Lambdas from First Principles

A Whirlwind Tour of C++

Plain old functions

```
int plus1(int x)
{
    return x+1;
}
```

```
__Z5plus1i:
    leal  1(%rdi), %eax
    retq
```

Function overloading

```
int plus1(int x)
{
    return x+1;
}

double plus1(double x)
{
    return x+1;
}
```

```
__Z5plus1i:
    leal  1(%rdi), %eax
    retq

__Z5plus1d:
    addsd LCPI1_0(%rip), %xmm0
    retq
```

Function templates

```
template<typename T>
T plus1(T x)
{
    return x+1;
}

auto y = plus1(42);
auto z = plus1(3.14);
```

```
__Z5plus1IiET_S0_:
    leal  1(%rdi), %eax
    retq

__Z5plus1IdET_S0_:
    addsd LCPI1_0(%rip), %xmm0
    retq
```

Class member functions

```
class Plus {
    int value;
  public:
    Plus(int v);

    int plusme(int x) const {
        return x + value;
    }
};
```

```
__ZN4PlusC1Ei:
    movl  %esi, (%rdi)
    retq

__ZN4Plus6plusmeEi:
    addl  (%rdi), %esi
    movl  %esi, %eax
    retq
```

“Which function do we call?”

```
auto plus = Plus(1);
auto x = plus.plusme(42);

assert(x == 43);
```

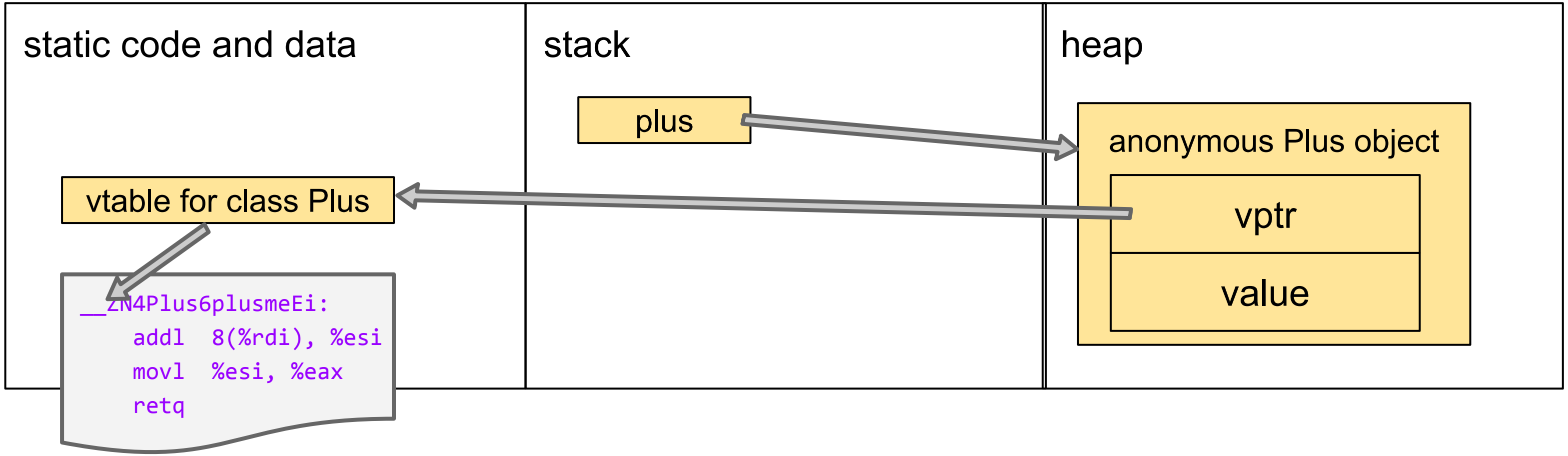
“The `plusme` function of the `Plus` class”

C++ is not Java!

The Java approach

```
auto plus = Plus(1);
auto x = plus.plusme(42);
assert(x == 43);
```

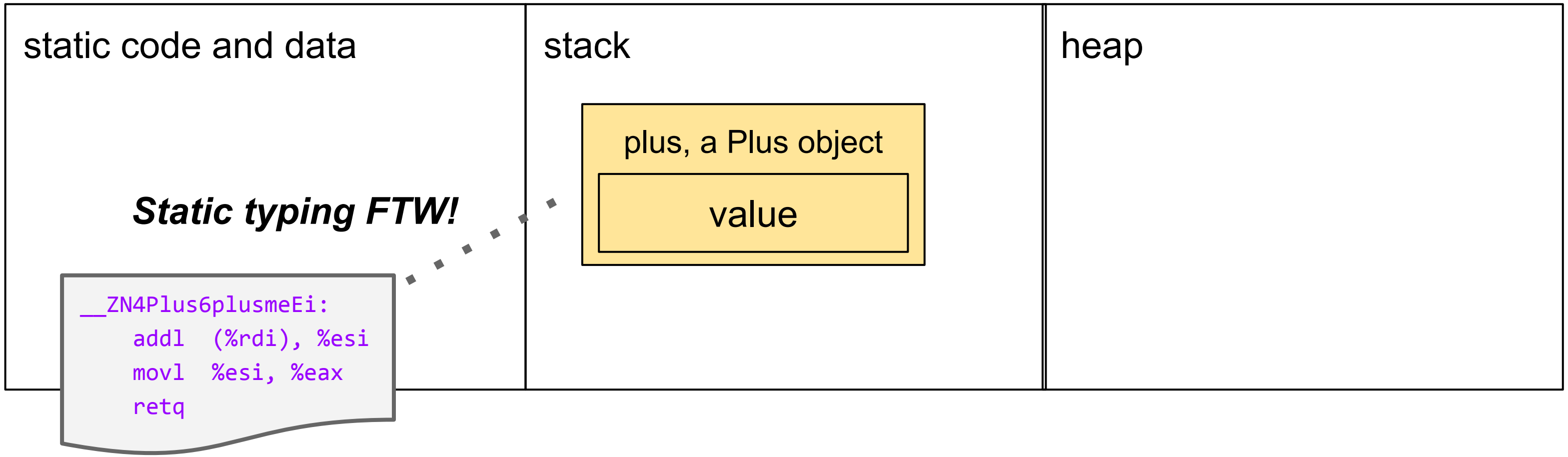
C++ lets you do this,
but it's not the default.



The C++ approach

```
auto plus = Plus(1);
auto x = plus.plusme(42);
assert(x == 43);
```

```
movl  $1, %esi
leaq  -16(%rbp), %rdi
callq __ZN4PlusC1Ei
movl  $42, %esi
leaq  -16(%rbp), %rdi
callq __ZN4Plus6plusmeEi
```



Class member functions (recap)

```
class Plus {
    int value;
  public:
    Plus(int v);

    int plusme(int x) const {
        return x + value;
    }
};
```

```
__ZN4PlusC1Ei:
    movl  %esi, (%rdi)
    retq

__ZN4Plus6plusmeEi:
    addl  (%rdi), %esi
    movl  %esi, %eax
    retq
```

```
auto plus = Plus(1);
auto x = plus.plusme(42);
```

Operator overloading

```
class Plus {
    int value;
  public:
    Plus(int v);

    int operator() (int x) const {
        return x + value;
    }
};
```

```
__ZN4PlusC1Ei:
    movl  %esi, (%rdi)
    retq

__ZN4PlusclEi:
    addl  (%rdi), %esi
    movl  %esi, %eax
    retq
```

```
auto plus = Plus(1);
auto x = plus(42);
```

So now we can make something kind of nifty...

Lambdas reduce boilerplate

```
class Plus {
    int value;
  public:
    Plus(int v): value(v) {}

    int operator() (int x) const {
        return x + value;
    }
};

auto plus = Plus(1);
assert(plus(42) == 43);
```

Lambdas reduce boilerplate

```
auto plus = [value=1](int x) { return x + value; };

assert(plus(42) == 43);
```

Same implementation

```
auto plus = [value=1](int x) {
    return x + value;
};
```

```
movl  $1, %esi
leaq  -16(%rbp), %rdi
callq __ZN3$_0C1Ei
movl  $42, %esi
leaq  -16(%rbp), %rdi
callq __ZN3$_0clEi
```

static code and data

Static typing FTW!

```
__ZN3$_0clEi:
    addl  (%rdi), %esi
    movl  %esi, %eax
    retq
```

stack

plus, a $_0 object

value

heap

Closures without garbage collection

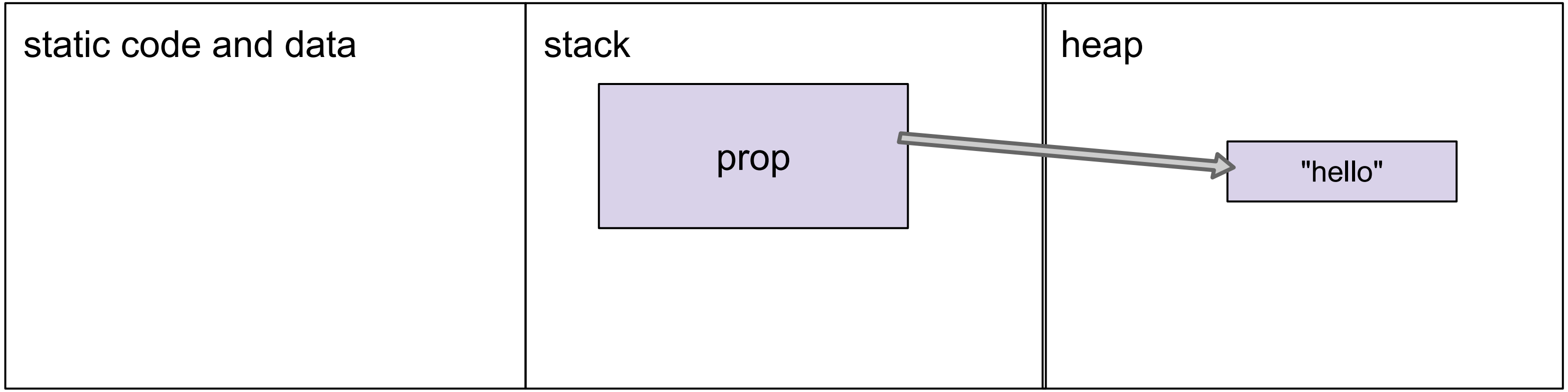
```
using object = std::map<std::string, int>;

void sort_by_property(std::vector<object>& v, std::string prop)
{
    auto pless = [p=prop](object& a, object& b) {
        return a[p] < b[p];
    };

    std::sort(v.begin(), v.end(), pless);
}
```

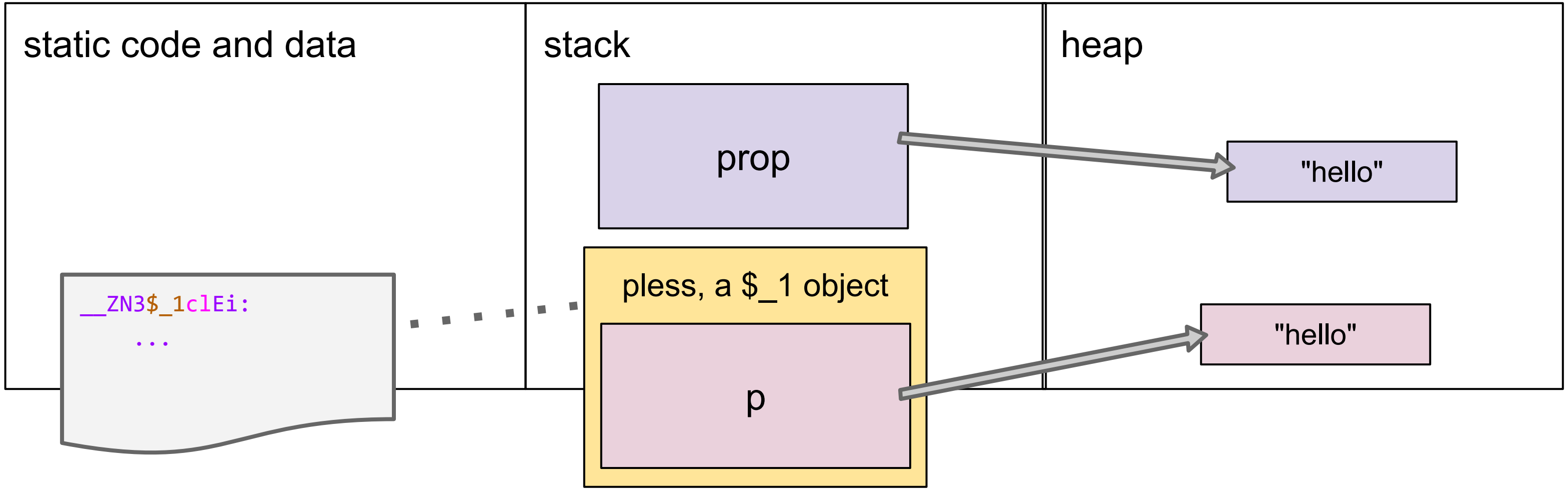
Closures without garbage collection

```
... std::string prop ...
```



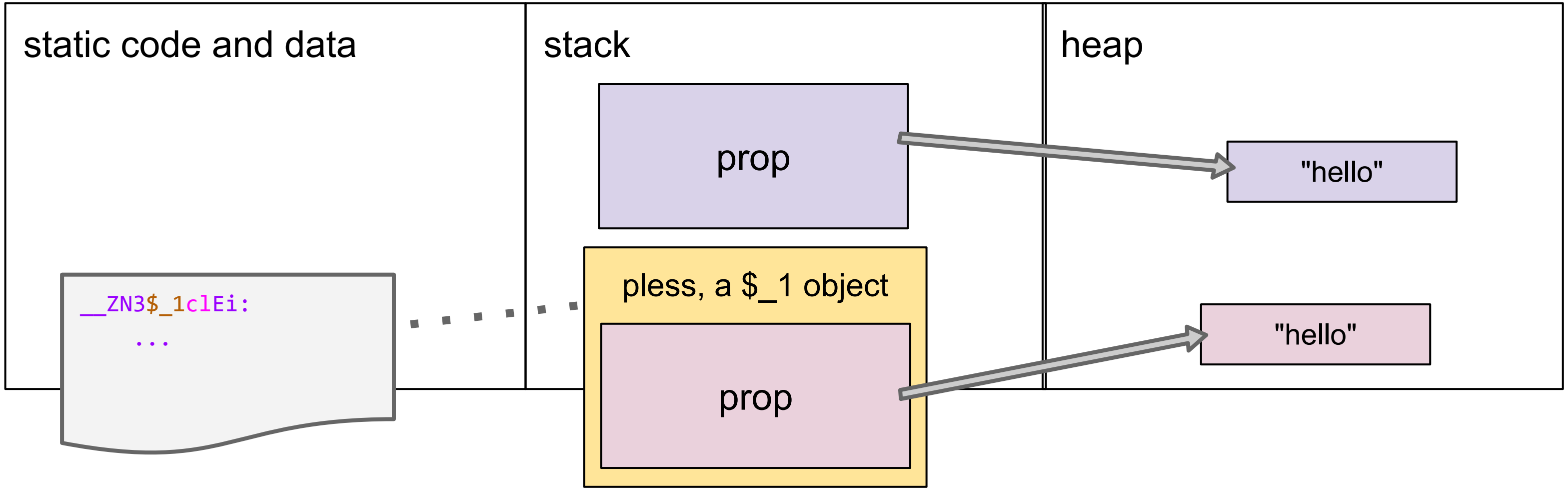
Closures without garbage collection

```
... std::string prop ...
    auto pless = [p=prop](object& a, object& b) {
        return a[p] < b[p];
    };
```



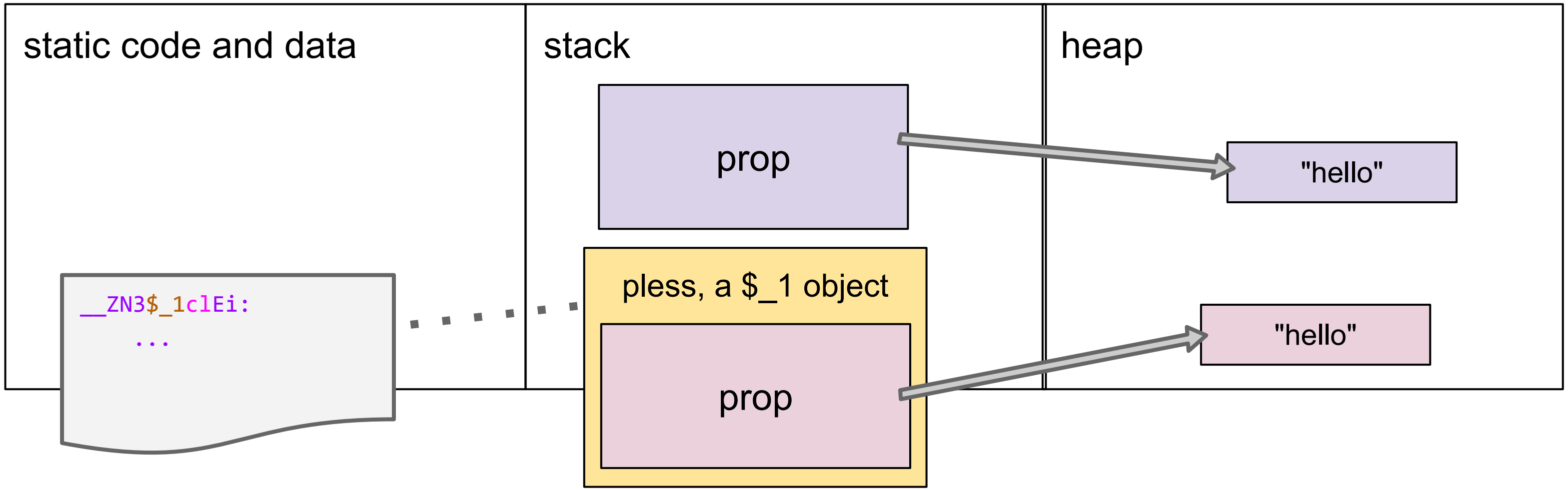
Closures without garbage collection

```
... std::string prop ...
    auto pless = [prop](object& a, object& b) {
        return a[prop] < b[prop];
    };
```



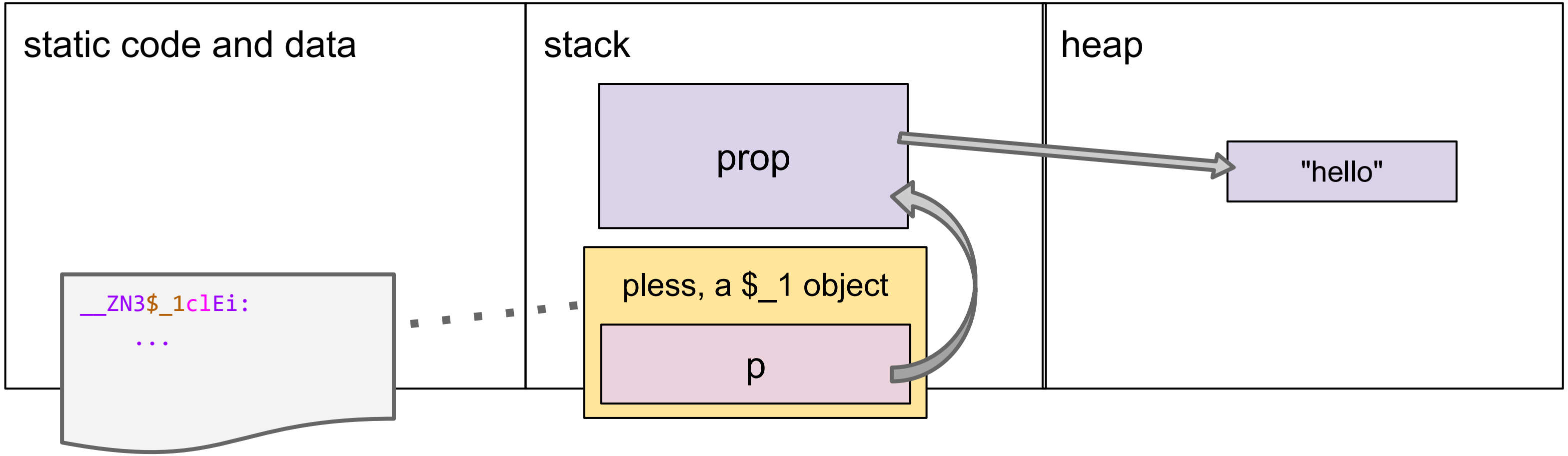
Copy semantics by default

```
... std::string prop ...
    auto pless = [=](object& a, object& b) {
        return a[prop] < b[prop];
    };
```



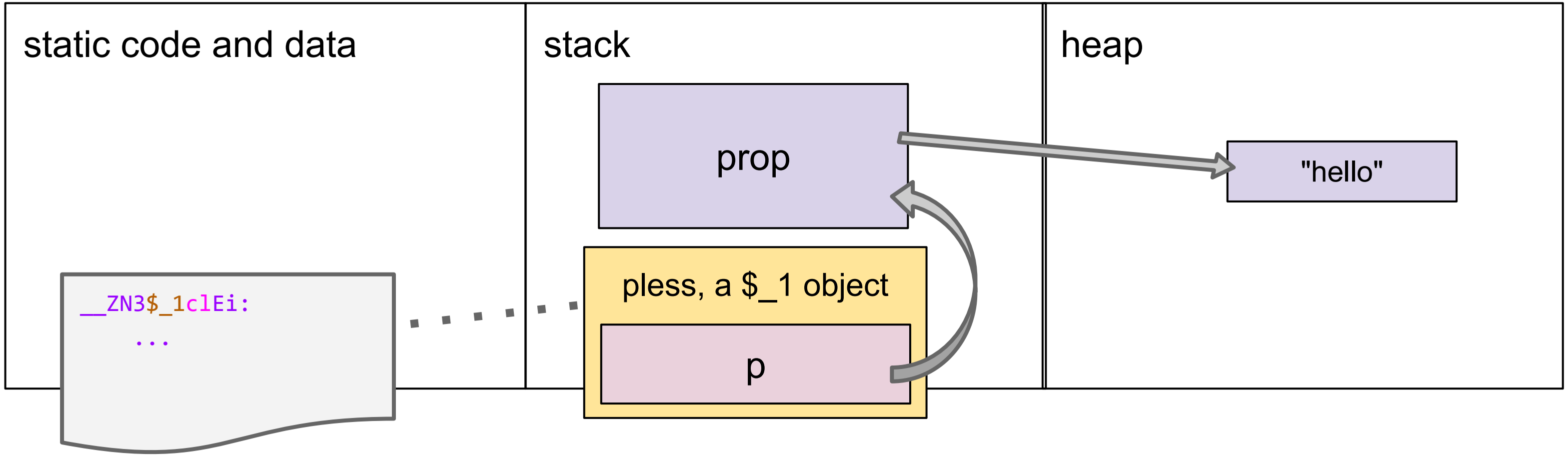
Capturing a reference

```
... std::string prop ...
    auto pless = [p=?????](object& a, object& b) {
        return a[p] < b[p];
    };
```



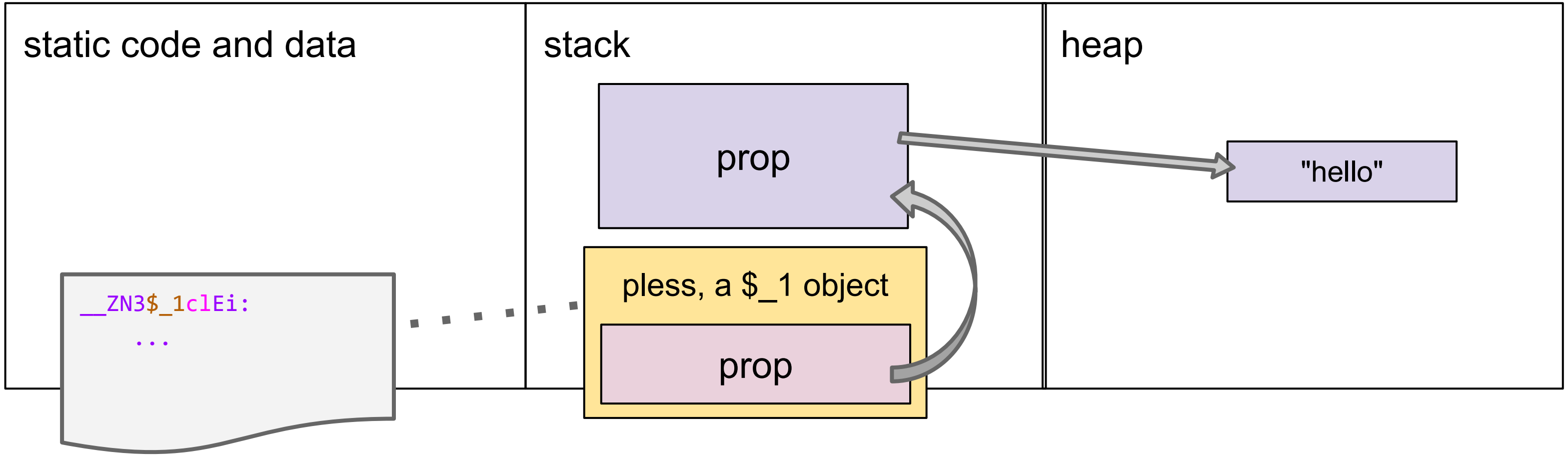
Capturing by reference

```
... std::string prop ...
    auto pless = [&p=prop](object& a, object& b) {
        return a[p] < b[p];
    };
```



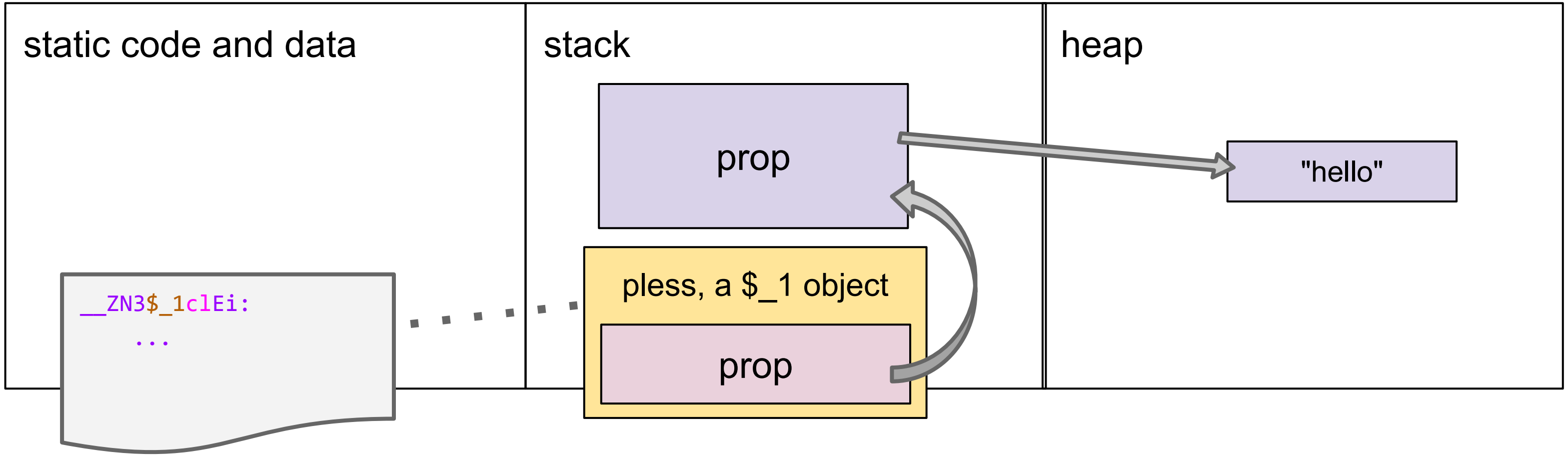
Capturing by reference

```
... std::string prop ...
    auto pless = [&prop](object& a, object& b) {
        return a[prop] < b[prop];
    };
```



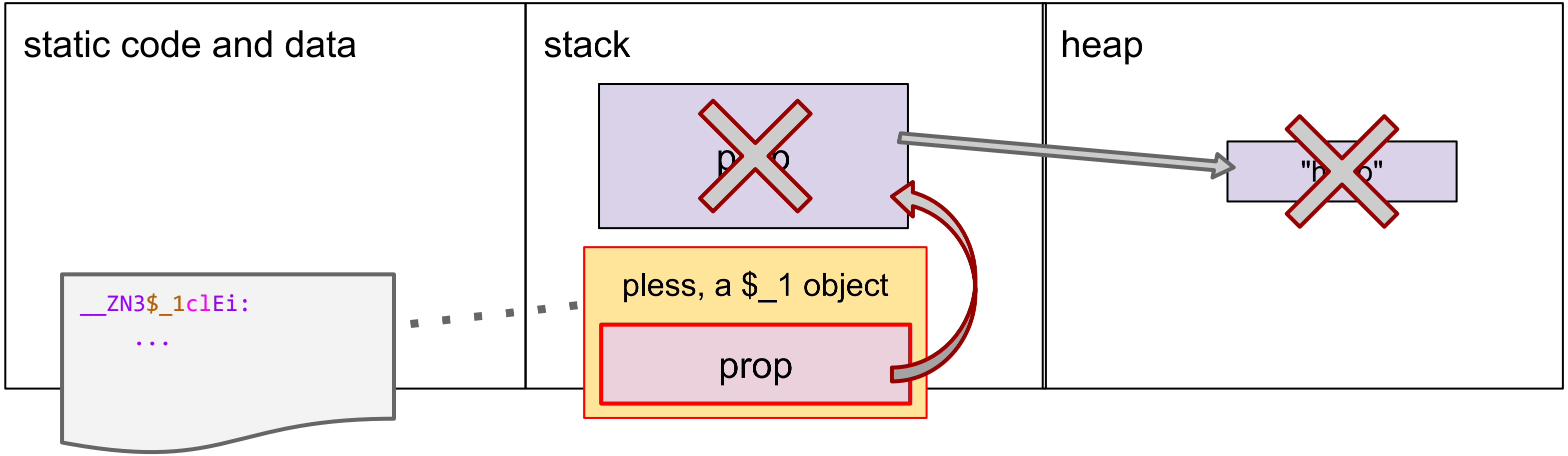
Capturing by reference

```
... std::string prop ...
    auto pless = [&](object& a, object& b) {
        return a[prop] < b[prop];
    };
```



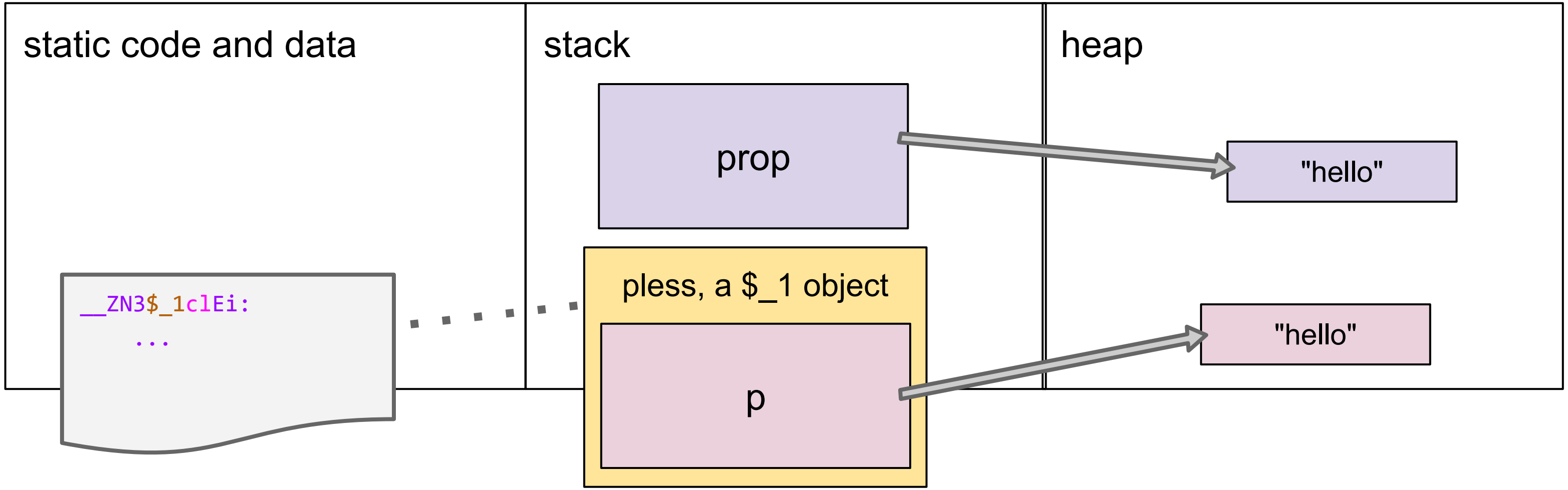
Beware of dangling references

```
... std::string prop ...
    auto pless = [&](object& a, object& b) {
        return a[prop] < b[prop];
    };
```



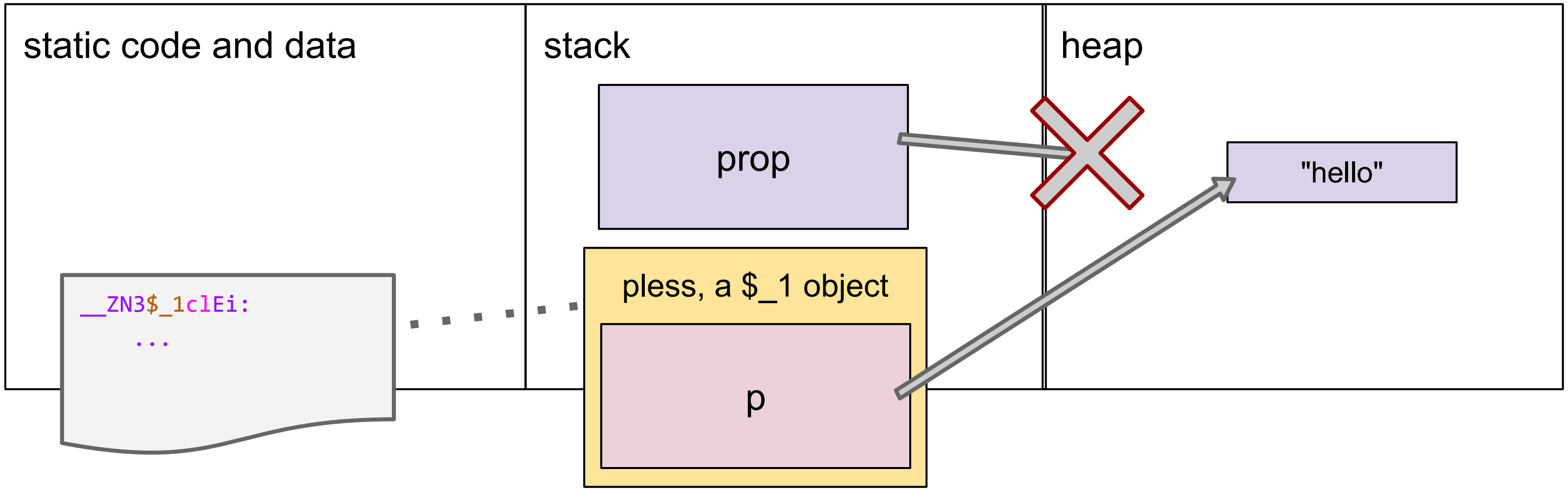
Capturing “by move”

```
... std::string prop ...
    auto pless = [p=prop](object& a, object& b) {
        return a[p] < b[p];
    };
```



Capturing “by move”

```
... std::string prop ...
    auto pless = [p=std::move(prop)](object& a, object& b) {
        return a[p] < b[p];
    };
```



Other features of lambdas

- Convertible to raw function pointer
  (when there are no captures involved)
- Variables with file/global scope are not captured
- Lambdas may have local state
  (but not in the way you think)

Puzzle

```
#include <stdio.h>

int g = 10;
auto kitten = [=]() { return g+1; };
auto cat = [g=g]() { return g+1; };

int main() {
    g = 20;
    printf("%d %d\n", kitten(), cat());
}
```

Puzzle

```
#include <stdio.h>

int g = 10;
auto kitten = [=]() { return g+1; };
auto cat = [g=g]() { return g+1; };

int main() {
    g = 20;
    printf("21 11\n", kitten(), cat());
}
```

Puzzle footnote

```
int g = 10;
auto ocelot = [g]() { return g+1; };
```

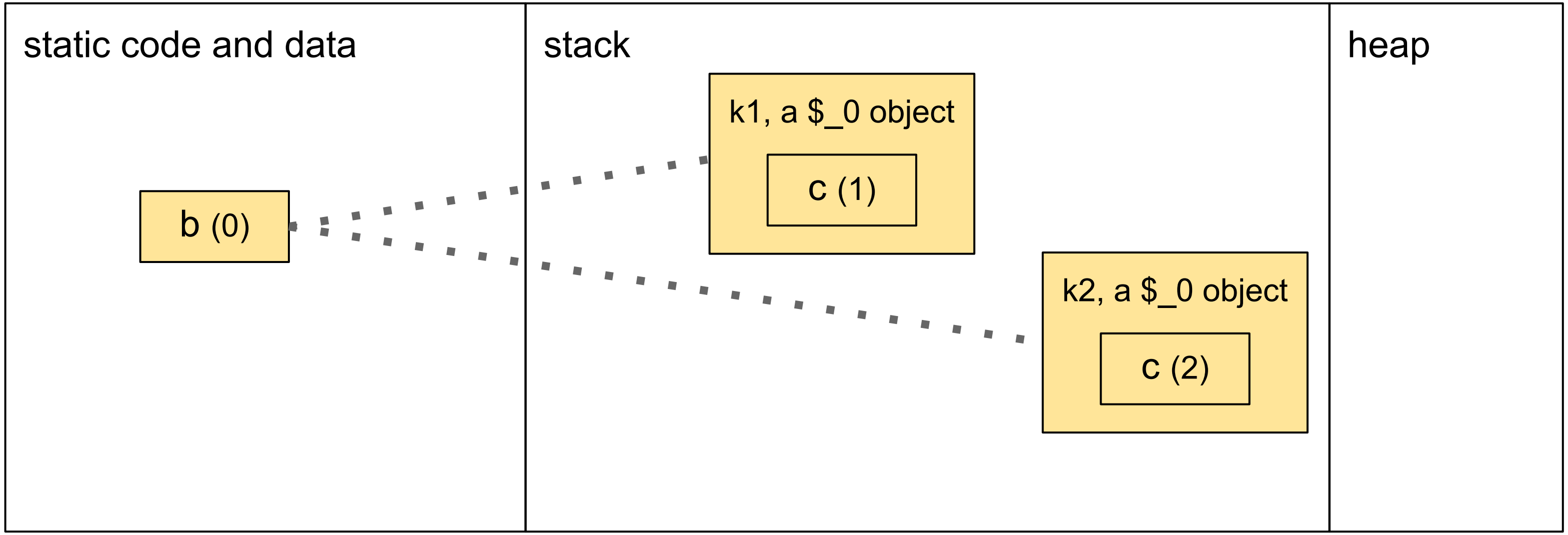
The above is ill-formed and requires a diagnostic.

> 5.1.2 [expr.prim.lambda]/10: The *identifier* in a *simple-capture* is looked up using the usual rules for unqualified name lookup (3.4.1); each such lookup **shall** find an entity. An entity that is designated by a *simple-capture* is said to be *explicitly captured*, and **shall** be `this` or a variable **with automatic storage duration** declared in the reaching scope of the local lambda expression.

In GCC this is just a warning, and the lambda does *not* capture g's value.

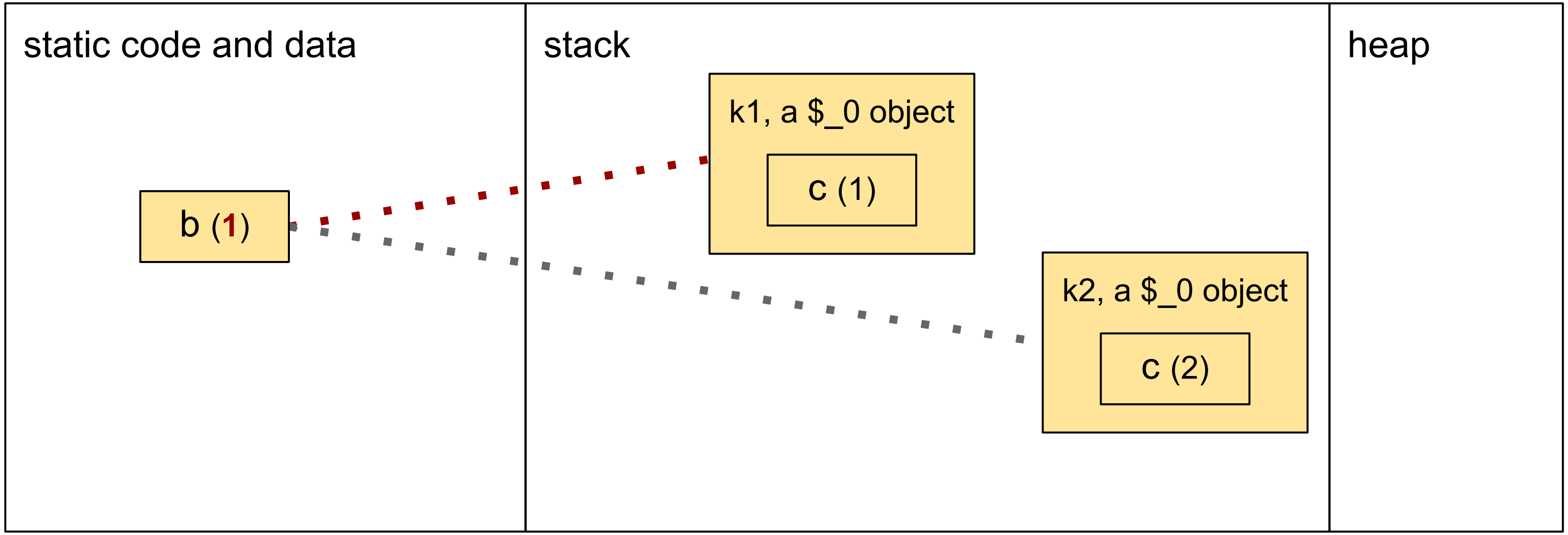
Per-lambda mutable state (wrong!)

```
... [c](int d) { static int b; ... } ...
```

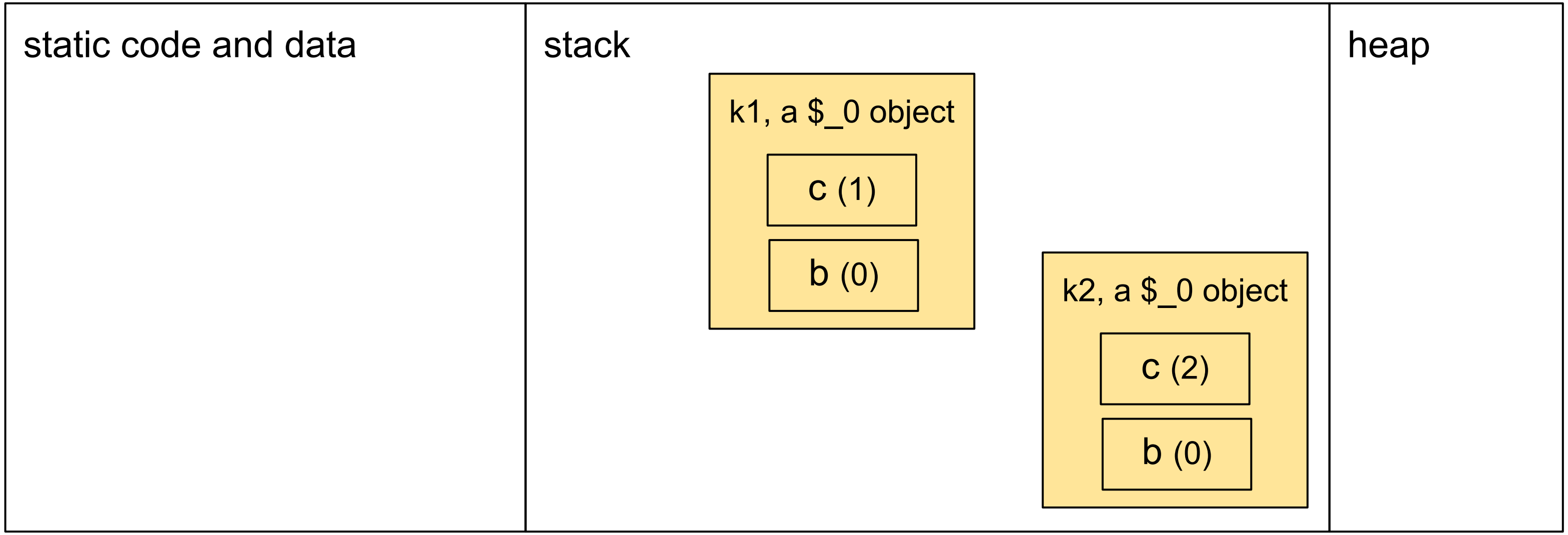


Per-lambda mutable state (wrong!)

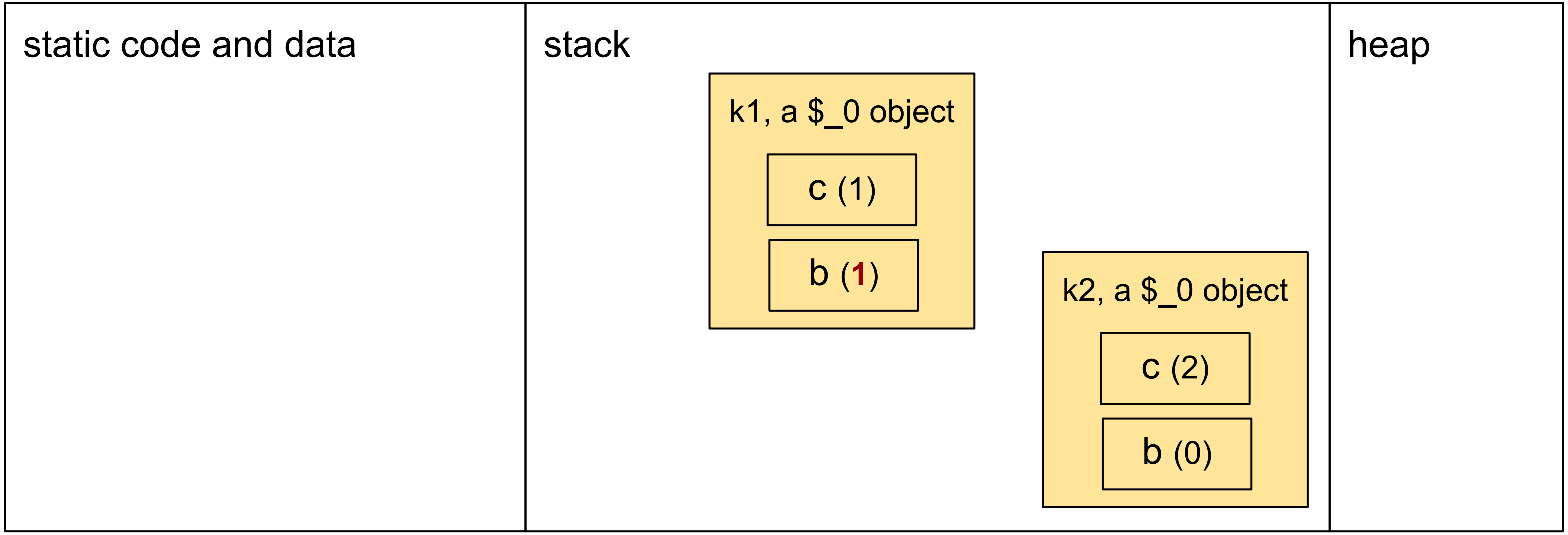
```
... [c](int d) { static int b; ... } ...
```



Per-lambda mutable state (right)



Per-lambda mutable state (right)



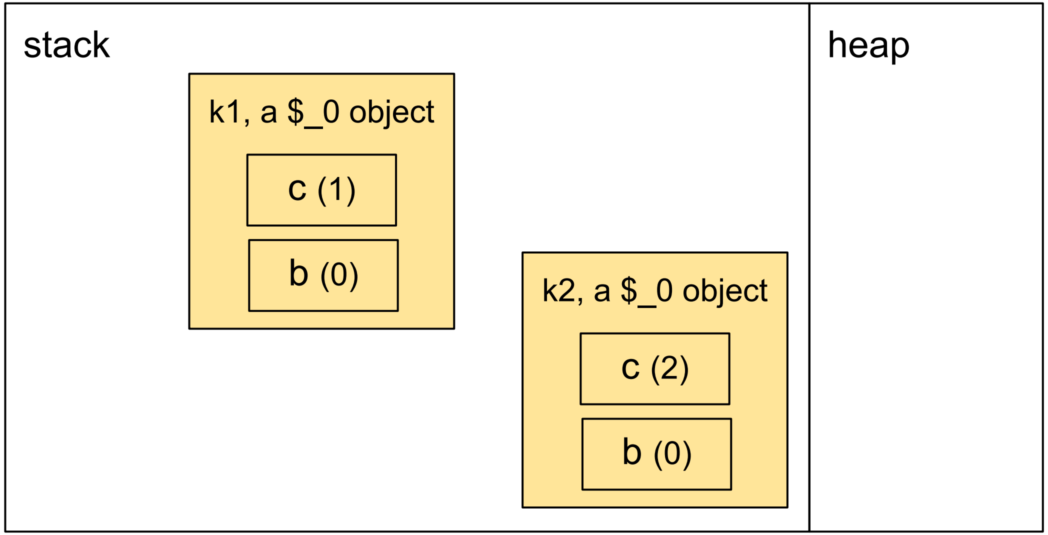
Per-lambda mutable state (right)

```
[c,b=0](int d) mutable { ... b++ ... }
```

Footnote:

`mutable` is all-or-nothing.

Generally speaking, captures aren't modifiable... and you usually don't want them to be.



Lambdas + Templates
=
Generic Lambdas

Class member function templates

```
class Plus {
    int value;
  public:
    Plus(int v);

    template<class T>
    T plusme(T x) const {
        return x + value;
    }
};
```

```
__ZNK4Plus6plusmeIiEET_S1_:
    addl  (%rdi), %esi
    movl  %esi, %eax
    retq

__ZNK4Plus6plusmeIdEET_S1_:
    cvtsi2sdl (%rdi), %xmm1
    addsd     %xmm0, %xmm1
    movaps    %xmm1, %xmm0
    retq
```

```
auto plus = Plus(1);
auto x = plus.plusme(42);
auto y = plus.plusme(3.14);
```

Class member function templates

```
class Plus {
    int value;
  public:
    Plus(int v);

    template<class T>
    T operator()(T x) const {
        return x + value;
    }
};
```

```
__ZNK4PlusclIiEET_S1_:
    addl  (%rdi), %esi
    movl  %esi, %eax
    retq

__ZNK4PlusclIdEET_S1_:
    cvtsi2sdl (%rdi), %xmm1
    addsd     %xmm0, %xmm1
    movaps    %xmm1, %xmm0
    retq
```

```
auto plus = Plus(1);
auto x = plus(42);
auto y = plus(3.14);
```

So now we can make something kind of nifty...

Generic lambdas reduce boilerplate

```
class Plus {
    int value;
  public:
    Plus(int v): value(v) {}

    template<class T>
    auto operator() (T x) const {
        return x + value;
    }
};

auto plus = Plus(1);
assert(plus(42) == 43);
```

Generic lambdas reduce boilerplate

```
auto plus = [value=1](auto x) { return x + value; };

assert(plus(42) == 43);
```

Generic lambdas are just templates under the hood.

Variadic function templates

```
class Plus {
  int value;
public:
  Plus(int v);

  template<class... A>
  auto operator()(A... a) {
    return sum(a..., value);
  }
};
```

```
__ZNK4PlusclIJidiEEEDaDpT_:
    cvtsi2sdl %esi, %xmm2
    addl (%rdi), %edx
    cvtsi2sdl %edx, %xmm1
    addsd %xmm1, %xmm0
    addsd %xmm2, %xmm0
    retq

__ZNK4PlusclIJPKciEEEDaDpT_:
    addl (%rdi), %edx
    movslq %edx, %rax
    addq %rsi, %rax
    retq
```

```
auto plus = Plus(1);
auto x = plus(42, 3.14, 1);
auto y = plus("foobar", 2);
```

Variadic lambdas reduce boilerplate

```
class Plus {
    int value;
  public:
    Plus(int v): value(v) {}

    template<class... P>
    auto operator() (A... a) const {
        return sum(a..., value);
    }
};

auto plus = Plus(1);
assert(plus(42, 3.14, 1) == 47.14);
```

Variadic lambdas reduce boilerplate

```
auto plus = [value=1](auto... a) {
    return sum(a..., value);
};

assert(plus(42, 3.14, 1) == 47.14);
```

What is `this` in a lambda?

What is `this` in a lambda?

```
... std::string prop ...
    auto pless = [p=prop](object& a, object& b) {
        return a[p] < b[p];
    };
```

You might think that `p` (being a member of the underlying closure instance) should also be accessible inside the lambda via "`this->p`."

Not so!

The underlying closure instance is just that: *underlying*. It's how the lambda is *implemented*. But at the source level, we want `this` to expose a different property...

What is `this` in a lambda?

```
class Widget {
    void work(int);

    void synchronous_foo(int x) {
        this->work(x);
    }

    void asynchronous_foo(int x) {
        fire_and_forget([=]() {
            this->work(x);
        });
    }
};
```

It's good that these two "`this`" expressions mean the same thing!

We can reuse code snippets without counting brackets so carefully.

Ways of capturing `this`

- `[=]() { this->work(); }`
- `[this]() { this->work(); }`
  - Both equivalent to `[ptr=this]() { ptr->work(); }`
- `[&]() { this->work(); }`
  - *Also* equivalent to `[ptr=this]() { ptr->work(); }`
- *New in C++17!* `[*this]() { this->work(x); }`
  - Equivalent to `[obj=*this]() { obj.work(); }`
- "Capture `*this` by move" has no shorthand equivalent.
  - Just write `[obj=std::move(*this)]() { obj.work(x); }`

“So are lambdas kind of like `std::function`, then?”

“Why does C++ have both?”

Type Erasure From Scratch

std::function is a *vocabulary type*

Before we can talk about `<math.h>`, we need `double`.

Before we can talk about stringstreams, we need `std::string`.

Before we can talk about callbacks, we need `std::function`.

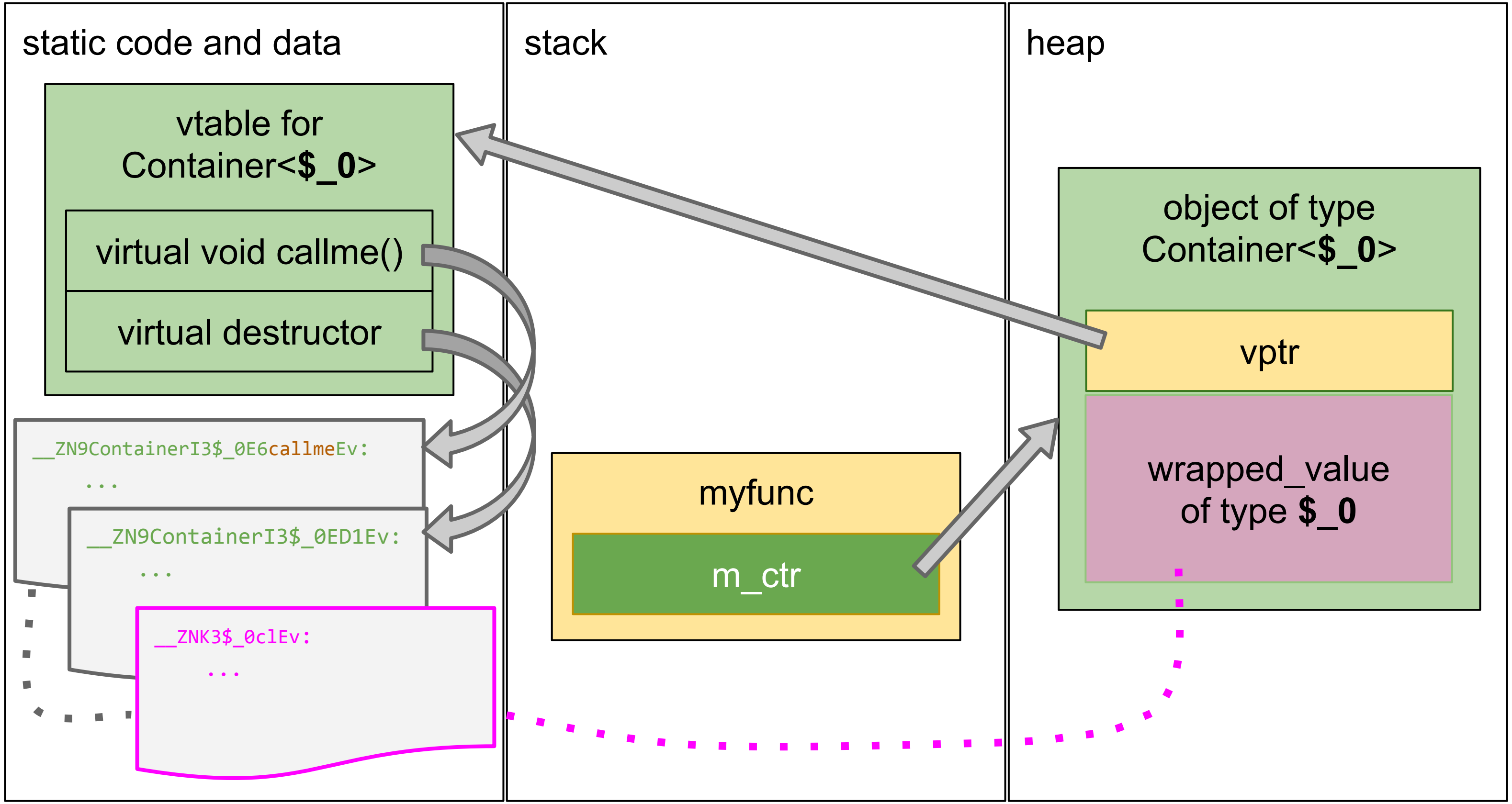
`std::function` allows us to pass lambdas, functor objects, etc., across *module boundaries*.

Type erasure in a nutshell

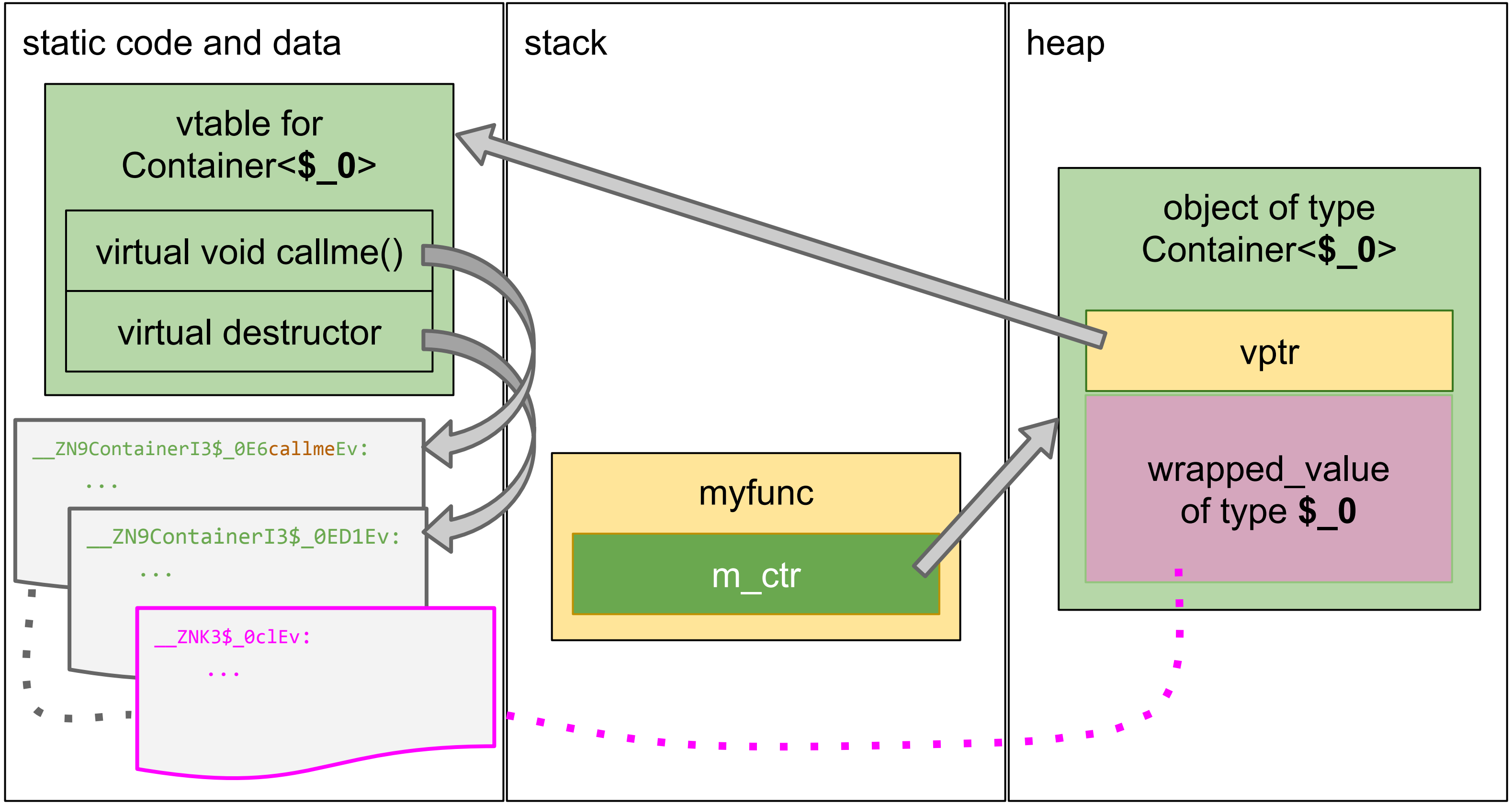
```
struct ContainerBase {
    virtual int callme(int) = 0;
    virtual ~ContainerBase() = default;
};
template <class Wrapped> struct Container : ContainerBase {
    Wrapped wrapped_value;
    Container(const Wrapped& wv) : wrapped_value(wv) {}
    int callme(int i) override { return wrapped_value(i); }
};

class i2i {  // equivalent to std::function<int(int)>
    ContainerBase *m_ctr;
public:
    template<class F> i2i(const F& wv)
      : m_ctr(new Container<F>(wv)) {}
    int operator()(int i) { return m_ctr->callme(i); }   // virtual dispatch
    ~i2i() { delete m_ctr; }                 // virtual dispatch
};
```

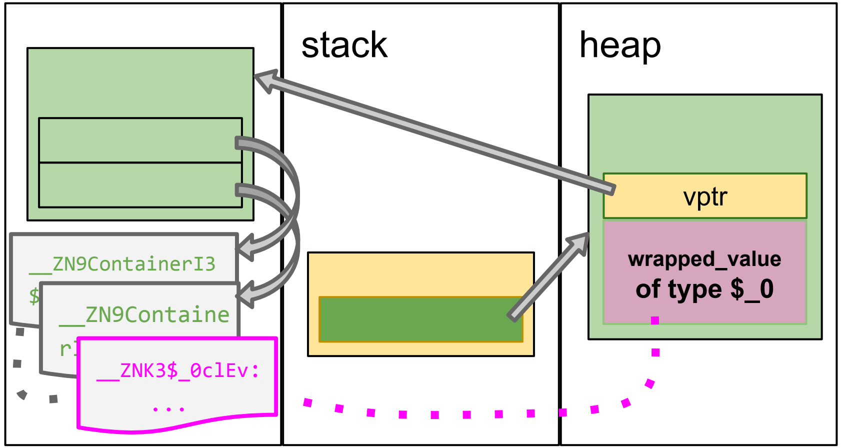
Type erasure diagrammatically



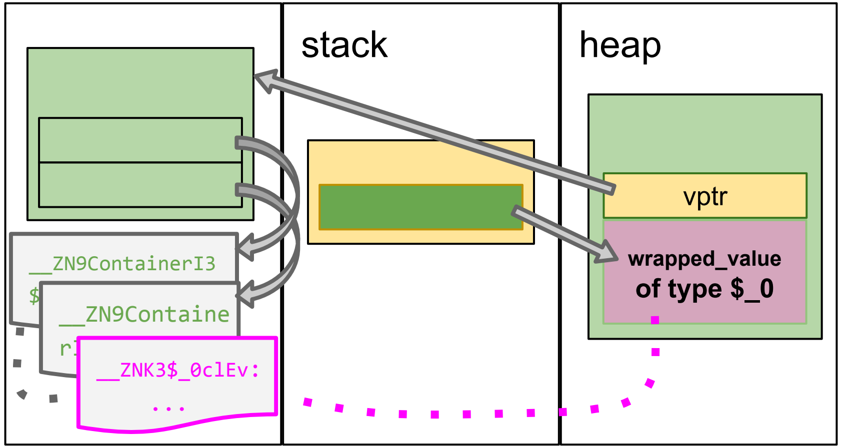
`myfunc` is nothrow moveable.



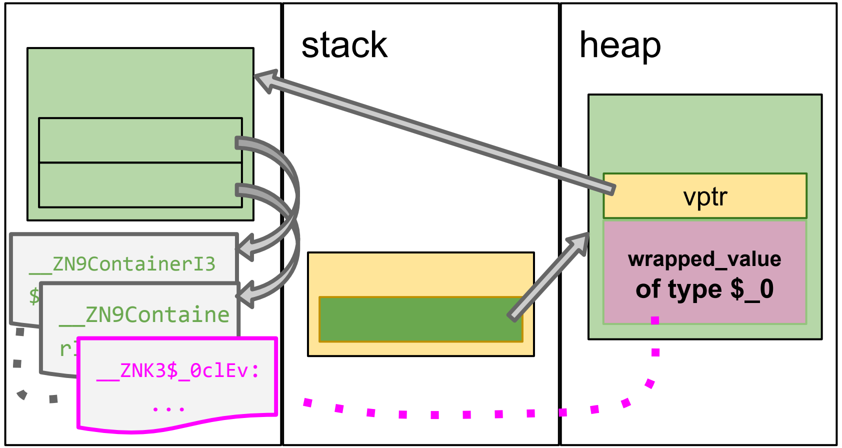
`myfunc` is nothrow moveable.



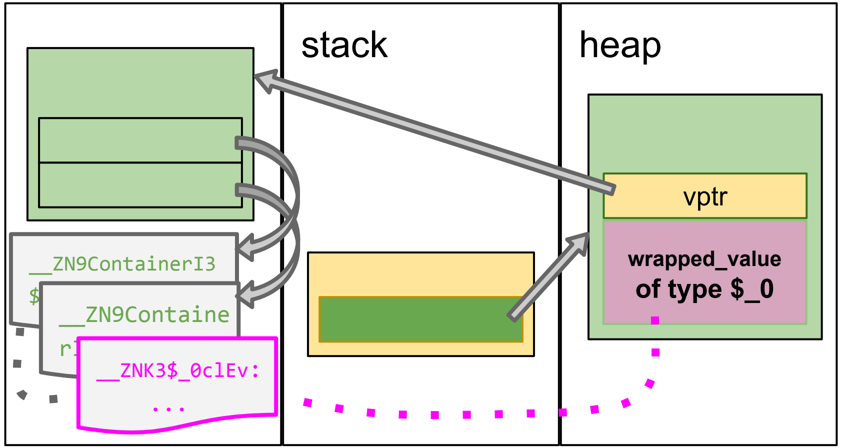
myfunc is nothrow moveable.



Is `myfunc` copyable?



Is `myfunc` copyable?



No, `myfunc` is not copyable.

In order to make a copy of `myfunc`, we'd have to allocate a second `Container<$_0>`, containing a copy of `myfunc`'s wrapped value.

But the copy constructor of `myfunc` (that is, `class i2i`'s copy constructor) doesn't remember the identity of type `$_0` anymore — we've erased it!

"Copying" is an operation, just like "calling with signature `int(int)`" is an operation. If we want it to be supported, we must explicitly support it via a virtual method in `ContainerBase`.

Make our `i2i` copyable

```
struct ContainerBase {
    virtual int callme(int) = 0;
    virtual ContainerBase *copyme() = 0;
    virtual ~ContainerBase() = default;
};
template <class Wrapped> struct Container : ContainerBase {
    Wrapped wrapped_value;
    Container(const Wrapped& wv) : wrapped_value(wv) {}
    int callme(int i) override { return wrapped_value(i); }
    ContainerBase *copyme() override { return new Container(wrapped_value); }
};

class i2i {  //  equivalent to std::function<int(int)>
    ContainerBase *m_ctr;
public:
    template<class F> i2i(const F& wv) : m_ctr(new Container<F>(wv)) {}
    i2i(const i2i& rhs) : m_ctr(rhs->copyme()) {}   // virtual dispatch
    int operator()(int i) { return m_ctr->callme(i); }   // virtual dispatch
    ~i2i() { delete m_ctr; }               // virtual dispatch
};
```

New

New

Even more

Lambdas may be copyable or not

```
std::unique_ptr<int> prop;
auto lamb = [p = std::move(prop)]() { };
auto lamb2 = std::move(lamb);  // OK
auto lamb3 = lamb; // error: call to implicitly-deleted copy constructor
```

A lambda’s type is copyable, moveable, or neither, depending as its captures are copyable, moveable, or neither.

`std::function` is always copyable.

Therefore, there are some lambdas that can’t be stored in a `std::function`.

```
std::function<void()> f = std::move(lamb);  // cascade of errors
```

Working around noncopyability (bad)

Hot-potato the single instance á là `auto_ptr`.

```
std::function<void()> f2 = AwfulPtr(lamb);
```

```
template<class T>
class AwfulPtr {
    std::optional<T> o;
public:
    explicit AwfulPtr(T& t) : o(std::move(t)) {}

    AwfulPtr(const AwfulPtr& rhs) : o((T&&)rhs.o.value()) {
        ((AwfulPtr&)rhs).o.reset();
    }

    template<class... Args>
    decltype(auto) operator()(Args&&... args) const {
        return o.value()(std::forward<Args>(args)...);
    }
};
```

Casting away constness

Casting away constness

Working around noncopyability (good)

```
auto lamb = something move-only;
```

Consider placing the single instance on the heap and *sharing* access to it:

```
std::function<void()> f3 = [
    p = std::make_shared<decltype(lamb)>(std::move(lamb))
]() { (*p)(); };
```

Or use a *moveable function type* such as `folly::Function`.

```
my::unique_function<void()> f4 = std::move(lamb);  // OK
```

Every codebase needs a moveable function type!

Questions?

Thanks for coming!