Back to Basics: Lambdas from Scratch

Arthur O’Dwyer
2019-09-20

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);
```

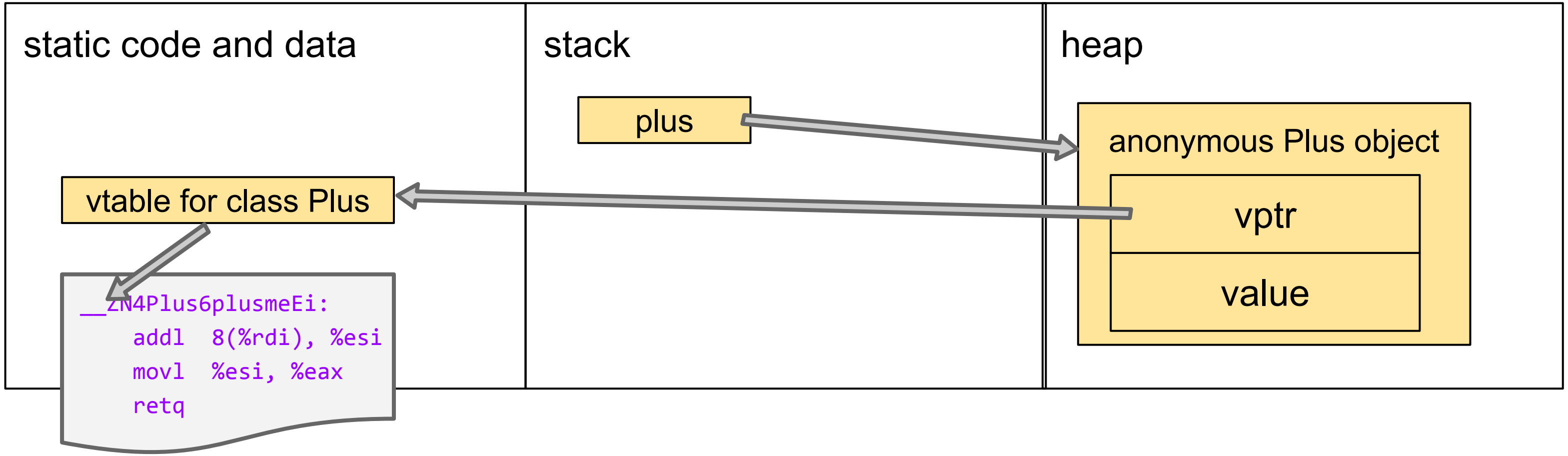
“The `plusme` function of the `Plus` class”

C++ is not Java!

The Java approach

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

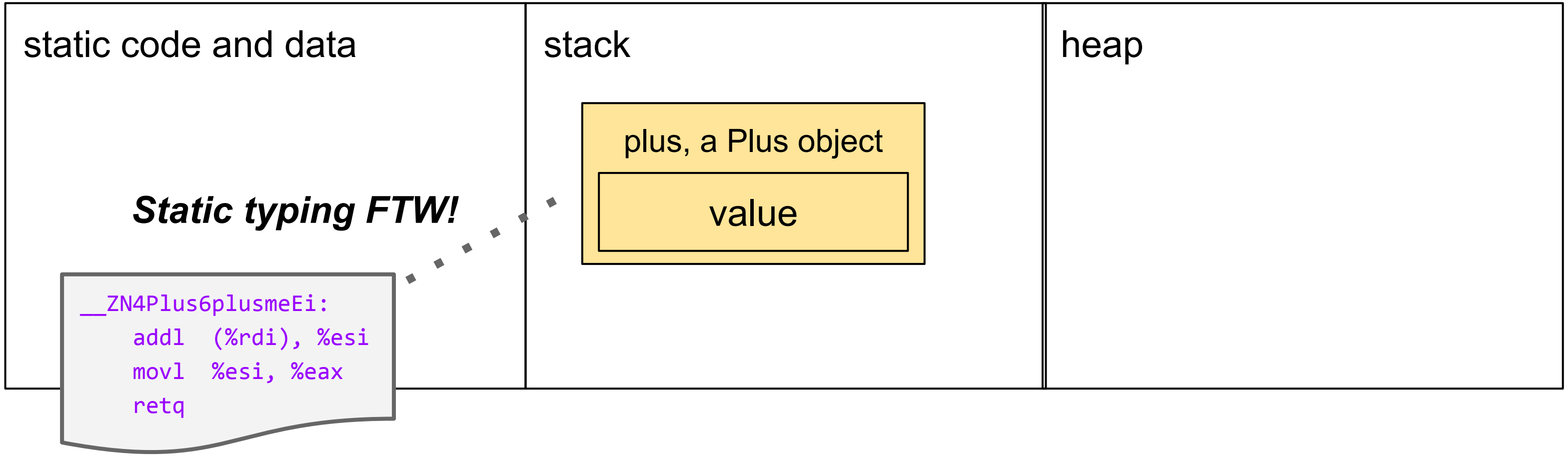
C++ lets you do this,
but it's not the default.



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

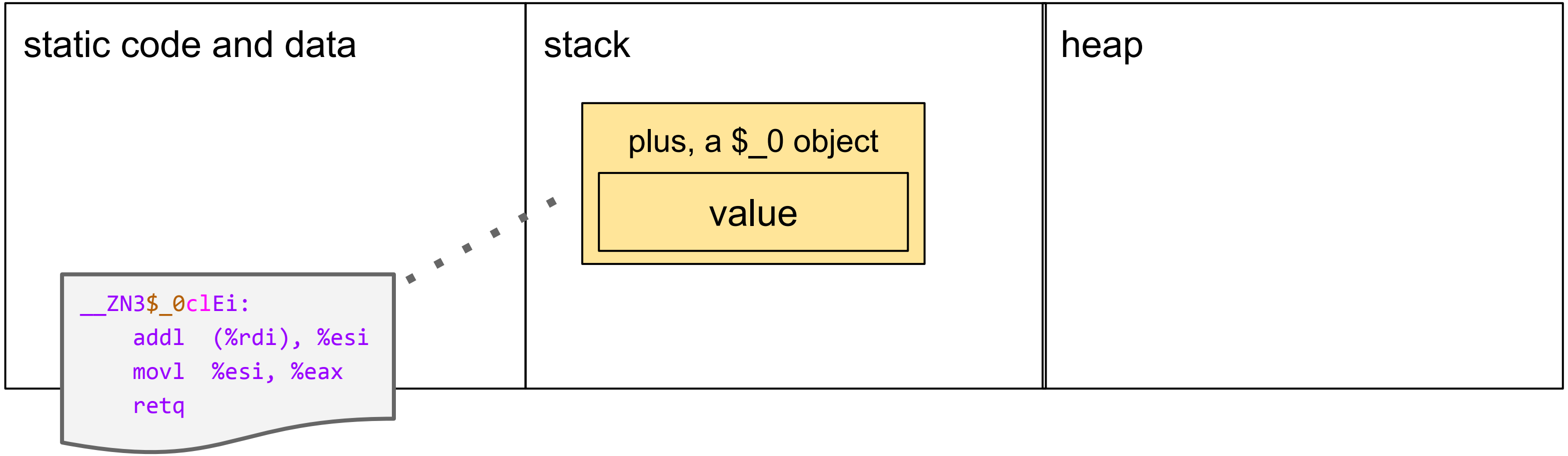
```
auto plus = [value=1](int x) { return x + value; };

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
```



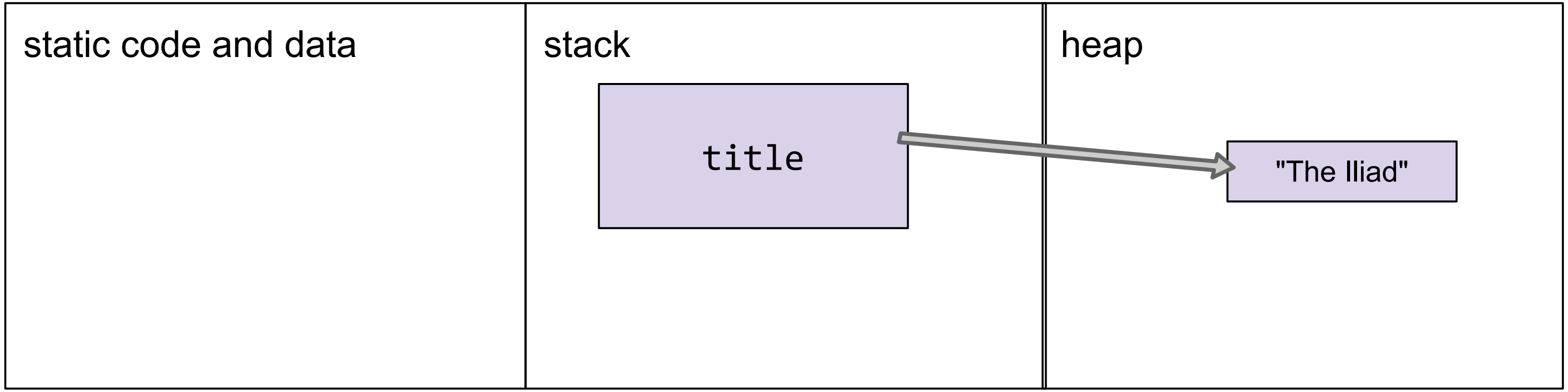
Closures without garbage collection

```
bool contains_title(const std::vector<Book>& shelf,
                    std::string title)
{
    auto has_title_t = [t=title](const Book& b) {
        return b.title() == t;
    };

    return v.end() !=
        std::find_if(v.begin(), v.end(), has_title_t);
}
```

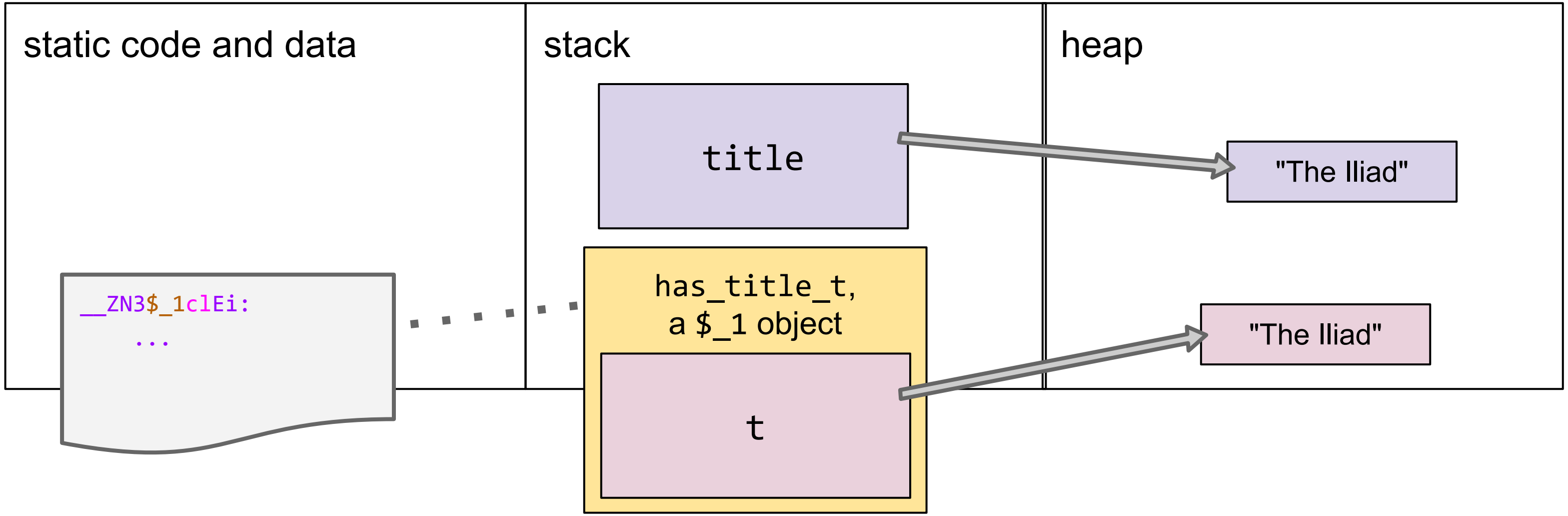
Closures without garbage collection

```
bool contains_title(const std::vector<Book>& shelf,
                    std::string title)
```



Closures without garbage collection

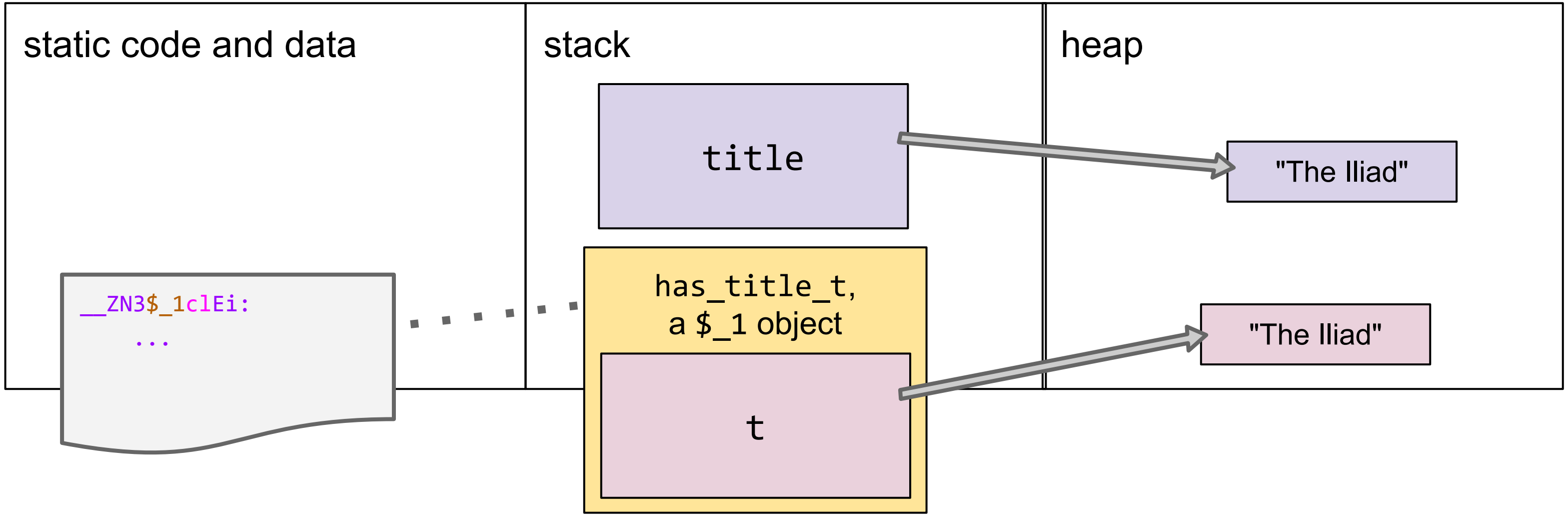
```
auto has_title_t = [t=title](const Book& b) {
    return b.title() == t;
};
```



Copy semantics by default

```
auto has_title_t = [t=title](const Book& b) {
    return b.title() == t;
};
```

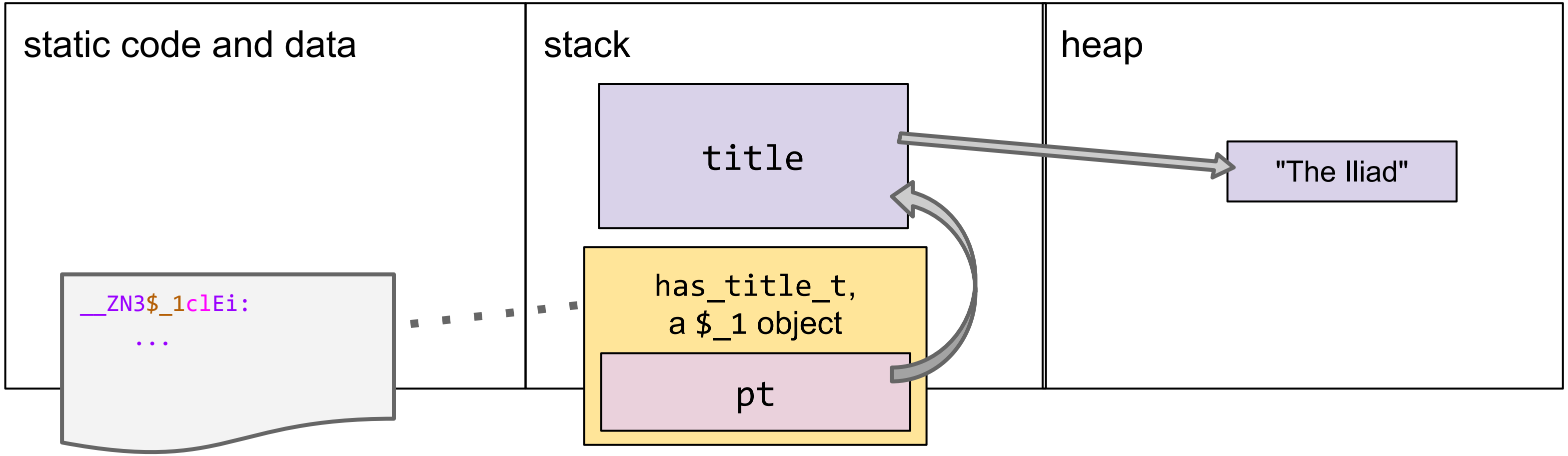
`auto t=title;` would make a copy of the string. Likewise, with `[t=title]`, the lambda captures a copy of the string.



Capturing a pointer

```
auto has_title_t = [pt=&title](const Book& b) {
    return b.title() == *pt;
};
```

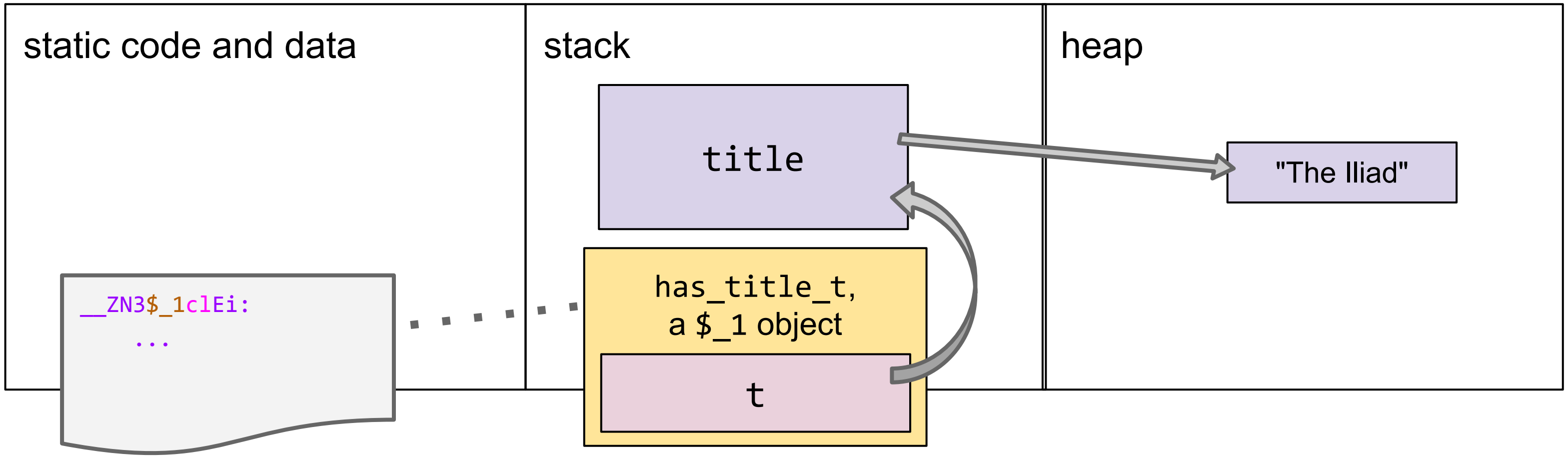
`auto pt=&title;` and likewise `[pt=&title]` mean to capture a pointer initialized with `&title`.



Capturing a reference

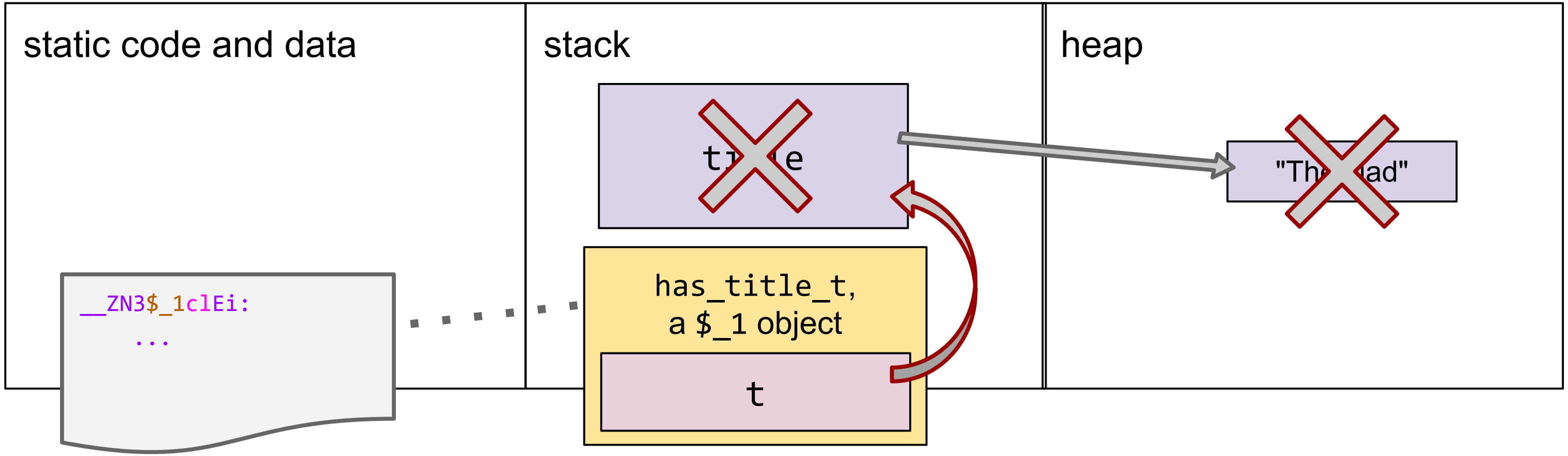
```
auto has_title_t = [&t=title](const Book& b) {
    return b.title() == t;
};
```

`auto &t=title;` and likewise `[&t=title]` mean to capture a reference that refers to `title`.



Beware of dangling references

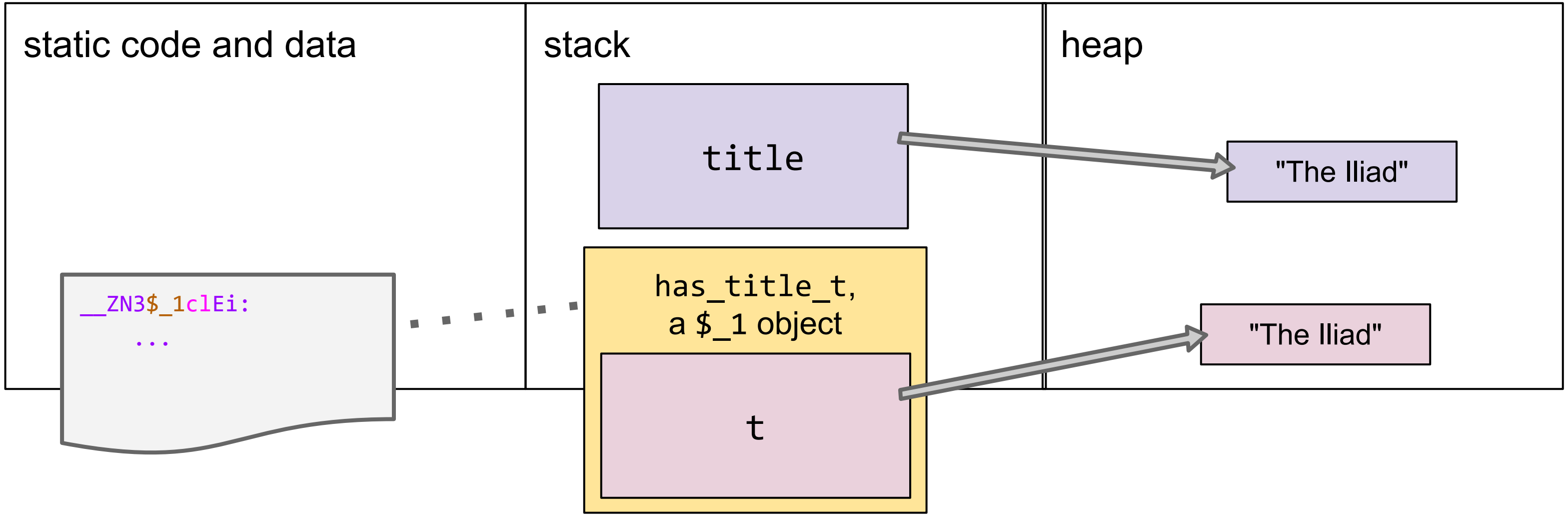
```
auto has_title_t = [&t=title](const Book& b) {
    return b.title() == t;
};
return has_title_t;
```



Capturing “by move”?

```
auto has_title_t = [t=title](const Book& b) {
    return b.title() == t;
};
```

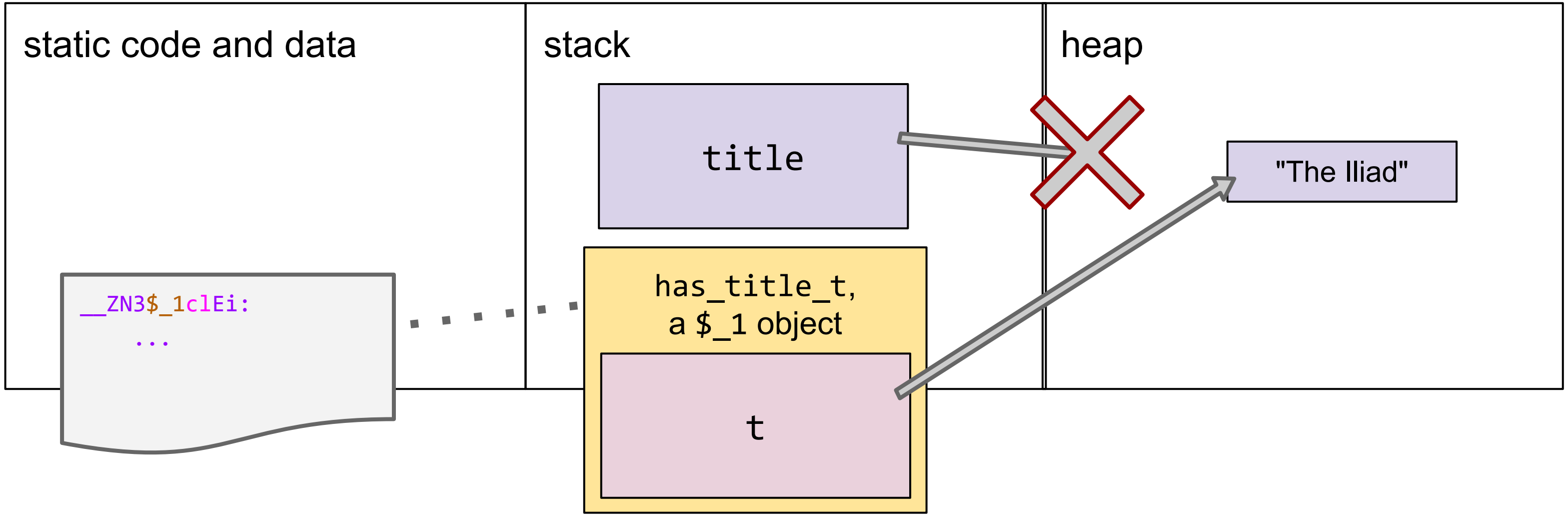
This *copies* `title` into the lambda’s captured state. What if we wanted to *move* it in, instead?



Capturing “by move”

```
auto has_title_t = [t=std::move(title)](const Book& b) {
    return b.title() == t;
};
```

`auto t=std::move(title);` and likewise `[t=std::move(title)]` capture a string by value, but prefer the move-constructor.



Many redundant shorthands

- `[t=title]() { decltype(title) ... use(t); }`
- `[title]() { decltype(title) ... use(title); }` *No array decay!*
- `[&t=title]() { use(t); }`
- `[&title]() { use(title); }`
- To capture ***only*** what is “needed” —
  - `[=]() { use(title); }`
  - `[&]() { use(title); }` *The most useful!*
  - Globals/statics aren’t captured; neither are unevaluated operands

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());
}
```

This will be important in my next talk!
Type Erasure From Scratch
1:30pm today in Aurora A

Other features of lambdas

- Convertible to raw function pointer (if captureless)

```
int (*fp)(int) = [](int x) { return x + 1; };
```

This is a very common idiom in some kinds of C++ code.

The unary + operator forces any expression to “scalar” type.
For most types that’s not very useful. But for lambdas it’s idiomatic!

```
template<class T> void fn(T t);

fn(+[](int x){ return x+1; }); // calls fn<int(*)(int)>
```

Other features of lambdas

- Default-constructible (if captureless)

```
auto lam = [](int x) { return x + 1; };
decltype(lam) copy;  // OK!
```

New in C++20!

- Constexpr by default, but ***not*** `noexcept` by default

```
static_assert(lam(42) == 43);  // OK!
static_assert(not noexcept(lam(42)));
```

New in C++17!

- Lambdas may have local state (but not in the way you think)

Per-lambda mutable state (wrong!)

```
auto counter = []() { static int i; return ++i; };
auto c1 = counter;
auto c2 = counter; // make two copies of the counter
std::cout << c1() << c1() << c1() << '\n';
std::cout << c2() << c2() << c2() << '\n';

  // 123
  // 456
```

Per-lambda mutable state (wrong!)

```
auto counter = []() { static int i; return ++i; };
```

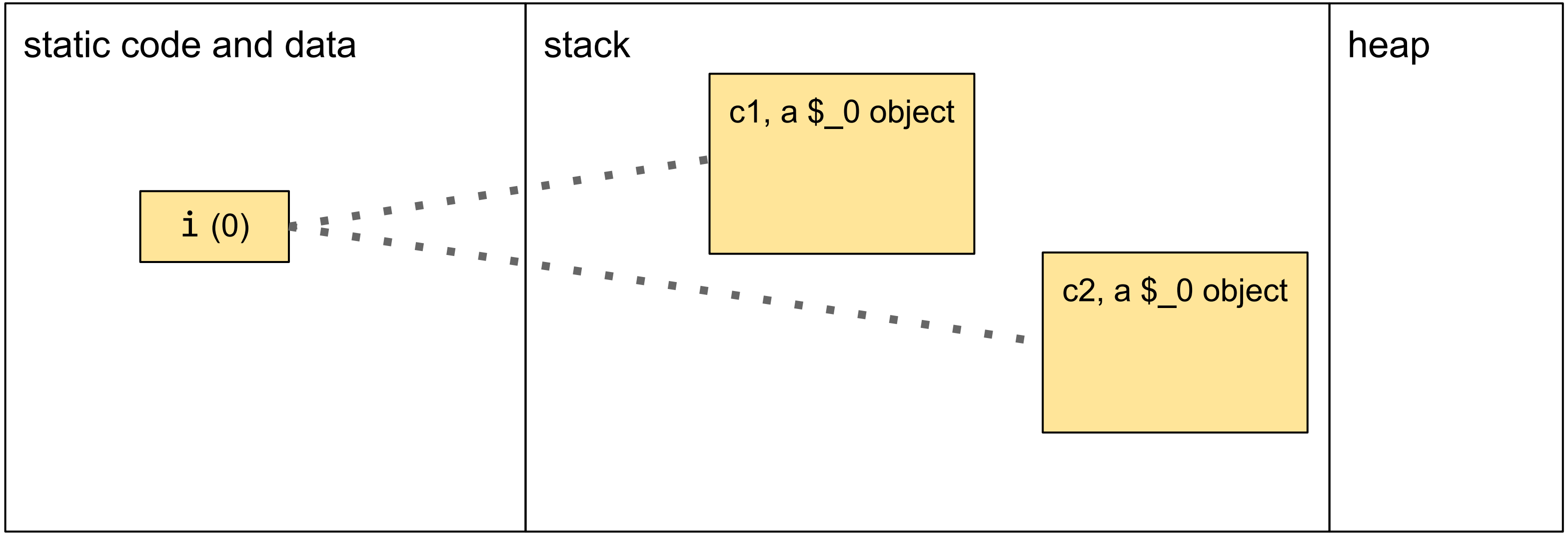
That closure type behaves just like the following class type:

```
class Counter {
    // no captured data members
public:
    int operator() const {
        static int i; return ++i;
    }
};
```

There is just one static `i`, shared by ***all*** callers of `Counter::operator()`!

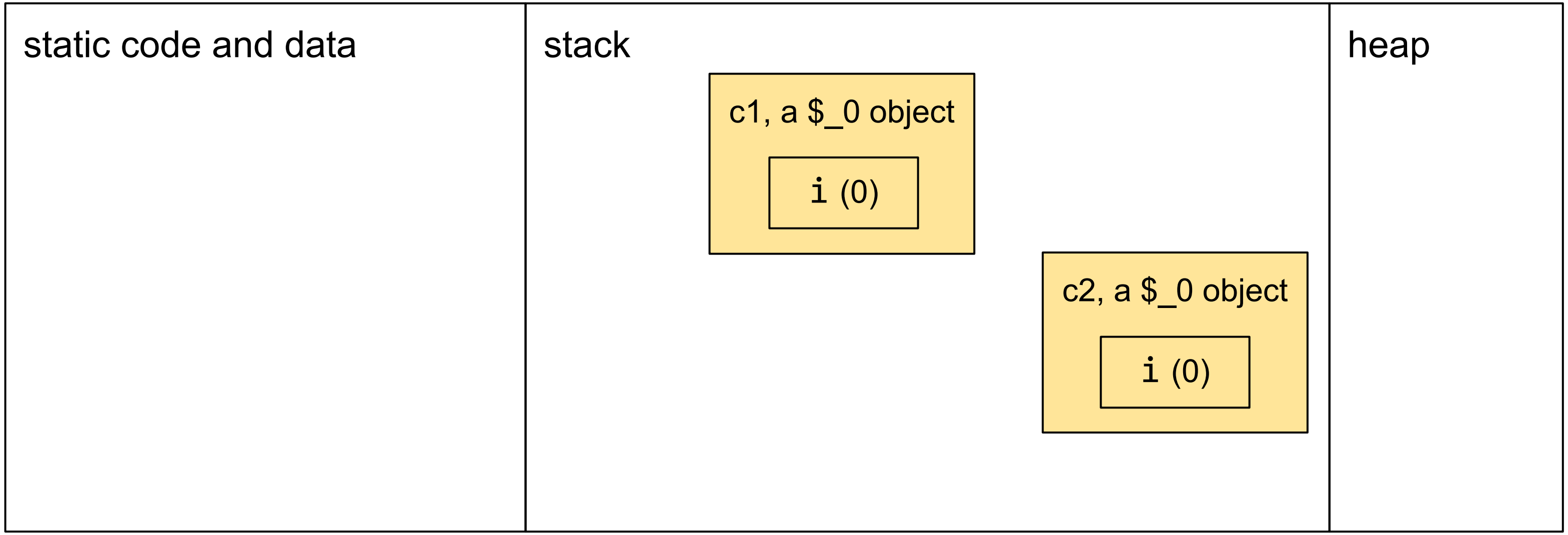
Per-lambda mutable state (wrong!)

```
[]() { static int i; return ++i; };
```



Per-lambda mutable state (wrong!)

```
[i=0]() { return ++i; };
```



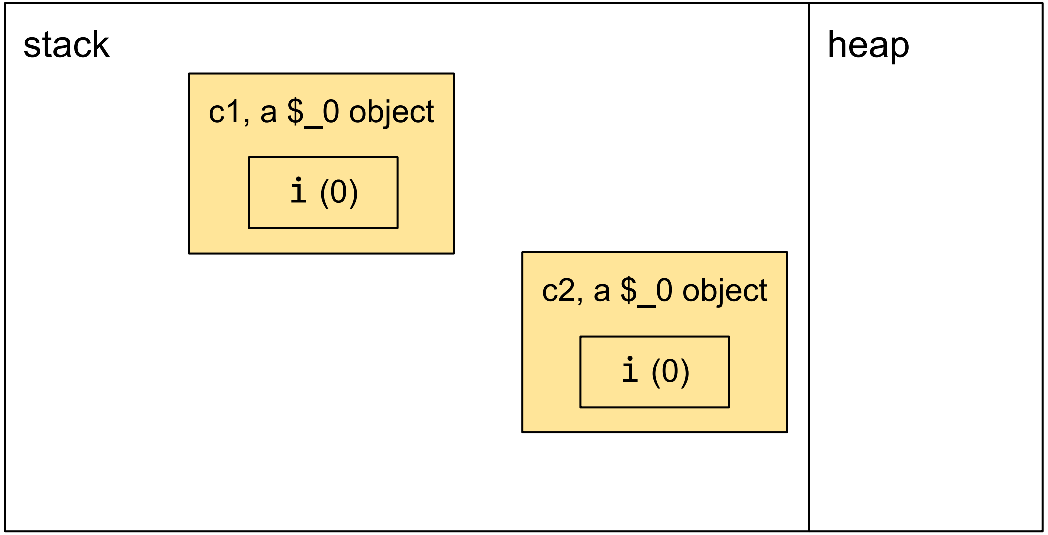
Per-lambda mutable state (wrong!)

```
[i=0]() { return ++i; };
```

Compiler errors!

```
error: increment of
read-only variable 'i'

error: cannot assign to
a variable captured by
copy in a non-mutable
lambda
```



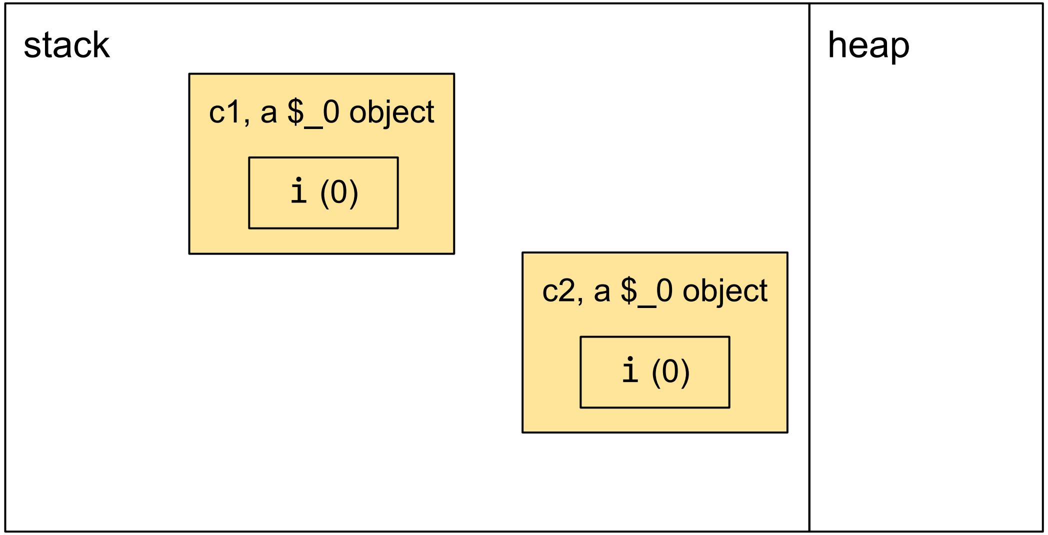
Per-lambda mutable state (right)

```
[i=0]() mutable { return ++i; };
```

`mutable` does not affect the constness of the data members themselves.

It affects the const-qualification of the lambda type's `operator()`.

Therefore it is all-or-nothing.



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... As>
  auto operator()(As... as) {
    return sum(as..., 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... As>
    auto operator() (As... as) const {
        return sum(as..., value);
    }
};

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

Variadic lambdas reduce boilerplate

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

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

What is `this` in a lambda?

What is `this` in a lambda?

```
auto has_title_t = [t=title](const Book& b) {
    return b.title() == t;
};
```

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

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(); }` *Deprecated in C++20!*
- `[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); }`

How do I name the parameter type(s) of a generic lambda?

What's the `T` in `std::forward<T>`?

```
auto plus = [](auto... args) {
    return sum(args...);
};

auto times = [](auto&&... args) {
    return product(std::forward<???>(args)...);
};
```

Two possible solutions (at least)

```
auto one = [](auto&&... args) {
    return product(std::forward<decltype(args)>(args)...);
};

auto two = []<class... Ts>(Ts&&... args) {
    return product(std::forward<Ts>(args)...);
};
```

New in C++20!

Note — `std::forward<Ts>` can always be spelled `static_cast<Ts&&>`, or `std::forward<decltype(a)>` can be spelled `static_cast<decltype(a)>`, to save some template instantiations.

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

“Why does C++ have both?”

Come to my next talk!

Type Erasure From Scratch

1:30pm today in Aurora A

How do I write functions that accept lambdas as arguments?

How do I pass lambdas around?

The “STL” way to accept lambdas is to make all your code into templates, and define those templates in header files.

```
class Shelf {
    template<class Func>
    void for_each_book(Func f) {
        for (const Book& b : books_) f(b);
    }
};

Shelf myshelf;
myshelf.for_each_book([](auto&& book){ book.print(); });
```

This template definition must be visible in the same TU as a declaration of `$_1::operator()` — so, the same TU as the lambda itself.

Suppose this lambda type gets mangled as $_1.

How do I pass lambdas around?

Alternatively, you can use ***type erasure*** to capture your lambda inside a library type that exposes just the call operator.

```
class Shelf {
    void for_each_book(ConcreteCBType f) {
        for (const Book& b : books_) f(b);
    }
};
```

We construct a `ConcreteCBType` object (by implicit conversion) from our original rvalue of type $_1.

This function definition must be in the same TU as a declaration of `ConcreteCBType::operator()` — so, you have more freedom where to place it.

```
Shelf myshelf;
myshelf.for_each_book([](auto&& book){ book.print(); });
```

`ConcreteCBType` might just be an alias for `std::function<void(const Book&)>`.

How do I pass lambdas around?

Perhaps write a template interface with a type-erased implementation.

Look, ma, no implicit conversions!

```
class Shelf {
    using ConcreteCBType = FunctionRef<void(const Book&)>;
    void for_each_book_impl(const ConcreteCBType& f);
public:
    template<class F>
    void for_each_book(const F& f) {
        for_each_book_impl(ConcreteCBType(f));
    }
};
```

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, movable, or neither, depending as its captures are copyable, movable, 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

```
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 *move-only function type* such as `folly::Function`.

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

Every codebase needs a move-only function type!

Questions?

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.