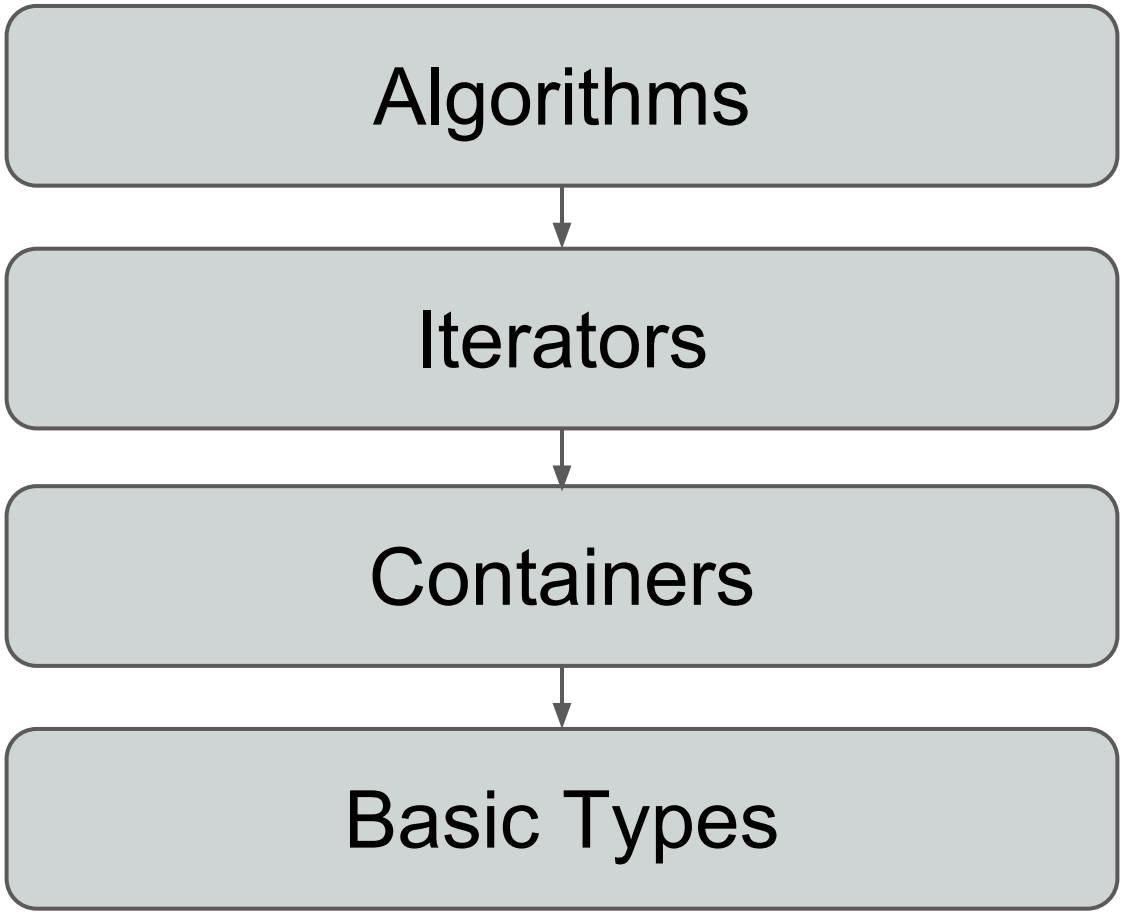
Functors

Reid Watson
(rawatson@stanford.edu)

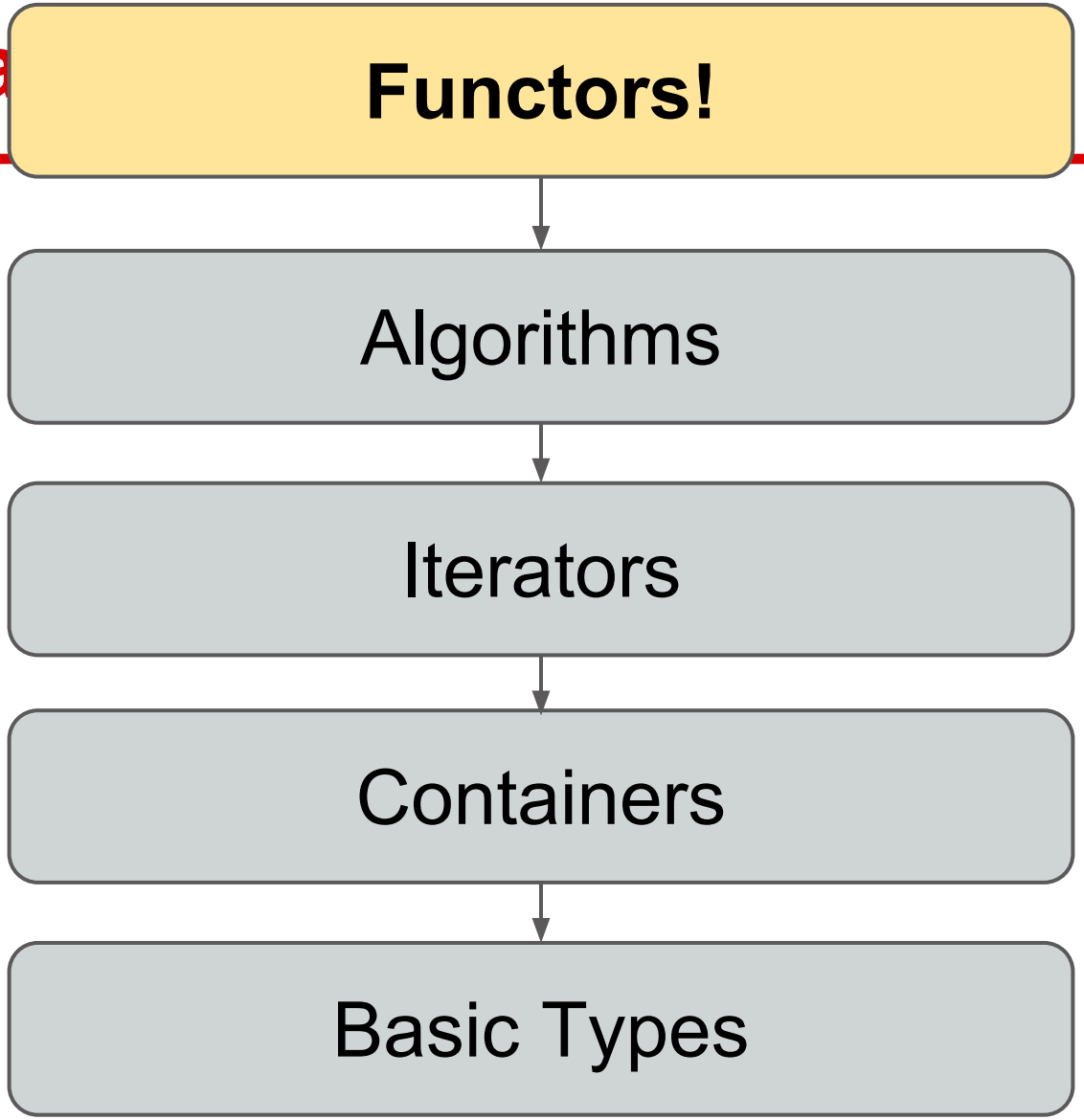
Administrivia

- We’ve started grading Evil Hangman
- KDTree is going out tonight!

Abstraction in the STL



Abstra



Functors

Functions in C++ take in parameters and produce a return value

... but you probably already knew that

Functors

- A **functor** is like a function, but it can do a bit more.
- Functors are *classes*, not functions
- Functors define **operator()**

Functors

```
int ExampleFunction(string s) {
   cout << s << endl;
   return 42;
}

struct ExampleFunctor {
   int operator()(string s) {
      cout << s << endl;
      return 42;
   }
};
```

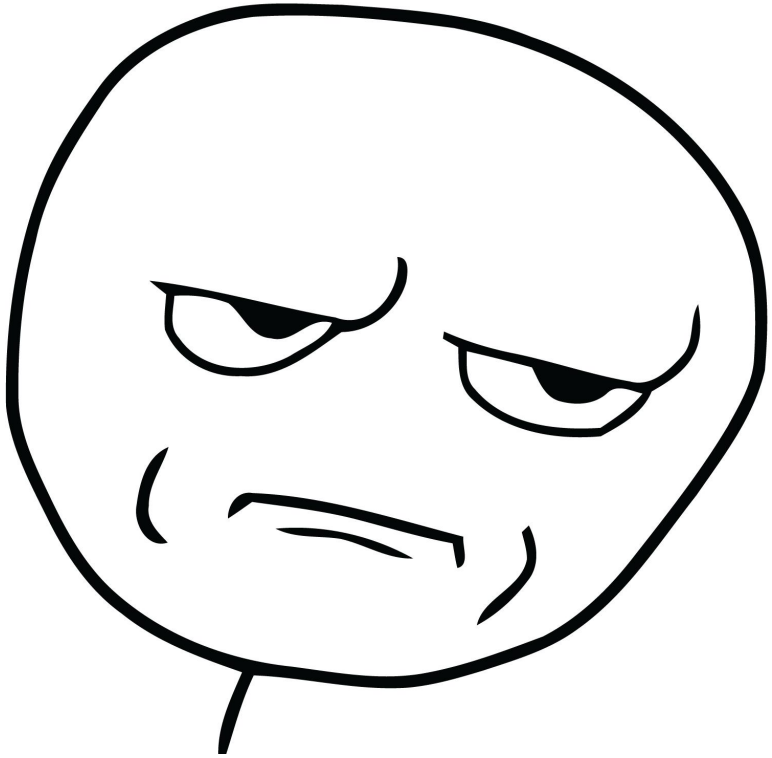
Functors

```
// Call a function
int a = ExampleFunction("Hello world!");

// Call a functor
ExampleFunctor f;
int b = f("Hello world!"); // or...
b = f.operator()("Hello world!");

// It's the same stuff!
assert(a == b);
```

Functors



That's totally pointless...

Functors

- There's more to functors than an awkward syntax for defining functions!
- Functors enable us to emulate **closures**, a very powerful concept in computer science
- Let's look at a basic example of using functors to count words of a certain size in a dictionary

Functors

The STL gives us a convenient way to count elements in a container using iterators and **unary predicate functions**, functions which take a single argument and return a boolean

Functors

Let's say that we needed to find the number of words of length five in the dictionary.

We can use the STL to solve this in a very simple manner.

Functors

See code in words-start.cpp

Functors

- Let's say the requirements change and we now need to read a word size from the user, and count words of that size
- Do we have to start over?
- Can we still use the `count_if` function?

Functors

See code in words.cpp

Functors

So, what separates functions and functors then, other than the syntax?

Functors

Functions have:

- Local vars
- Parameters
- Global vars

Functors have:

- Local vars
- Parameters
- Global vars
- **Instance vars**

We can use this tiny little difference to enable an entire world of functionality

Functors

- This is why functors are so powerful
- Defining one functor can define an infinite number of functions
- Functors can remember information about the context in which they were called (emulating closures)
- Functors can remember information between calls to the same function

Using Functors

```
template <typename InputIter, typename CriteriaFn>
size_t count_if(InputItr begin, InputItr end, CriteriaFn crit) {
    size_t count = 0;
    while (begin != end) {
        if (crit(*begin))
            ++count;
        ++begin;
    }
    return count;
}
```

Using Functors

```
template <typename InputIter, typename CriteriaFn>
size_t count_if(InputItr begin, InputItr end, CriteriaFn crit) {
    size_t count = 0;
    while (begin != end) {
        if (crit(*begin))
            ++count;
        ++begin;
    }
    return count;
}
```

Using Functors

- CriteriaFn could be either a function or a functor
- CriteriaFn must return a boolean (or something that can be implicitly converted to a boolean).

Uses of Functors

There are a couple common conventions used for functors in the STL:

- **Unary Predicate Function**: Takes a single argument, returns a boolean
- **Comparison Function:** Takes two arguments, returns true if the first should be "ordered" before the second
- **Unary Operation**: Takes a single argument and returns a value of any type

Uses of Functors

Unary Predicate Functions are used when considering only a certain type of element for some operation:

- `all_of, any_of, none_of`
- `find_if, count_if, copy_if, replace_if, ...`

Uses of Functors

Comparison Functions are used to define an **ordering** on elements. This will be useful whenever there's a notion of elements being less than one another:

- `sort, partial_sort, is_sorted`
- `min, max_element, min_element`
- `lexicographical_compare`

Uses of Functors

Unary Operations are used whenever you want to apply a single operation a single element

- `transform`
- `for_each` (return value isn't used)

Functors and Closures

Think back to our words-of-a-certain-size counting example.

We had to do a lot of work to accomplish something fairly simple.

Functors and Closures

Incoming C++11 funtimes!

Functors and Closures

There's a lot of boilerplate involved in defining a functor we're only going to use once.

Functors and Closures

C++11 introduces the idea of "lambdas", which allow programmers to define small functors with very little effort.

Let's try this out in our code...

Functors and Closures

Let's look at that one crazy line of code...

```
// So much magic!
auto SizeFn = [length](const string& word)
{return word.size() == length;};

// Respaced, becomes...
auto SizeFn = [length](const string& word){
   return word.size() == length;
};
```

Functors and Closures

```
// Respaced, becomes...
auto SizeFn = STUFF;
```

We're creating a variable called `SizeFn`, but I'm not going to say what its type is, the compiler is going to figure that out for me.

Functors and Closures

```
auto SizeFn = [length](const string& word){
   return word.size() == length;
};
```

We want the variable length to be accessible inside the body of our lambda

Functors and Closures

```
auto SizeFn = [length](const string& word){
   return word.size() == length;
};
```

`SizeFn` can be called as a function with one parameter, which will be a const reference to a string.

Functors and Closures

```
auto SizeFn = [length](const string& word){
   return word.size() == length;
};
```

This works just like a regular function body -- you can do whatever you like!

Functors and Closures

We've reduced our class, member variable, and member function into a single line of code!