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
```

Function templates

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

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

Footnotes:

Template type parameter T is *deduced* from the type of the argument passed in by the caller.

42 is an `int`, so the compiler deduces that the call must be to `plus1<int>`.

3.14 is a `double`, so the compiler deduces that the call must be to `plus1<double>`.

Function templates

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

auto y = plus1<double>(42);
int (*z)(int) = plus1;
```

Footnotes:

We can call `plus1<double>` directly, via *explicit specialization*.

The compiler deduces `T` in a few other contexts, too, such as in contexts requiring a function pointer of a specific type.

Function templates

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

Footnote:

Using the name plus1 in contexts where its meaning is ambiguous is not allowed. The compiler will diagnose your error.

```
auto err = plus1; // oops
```

test.cc:7: ... incompatible initializer of type '<overloaded function type>'

Puzzle #1

```
template <class T>
auto kitten(T t) {
    static int x = 0;
    return (++x) + t;
}

int main() {
    printf("%d ", kitten(1));
    printf("%g\n",   kitten(3.14));
}
```

Puzzle #1

```
template <class T>
auto kitten(T t) {
    static int x = 0;
    return (++x) + t;
}

int main() {
    printf("2 " , kitten(1));
    printf("4.14\n", kitten(3.14));
}
```

Puzzle #1

```
template <class T>
auto kitten(T t) {
    static int x = 0;
    return (++x) + t;
}

int main() {
    printf("2 " , kitten(1));
    printf("4.14\n", kitten(3.14));
}
```

```
__ZZ6kittenIiEDaT_E1x:
    .long 0
__Z6kittenIiEDaT_:
    movq  __ZZ6kittenIiEDaT_E1x, %rax
    movl  (%rax), %ecx
    leal  1(%rcx), %edx
    movl  %edx, (%rax)
    leal  1(%rcx,%rdi), %eax
    retq

__ZZ6kittenIdEDaT_E1x:
    .long 0
__Z6kittenIdEDaT_:
    movq      __ZZ6kittenIdEDaT_E1x, %rax
    movl      (%rax), %ecx
    incl      %ecx
    movl      %ecx, (%rax)
    cvtsi2sdl %ecx, %xmm1
    addsd     %xmm0, %xmm1
    movaps    %xmm1, %xmm0
    retq
```

The rationale is made more apparent when we...

Puzzle #1

```
template <class T>
auto kitten(T t) {
    static T x = 0;
    return (x += 1) + t;
}

int main() {
    printf("2 " , kitten(1));
    printf("4.14\n", kitten(3.14));
}
```

```
__ZZ6kittenIiEDaT_E1x:
    .long 0                ## int 0
__Z6kittenIiEDaT_:
    movq  __ZZ6kittenIiEDaT_E1x, %rax
    movl  (%rax), %ecx
    leal  1(%rcx), %edx
    movl  %edx, (%rax)
    leal  1(%rcx,%rdi), %eax
    retq

__ZZ6kittenIdEDaT_E1x:
    .quad 0                ## double 0.0
__Z6kittenIdEDaT_:
    movq      __ZZ6kittenIdEDaT_E1x, %rax
    movl      (%rax), %ecx
    incl      %ecx
    movl      %ecx, (%rax)
    cvtsi2sdl %ecx, %xmm1
    addsd     %xmm0, %xmm1
    movaps    %xmm1, %xmm0
    retq
```

We’ll come back to templates later.

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?”

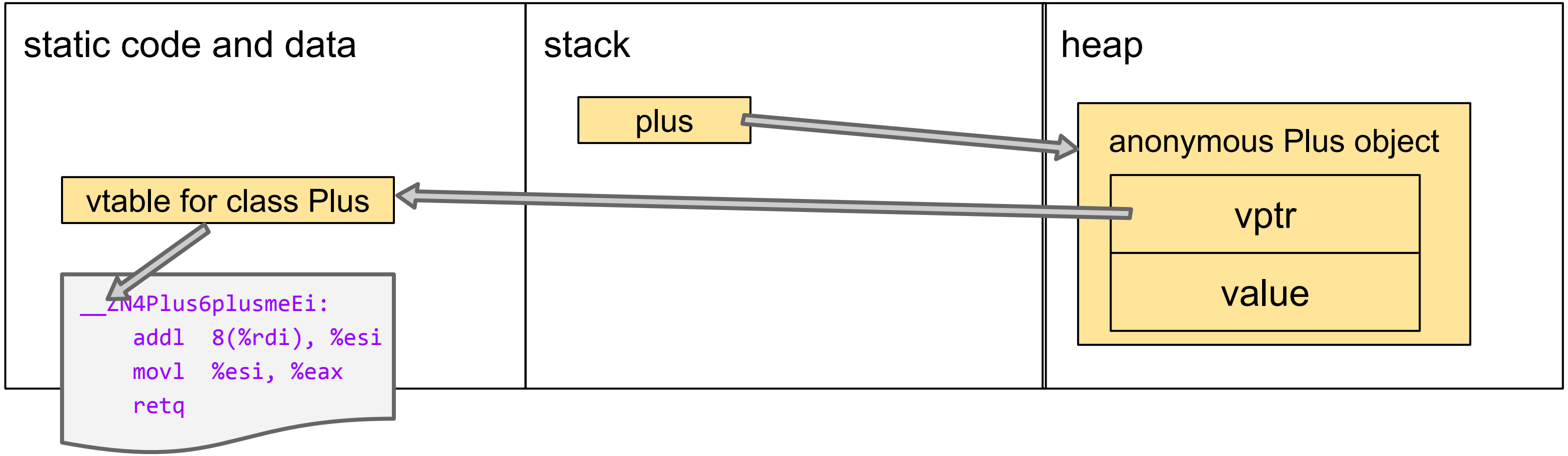
```
auto plus = Plus(1);
auto x = plus.plusme(42);

assert(x == 43);
```

C++ is not Java!

The Java approach

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

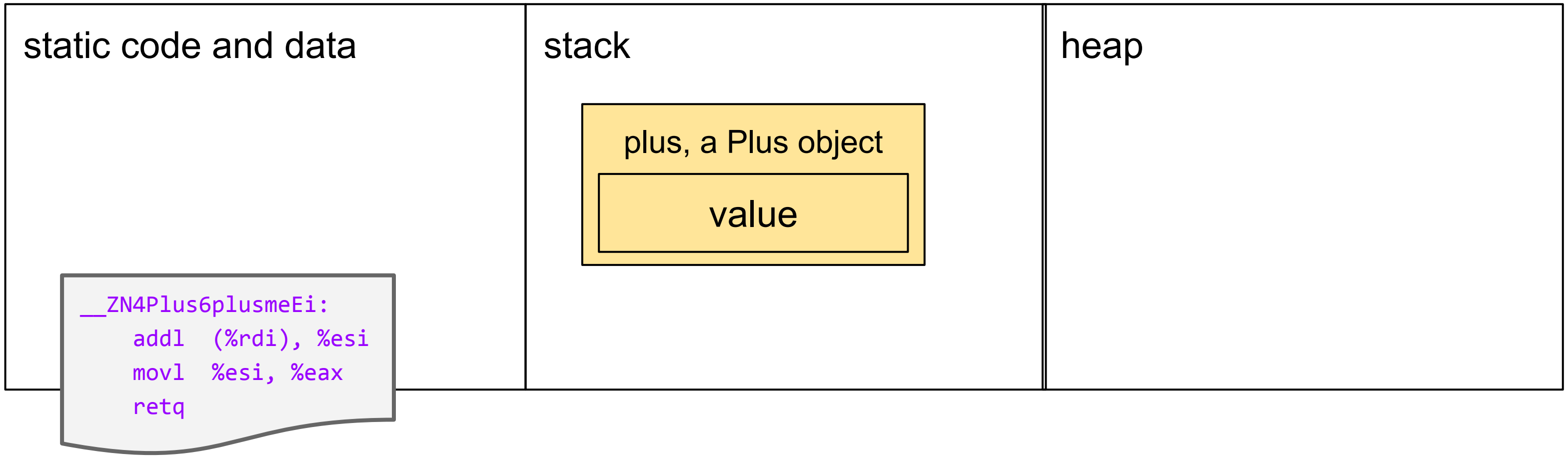
C++ lets you do this,
but it's not the default.



The C++ approach

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

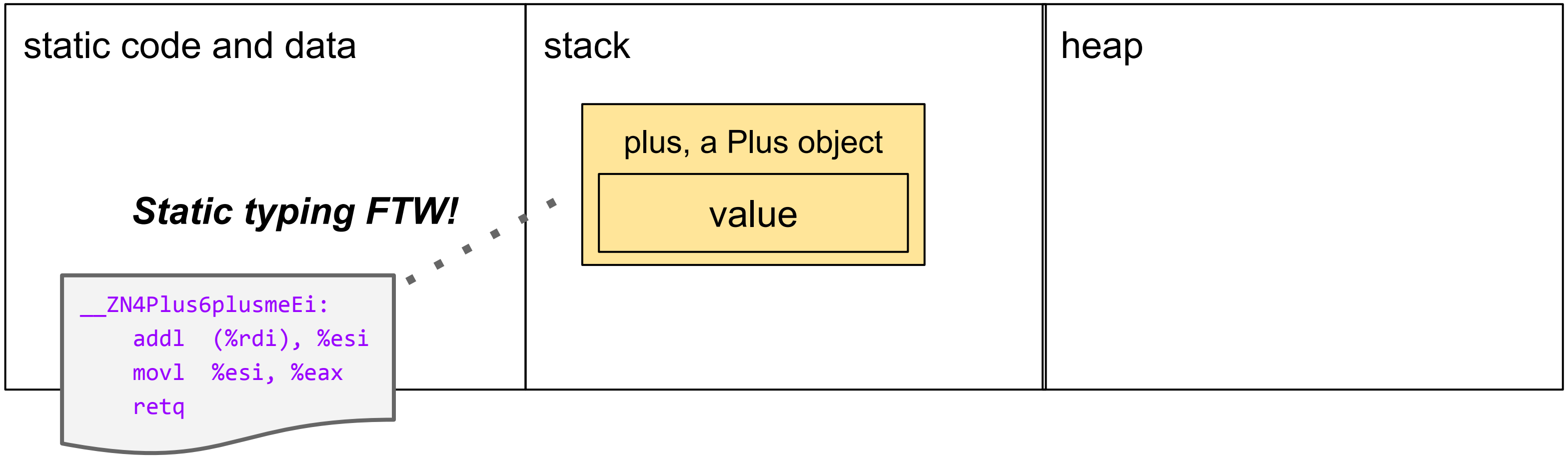
```
movl  $1, %esi
leaq  -16(%rbp), %rdi
callq __ZN4PlusC1Ei
movl  $42, %esi
leaq  -16(%rbp), %rdi
callq __ZN4Plus6plusmeEi
```



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);
```

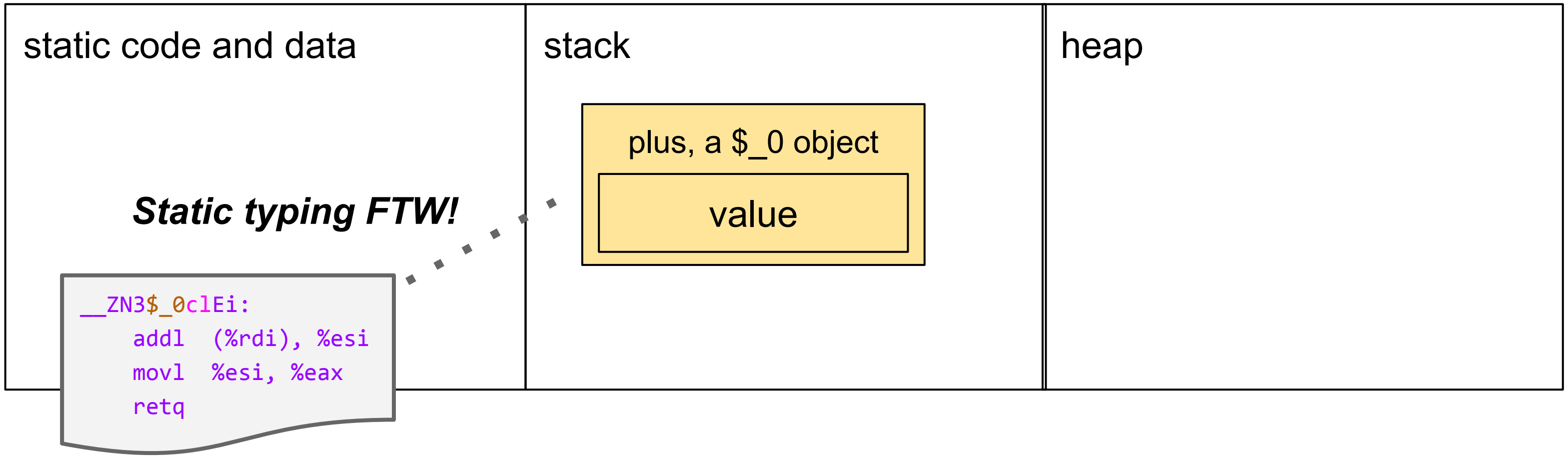
Lambdas reduce boilerplate

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

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

Same implementation

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

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

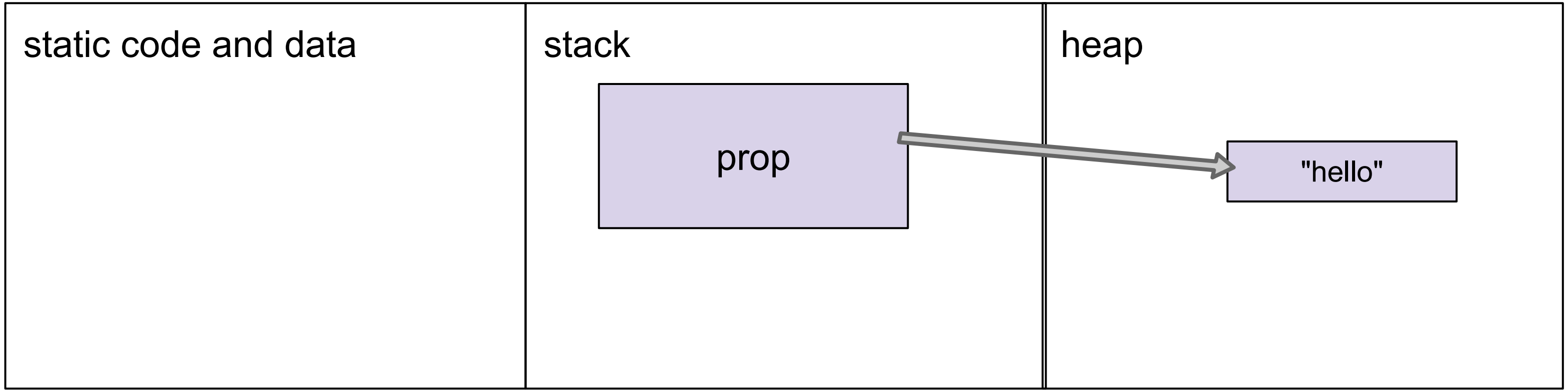


Closures without garbage collection

```
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);
}
```

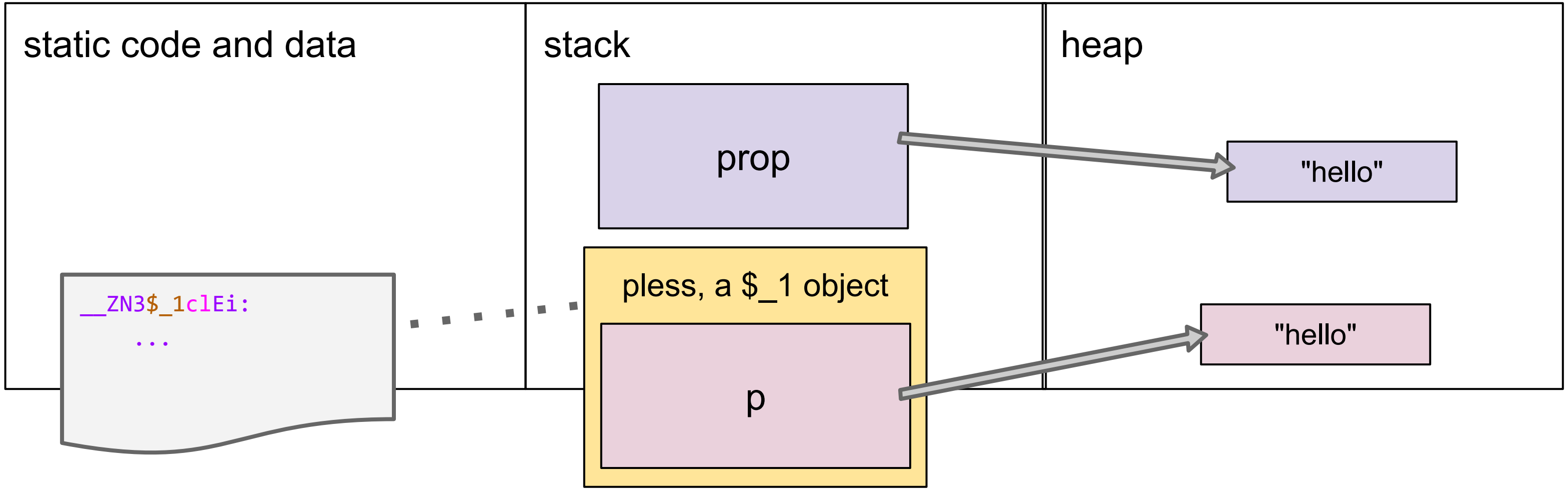
Closures without garbage collection

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



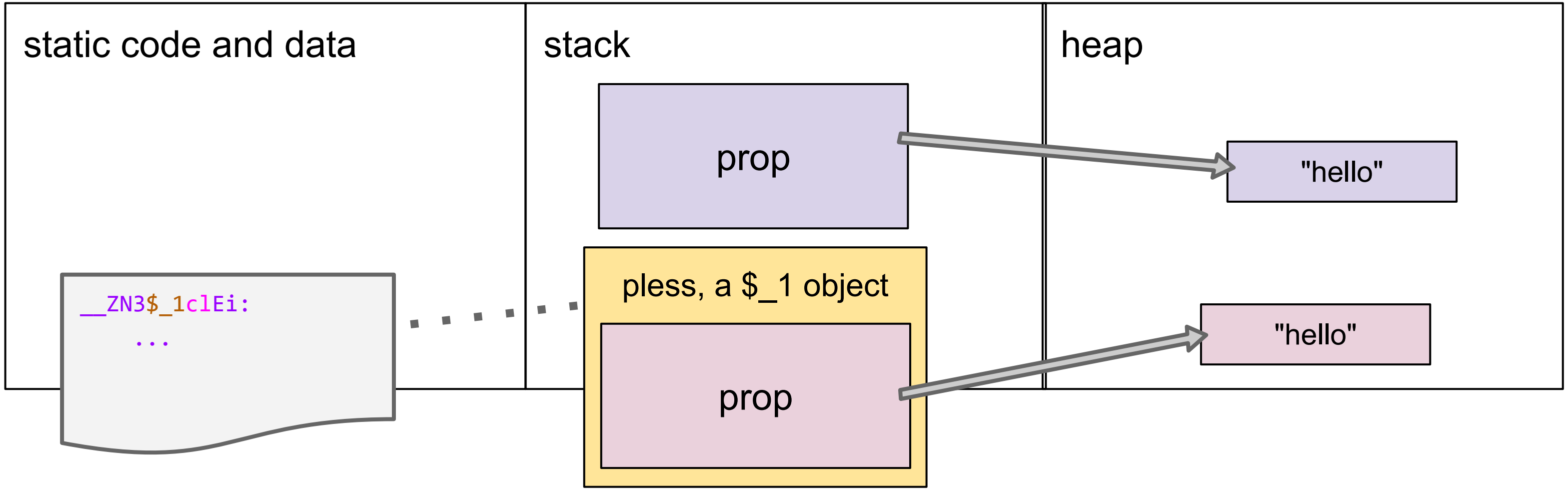
Closures without garbage collection

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



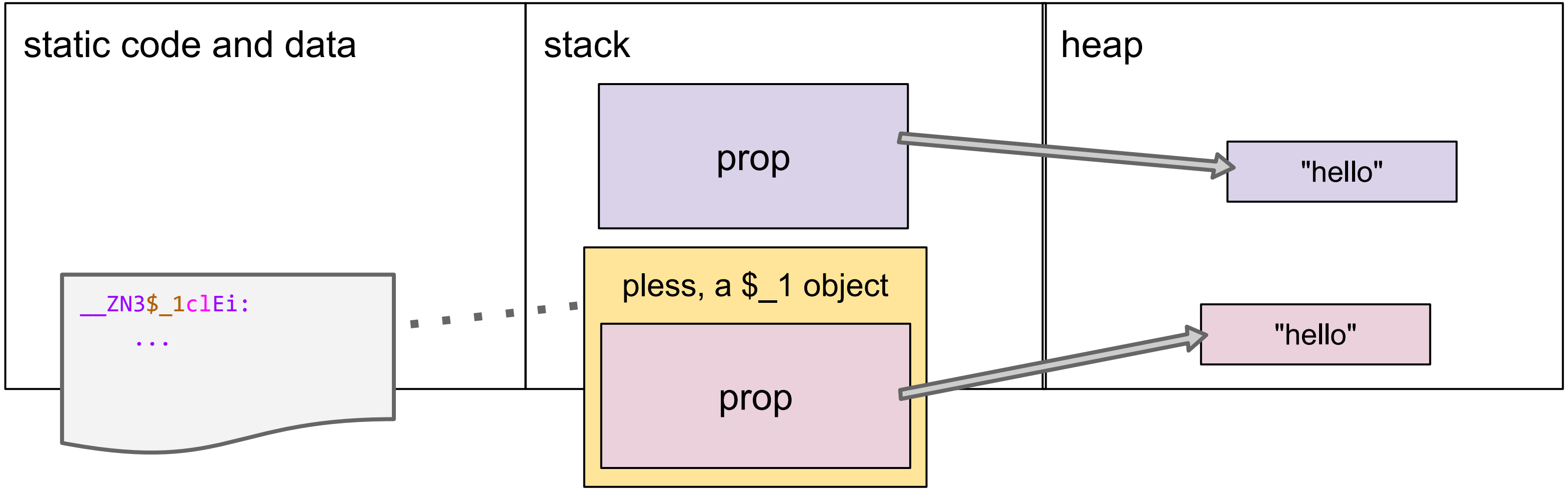
Closures without garbage collection

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



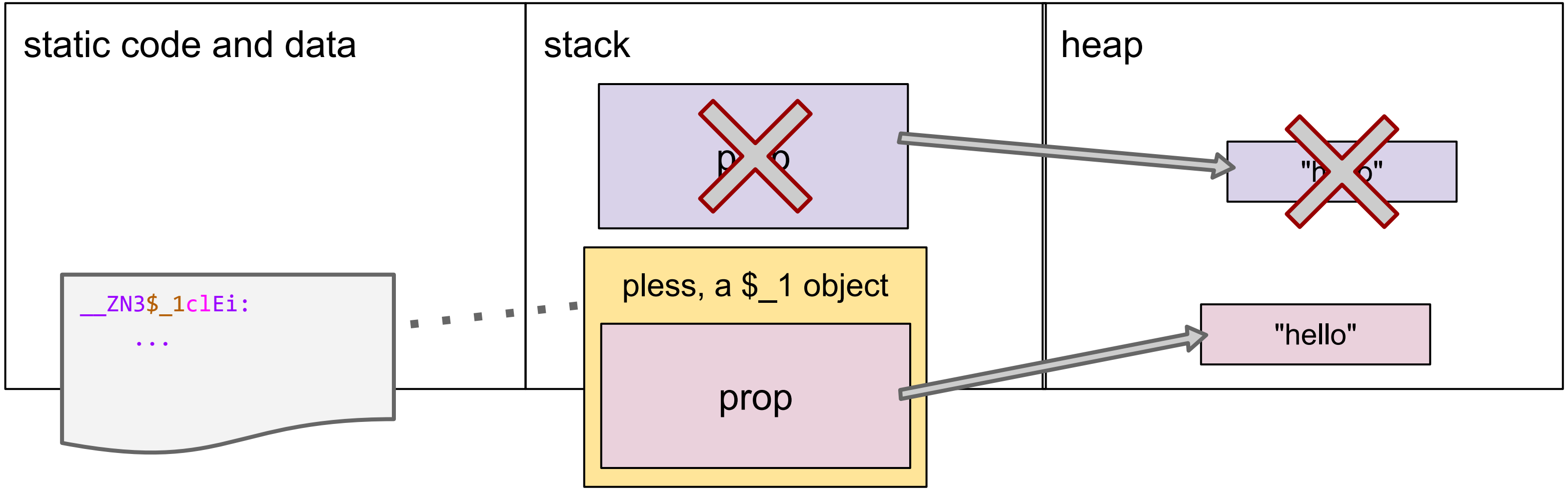
Copy semantics by default

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



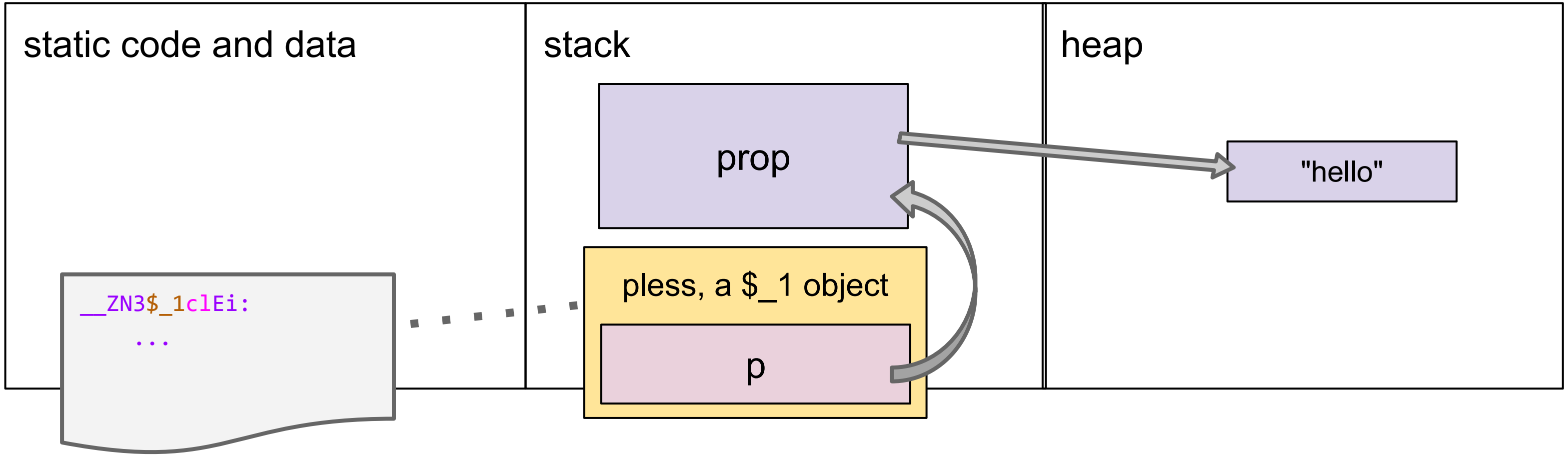
Copy semantics by default

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



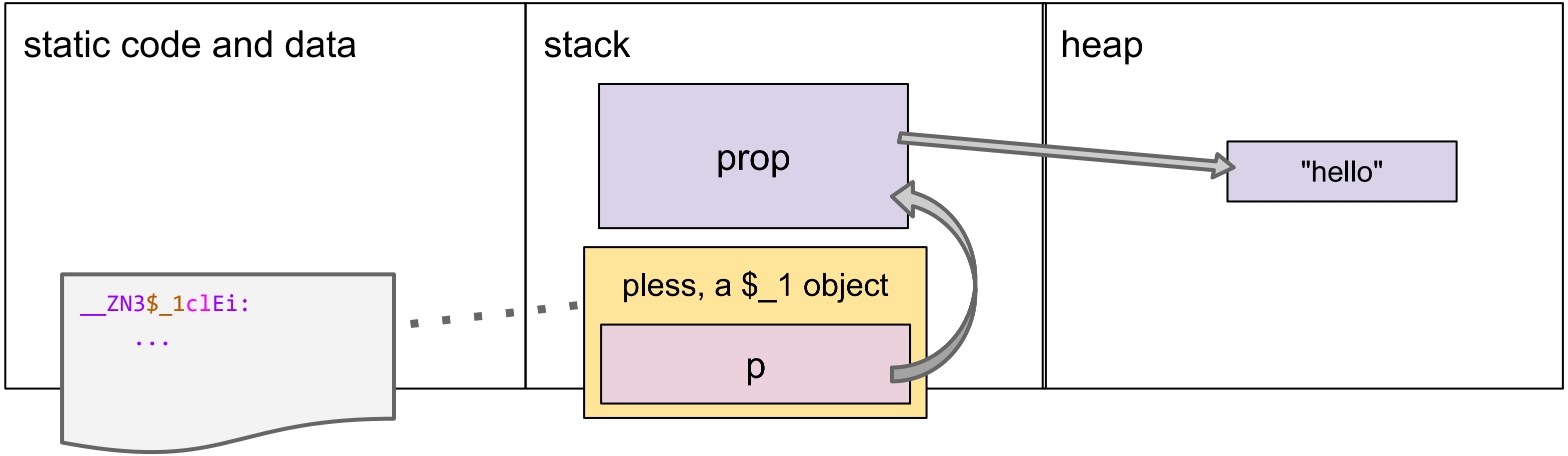
Capturing a reference

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



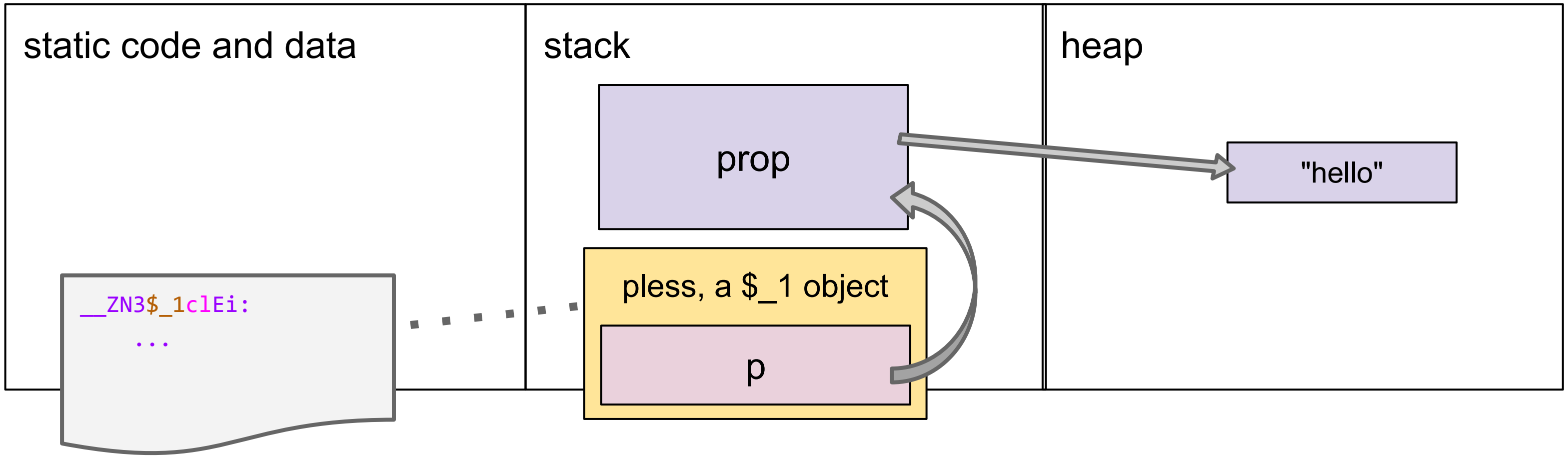
Capturing a reference

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



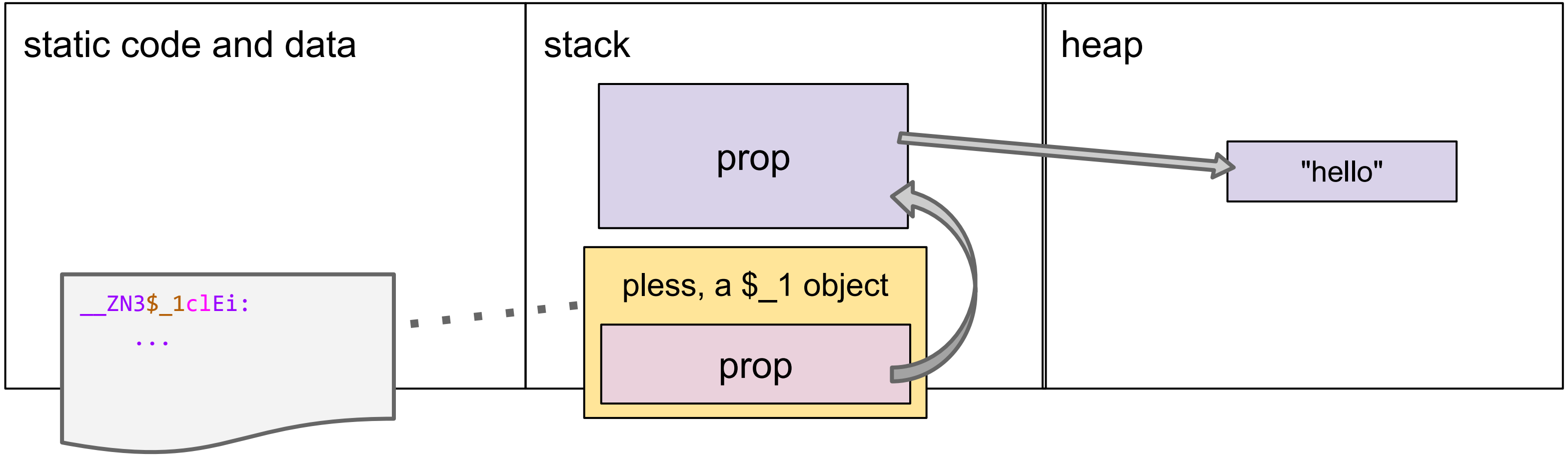
Capturing by reference

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



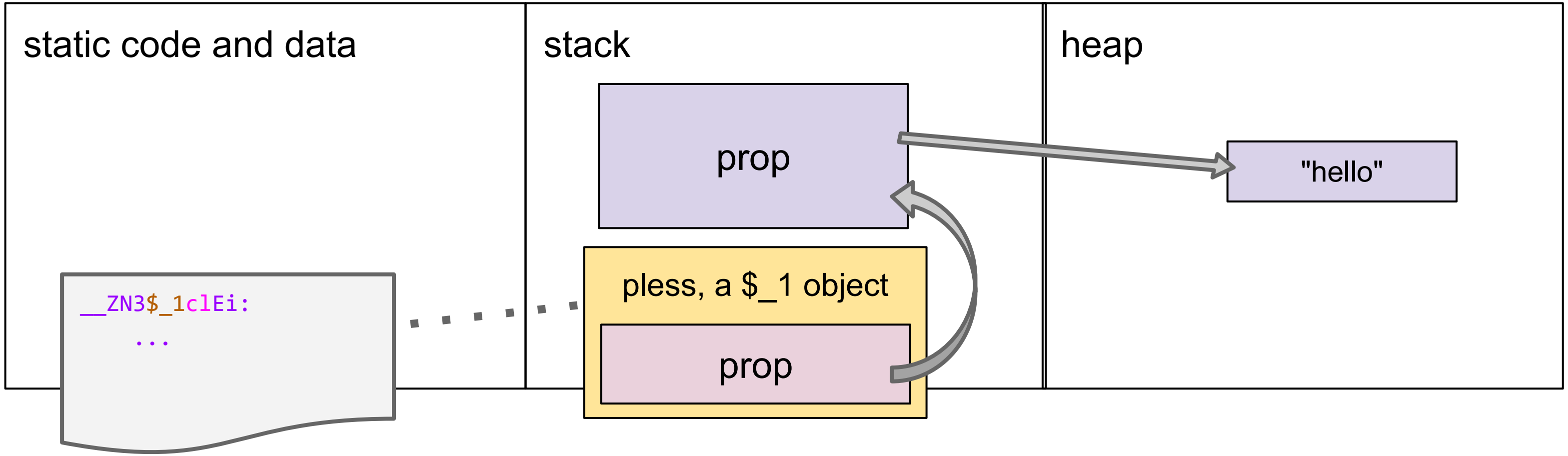
Capturing by reference

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



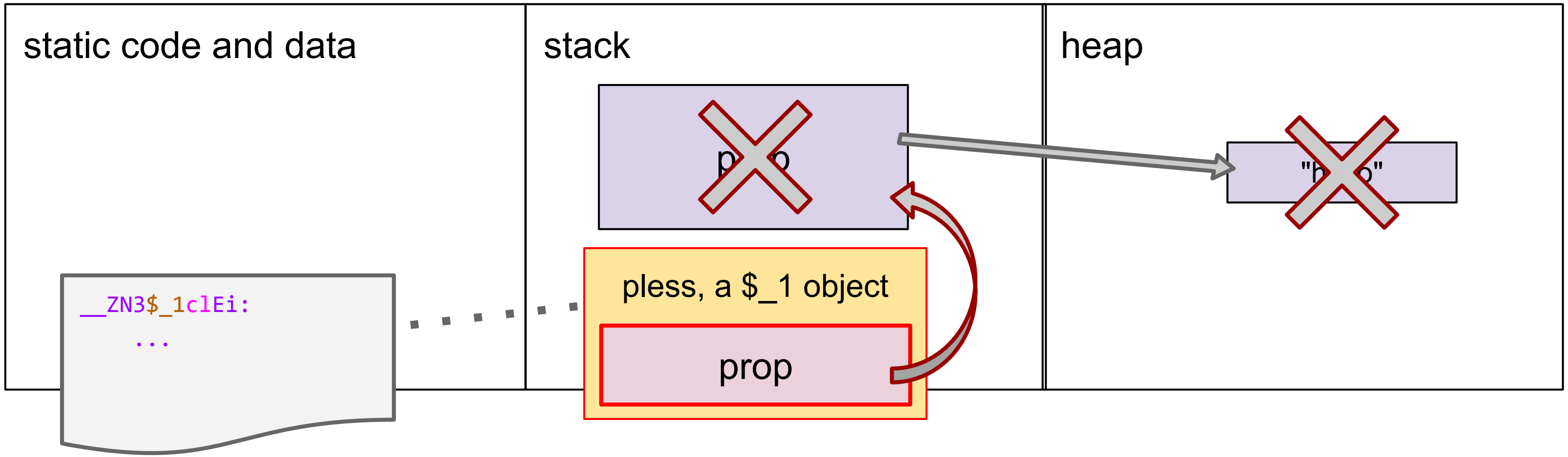
Capturing by reference

```
... 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 value vs. by reference

```
auto GOOD_increment_by(int y) {
    return [=](int x) { return x+y; };
}

auto BAD_increment_by(int y) {
    return [&](int x) { return x+y; };
}

auto plus5 = GOOD_increment_by(5);
int seven = plus5(2);
```

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 #2

```
#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 #2

```
#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 #2 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.

However, this is just a warning in GCC (it’s an error in Clang).

Puzzle #3

```
auto make_kitten(int c) {
    static int a = 0;
    return [=](int d) {
        static int b = 0;
        return (a++) + (b++) + c + d;
    };
}

int main() {
    auto k1 = make_kitten(1), k2 = make_kitten(2);
    printf("%d ", k1(20)); printf("%d\n", k1(30));
    printf("%d ", k2(20)); printf("%d\n", k2(30));
}
```

a — static outside the lambda
b — static inside the lambda
c — captured by value
d — the lambda’s own argument

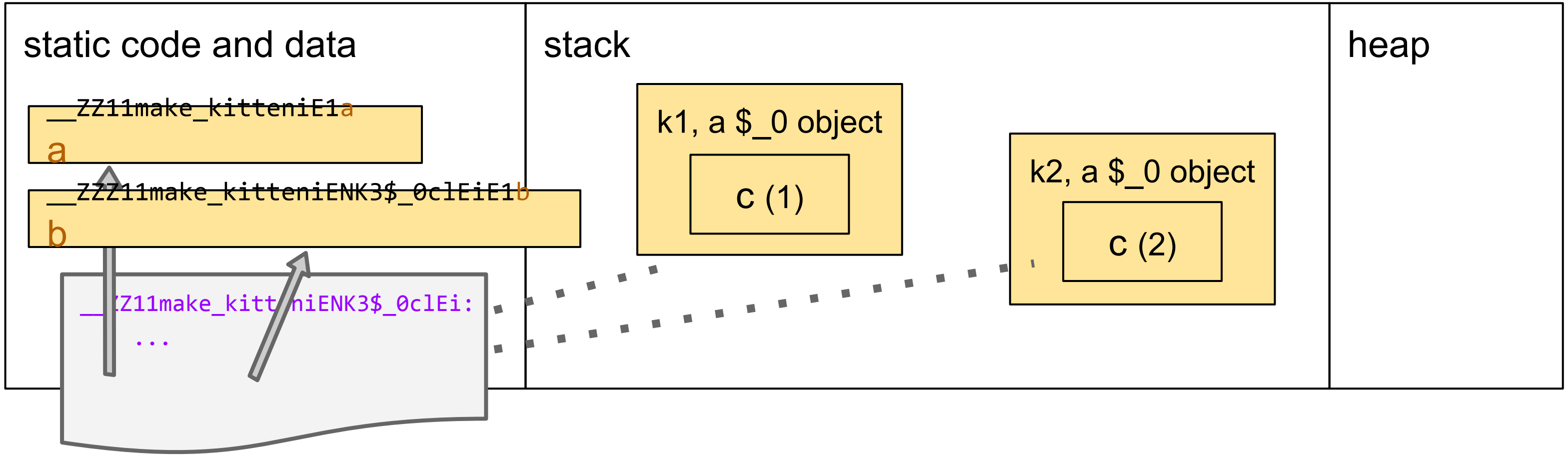
Puzzle #3

```
auto make_kitten(int c) {
    static int a = 0;
    return [=](int d) {
        static int b = 0;
        return (a++) + (b++) + c + d;
    };
}

int main() {
    auto k1 = make_kitten(1), k2 = make_kitten(2);
    printf("21 ", k1(20)); printf("33\n", k1(30));
    printf("26 ", k2(20)); printf("38\n", k2(30));
}
```

a — static outside the lambda
b — static inside the lambda
c — captured by value
d — the lambda's own argument

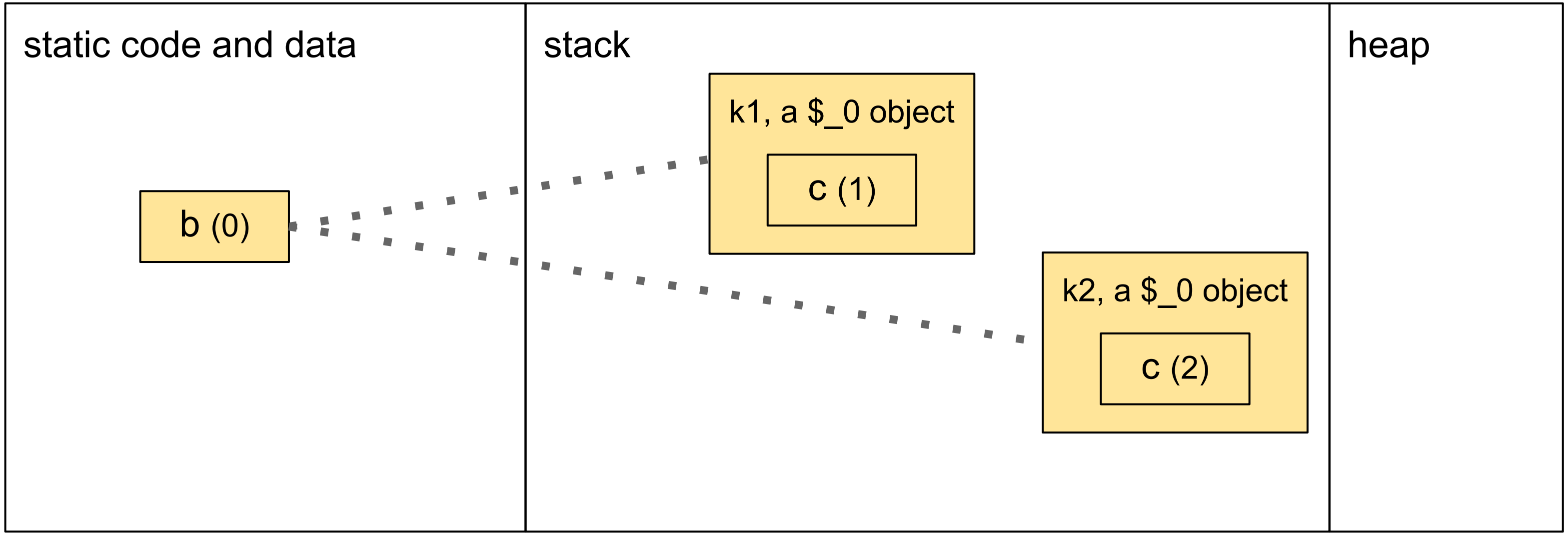
Puzzle #3

```
... static int a = 0; return [=](int d) {
                    static int b = 0;
                    return (a++) + (b++) + c + d; };
... auto k1 = make_kitten(1), k2 = make_kitten(2); ...
```



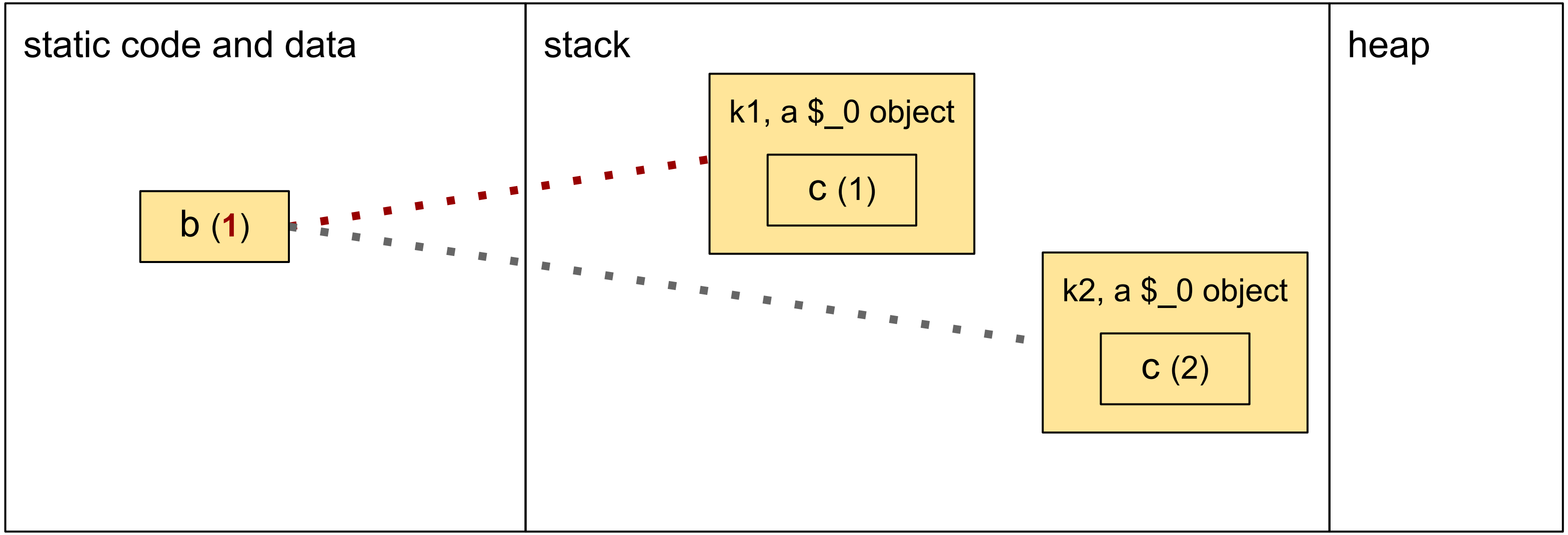
Per-lambda mutable state

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

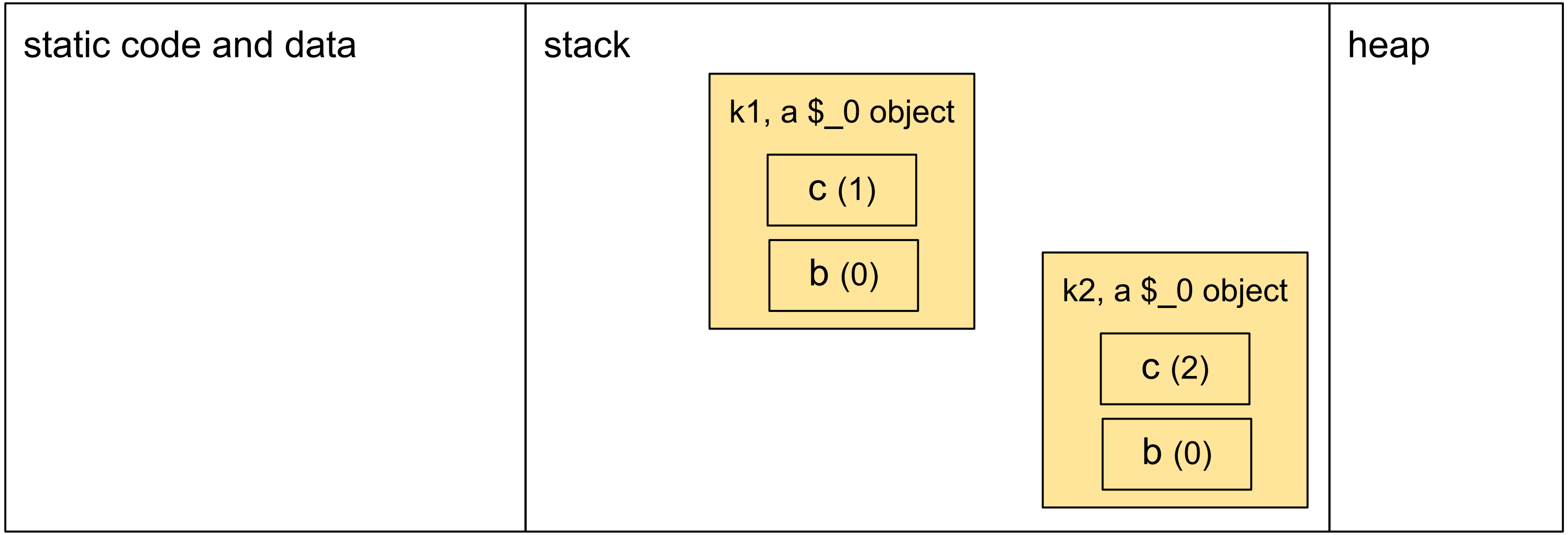


Per-lambda mutable state

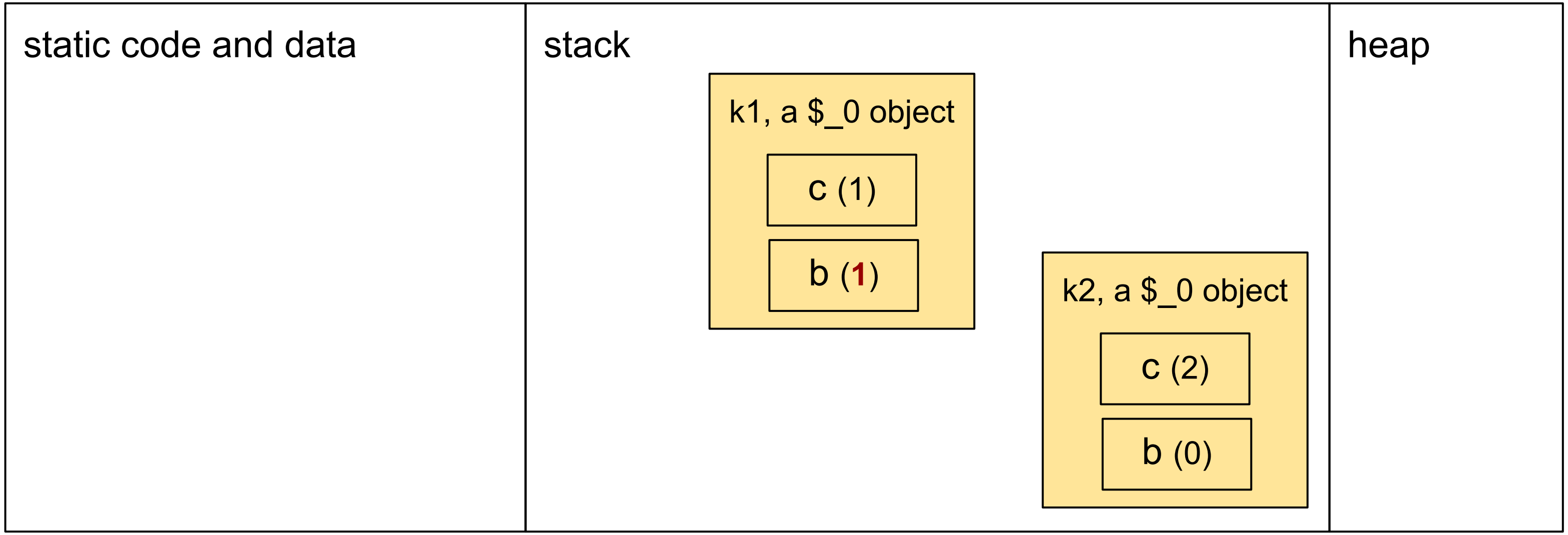
```
... [c](int d) { static int b; ... } ...
```



Per-lambda mutable state



Per-lambda mutable state



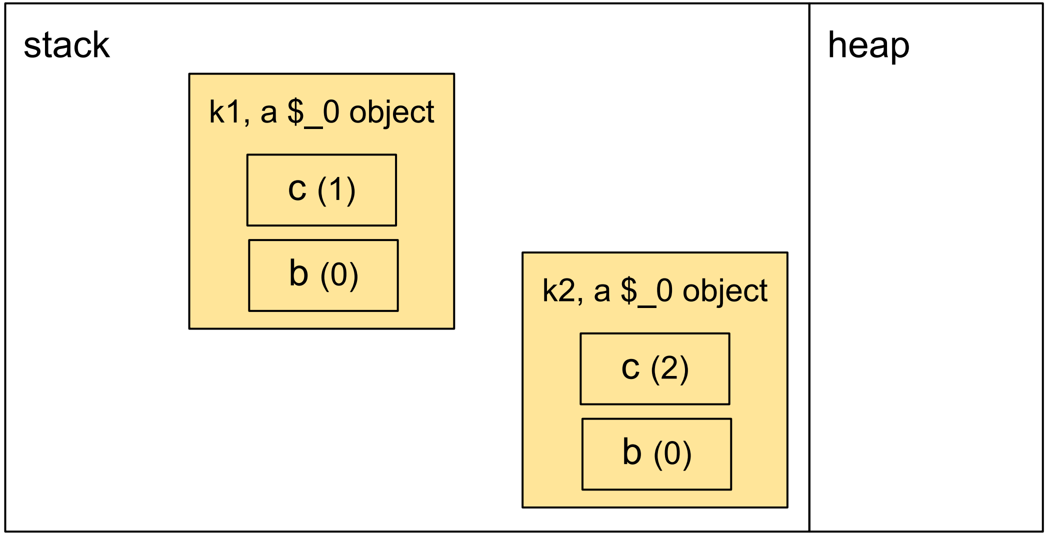
Per-lambda mutable state

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

Footnote:

mutable is all-or-nothing.

Generally speaking, captures aren’t modifiable... and you wouldn’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);
```

Puzzle #3 redux

```
auto kitten = [](auto t) {
    static int x = 0;
    return (++x) + t;
};

int main() {
    printf("%d ", kitten(1));
    printf("%g\n",   kitten(3.14));
}
```

Puzzle #1 redux

```
auto kitten = [](auto t) {
    static int x = 0;
    return (++x) + t;
};

int main() {
    printf("2 " , kitten(1));
    printf("4.14\n", kitten(3.14));
}
```

```
__ZZNK3$_0clIiEEDaT_E1x:
    .long 0
__ZN3$_08__invokeIiEEDaT_:
    movq   __ZZNK3$_0clIiEEDaT_E1x, %rax
    movl   (%rax), %ecx
    leal   1(%rcx), %edx
    movl   %edx, (%rax)
    leal   1(%rcx,%rdi), %eax
    retq

__ZNK3$_0clIdEEDaT_E1x:
    .long 0
__ZN3$_08__invokeIdEEDaT_:
    movq      __ZZNK3$_0clIdEEDaT_E1x, %rax
    movl      (%rax), %ecx
    incl      %ecx
    movl      %ecx, (%rax)
    cvtsi2sdl %ecx, %xmm1
    addsd     %xmm0, %xmm1
    movaps    %xmm1, %xmm0
    retq
```

Generic lambdas are just templates under the hood.

Variadic function templates

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

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

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

Variadic lambdas reduce boilerplate

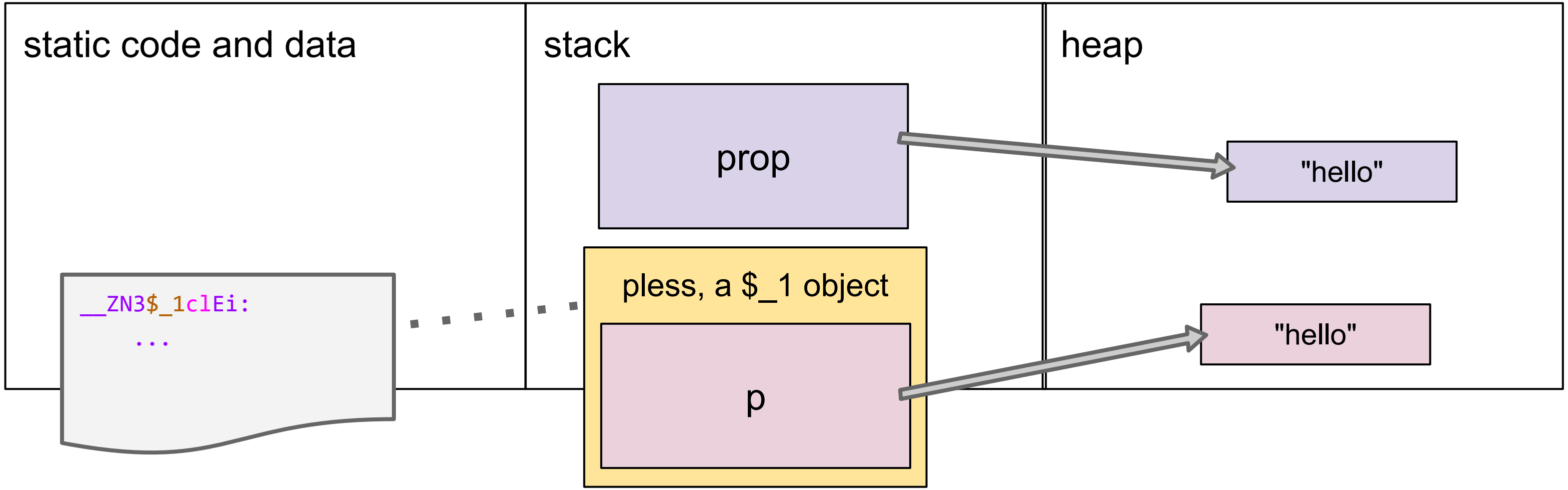
```
auto plus = [value=1](auto... p) {
    return sum(p..., value);
};

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

Capturing a whole parameter-pack

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

template<typename... P>
void sort_by_properties(std::vector<object>& v, P... props)
{
    auto pless = [props...](object& a, object& b) {
        return tie(a[props]...) < tie(b[props]...);
    };

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

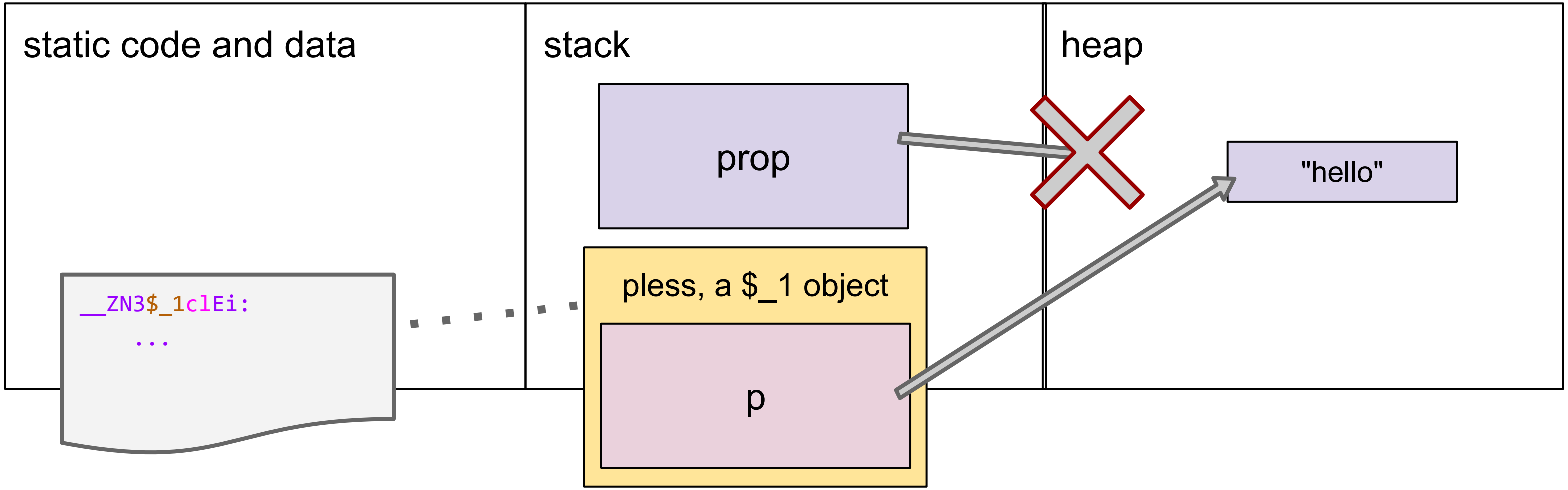
Capturing "by move"

```
... 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];
    };
```



Capturing a whole parameter-pack "by move"?

```
template<typename... P>
void sort_by_properties(std::vector<object>& v, P... props)
{
    auto pless = [p=std::move(props)...](object& a, object& b) {
        return tie(a[p]...) < tie(b[p]...);
    };

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

Capturing a whole parameter-pack “by move”?

```
template<typename... P>
void sort_by_properties(std::vector<object>& v, P... props)
{
    auto pless = [p=std::move(props)...](object& a, object& b) {
        return tie(a[p]...) < tie(b[p]...);
    };

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

Capturing a whole parameter-pack “by move”?

```
template<typename... P>
void sort_by_properties(std::vector<object>& v, P... props)
{
    auto tprops = std::make_tuple(std::move(props)...);
    auto pless = [tp=std::move(tprops)](object& a, object& b) {
        return YUCK;
    };

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

Questions?

Otherwise we'll talk about std::bind.

std::bind is obsolete as of C++14

It lets you wrap up certain arguments to a function call while leaving others unspecified until later. But you have to define the code itself out-of-line.

```
int add(int x, int y) {
    return x + y;
}

auto plus5 = std::bind(add, std::placeholders::_1, 5);
auto plus5 = [](auto x, auto...) {
    return add(std::forward<decltype(x)>(x), 5);
};
auto z = plus5(42);
assert(z == 47);
```

EMC++ Item 34

```
// at time t, make sound s for duration d
void setAlarm(Time t, Sound s, Duration d);

// setSoundL ("L" for "Lambda") is a function object
// allowing a sound to be specified for a 30-sec alarm
// to go off an hour after it's set
auto setSoundL = [](Sound s) {
    using namespace std::chrono;
    using namespace std::literals;
    setAlarm(steady_clock::now() + 1h, s, 30s);
};
```

EMC++ Item 34

```
// at time t, make sound s for duration d
void setAlarm(Time t, Sound s, Duration d);

// setSoundB ("B" for "Bind") is a function object
// allowing a sound to be specified for a 30-sec alarm
// to go off an hour after it's set... or is it?
using namespace std::chrono;
using namespace std::literals;
auto setSoundB = std::bind(
    setAlarm, steady_clock::now() + 1h, _1, 30s
);
```

We must defer the call to now()

```
// at time t, make sound s for duration d
void setAlarm(Time t, Sound s, Duration d);

// setSoundB ("B" for "Bind") is a function object
// allowing a sound to be specified for a 30-sec alarm
// to go off an hour after it's set... or is it?
using namespace std::chrono;
using namespace std::literals;
auto setSoundB = std::bind(
    setAlarm, std::bind(steady_clock::now) + 1h, _1, 30s
);
```

We must defer the call to operator+

```
// at time t, make sound s for duration d
void setAlarm(Time t, Sound s, Duration d);

// setSoundB ("B" for "Bind") is a function object
// allowing a sound to be specified for a 30-sec alarm
// to go off an hour after it's set... or is it?
using namespace std::chrono;
using namespace std::literals;
auto setSoundB = std::bind(
    setAlarm, std::bind(steady_clock::now) + 1h, _1, 30s
);
```

The corrected std::bind code

```
using namespace std::chrono;
using namespace std::literals;
auto setSoundB = std::bind(
    setAlarm,
    std::bind(
        std::plus<>{},
        std::bind(
            steady_clock::now),
        1h),
    _1,
    30s);
```

Lambdas FTW

```
auto setSoundL = [](Sound s) {
    using namespace std::chrono;
    using namespace std::literals;
    setAlarm(steady_clock::now() + 1h, s, 30s);
};
```

Questions?

Otherwise we'll talk about std::function.

std::function provides *type erasure*

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

static_assert(!is_same_v<decltype(fplus), decltype(lplus)>);  // different

using i2i = std::function<int(int)>;

i2i wrappedf = fplus;
i2i wrappedl = lplus;

static_assert(is_same_v<decltype(wrappedf), decltype(wrappedl)>);  // same
```

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`.

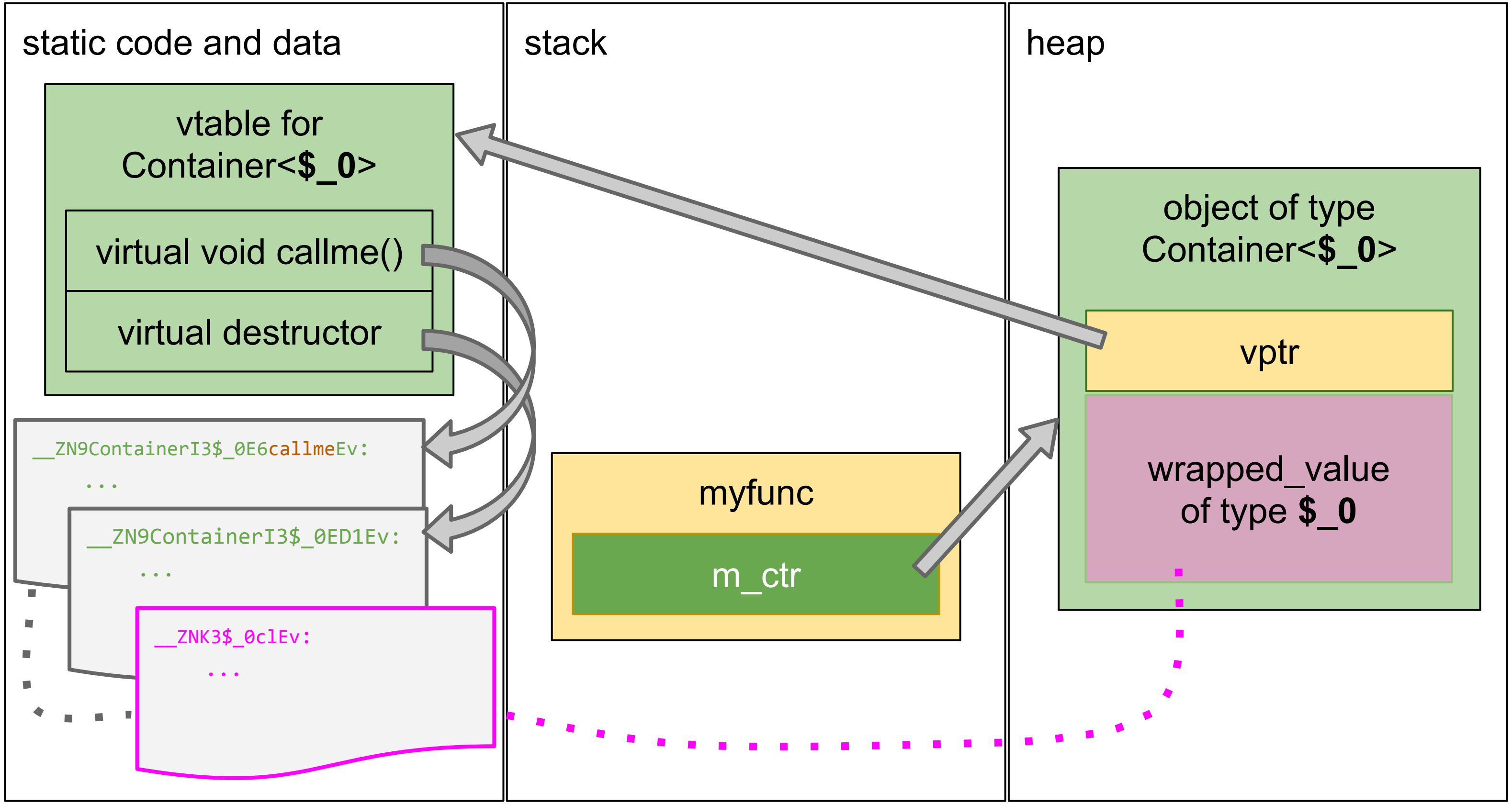
Before we can talk about callbacks, we need `std::function`.

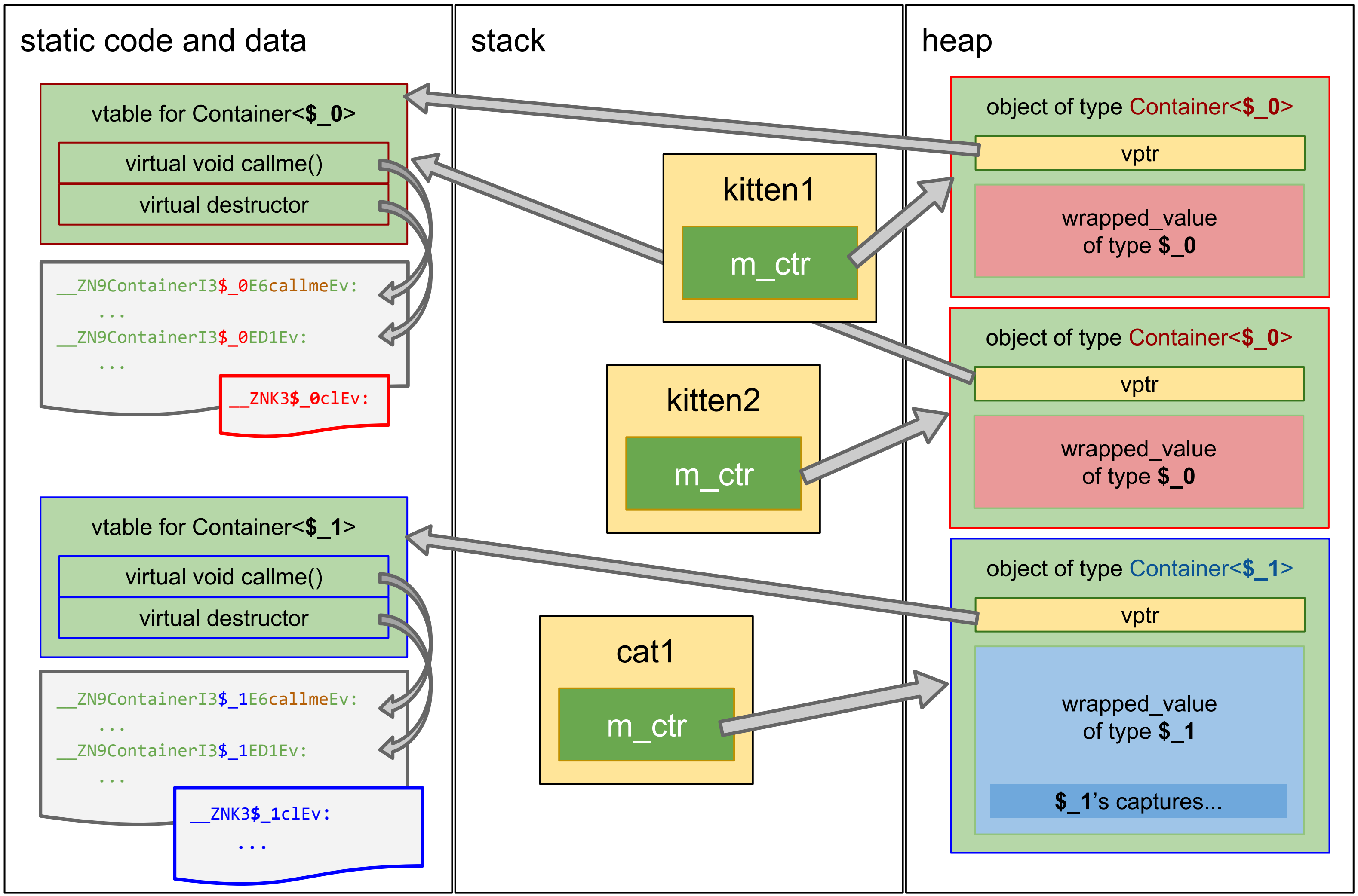
`std::function` allows us to pass lambdas, functor objects, etc., across *module boundaries*.

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) {}
    virtual 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)) {}
    void operator()(int i) { return m_ctr->callme(i); }   // virtual dispatch
    ~i2i() { delete m_ctr; }                 // virtual dispatch
};
```

Java++


static code and data
stack
heap
vtable for
Container<**$_0**>
virtual void callme()
virtual destructor
__ZN9ContainerI3$_0E6callmeEv:
...
__ZN9ContainerI3$_0ED1Ev:
...
__ZNK3$_0clEv:
...
myfunc
m_ctr
object of type
Container<**$_0**>
vptr
wrapped_value
of type **$_0**


static code and data
vtable for Container<$_0>
virtual void callme()
virtual destructor
__ZN9ContainerI3$_0E6callmeEv:
...
__ZN9ContainerI3$_0ED1Ev:
...
__ZNK3$_0clEv:
vtable for Container<$_1>
virtual void callme()
virtual destructor
__ZN9ContainerI3$_1E6callmeEv:
...
__ZN9ContainerI3$_1ED1Ev:
...
__ZNK3$_1clEv:
...
stack
kitten1
m_ctr
kitten2
m_ctr
cat1
m_ctr
heap
object of type Container<$_0>
vptr
wrapped_value of type $_0
object of type Container<$_0>
vptr
wrapped_value of type $_0
object of type Container<$_1>
vptr
wrapped_value of type $_1
$_1's captures...