# Heavy Lifting

## Caphyon Lightning Talks

Victor Ciura | @ciura_victor

Principal Engineer, CAPHYON

February 2020

# Type Constructors

There are various ways to hide a value:

- `unique_ptr<T> p;`
- `shared_ptr<T> p;`
- `vector<T> v;`
- `optional<T> o;`
- `function<T(int)> f;`

Access the value within:

- `*p | p.get()`
- `*p | p.get()`
- `v[0] | *v.begin()`
- `*o | o.value()`
- `f(5)`

# Functor

Performing actions on the *hidden* value, without breaking the BOX.

# Example

Calling the a function on the std::string value inside the **std::optional** box.

```
string capitalize(string str);
...
optional<string> str = ...; // from an operation that could fail
string cap;
if (str)
  cap = capitalize(str.value()); // capitalize(*str);
```

# Example

Calling the a function on the `std::string` value inside the **std::optional** box.

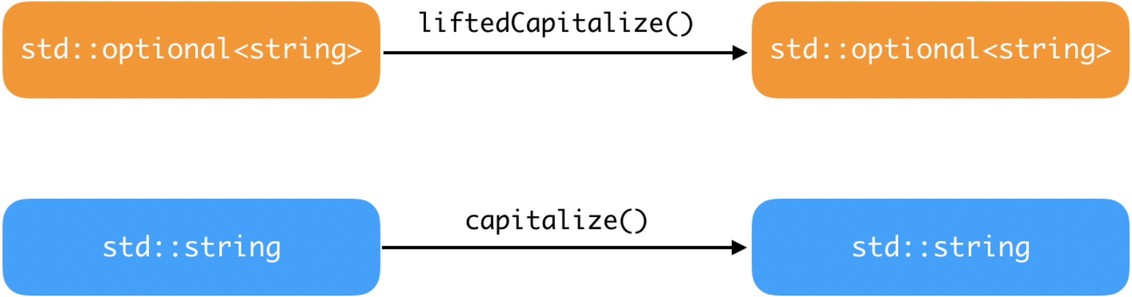
```
string capitalize(string str);
...
optional<string> str = ...; // from an operation that could fail
optional<string> cap;
if (str)
  cap = capitalize(str.value()); // capitalize(*str);
```

# Lifting capitalize()

Lifted capitalize() operates on optional<string> and produces optional<string>

```
optional<string> liftedCapitalize(const optional<string> & s)
{
  optional<string> result;
  if (s)
    result = capitalize(*s);
  return result;
}
```

# Lifting capitalize()



std::optional<string>

liftedCapitalize()

std::optional<string>

std::string

capitalize()

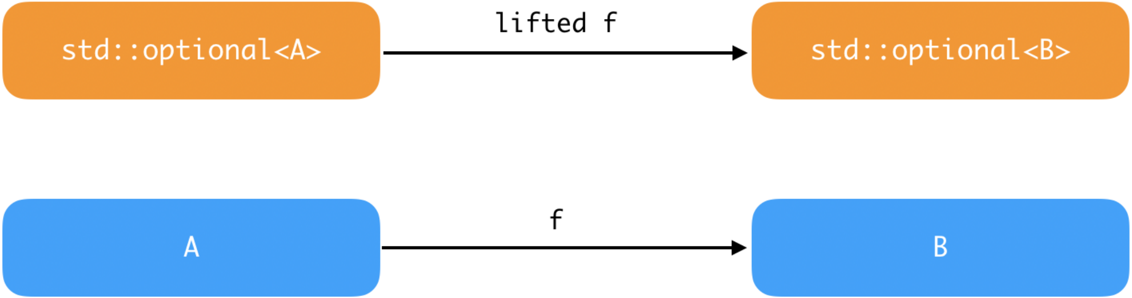
std::string

# Lifting any function

Lifted f operates on optional<A> and produces optional<B>

```
template<class A, class B>
optional<B> fmap(function<B(A)> f, const optional<A> & o)
{
  optional<B> result;
  if (o)
    result = f(*o); // wrap a <B>
  return result;
}
```

# Lifting any function



std::optional<A>

lifted f

std::optional<B>

A

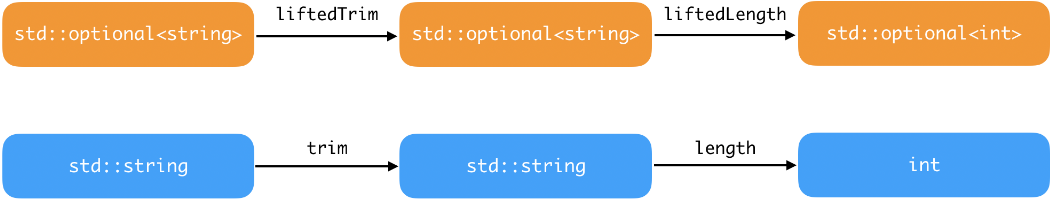
f

B

# Composition of lifted functions

The real power of lifted functions shines when composing functions.

```
optional<string> str{" Some text "};
auto len = fmap<string, int>(&length,
                             fmap<string, string>(&trim, str);
```



# Lifting any function (take 2)

```
template<typename T, typename F>
auto fmap(const optional<T> & o, F f) -> decltype( f(o.value()) )
{
  if (o)
    return f(o.value());
  else
    return {}; // std::nullopt
}
```

# Composition Example

Let’s build a symbol table for a debugged program.

```
optional<int64_t> current_pc = ... ; // function address
...
optional<string> debug_location()
{
  if (!current_pc)
    return {};

  const auto function = dsym::load_symbol(current_pc.value());
  if (!function)
    return {};

  return dsym::to_string(function.value()); // function name
}
```

# Composition Example (take 2)

Let's build a symbol table for a debugged program.

```
optional<int64_t> current_pc = ... ; // function address
...
optional<string> debug_location()
{
  return fmap(
    fmap(current_pc, dsym::load_symbol),
    dsym::to_string
  );
}
```

# Composition Example (take 3)

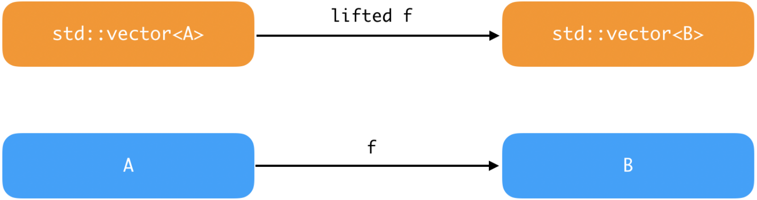
We could create an `fmap` transformation that has the pipe | syntax as ranges:

```
optional<int64_t> current_pc = ... ; // function address
...
optional<string> debug_location()
{
  return current_pc
          | fmap(dsym::load_symbol)
          | fmap(dsym::to_string);
}
```

# Lifting a function to a vector

Lifted f operates on vector<A> and produces vector<B>

```
template<class A, class B>
vector<B> fmap(function<B(A)> f, vector<A> v)
{
  vector<B> result;
  std::transform(v.begin(), v.end(), back_inserter(result), f);
  return result;
}
```



# Lifting a function to a vector

Lifted `length` operates on `vector<string>` and produces `vector<int>`

```
vector<string> names{...};
vector<int> lengths = fmap<string, int>(&length, names);
```

# Functor

- Type constructor
  - create a **box** type that wraps another type
  - encapsulates the values of another type into a *context*
- Function lifting
  - create a *higher-order* function (fmap)
  - for any function `A->B` create a function `box<A> -> box<B>`
- Why?
  - no need to break encapsulation
  - better composition (chaining)