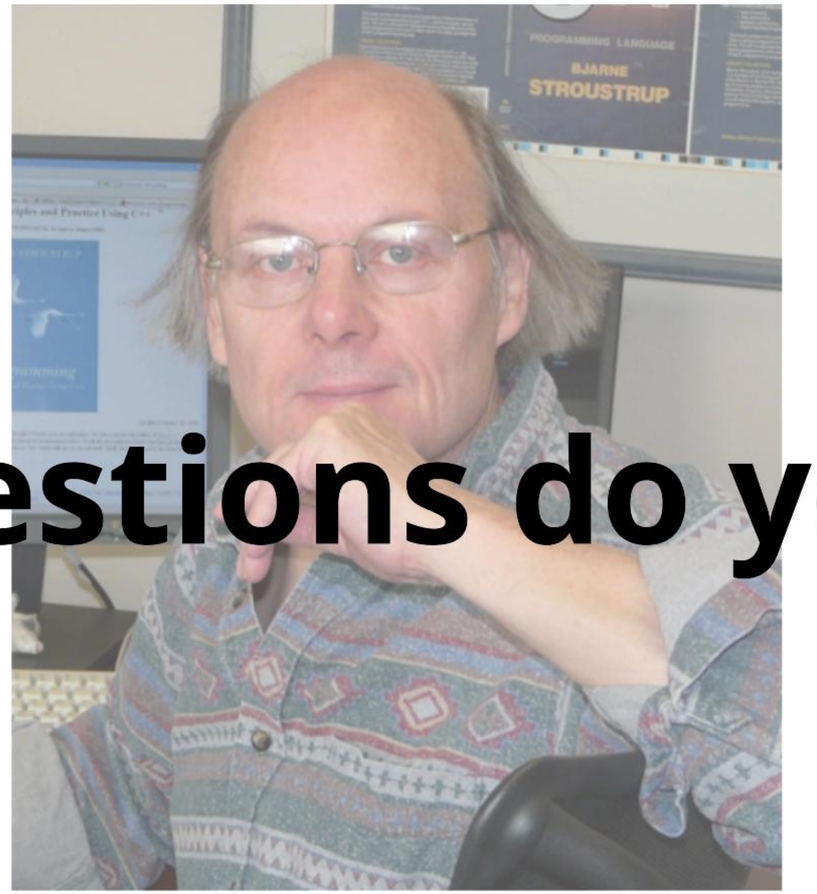
Compiler will figure this out for us using implicit instantiation!

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



What questions do you have?



bjarne_about_to_raise_hand

Using our `find_if` function

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);
```

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

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};
```

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());
```

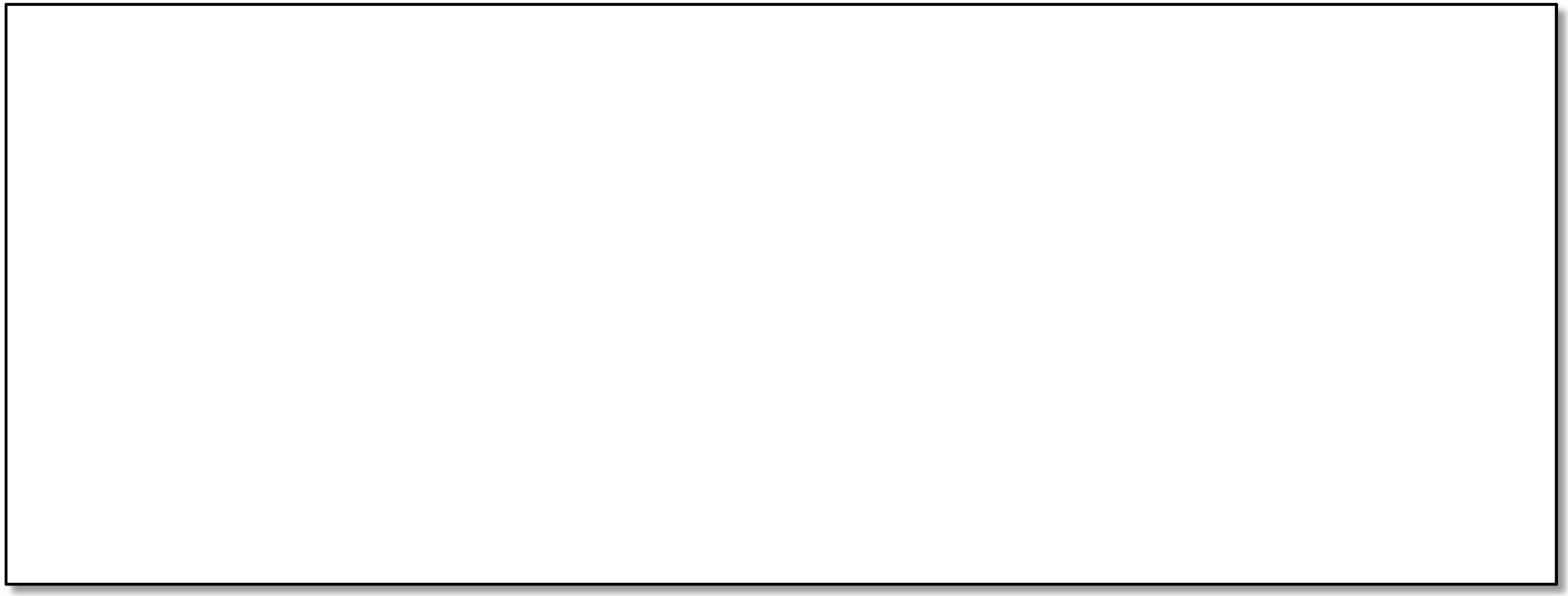
You: ”What type is this!!?”

Compiler: “I gottttchuuu man”

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

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

Pred is a function pointer



Pred is a function pointer

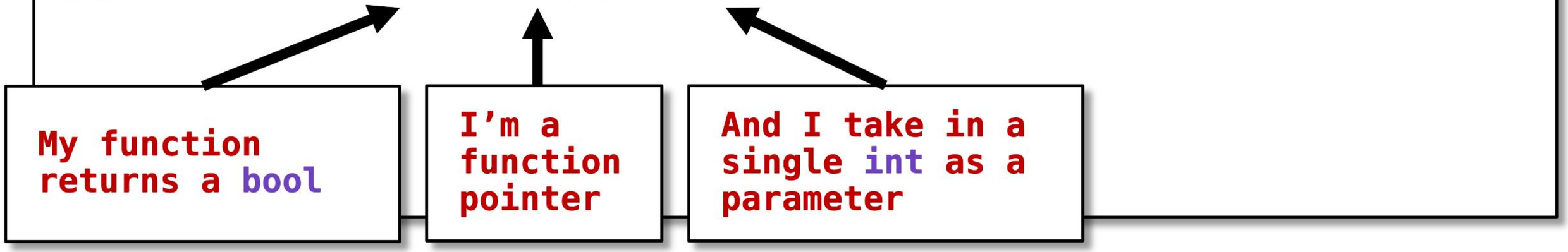
```
find_if(corlys.begin(), corlys.end(), isVowel);


find_if(ints.begin(), ints.end(), isPrime);
```

Pred is a function pointer

```
find_if(corlys.begin(), corlys.end(), isVowel);


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

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)
```



Function pointers generalize poorly

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

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; }



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

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

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

Lambda Syntax

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

Parameters

Function parameters, exactly like a normal function

Function body

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

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;
}
```

A note on captures

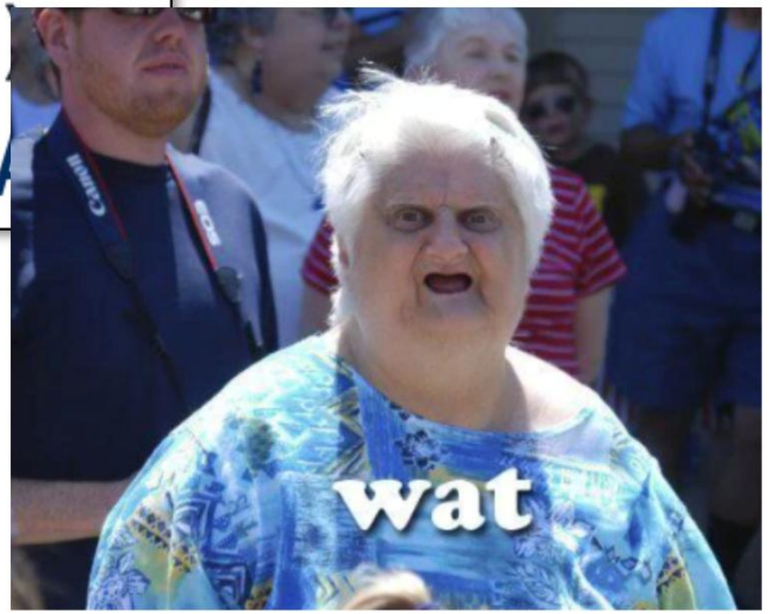
```
auto lambda = [capture-values](arguments) {
   return expression;
}

[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
```

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’;
   });
```

```
std::string corlys =
auto it = find_if(corl
  [](auto c) {
      c = toupper(c)
      return c == 'A
```

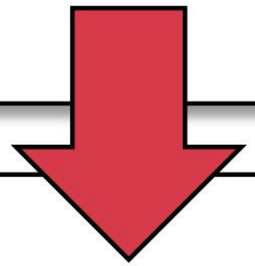


`auto` parameters are shorthand for templates

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

`auto` parameters are shorthand for templates

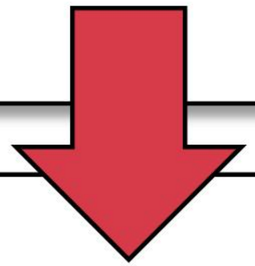
```
auto lessThanN = [n](auto x) {
   return x < n;
};
```



```
template <typename T>
auto lessThanN = [n](T x) {
   return x < n;
};
```

`auto` parameters are shorthand for templates

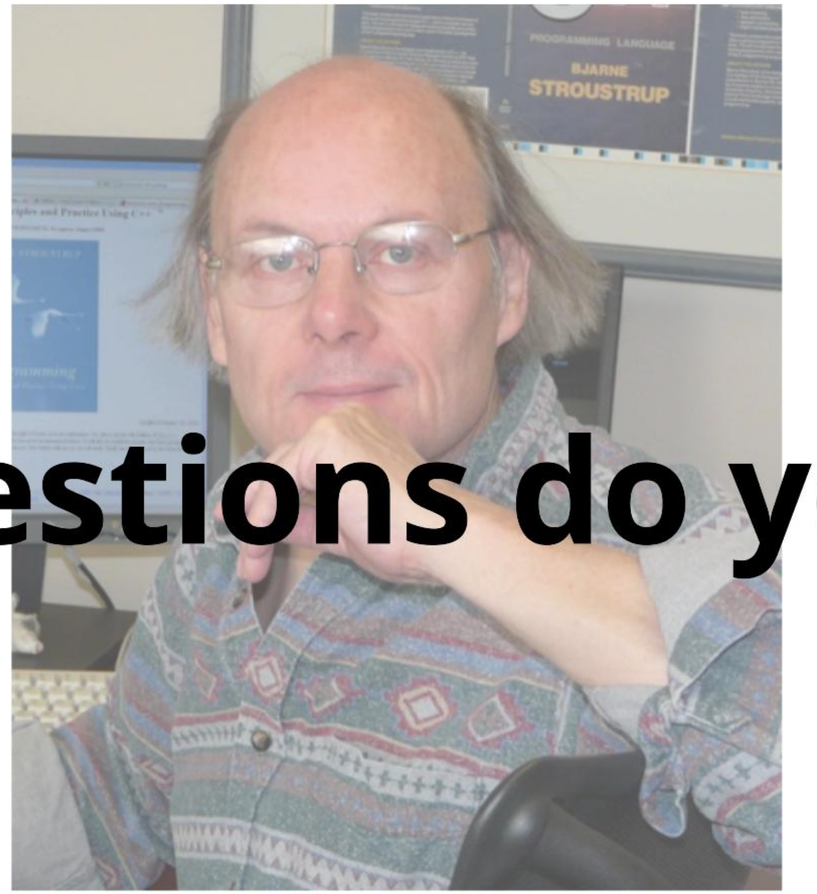
```
auto lessThanN = [n](auto x) {
  return x < n;
};
```



```
template <typename T>
auto lessThanN = [n](T x) {
  return x < n;
};
```

This is true wherever you see an `auto` parameter, not just in lambda functions!

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()`

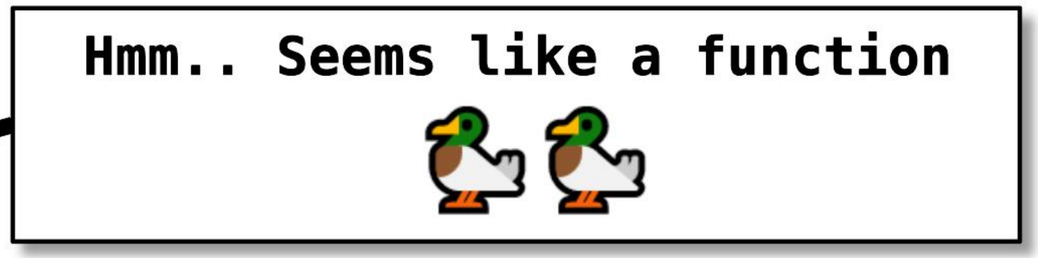
In English: an object that acts like a function

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;
   }
};
```

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
```



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 ...;
   }
};
```

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

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;
};
```

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);
```

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

...is equivalent to this code!

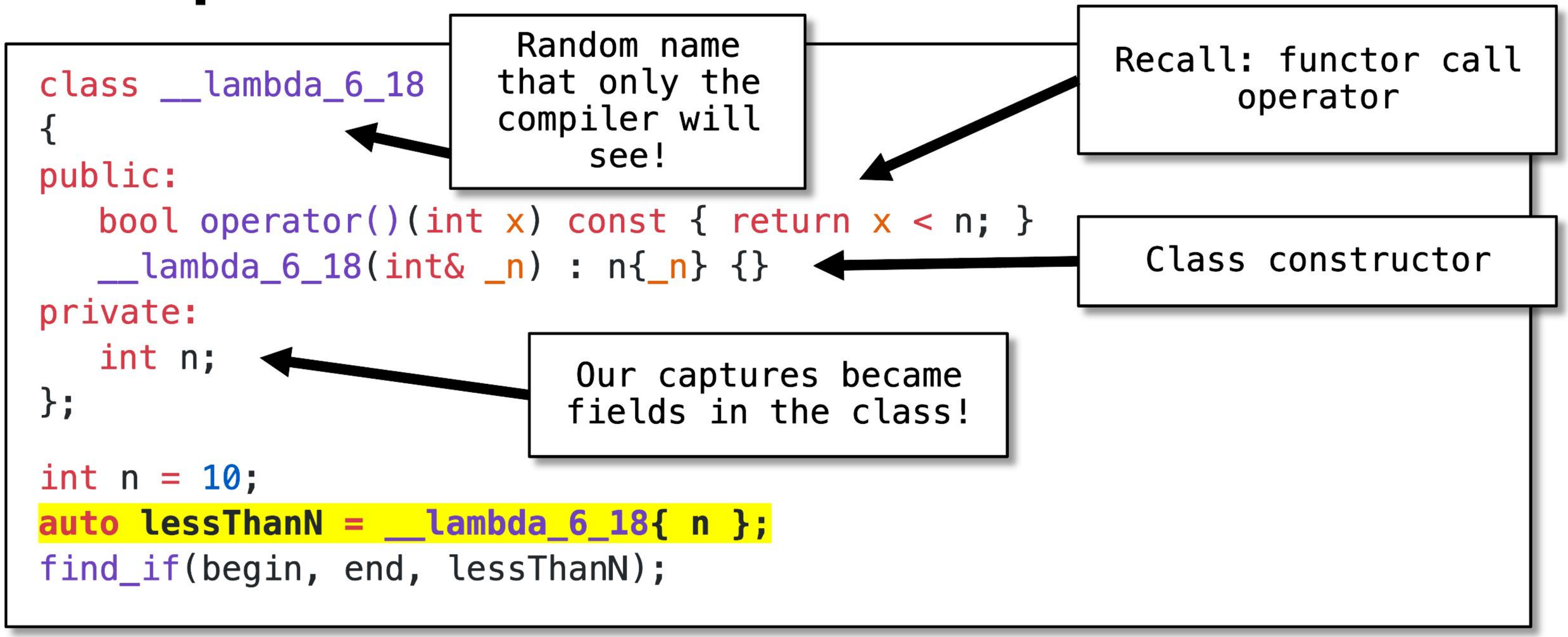
Random name that only the compiler will see!

Recall: functor call operator

```
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);
```

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

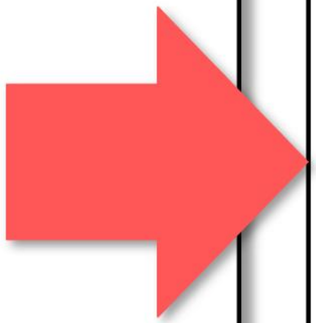
...is equivalent to this code!



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

You've seen this kind of thing before...

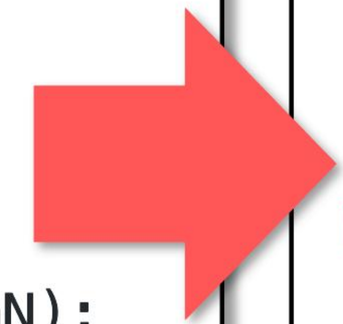
```
std::vector<int> v {1,2,3};
for (const int& e : v)
{
    // ...
}
```



```
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

Functions & Lambdas Recap

- Use functions/lambdas to pass around **behaviour** as variables
- Aside: `std::function` is an overarching type for functions/lambdas

```
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; };
```