



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



Template Functions

What else in C++ can be generalized? What is the philosophy behind generalization?

CS106L - Fall 22



Attendance!

<https://bit.ly/3gE1jWx>





Agenda



01. Recap: Iterators & Template Classes



02. Generic Programming

A coding philosophy

03. Template Functions

Type deduction, lvalues and rvalues, and metaprogramming

04. Practice!

Motivating our talk on lambdas next time!



Agenda



01. Recap: Iterators & Template Classes



02. Generic Programming

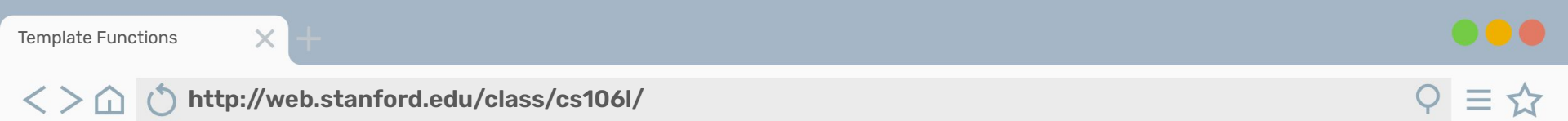
A philosophy

03. Template Functions

Type deduction, lvalues and rvalues, and metaprogramming

04. Practice!

Motivating our talk on lambdas next time!



Review: Iterators

Containers all implement something called an iterator to do this!

- Iterators let you access **all** data in **all** containers programmatically!
- An iterator has a certain **order**; it “knows” what element will come next
 - Not necessarily the same each time you iterate!

Review: Iterators

All containers implement iterators, but they're not all the same!

- Each container has its own iterator, which can have different behavior.
- All iterators implement a few shared operations:
 - Initializing → `iter = s.begin();`
 - Incrementing → `++iter;`
 - Dereferencing → `*iter;`
 - Comparing → `iter != s.end();`
 - Copying → `new_iter = iter;`

Review: Iterators

```
for ( auto iter=set.begin() ; iter != set.end(); ++iter ) {
```

Now we can access each element individually!

If we want the element and not just a reference to it, we dereference (*iter).

```
const auto& elem = *iter;
```

Review: Template Classes

- Add `template<typename T1, typename T2...>` before class definition in .h
- Add `template<typename T1, typename T2...>` before all function signature in .cpp
- When returning nested types (like iterator types), put `typename ClassName<T1, T2...>::member_type` as return type, not just `member_type`
- Templates don't emit code until instantiated, so `#include` the .cpp file in the .h file, not the other way around!

Review: Const and Const Correctness

- Use const parameters and variables wherever you can in application code
- Every member function of a class that doesn't change its member variables should be marked `const`
- `auto` will drop all const and `&`, so be sure to specify
- Make iterators and `const_iterators` for all your classes!
 - `const_iterator` = cannot increment the iterator, can dereference and change underlying value
 - `const_iterator` = can increment the iterator, cannot dereference and change underlying value
 - `const const_iterator` = cannot increment iterator, cannot change underlying value



Agenda



01. Recap: Iterators & Template Classes



02. Generic Programming

A philosophy

03. Template Functions

Type deduction, lvalues and rvalues, and metaprogramming

04. Practice!

Motivating our talk on lambdas next time!



Why do we want generic C++?

C++ is strongly typed, but generic C++ lets you parametrize data types!

Why do we want generic C++?

C++ is strongly typed, but generic C++ lets you parametrize data types!

- Ex. variable return type or input in a class (template classes)

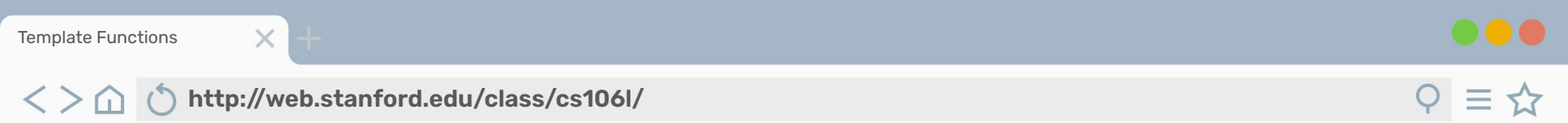
Why do we want generic C++?

C++ is strongly typed, but generic C++ lets you parametrize data types!

- Ex. variable return type or input in a class (template classes)

Can we parametrize even more?

Can we write a function that works on **any data type**?



Why not!

Let's say we want a function to return the min of two ints!



Why not!

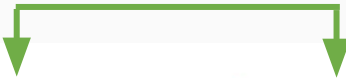
Let's say we want a function to return the min of two ints!

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

Why not!

Let's say we want a function to return the min of two ints!

We take in two
ints...




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


Why not!

Let's say we want a function to return the min of two ints!

We take in two
ints...

And return an
int!



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

Why not!

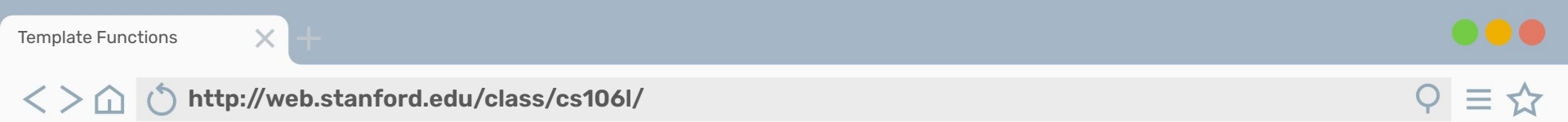
Let's say we want a function to return the min of two ints!

We take in two
ints...

And return an
int!

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

What about doubles? Floats? Longs?



What about function overloading?

Sure, we
could...

What about function overloading?

Sure, we
could...

```
int myMin(int a, int b) {  
    return a < b ? a : b;  
}  
  
// exactly the same except for types  
std::string my_min(std::string a, std::string b) {  
    return a < b ? a : b;  
}  
  
int main() {  
    auto min_int = myMin(1, 2);           // 1  
    auto min_name = myMin("Sarah", "Haven"); // Haven  
}
```

What about function overloading?

Sure, we
could...

**What about
other types?**

```
int myMin(int a, int b) {  
    return a < b ? a : b;  
}  
  
// exactly the same except for types  
std::string my_min(std::string a, std::string b) {  
    return a < b ? a : b;  
}  
  
int main() {  
    auto min_int = myMin(1, 2);           // 1  
    auto min_name = myMin("Sarah", "Haven"); // Haven  
}
```



Agenda



01. Recap: Iterators & Template Classes



02. Generic Programming

A philosophy

03. Template Functions

Type deduction, lvalues and rvalues, and metaprogramming

04. Practice!

Motivating our talk on lambdas next time!



Template functions are completely generic functions!

Just like classes, they work regardless of type!

Let's break it down:

Template functions are completely generic functions!

Just like classes, they work regardless of type!

Let's break it down:

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


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;
}
```

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

The class keyword is interchangeable!

List of your template variables

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

Default Types

We can define default parameter types!

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

Default Types

We can define default parameter types!

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

What does it look like to use a template function?

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
```

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
```

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;
}
```

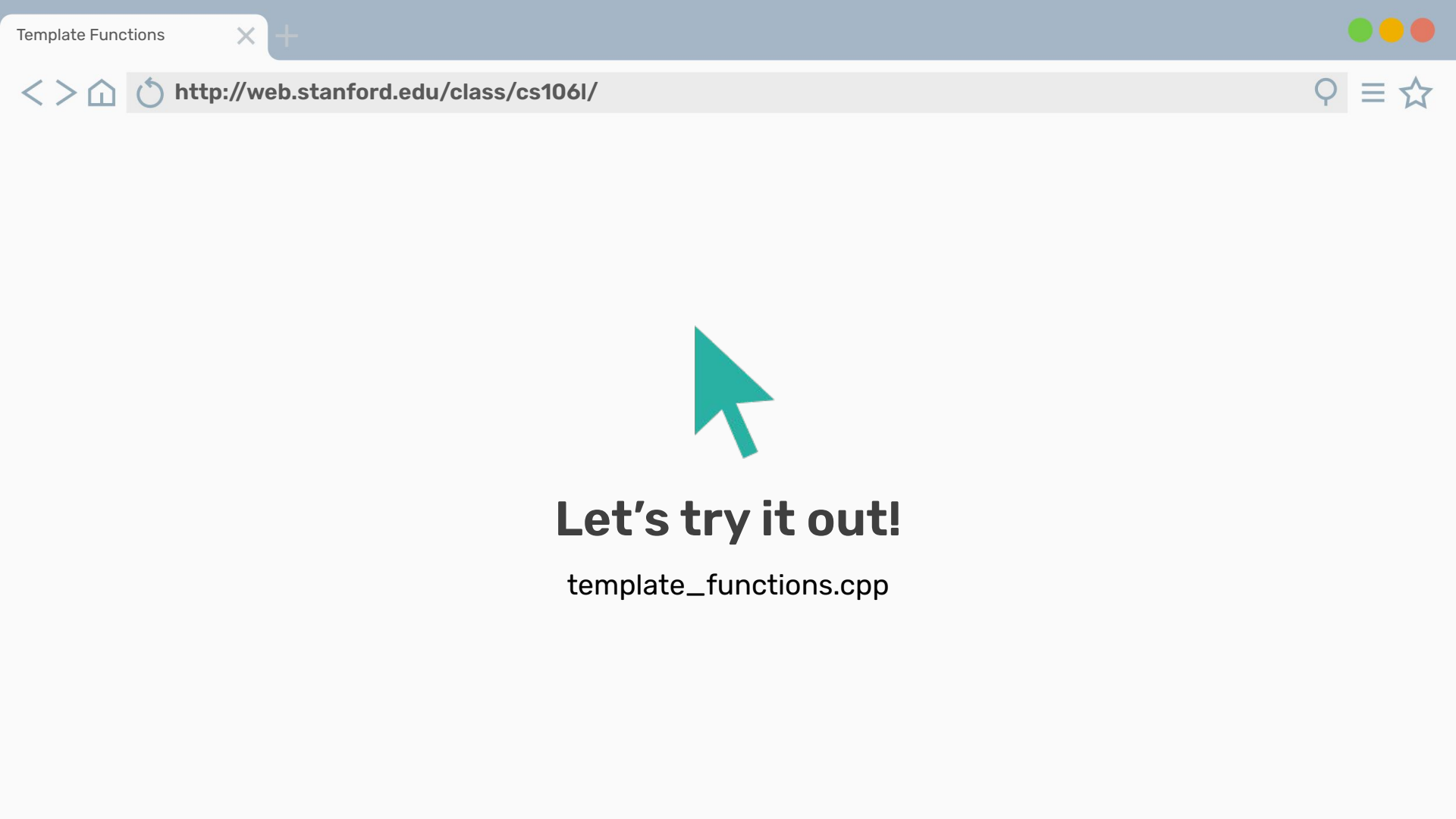
```
// int main() {} will be omitted from future examples
// we'll instead show the code that'd go inside it
cout << myMin(3.2, 4) << endl; // 3.2
```


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
```



Let's try it out!

template_functions.cpp



Behind the Instantiation Scenes

Remember: like in template classes, **template functions are not compiled until used!**

Behind the Instantiation Scenes

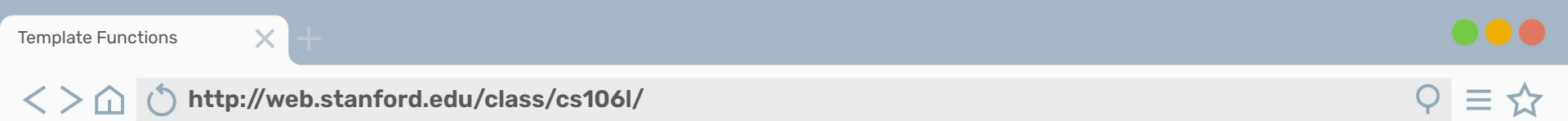
Remember: like in template classes, **template functions are not compiled until used!**

- For each instantiation with different parameters, the compiler generates a new specific version of your template

Behind the Instantiation Scenes

Remember: like in template classes, **template functions are not compiled until used!**

- For each instantiation with different parameters, the compiler generates a new specific version of your template
- After compilation, it will look like you wrote each version yourself



Wait a minute...

The code doesn't exist until you instantiate it, which runs quicker.

Can we take advantage of this behavior?

Wait a minute...

The code doesn't exist until you instantiate it, which runs quicker.

Can we take advantage of this behavior?



Wait a minute...

The code doesn't exist until you instantiate it, which runs quicker.

Can we take advantage of this behavior?

Yes.





Agenda



01. Recap: Iterators & Template Classes



02. Generic Programming

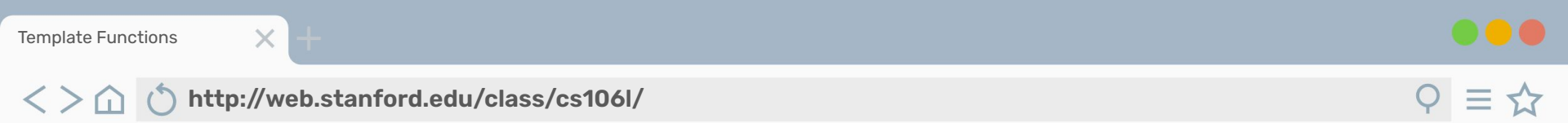
A philosophy

03. Template Functions

Type deduction, lvalues and rvalues, and a demo

04. Template Metaprogramming

Gaming the system



Templates can be used for efficiency!

Normally, code runs during **runtime**.

Templates can be used for efficiency!

Normally, code runs during **runtime**.

With template metaprogramming, code runs **once** during
compile time!

Templates can be used for efficiency!

Normally, code runs during **runtime**.

With template metaprogramming, code runs **once** during **compile time**!

```
template<unsigned n>
struct Factorial {
    enum { value = n * Factorial<n - 1>::value };
};

template<> // template class "specialization"
struct Factorial<0> {
    enum { value = 1 };
};

std::cout << Factorial<10>::value << endl; // prints 3628800, but run during compile time!
```

How?

```
template<unsigned n>
struct Factorial {
    enum { value = n * Factorial<n - 1>::value };
};

template<> // template class "specialization"
struct Factorial<0> {
    enum { value = 1 };
};

std::cout << Factorial<10>::value << endl; // prints 3628800, but run during compile time!
```

How?

```
template<unsigned n>
struct Factorial {
    enum { value = n * Factorial<n - 1>::value };
};

template<> // template class "specialization"
struct Factorial<0> {
    enum { value = 1 };
};

std::cout << Factorial<10>::value << endl; // prints 3628800, but run during compile time!
```

How?

```
template<unsigned n>
struct Factorial {
    enum { value = n * Factorial<n - 1>::value };
};

template<> // template class "specialization"
struct Factorial<0> {
    enum { value = 1 };
};

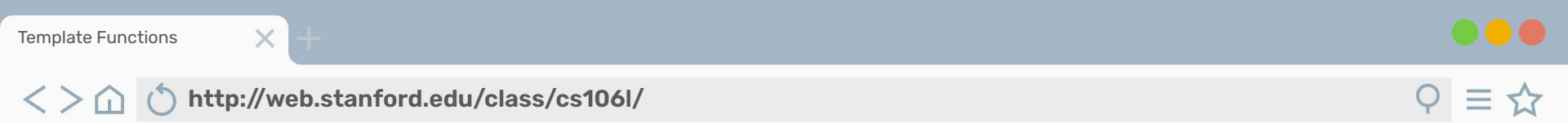
std::cout << Factorial<10>::value << endl; // prints 3628800, but run during compile time!
```

How?

```
template<unsigned n>
struct Factorial {
    enum { value = n * Factorial<n - 1>::value };
};

template<> // template class "specialization"
struct Factorial<0> {
    enum { value = 1 };
};

std::cout << Factorial<10>::value << endl; // prints 3628800, but run during compile time!
```

Why?

Overall, can increase performance for these pieces!

- Compiled code ends up being smaller



Why?

Overall, can increase performance for these pieces!

- Compiled code ends up being smaller
- Something runs once during compiling and can be used as many times as you like during runtime

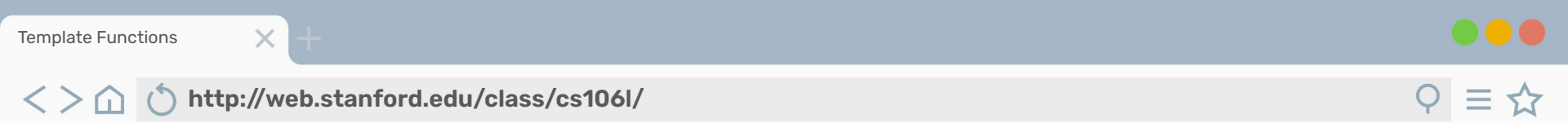


Why?

Overall, can increase performance for these pieces!

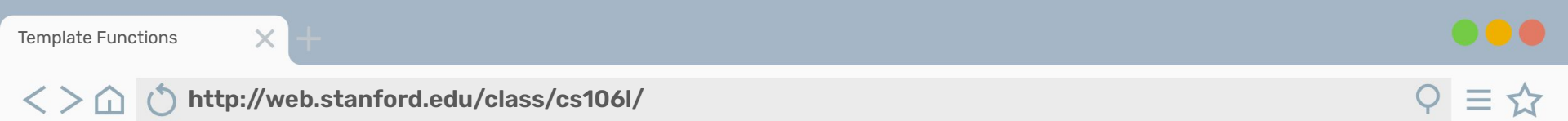
- Compiled code ends up being smaller
- Something runs once during compiling and can be used as many times as you like during runtime

TMP was an accident; it was discovered, not invented!



Applications of TMP

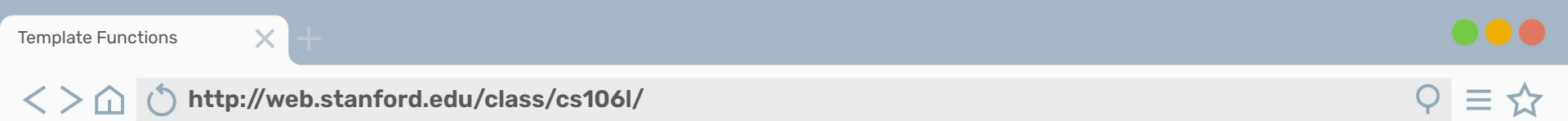
TMP isn't used that much, but it has some interesting implications:



Applications of TMP

TMP isn't used that much, but it has some interesting implications:

- Optimizing matrices/trees/other mathematical structure operations



Applications of TMP

TMP isn't used that much, but it has some interesting implications:

- Optimizing matrices/trees/other mathematical structure operations
- Policy-based design



Applications of TMP

TMP isn't used that much, but it has some interesting implications:

- Optimizing matrices/trees/other mathematical structure operations
- Policy-based design
- Game graphics



Solving problems with generics

What if we wanted to count all the occurrences of a character in a string?

Solving problems with generics

What if we wanted to count all the occurrences of a character in a string?

Or a number in a vector?



Solving problems with generics

What if we wanted to count all the occurrences of a character in a string?

Or a number in a vector?

Or a word in a stream?

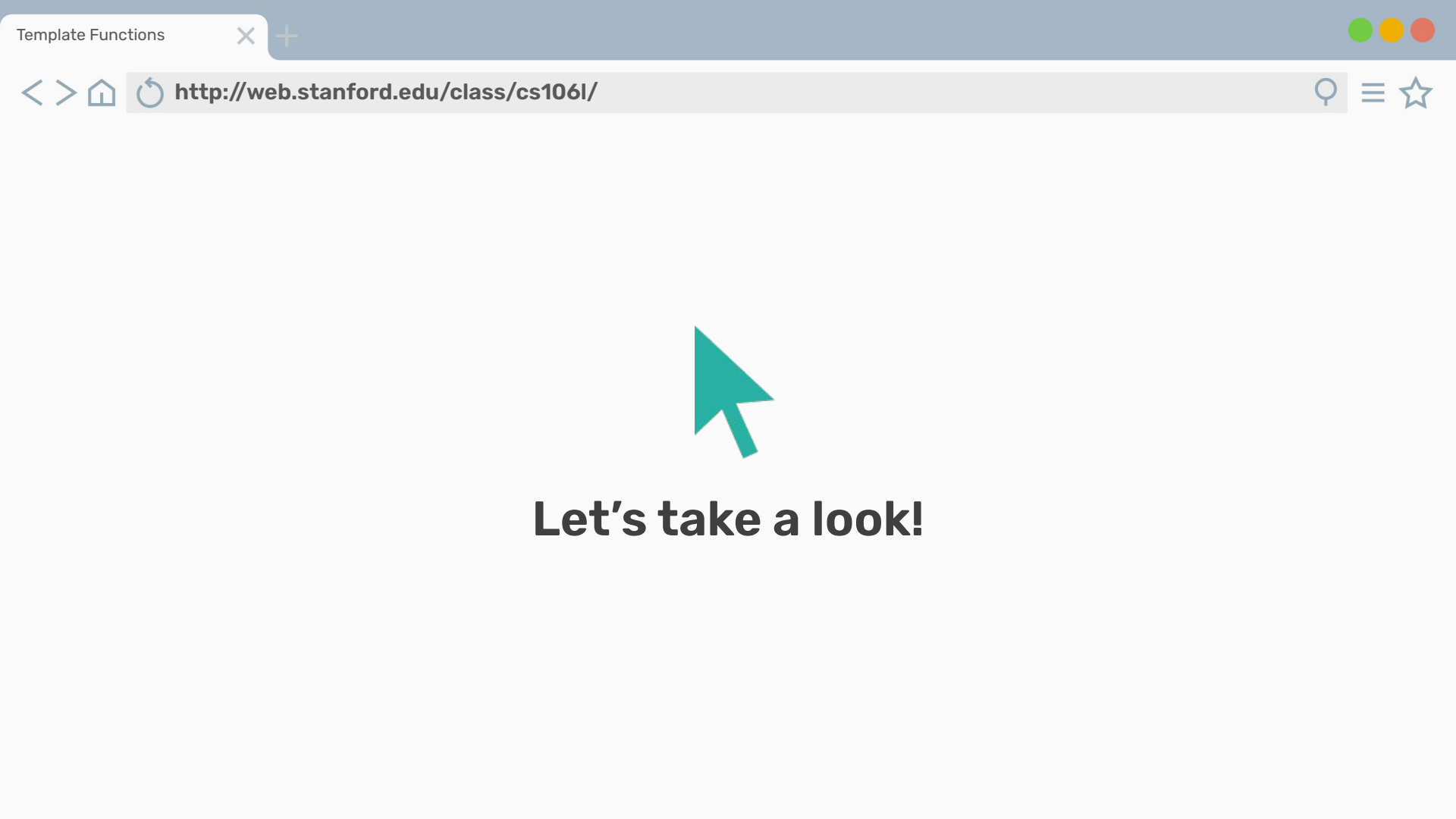
Solving problems with generics

What if we wanted to count all the occurrences of a character in a string?

Or a number in a vector?

Or a word in a stream?

These are all the same problem!



Let's take a look!

Summary

- 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



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



Thanks!

Next up: Functions and Lambdas!