



 <http://web.stanford.edu/class/cs106l/>



## Functions and Lambdas

How can we make template functions even more general?

CS106L - Spring 23

# Attendance!

<https://bit.ly/44knqp8>





# Announcements!

- No class **next week** – midquarter break!
  - Office hours during class time (3-4:30pm)
  - Review material from the lectures so far!
- After this lecture, you will be able to complete Assignment 1!
  - Due **May 12th!**



## CONTENTS



### 01. Recap: Template Functions



### 02. Functions and Lambdas

Passing input outside of parameters

### 03. Algorithms



## CONTENTS



### 01. Recap: Template Functions



### 02. Functions and Lambdas

Passing input outside of parameters

### 03. Algorithms

# Template functions are completely generic functions!

Just like classes, they work regardless of type!

Let's break it down:

Indicating this  
function is a template

Specifies that  
Type is generic

List of your  
template  
variables

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

## Calling template functions

We can **explicitly** define what type we will pass, like this:

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

```
// int main() {} will be omitted from future examples
// we'll instead show the code that'd go inside it
cout << myMin<int>(3, 4) << endl; // 3
```



**Just like in  
template classes!**

## Calling template functions

We can also **implicitly** leave it for the compiler to deduce!

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

```
// int main() {} will be omitted from future examples  
// we'll instead show the code that'd go inside it  
cout << myMin(3.2, 4) << endl; // 3.2
```



## Review: Template Functions

- Template functions allow you to parametrize the type of a function to be anything without changing functionality
- Generic programming can solve a complicated conceptual problem for any specifics – powerful and flexible!
- Template code is instantiated at compile time; template metaprogramming takes advantage of this to run code at compile time



## CONTENTS



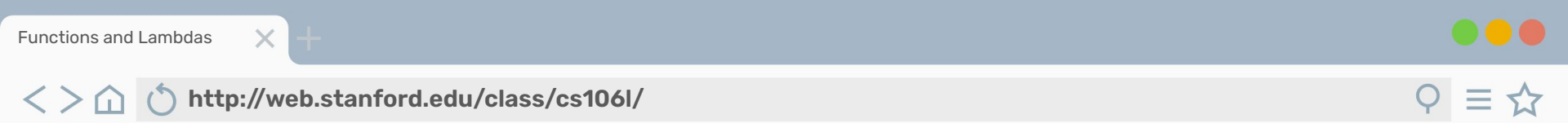
### 01. Recap: Template Functions



### 02. Functions and Lambdas

Passing input outside of parameters

### 03. Algorithms



**Let's review that  
count\_occurrences function!**

## This is a successfully templated function!

This code will work for any containers with any types, for a single specific target.

```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) count++;
    }
    return count;
}
```

Usage: `std::string str = "Xadia";`  
`count_occurrences(str.begin(), str.end(), 'a');`

## This is a successfully templated function!

This code will work for any containers with any types, for a single specific target.

Will this work for a more general category of targets than one specific value?

```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) count++;
    }
    return count;
}
```

Usage: `std::string str = "Xadia";`  
`count_occurrences(str.begin(), str.end(), 'a');`

## This is a successfully templated function!

This code will work for any containers with any types, for a single specific target.

Will this work for a more general category of targets than one specific value?

```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) count++;
    }
    return count;
}
```

Usage: `std::string str = "Xadia";`  
`count_occurrences(str.begin(), str.end(), 'a');`

**What if we wanted to find all the vowels in "Xadia"?**

## This is a successfully templated function!

This code will work for any containers with any types, for a single specific target.

Will this work for a more general category of targets than one specific value?

```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) count++;
    }
    return count;
}
```

Usage: `std::string str = "Xadia";`  
`count_occurrences(str.begin(), str.end(), 'a');`

## This is a successfully templated function!

This code will work for any containers with any types, for a single specific target.

Will this work for a more general category of targets than one specific value?

```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end;
        if (*iter == val) count++;
    }
    return count;
}
```

isVowel(\*iter) ?

Usage: `std::string str = "Xadia";`  
`count_occurrences(str.begin(), str.end(), 'a');`





# Predicate Functions

Any function that returns a boolean value is a predicate!



# Predicate Functions

Any function that returns a boolean value is a predicate!

- `isVowel()` is an example of a predicate, but there are tons of others we might want!

# Predicate Functions

Any function that returns a boolean value is a predicate!

- `isVowel()` is an example of a predicate, but there are tons of others we might want!

```
bool isLowercaseA(char c) {  
    return c == 'a';  
}  
  
bool isVowel(char c) {  
    std::string vowels = "aeiou";  
    return vowels.find(c) != std::string::npos;  
}
```

```
bool isMoreThan(int num, int limit) {  
    return num > limit;  
}  
  
bool isDivisibleBy(int a, int b) {  
    return (a % b == 0);  
}
```

# Predicate Functions

Any function that returns a boolean value is a predicate!

- `isVowel()` is an example of a predicate, but there are tons of others we might want!
- A predicate can have any amount of parameters...

```
bool isLowercaseA(char c) {  
    return c == 'a';  
}  
  
bool isVowel(char c) {  
    std::string vowels = "aeiou";  
    return vowels.find(c) != std::string::npos;  
}
```

```
bool isMoreThan(int num, int limit) {  
    return num > limit;  
}  
  
bool isDivisibleBy(int a, int b) {  
    return (a % b == 0);  
}
```

# Predicate Functions

Any function that returns a boolean value is a predicate!

- `isVowel()` is an example of a predicate, but there are tons of others we might want!
- A predicate can have any amount of parameters...

## Unary

```
bool isLowercaseA(char c) {  
    return c == 'a';  
}  
  
bool isVowel(char c) {  
    std::string vowels = "aeiou";  
    return vowels.find(c) != std::string::npos;  
}
```

## Binary

```
bool isMoreThan(int num, int limit) {  
    return num > limit;  
}  
  
bool isDivisibleBy(int a, int b) {  
    return (a % b == 0);  
}
```

## Let's use that!

```
template <typename InputIt, typename DataType, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val pred(*iter)) count++;
    }
    return count;
}
```

```
bool isVowel(char c) {
    std::string vowels = "aeiou";
    return vowels.find(c) != std::string::npos;
}
```

```
Usage: std::string str = "Xadia";
       count_occurrences(str.begin(), str.end(), isVowel);
```

## Let's use that!

```
template <typename InputIt, typename DataType, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val pred(*iter)) count++;
    }
    return count;
}
```

```
bool isVowel(char c) {
    std::string vowels = "aeiou";
    return vowels.find(c) != std::string::npos;
}
```

Usage: `std::string str = "Xadia";`  
`count_occurrences(str.begin(), str.end(), isVowel);`

## Let's use that!

```
template <typename InputIt, typename DataType, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val pred(*iter)) count++;
    }
    return count;
}
```

```
bool isVowel(char c) {
    std::string vowels = "aeiou";
    return vowels.find(c) != std::string::npos;
}
```

```
Usage: std::string str = "Xadia";
       count_occurrences(str.begin(), str.end(), isVowel);
```



## Let's use that!

```
template <typename InputIt, typename DataType, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val pred(*iter)) count++;
    }
    return count;
}
```

```
bool isVowel(char c) {
    std::string vowels = "aeiou";
    return vowels.find(c) != std::string::npos;
}
```

Usage: `std::string str = "Xadia";`  
`count_occurrences(str.begin(), str.end(), isVowel);`

What type is UniPred???

## Let's use that!

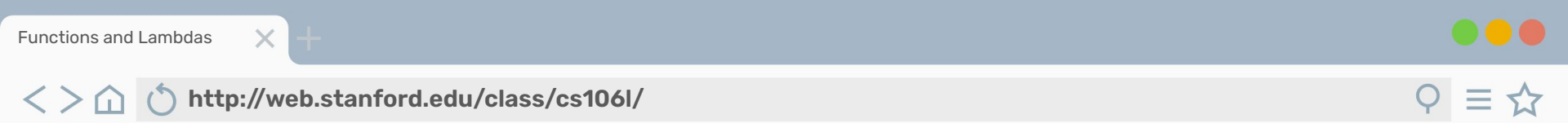
```
template <typename InputIt, typename DataType, typename UniPred>  
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {  
    int count = 0;  
    for (auto iter = begin; iter != end; ++iter) {  
        if (pred(*iter)) ++count;  
    }  
    return count;  
}
```

```
bool isVowel(char c) {  
    std::string vowels = "aeiou";  
    return vowels.find(c) != std::string::npos;  
}
```

```
Usage: std::count_occurrences(begin, end, isVowel);
```



What type is UniPred???



# Function Pointers

UniPred is what's called a **function pointer**!



# Function Pointers

UniPred is what's called a **function pointer**!

- Function pointers can be treated just like other pointers



# Function Pointers

UniPred is what's called a **function pointer**!

- Function pointers can be treated just like other pointers
- They can be passed around like variables as parameters or in template functions!

# Function Pointers

UniPred is what's called a **function pointer**!

- Function pointers can be treated just like other pointers
- They can be passed around like variables as parameters or in template functions!
- They can be called like functions!

# Is this good enough?

Are there any ways this could be an issue?

```
template <typename InputIt, typename DataType, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val pred(*iter)) count++;
    }
    return count;
}
```

```
bool isVowel(char c) {
    std::string vowels = "aeiou";
    return vowels.find(c) != std::string::npos;
}
```

```
Usage: std::string str = "Xadia";
       count_occurrences(str.begin(), str.end(), isVowel);
```

## Poor Generalization

Unary predicates are pretty limited and don't generalize well.

```
bool isMoreThan3(int num) {  
    return num > 3;  
}  
  
bool isMoreThan4(int num) {  
    return num > 4;  
}  
  
bool isMoreThan5(int num) {  
    return num > 5;  
}
```



## Poor Generalization

Unary predicates are pretty limited and don't generalize well.

Ideally, we'd like something like this!

```
bool isMoreThan3(int num) {  
    return num > 3;  
}  
  
bool isMoreThan4(int num) {  
    return num > 4;  
}  
  
bool isMoreThan5(int num) {  
    return num > 5;  
}  
  
// a generalized version of the above  
bool isMoreThan(int num, int limit) {  
    return num > limit;  
}
```

## Can we use binary predicates?


If we could, it would be nice to use a binary predicate to handle this!

```
template <typename InputIt, typename BinPred>
int count_occurrences(InputIt begin, InputIt end, BinPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (pred(*iter, ???)) count++;
    }
    return count;
}
```

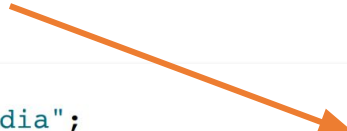
## Can we use binary predicates?

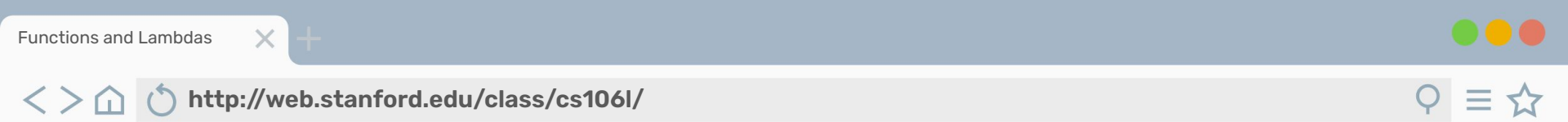
How do we know what value to use? What about unary (or any other number of arguments) predicates?

```
template <typename InputIt, typename BinPred>
int count_occurrences(InputIt begin, InputIt end, BinPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (pred(*iter, ???)) count++;
    }
    return count;
}
```

 **We can't pass this in from the predicate!**

Usage: `std::string str = "Xadia";`  
`count_occurrences(str.begin(), str.end(), isVowel);`





## The Catch-22

We want our function to know more information about our predicate.

## The Catch-22

We want our function to know more information about our predicate.

However, we can't pass in more than one parameter.

## The Catch-22

We want our function to know more information about our predicate.

However, we can't pass in more than one parameter.

How can we pass along information without needing another parameter?

## Let's use lambdas!

Lambdas are inline, anonymous functions that can know about functions declared in their same scope!

```
auto var = [capture-clause] (auto param) -> bool
{
    . . .
}
```

## Let's use lambdas!

Lambdas are **inline**, anonymous functions that can know about variables declared in their same scope!

```
auto var = [capture-clause] (auto param) -> bool
{
    ...
}
```



## Let's use lambdas!

Lambdas are inline, **anonymous** functions that can know about variables declared in their same scope!

```
auto var = [capture-clause] (auto param) -> bool
{
    ...
}
```

## Let's use lambdas!

Lambdas are inline, **anonymous** functions that can know about variables declared in their same scope!

Outside parameters  
go here

Specifies that  
Type is generic

```
auto var = [capture-clause] (auto param) -> bool
{
    ...
}
```

Function body  
goes here!

## Let's use lambdas!

It might look something like this!

```
int limit = 5;  
auto isMoreThan = [limit] (int n) { return n > limit; };  
isMoreThan(6); // true
```

## Let's use lambdas!

It might look something like this!

```
int limit = 5;  
auto isMoreThan = [limit] (int n) { return n > limit; };  
isMoreThan(6); // true
```

## Capture Clauses

You can capture any outside variable, both by reference and by value.

```
[ ]           // captures nothing
[limit]       // captures limit by value
[&limit]      // captures limit by reference
[&limit, upper] // captures limit by reference, upper by value
[&, limit]    // captures everything except limit by reference
[&]          // captures everything by reference
[=]          // captures everything by value
```

## Capture Clauses

You can capture any outside variable, both by reference and by value.

- Use just the = symbol to capture everything by value, and just the & symbol to capture everything by reference

```
[ ]           // captures nothing
[limit]       // captures limit by value
[&limit]      // captures limit by reference
[&limit, upper] // captures limit by reference, upper by value
[&, limit]    // captures everything except limit by reference
[&]           // captures everything by reference
[=]           // captures everything by value
```

## We've solved our problem!

```
template <typename InputIt, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (pred(*iter)) count++;
    }
    return count;
}
```

Usage:

```
int limit = 5;
auto isMoreThan = [limit] (int n) { return n > limit; };
std::vector<int> nums = {3, 5, 6, 7, 9, 13};

count_occurrences(nums.begin(), nums.end(), isMoreThan);
```

## We've solved our problem!

```
template <typename InputIt, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (pred(*iter)) count++;
    }
    return count;
}
```

Usage:

```
int limit = 5;
auto isMoreThan = [limit] (int n) { return n > limit; };
std::vector<int> nums = {3, 5, 6, 7, 9, 13};

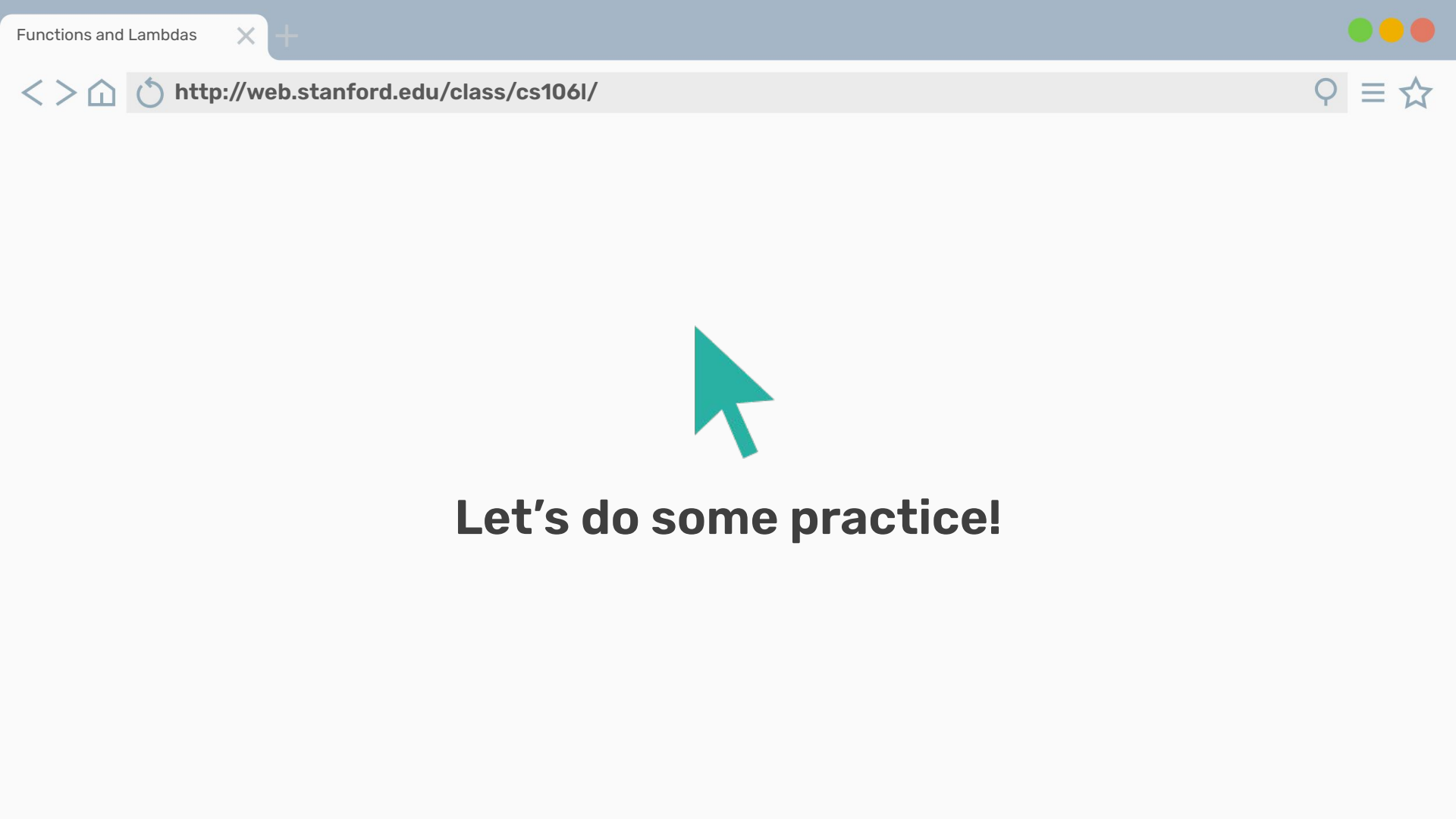
count_occurrences(nums.begin(), nums.end(), isMoreThan);
```



## Using Lambdas

Lambdas are pretty computationally cheap and a great tool!

- Use a lambda when you need a short function or to access local variables in your function.
- If you need more logic or overloading, use function pointers.



**Let's do some practice!**

## Aside: What the Functor?

A **functor** is any class that provides an implementation of `operator()`.

```
class functor {  
public:  
    int operator() (int arg) const { // parameters and function body  
        return num + arg;  
    }  
private:  
    int num; // capture clause  
};  
  
int num = 0;  
auto lambda = [&num] (int arg) { num += arg; };  
lambda(5);
```

## Aside: What the Functor?

A **functor** is any class that provides an implementation of `operator()`.

- They can create **closures** of “customized” functions!

```
class functor {  
public:  
    int operator() (int arg) const { // parameters and function body  
        return num + arg;  
    }  
private:  
    int num; // capture clause  
};  
  
int num = 0;  
auto lambda = [&num] (int arg) { num += arg; };  
lambda(5);
```

## Aside: What the Functor?

A **functor** is any class that provides an implementation of `operator()`.

- They can create **closures** of “customized” functions!

```
class functor {  
public:  
    int operator() (int arg) const { // parameters and function body  
        return num + arg;  
    }  
private:  
    int num; // capture clause  
};  
  
int num = 0;  
auto lambda = [&num] (int arg) { num += arg; };  
lambda(5);
```

Closure: a single instantiation of a functor object

## Aside: What the Functor?

A **functor** is any class that provides an implementation of operator().

- They can create **closures** of “customized” functions!
- Lambdas are just a reskin of functors!

```
class functor {  
public:  
    int operator() (int arg) const { // parameters and function body  
        return num + arg;  
    }  
private:  
    int num; // capture clause  
};  
  
int num = 0;  
auto lambda = [&num] (int arg) { num += arg; };  
lambda(5);
```

Closure: a single instantiation of a functor object



## Tying it all together

So far, we've talked about lambdas, functors, and function pointers.

## Tying it all together

So far, we've talked about lambdas, functors, and function pointers.

The STL has an overarching, standard function object!

```
std::function<return_type (param_types)> func;
```



## Tying it all together

So far, we've talked about lambdas, functors, and function pointers.

The STL has an overarching, standard function object!

```
std::function<return_type (param_types)> func;
```

Everything (lambdas, functors, function pointers) can be cast to a standard function!

## Tying it all together

So far, we've talked about lambdas, functors, and function pointers.

The STL has an **overarching, standard function object!**

```
std::function<return_type (param_types)> func;
```

Everything (lambdas, functors, function pointers) can be cast to a standard function!



**Much bigger and more expensive than a function pointer or lambda!**

## Aside: Virtual Functions

Be careful using function pointers with classes, especially if you have a subclass of another class!

```
class Animal {  
    // constructors and other methods go here!  
    void speak() {  
        std::cout << "I'm an animal!" << std::endl;  
    } // private information and the rest of the class goes here!  
}  
  
class Dog : Animal { // this syntax tells us we're a subclass of Animal!  
    // constructors and private information here!  
    void speak() {  
        std::cout << "I'm an animal!" << std::endl;  
    } // private information and the rest of the class goes here!  
}
```

## Aside: Virtual Functions

What happens if we try to pass a Dog object to a function that expects an Animal?

```
void func(Animal* animal) { // can take in any animal and make it speak!
    animal->speak();
}

int main() {
    Animal* myAnimal = new Animal;
    Dog* myDog = new Dog;
    func(myAnimal);
    func(myDog);
}
```

## Aside: Virtual Functions

What happens if we try to pass a Dog object to a function that expects an Animal?

```
void func(Animal* animal) { // can take in any animal and make it speak!
    animal->speak();
}

int main() {
    Animal* myAnimal = new Animal;
    Dog* myDog = new Dog;
    func(myAnimal); \\ I'm an animal!
    func(myDog);    \\ I'm an animal!
}
```

The function expects an Animal, so it will try to use the Animal speak function! It doesn't know it's been overridden!



## Aside: Virtual Functions

If you have a function that can take in a pointer to the superclass, it won't know to use the subclass's function!

## Aside: Virtual Functions

If you have a function that can take in a pointer to the superclass, it won't know to use the subclass's function!

The same issue happens if we create a superclass pointer to an existing subclass object.

## Aside: Virtual Functions

If you have a function that can take in a pointer to the superclass, it won't know to use the subclass's function!

The same issue happens if we create a superclass pointer to an existing subclass object.

To fix this, we can mark the overridden function as **virtual** in the header!



## Aside: Virtual Functions

If you have a function that can take in a pointer to the superclass, it won't know to use the subclass's function!

The same issue happens if we create a superclass pointer to an existing subclass object.

To fix this, we can mark the overridden function as **virtual** in the header!

Virtual functions are functions in the superclass we expect to be overridden later on.

## Aside: Virtual Functions

Let's change Animal to have a virtual implementation of speak()!

```
class Animal {  
    // constructors and other methods go here!  
    virtual void speak() {  
        std::cout << "I'm an animal!" << std::endl;  
    } // private information and the rest of the class goes here!  
}  
  
class Dog : Animal { // this syntax tells us we're a subclass of Animal!  
    // constructors and private information here!  
    void speak() {  
        std::cout << "I'm an animal!" << std::endl;  
    } // private information and the rest of the class goes here!  
}
```

## Aside: Virtual Functions

Let's change Animal to have a virtual implementation of speak()!

```
void func(Animal* animal) { // can take in any animal and make it speak!
    animal->speak();
}

int main() {
    Animal* myAnimal = new Animal;
    Dog* myDog = new Dog;
    func(myAnimal); \\ I'm an animal!
    func(myDog);    \\ I'm a dog!
}
```

**Now calling speak() will use  
the correct subclass version!**



 <http://web.stanford.edu/class/cs106l/>



## CONTENTS



### 01. Recap: Template Functions



### 02. Functions and Lambdas

Passing input outside of parameters

### 03. Algorithms



# Coding Philosophy 101

There are few universal, scientifically proven  
pieces of wisdom that will lead to a happier life:

# Coding Philosophy 101

There are few universal, scientifically proven pieces of wisdom that will lead to a happier life:

1. Look both ways before crossing the street.



# Coding Philosophy 101

There are few universal, scientifically proven pieces of wisdom that will lead to a happier life:

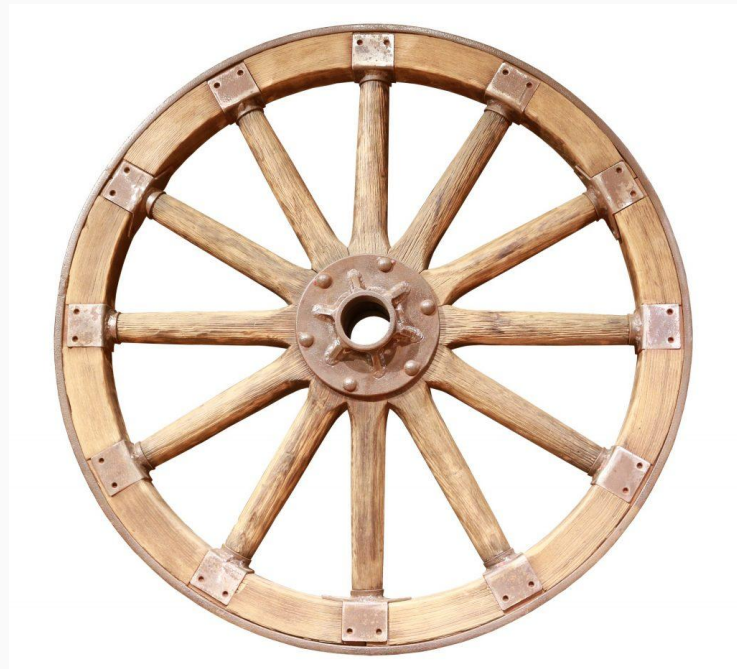
1. Look both ways before crossing the street.
2. Never tell a pre-med you're stressed.



## Coding Philosophy 101

There are few universal, scientifically proven pieces of wisdom that will lead to a happier life:

1. Look both ways before crossing the street.
2. Never tell a pre-med you're stressed.
3. When coding, never reinvent the wheel.





# New toys!

The STL implements an entire library of algorithms written by C++ developers!

- To utilize, `#include <algorithm>` in your file!
- All algorithms are **fully generic, templated** functions!

```

Constrained algorithms and algorithms on ranges (C++20)
Constrained algorithms, e.g. ranges::copy, ranges::sort, ...
Execution policies (C++17)
execution::seq (C++17) execution::sequenced_policy
execution::par (C++17) execution::parallel_policy
execution::par_unseq (C++17) execution::parallel_unsequenc
execution::unseq (C++20) execution::parallel_unsequenc
is_execution_policy (C++17)

Non-modifying sequence operations
all_of (C++11) count search
any_of (C++11) count_if search_n
none_of (C++11) mismatch lexicographical_compare
for_each equal lexicographical_compare_three
for_each_n (C++17) adjacent_find

Modifying sequence operations
copy fill remove
copy_if (C++11) fill_n remove_if
copy_n (C++11) generate replace
copy_backward generate_n replace_if
move (C++11) swap reverse
move_backward (C++11) iter_swap rotate
shift_left (C++20) swap_ranges unique
shift_right (C++20) sample (C++17) random_shuffle (until C++17)
transform

Partitioning operations
is_partitioned (C++11) partition stable_partition
partition_point (C++11) partition_copy (C++11)

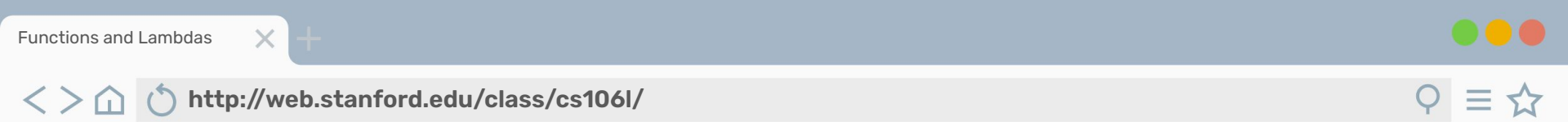
Sorting operations
is_sorted (C++11) sort partial_sort
is_sorted_until (C++11) stable_sort partial_sort_copy

Binary search operations
lower_bound upper_bound binary_search

Set operations (on sorted ranges)
merge set_difference set_symmetric_difference
inplace_merge set_intersection set_union

Heap operations

```



## What kind of algorithms?

With the algorithm library, we can...

```
#include <algorithm>:
```

## What kind of algorithms?

With the algorithm library, we can...

- check if a condition is true across any elements

```
#include <algorithm>:
```

```
any_of  all_of  none_of
```

## What kind of algorithms?

With the algorithm library, we can...

- check if a condition is true across any elements
- apply a function to all elements in a container

```
#include <algorithm>:  
  
any_of    all_of    none_of  
  
for_each
```



## What kind of algorithms?

With the algorithm library, we can...

- check if a condition is true across any elements
- apply a function to all elements in a container
- search for specific elements or a range

```
#include <algorithm>:
```

```
any_of  all_of  none_of
```

```
for_each
```

```
find      search
```

## What kind of algorithms?

With the algorithm library, we can...

- check if a condition is true across any elements
- apply a function to all elements in a container
- search for **specific elements** or a **range**

```
#include <algorithm>:
```

```
any_of  all_of  none_of
```

```
for_each
```

```
find
```

```
search
```

## What kind of algorithms?

With the algorithm library, we can...

- check if a condition is true across any elements
- apply a function to all elements in a container
- search for specific elements or a range
- copy, remove, add elements from one container to another

```
#include <algorithm>:  
  
any_of  all_of  none_of  
  
        for_each  
  
        find      search  
  
        copy
```

## What kind of algorithms?

With the algorithm library, we can...

- check if a condition is true across any elements
- apply a function to all elements in a container
- search for specific elements or a range
- copy, remove, add elements from one container to another
- and much, much more!

```
#include <algorithm>:
```

```
any_of  all_of  none_of
```

```
for_each
```

```
find      search
```

```
copy
```



## Look familiar?

**count\_occurrences**

```
template <typename InputIt, typename UniPred>  
int count_occurrences(InputIt begin, InputIt end, UniPred pred);
```

## Look familiar?

**count\_occurrences**

```
template <typename InputIt, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred);
```

**std::count\_if**

```
template< class InputIt, class T >
typename iterator_traits<InputIt>::difference_type
count( InputIt first, InputIt last, const T& value );
```

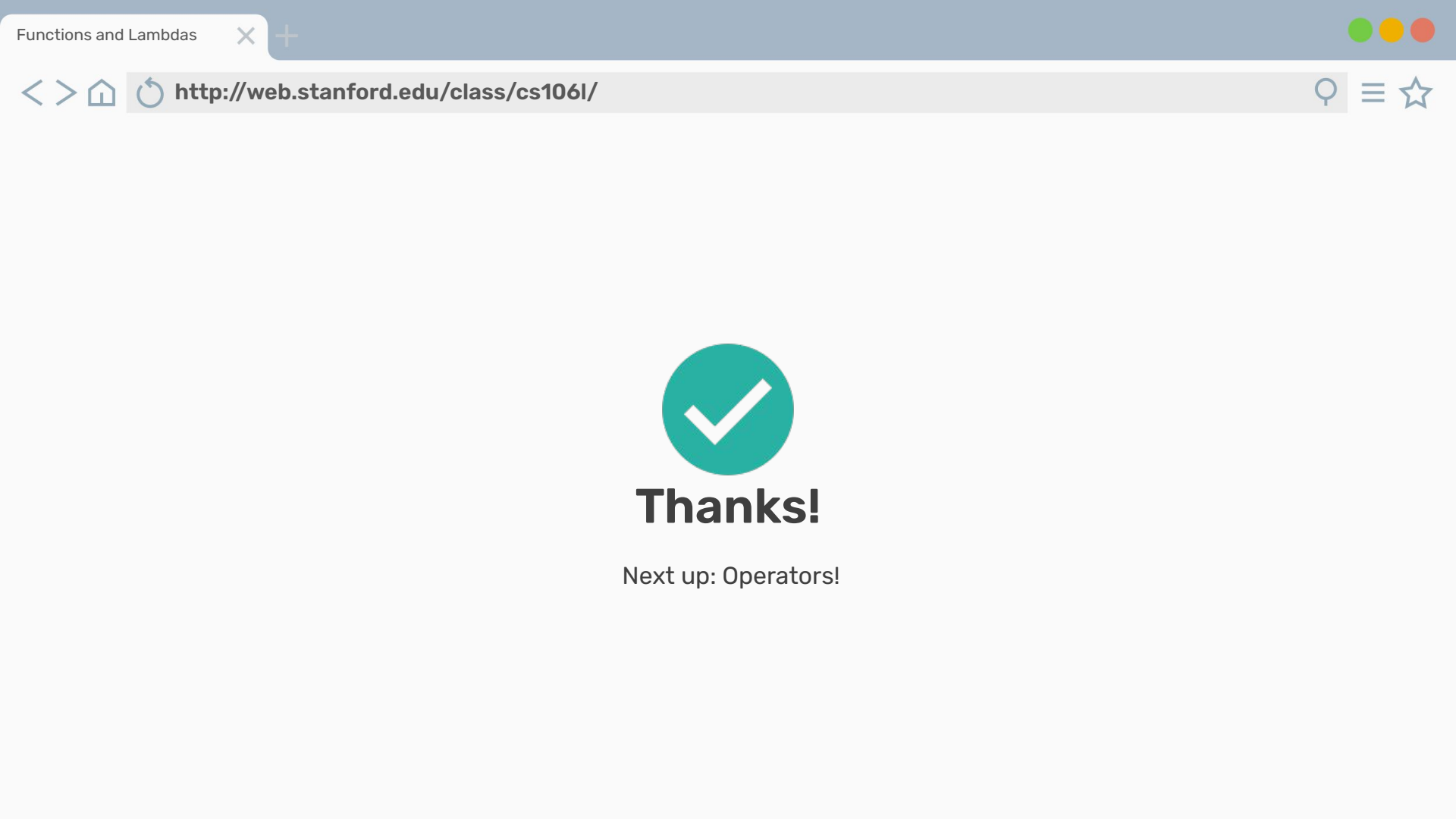
# Algorithms

All standard algorithms work on iterators.

- Efficient searching, sorting, complex data structure operations, smart pointers, and more are all there for you to use!
- Check out the documentation to get more information!

## Summary

- Lambda functions are inline functions that let you pass outside variables in using capture clauses!
- Lambdas can be used to pass predicate function pointers to template functions for more generalizability.
- The STL implements tons of cool algorithms that we can use without rewriting them!



**Thanks!**

Next up: Operators!