Back to Basics: Smart Pointers

Arthur O’Dwyer
2019-09-19

Outline

- Unique ownership with `std::unique_ptr` [5–18]
- Shared ownership with `std::shared_ptr` [19–30]
- `make_shared` and `make_unique` [31–39]
- `std::weak_ptr` [40–46]
- `std::enable_shared_from_this` [47–52]
- Questions?

C++11 (and up) smart pointer types

`auto_ptr`

C++98. Deprecated in C++11. Removed in C++17.

`unique_ptr`

C++11 replacement for `auto_ptr`. C++14 adds `make_unique`.

`shared_ptr`

C++11. Reference-counting. C++17 adds `shared_ptr<T[]>`.

`weak_ptr`

C++11. "Weak" references.

All the standard smart pointers, plus `make_shared<T>`, arrived together in C++11.

`make_shared<T[]>` finally arrives in C++20.

Smart pointers look familiar, by design

```
std::shared_ptr<String> p =
    std::make_shared<String>("Hello");

auto q = p;

p = nullptr;

if (q != nullptr) {
    std::cout << q->Length() << *q << '\n';
}
```

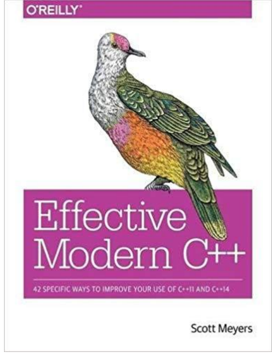
Creation of heap-allocated objects is slightly novel (more on this later)

Initialization (and assignment) works as you'd expect

So does `nullptr`

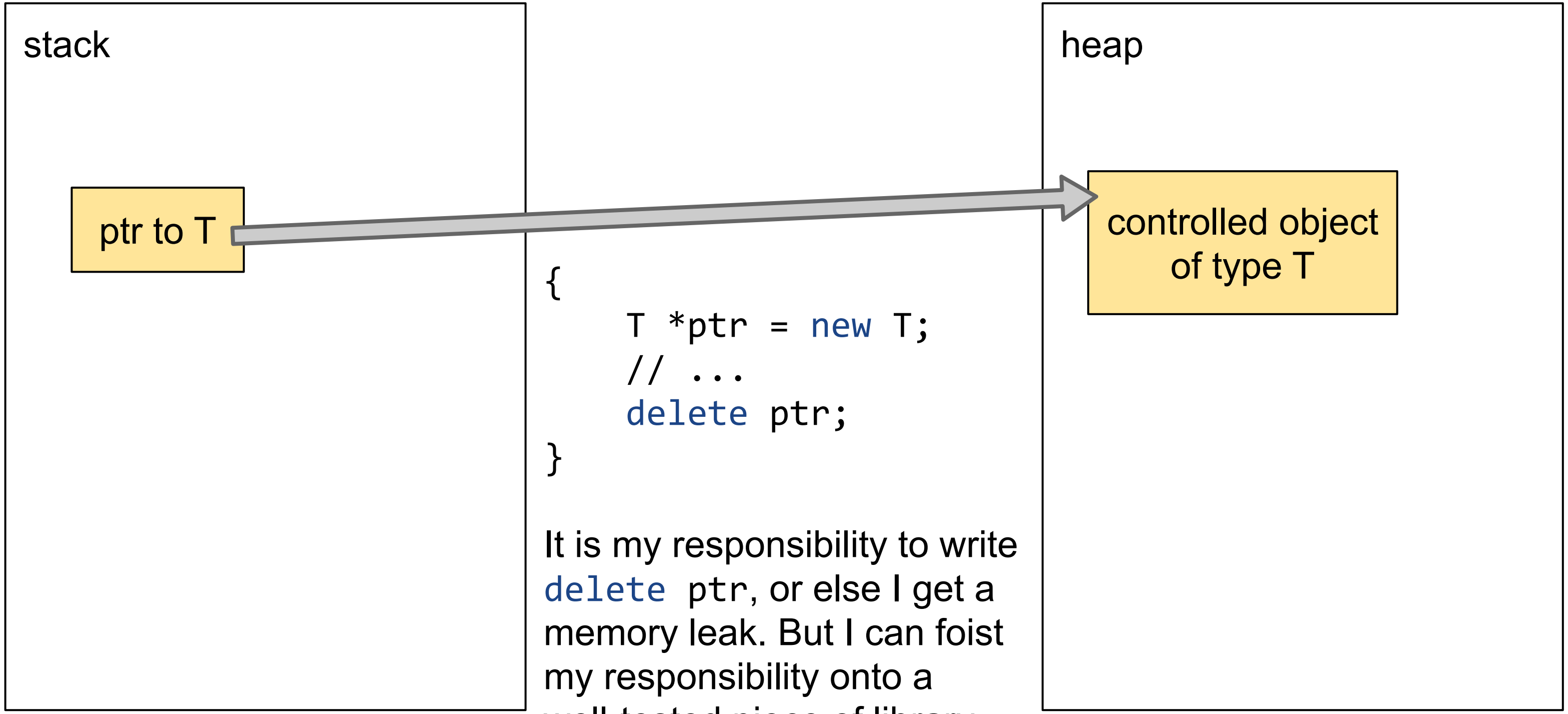
So does comparison and dereferencing

Scott Meyers EMC++ Item 18



Use **`std::unique_ptr`** for exclusive-ownership resource management.

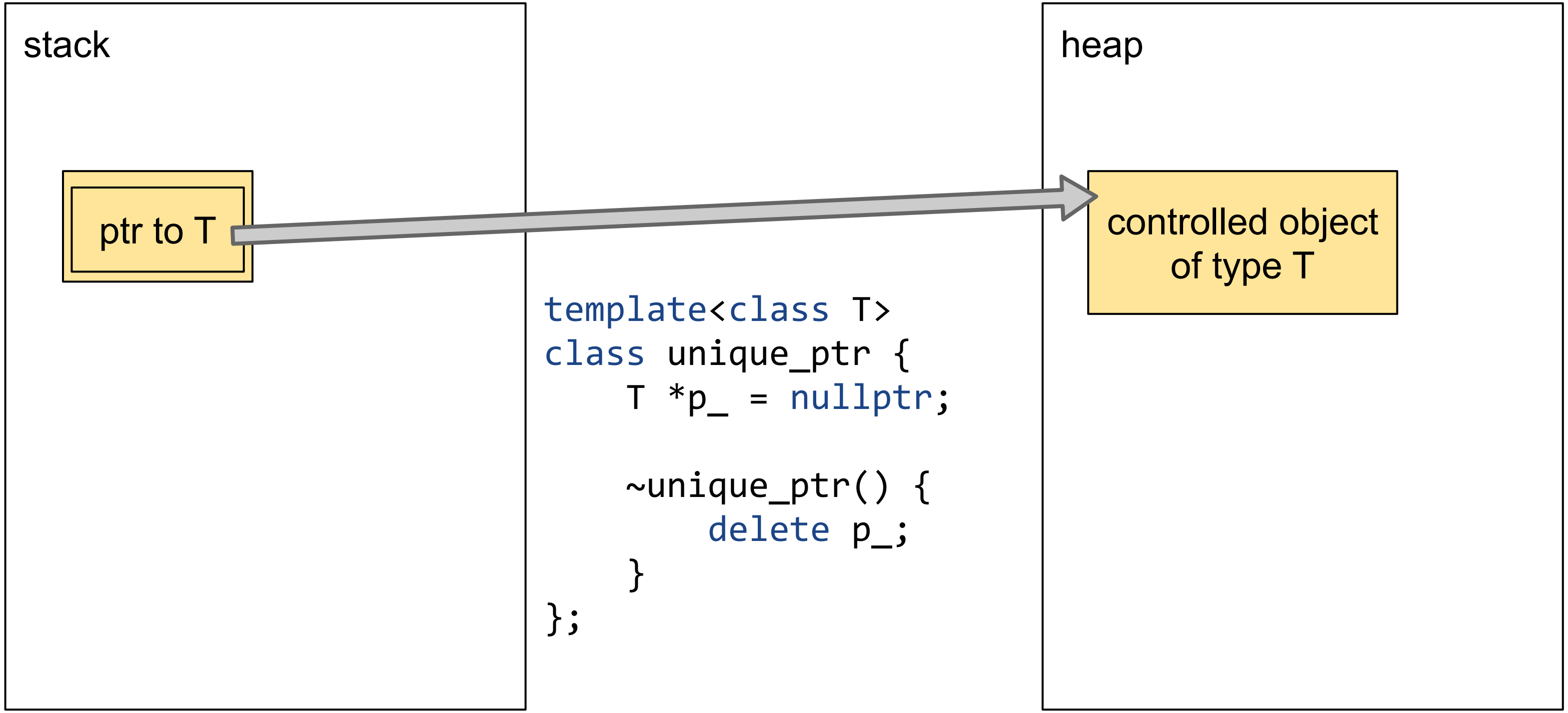
Exclusive ownership




```
{
    T *ptr = new T;
    // ...
    delete ptr;
}
```

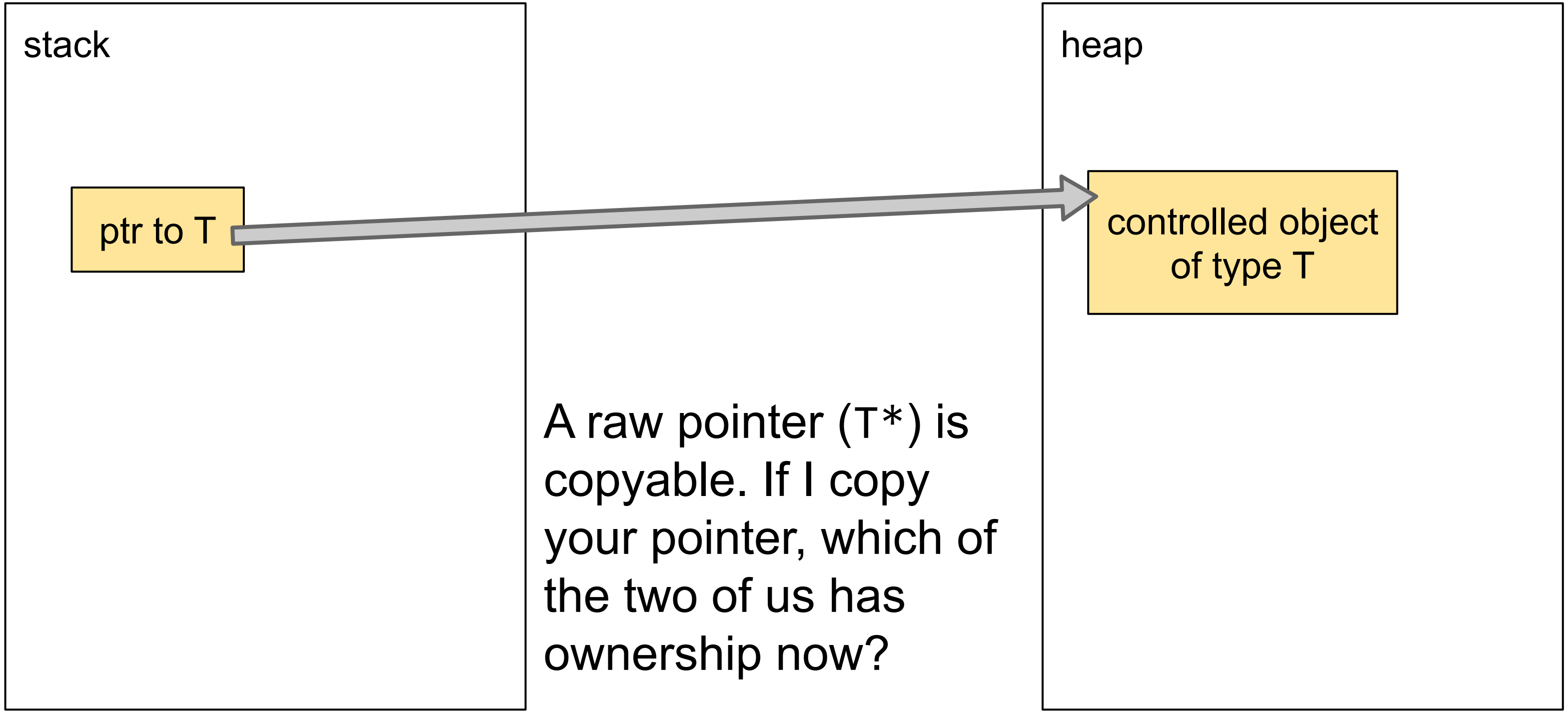
It is my responsibility to write `delete ptr`, or else I get a memory leak. But I can foist my responsibility onto a well-tested piece of library code.

std::unique_ptr<T>



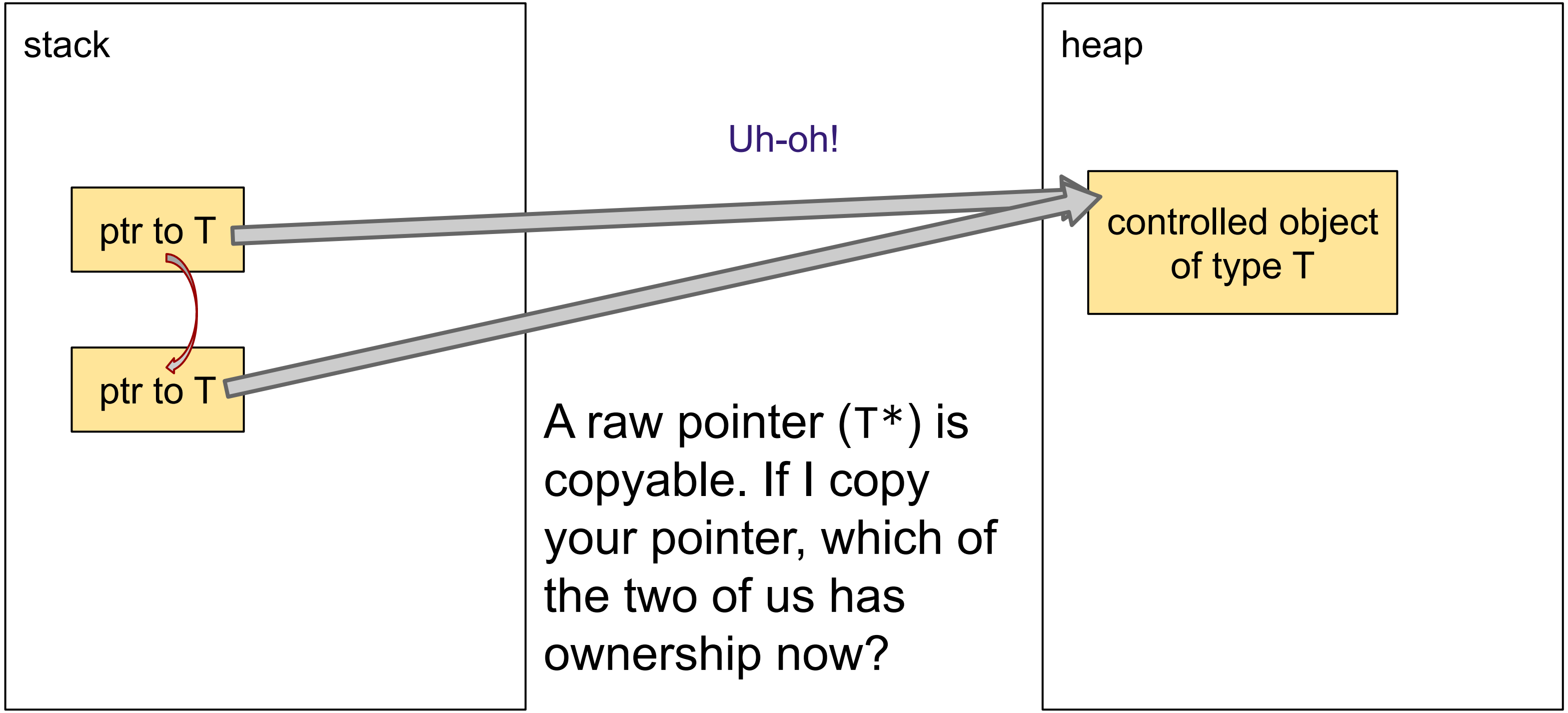

```cpp
template<class T>
class unique_ptr {
    T *p_ = nullptr;

    ~unique_ptr() {
        delete p_;
    }
};
```

`unique_ptr` is moveable (move-only)



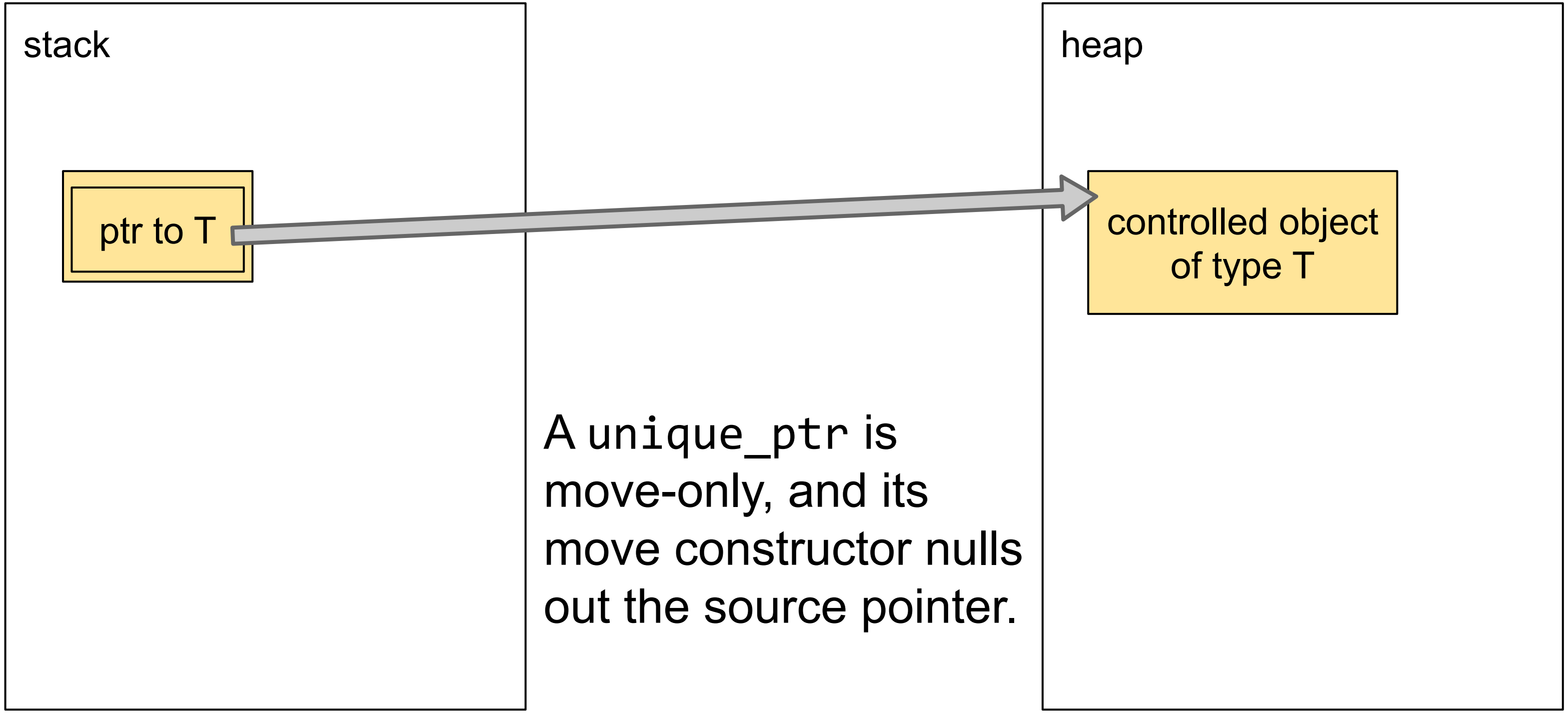
A raw pointer (`T*`) is copyable. If I copy your pointer, which of the two of us has ownership now?

`unique_ptr` is moveable (move-only)



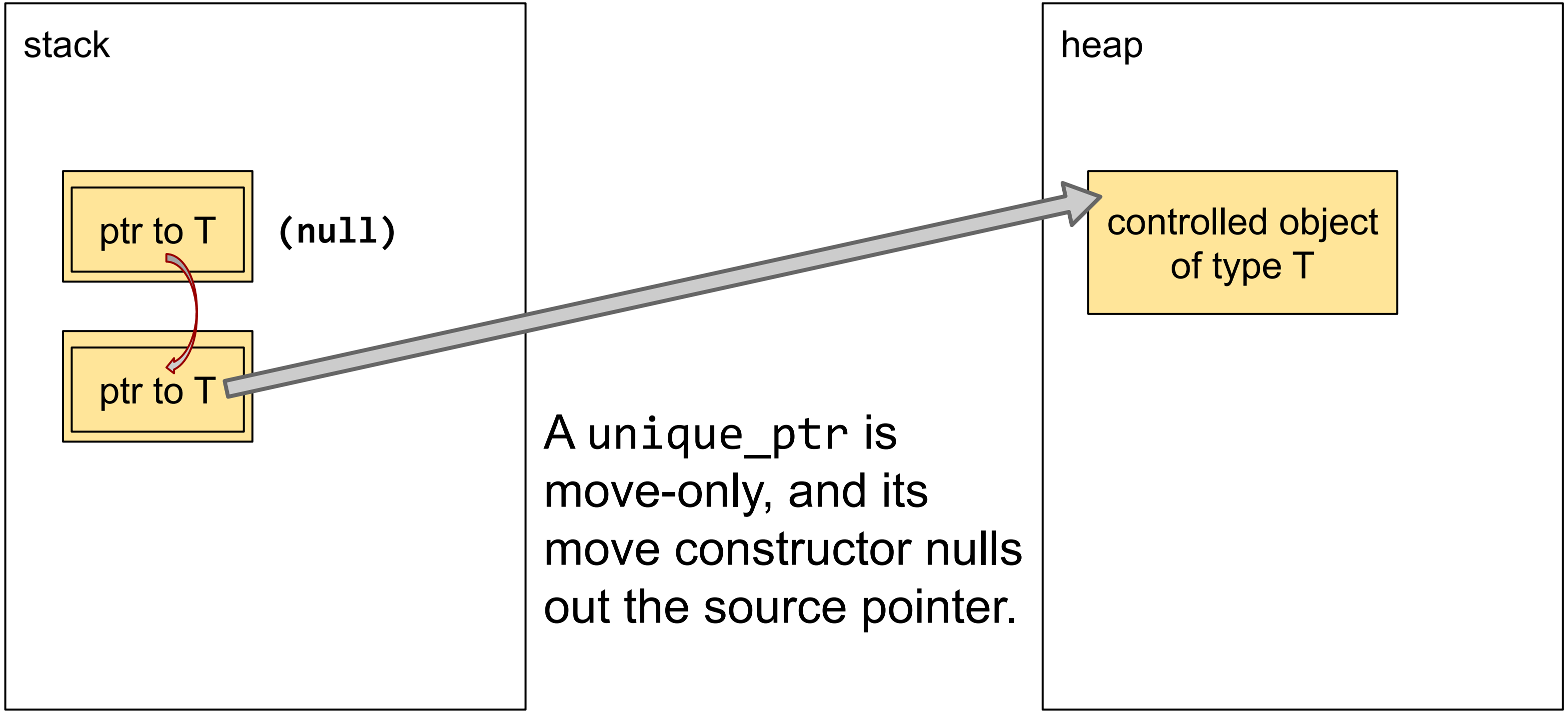
A raw pointer (T*) is copyable. If I copy your pointer, which of the two of us has ownership now?

unique_ptr is moveable (move-only)



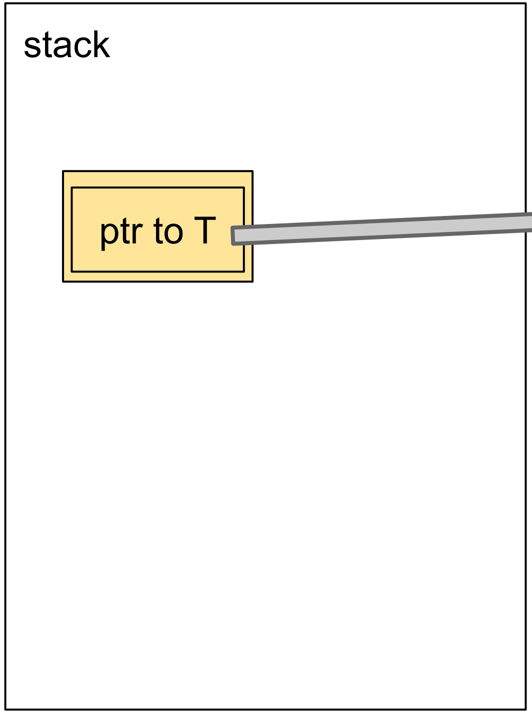
A unique_ptr is move-only, and its move constructor nulls out the source pointer.

`unique_ptr` is moveable (move-only)



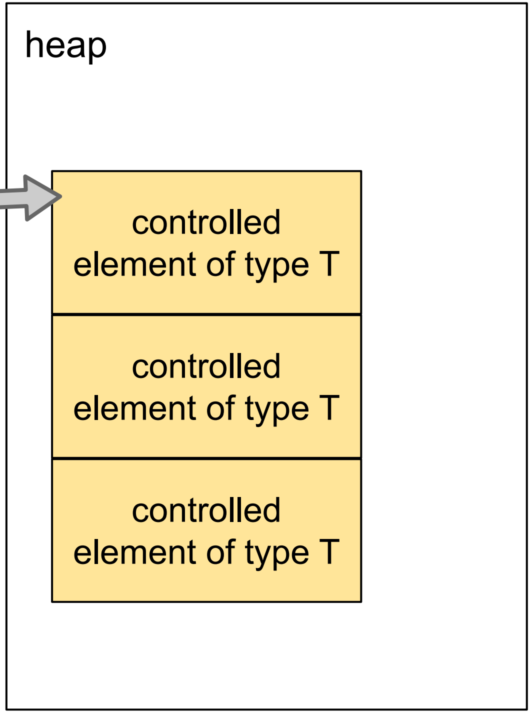
A `unique_ptr` is move-only, and its move constructor nulls out the source pointer.

`std::unique_ptr<T[]>`

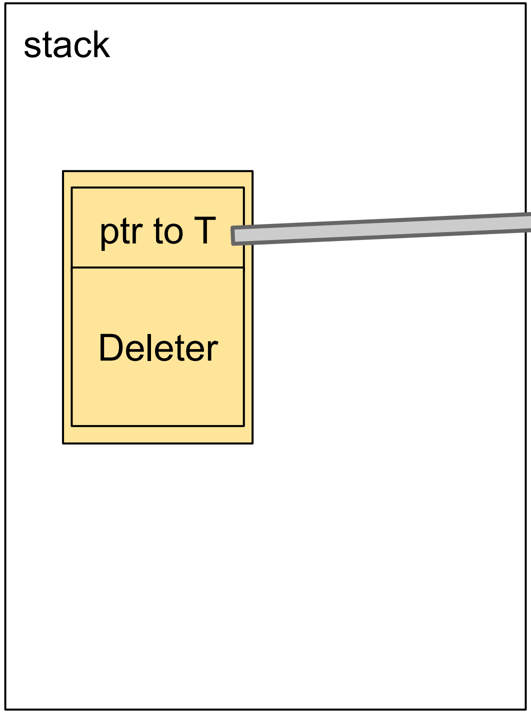


There's a specialization for array types.

```
template<class T>
class unique_ptr<T[]> {
    T *p_ = nullptr;

    ~unique_ptr() {
        delete [] p_;
    }
};
```

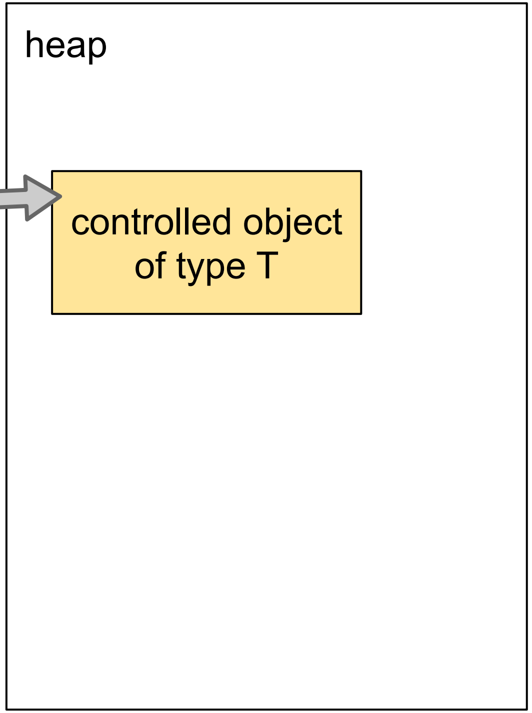


std::unique_ptr<T, Deleter>



In fact, there are specializations for *arbitrary* "cleanup."

```
template<class T,
         class Deleter>
class unique_ptr {
    T *p_ = nullptr;
    Deleter d_;

    ~unique_ptr() {
        if (p_) d_(p_);
    }
};
```



std::unique_ptr<T, Deleter>

```
template<class T, class Deleter = std::default_delete<T>>
class unique_ptr {
    T *p_ = nullptr;
    Deleter d_;

    ~unique_ptr() {
        if (p_) d_(p_);
    }
};

template<class T>
struct default_delete {
    void operator()(T *p) const {
        delete p;
    }
};
```

`unique_ptr` is always a template of two parameters. If you provide no second parameter, it is defaulted to `std::default_delete<T>`.

Even this part is customizable. If `Deleter::pointer` names a type, then this member will be of that type instead of `T*`.

Custom deleters can do neat things

```
struct FileCloser {
    void operator()(FILE *fp) const {
        assert(fp != nullptr);
        fclose(fp);
    }
};

FILE *fp = fopen("input.txt", "r");
std::unique_ptr<FILE, FileCloser> uptr(fp);
```

Custom deleters can do neat things

My codebase's OpenSSL code is peppered with this pattern:

```
struct REQDeleter {
    void operator()(X509_REQ *p) const { X509_REQ_free(p); }
};

class MyCSR {
    std::unique_ptr<X509_REQ, REQDeleter> p_;
public:
    explicit MyCSR(std::unique_ptr<X509_REQ, REQDeleter> p) :
        p_(std::move(p)) {}
};
```

Custom deleters can do neat things

```
explicit MyCSR(std::unique_ptr<X509_REQ, REQDeleter> p) :
    p_(std::move(p)) {}
```

This constructor takes a `unique_ptr` by value.

Functions taking owning pointers by value are sometimes called ***sinks***.

A “`unique_ptr` sink” clearly conveys ***transfer of ownership***.

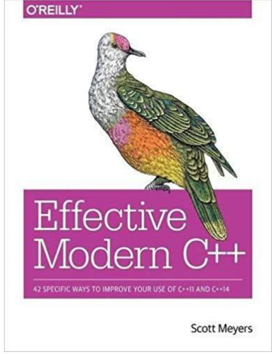
To call this constructor at all, you must have a `unique_ptr` already — so, you must have ownership of an `X509_REQ` structure. By taking your `unique_ptr` by value (by move), the constructor shows that it ***takes*** ownership of that `X509_REQ` structure away from you.

`unique_ptr` is the glue between the low-level, non-RAII, raw resource (`X509_REQ`) and the high-level RAII business object (`MyCSR`).

Rules of thumb for smart pointers

- Treat smart-pointer types just like raw pointer types.
  - Pass by value.
  - Return by value (of course).
  - Passing a pointer by reference is usually a code smell.
  - Same goes for smart pointers.
- A function taking a `unique_ptr` by value shows ***transfer of ownership***.
  - Even shows exactly ***what*** responsibility is being transferred, because the responsibility is encoded in the deleter type.
  - Usually the responsibility is simply "to call `delete`."
- Smart pointers are frequently implementation details and glue.
  - To bake `unique_ptr` or `shared_ptr` into your ***interface*** might be a code smell. Try to deal in business classes like `MyCSR` instead.

EMC++ Item 19



Use `std::shared_ptr` for shared-ownership resource management.

std::shared_ptr<T>

`shared_ptr` looks similar to `unique_ptr` on the surface...

```
std::unique_ptr<int> uptr = std::make_unique<int>(42);
std::shared_ptr<int> sptr = std::make_shared<int>(42);
```

But it is vastly more complicated on the inside!

`shared_ptr` expresses shared ownership. Specifically, ***reference-counting***.

Simple reference counting

“Will the last person out of the room please turn out the lights.”

How do you know if you’re the last person in the room? (It’s a very big room.)

We keep a jar of tokens by the door. When anyone enters, they put a token in the jar. When anyone leaves, they take a token with them. If you take the ***last*** token from the jar, then you must be the last person to leave the room.

Notice that everyone involved must cooperate in our scheme. Anyone who enters without leaving a token in the jar, risks being left in the dark!

Simple reference counting

```
struct Widget {
    std::atomic<int> usecount_{1};
    Widget *retain() { ++usecount_; return this; }
    void release() { if (--usecount_ == 0) delete this; }
};
{
    Widget *p = new Widget;
    Widget *q = p->retain();
    p->release();
    q->release();  // causes the Widget to be deleted
}
```

shared_ptr automates refcounting

```
struct Widget {
    std::atomic<int> usecount_{1};
    Widget *retain() { ++usecount_; return this; }
    void release() { if (--usecount_ == 0) delete this; }
};
{
    Widget *p = new Widget;
    Widget *q = p->retain();
    p->release();
    q->release();
}
```

`shared_ptr<Widget>` will have a copy-constructor that increments the refcount, and a destructor that decrements the refcount.

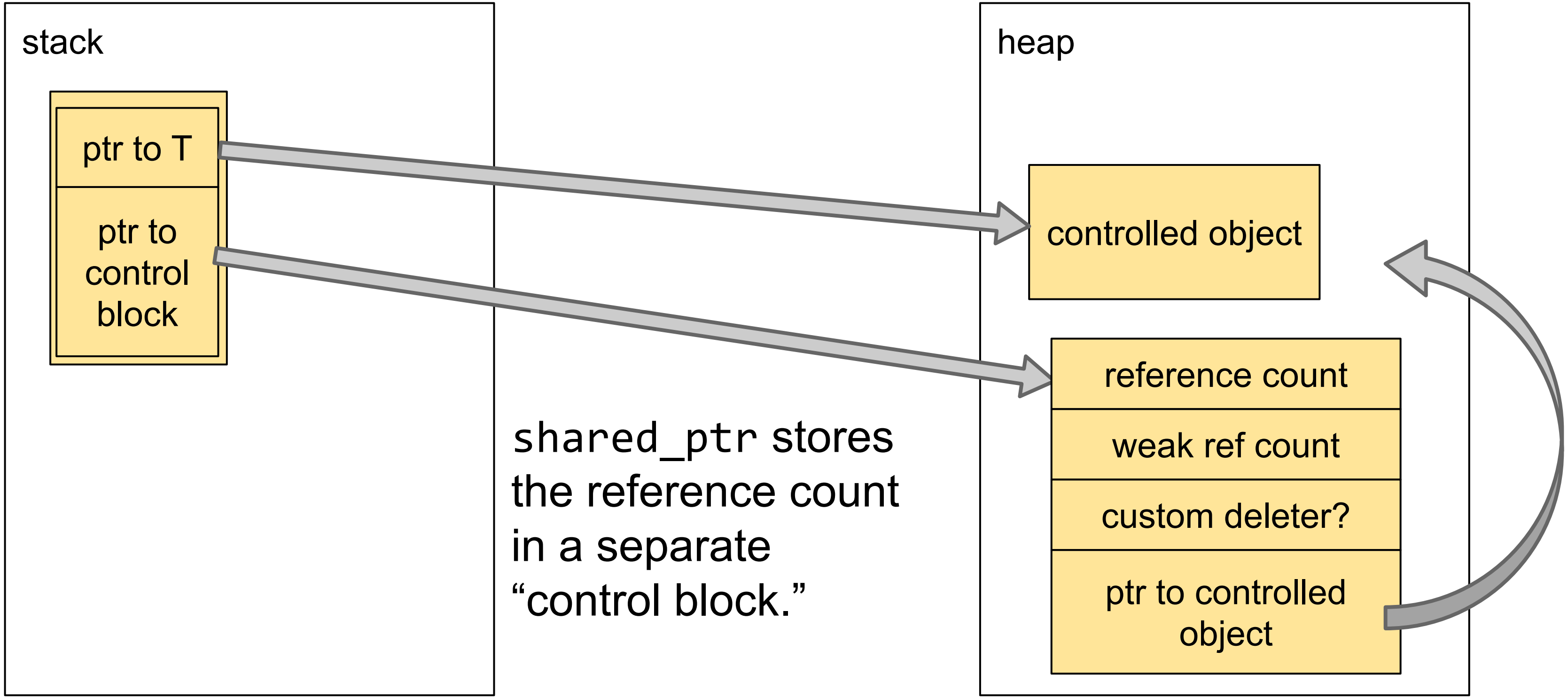
But there's a problem...

C++ objects don’t contain refcounts!

```
struct Widget {
    std::atomic<int> usecount_{1};
    Widget *retain() { ++usecount_; return this; }
    void release() { if (--usecount_ == 0) delete this; }
};
{
    Widget *p = new Widget;
    Widget *q = p->retain();
    p->release();
    q->release();
}
```

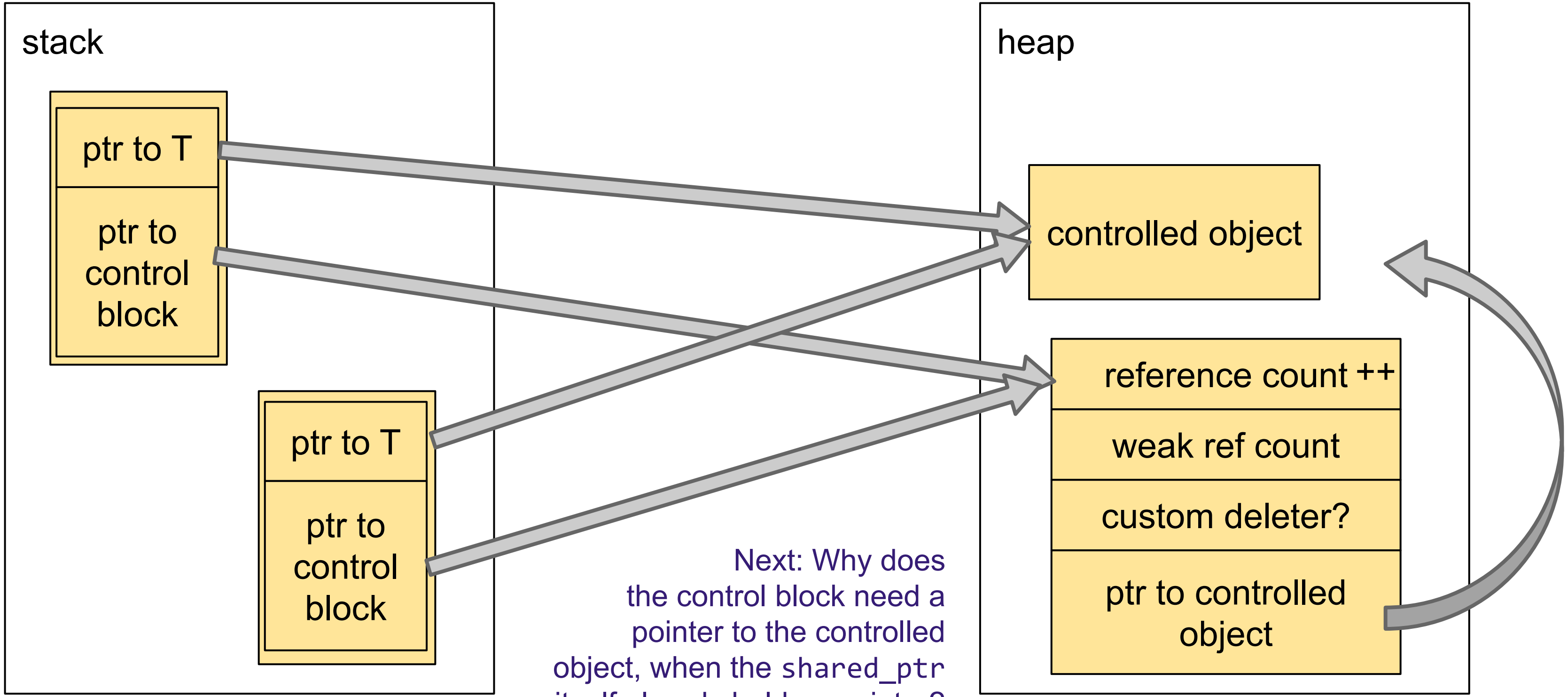
What about `shared_ptr<int>`? Where could we store the reference count for an `int` object?

`std::shared_ptr<T>`



`shared_ptr` stores the reference count in a separate "control block."

Copying a `std::shared_ptr`



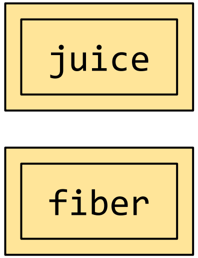
Next: Why does the control block need a pointer to the controlled object, when the `shared_ptr` itself already holds a pointer?

Detour: Physical class layout

```
struct Fruit { int juice; };
struct Vegetable { int fiber; };
```

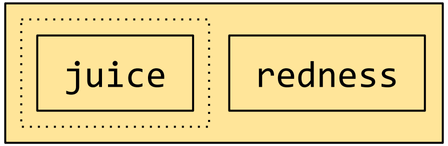
Fruit

Vegetable



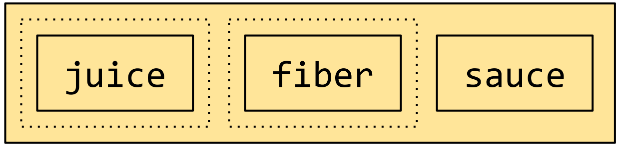

```
struct Apple : Fruit { int redness; };
```

Apple

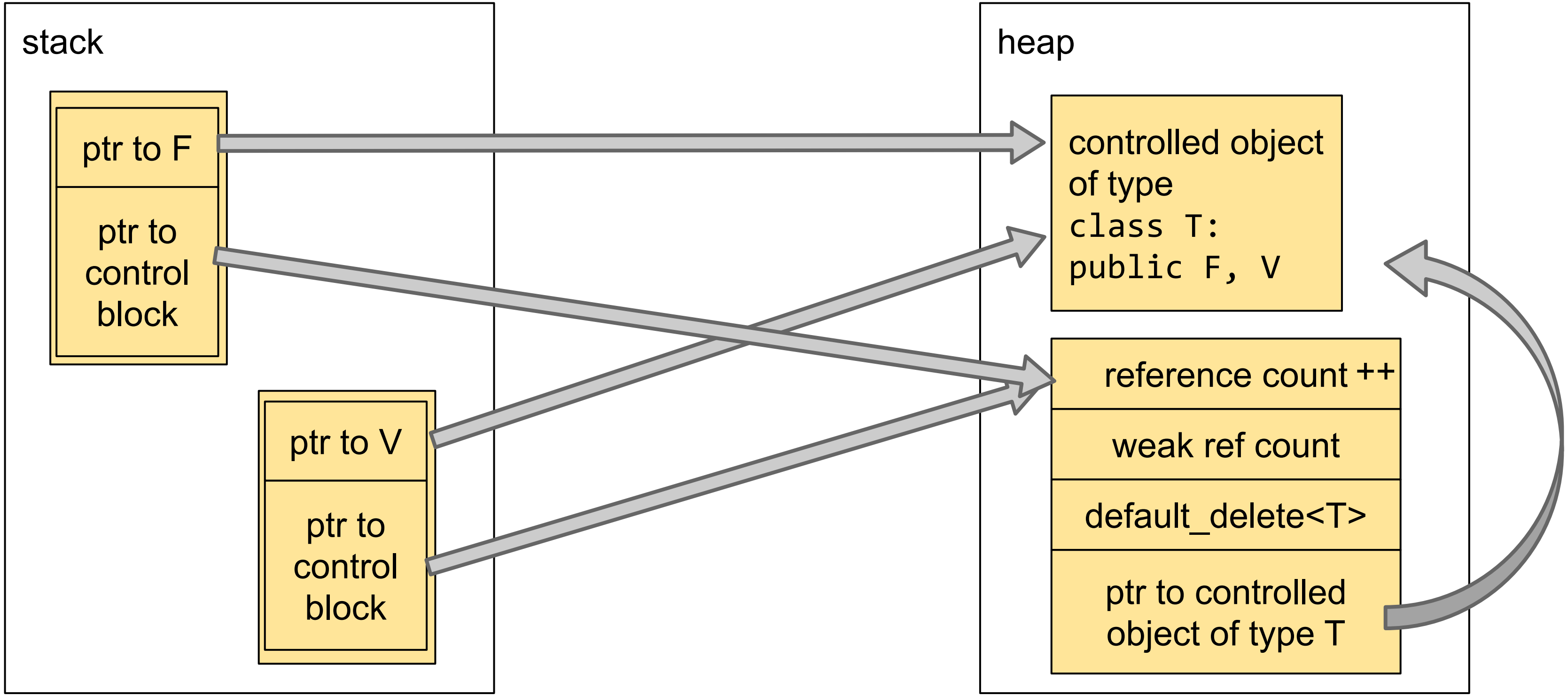



```
struct Tomato : Fruit, Vegetable { int sauce; };
```

Tomato



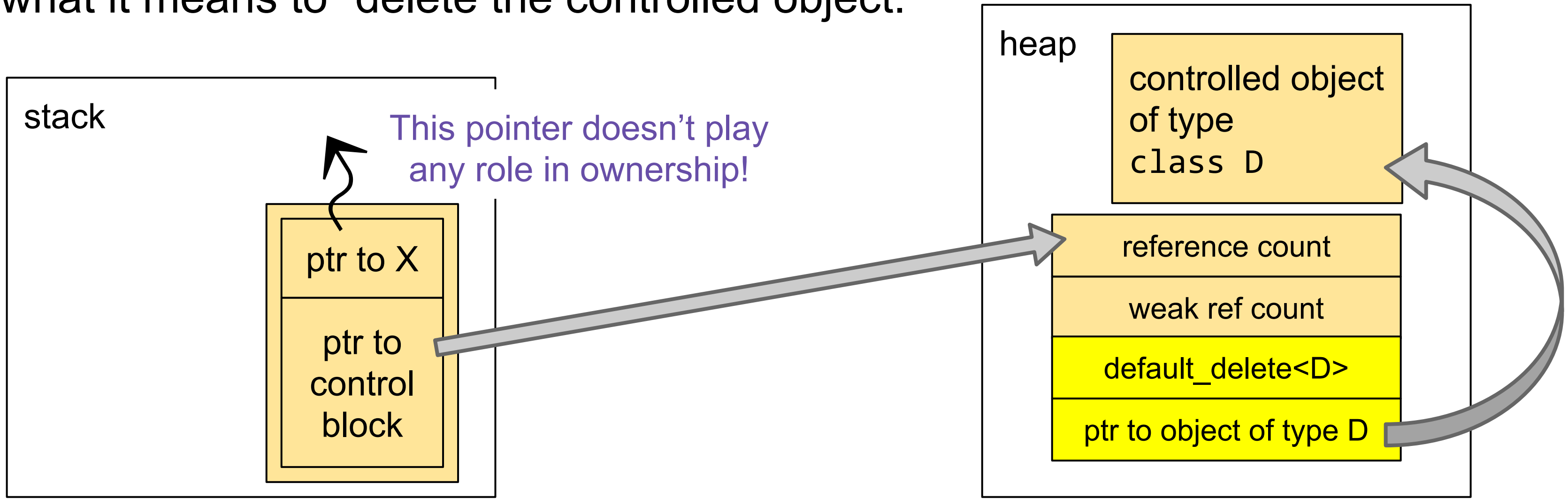
`std::shared_ptr` to base class



`shared_ptr` type-erases the deleter

`shared_ptr`, unlike `unique_ptr`, places a layer of indirection between the physical heap-allocated object and the notion of ownership.

Your `shared_ptr` instances are essentially participating in ref-counted ownership of the ***control block***. The control block itself is sole arbiter of what it means to "delete the controlled object."

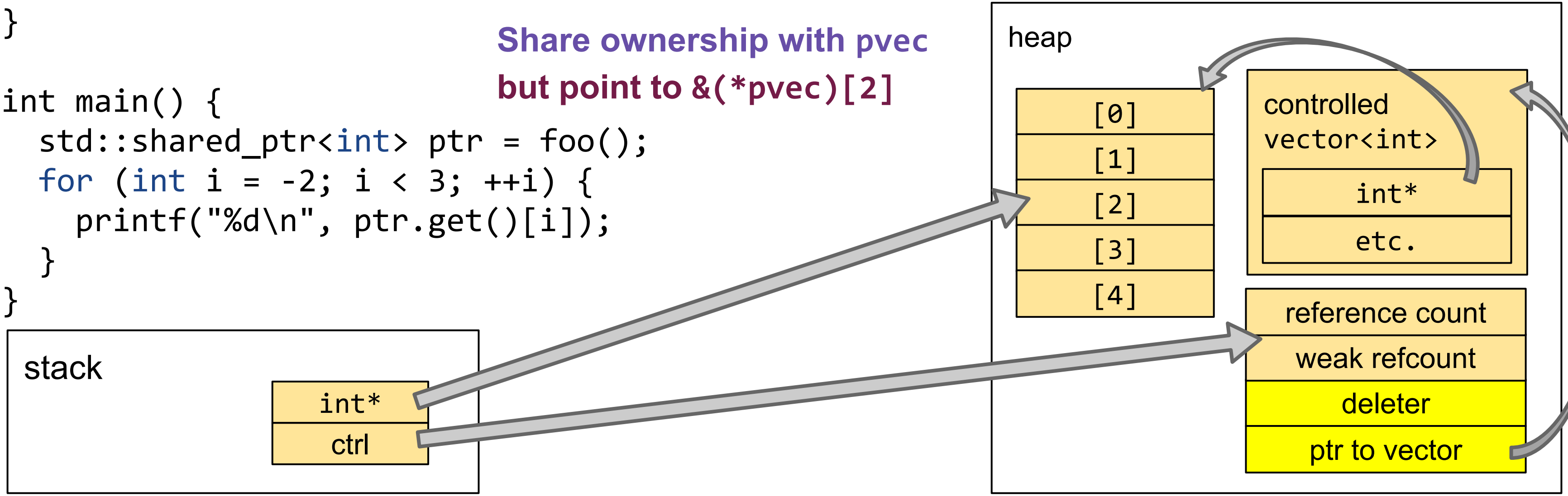


shared_ptr’s “aliasing constructor”

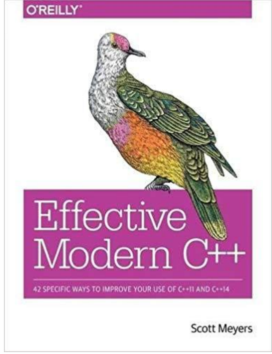
```
using Vec = std::vector<int>;

std::shared_ptr<int> foo() {
  auto elts = { 0,1,2,3,4 };
  std::shared_ptr<Vec> pvec = std::make_shared<Vec>(elts);
  return std::shared_ptr<int>(pvec, &(*pvec)[2]);
}

int main() {
  std::shared_ptr<int> ptr = foo();
  for (int i = -2; i < 3; ++i) {
    printf("%d\n", ptr.get()[i]);
  }
}
```

Share ownership with pvec

but point to &(*pvec)[2]



EMC++ Item 21



Prefer **`std::make_unique`**
and **`std::make_shared`**
to direct use of **`new`**.

Goal: No raw `new` or `delete`

The first rule of C++ memory management is: Every `new` must have a matching `delete`, and vice versa.

The second rule of C++ memory management is: Stop using manual calls to `delete`! Use smart pointers whose destructors automatically call `delete` at the right time.

But if we stop writing `delete`... by the first rule, mustn't we stop writing `new`?

```
auto *w = new Widget();
use(w); // Is this code OK?
```

```
std::shared_ptr<Widget> w(new Widget());
use(w); // Is this code any better?
```

Goal: No raw `new` or `delete`

Yes! If we're not going to write `delete`, we ***should*** stop writing `new`.

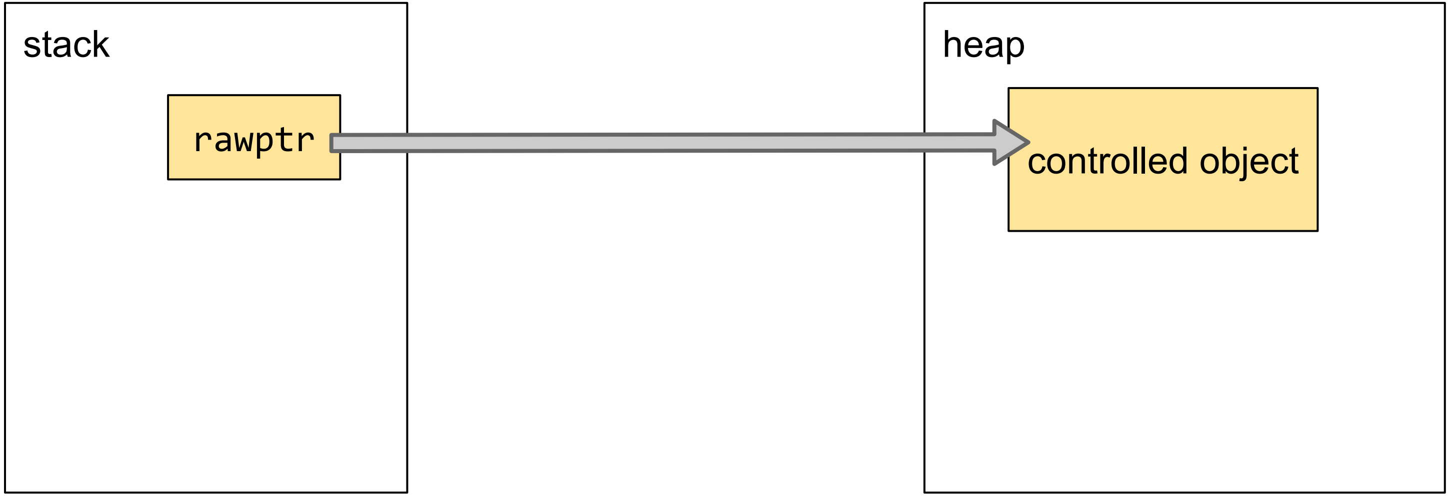
Whenever we heap-allocate something, we'll use a factory function that returns it ***already wrapped in a smart pointer***.

We should never have to touch raw pointers with our hands.

```
auto w = std::make_shared<Widget>();
use(w); // Now we have no reason
        // to flag this code.
        // No new, no delete.
```

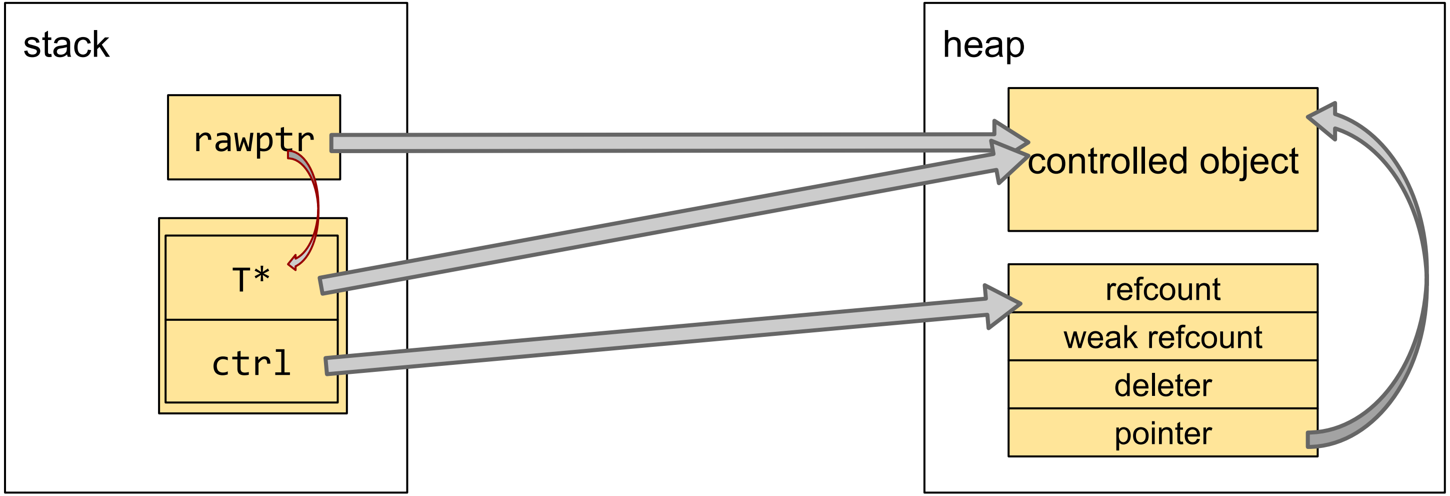
make_shared can even be optimized

```
template<class T, class... Args>
std::shared_ptr<T> make_shared(Args&&... args) {
    T *rawptr = new T(std::forward<Args>(args)...);
    return std::shared_ptr<T>(rawptr);  // take ownership from rawptr
}
```



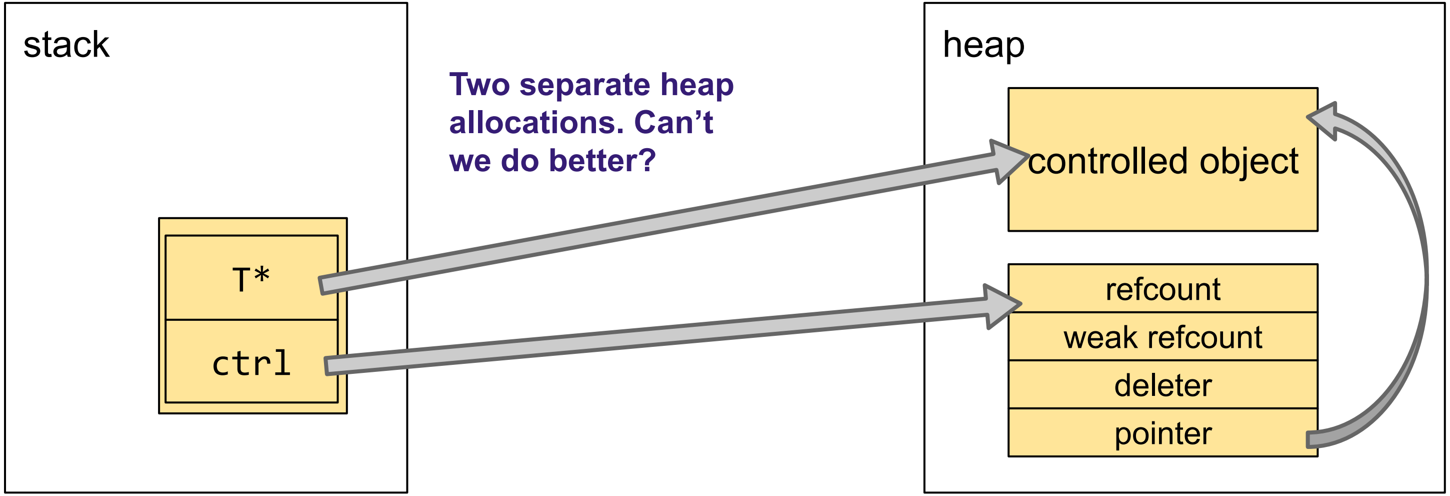
make_shared can even be optimized

```
template<class T, class... Args>
std::shared_ptr<T> make_shared(Args&&... args) {
    T *rawptr = new T(std::forward<Args>(args)...);
    return std::shared_ptr<T>(rawptr);  // take ownership from rawptr
}
```



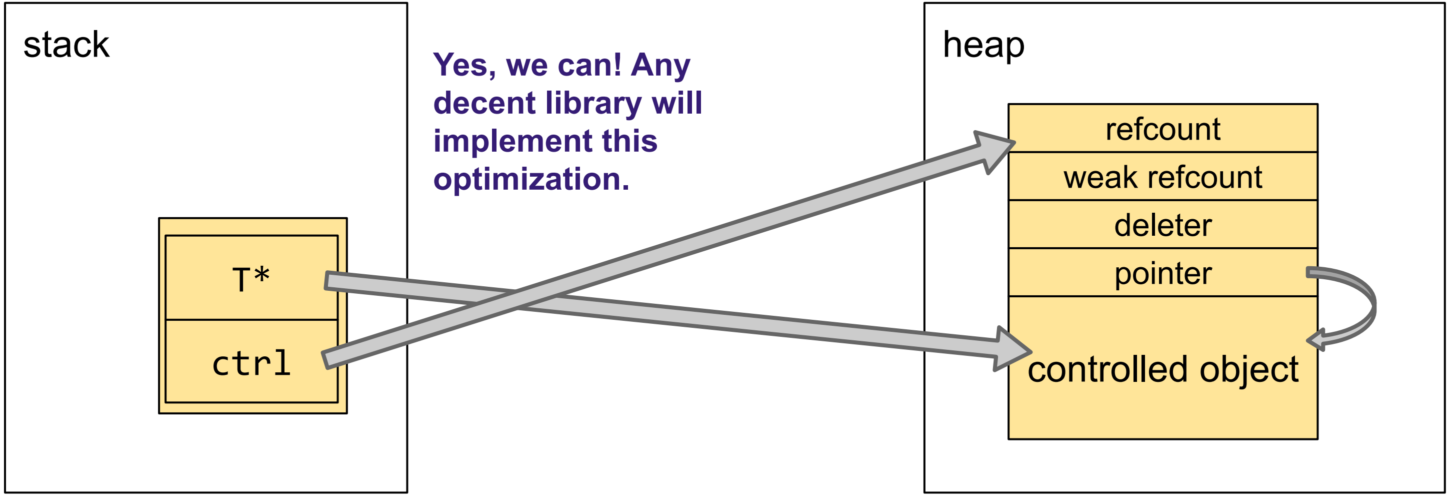
make_shared can even be optimized

```
template<class T, class... Args>
std::shared_ptr<T> make_shared(Args&&... args) {
    T *rawptr = new T(std::forward<Args>(args)...);
    return std::shared_ptr<T>(rawptr);  // take ownership from rawptr
}
```



make_shared can even be optimized

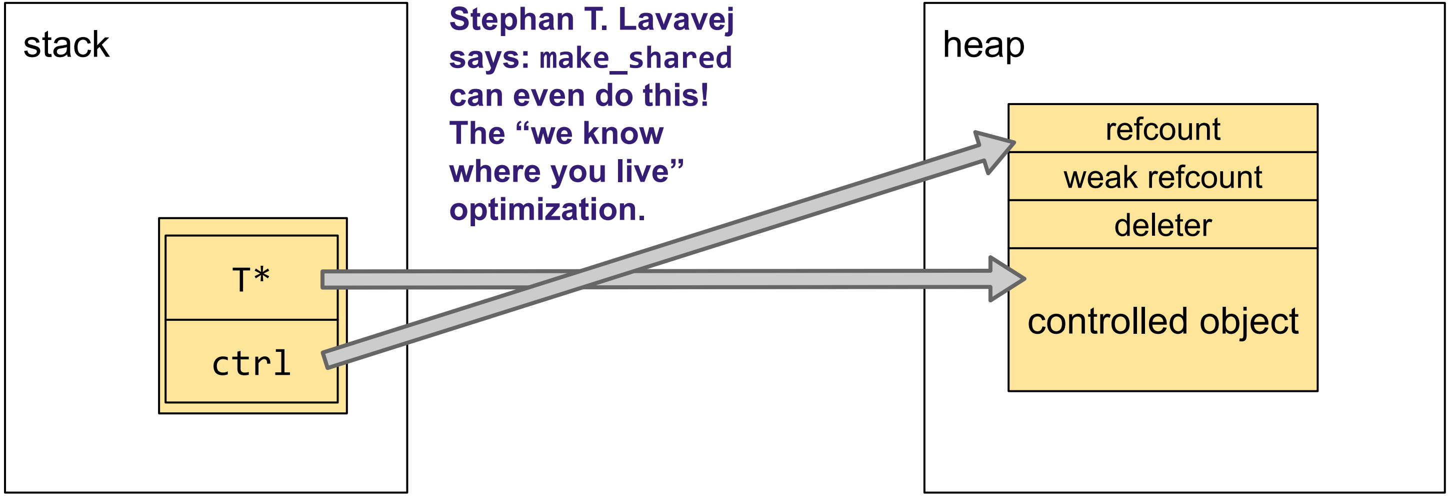
```
template<class T, class... Args>
std::shared_ptr<T> make_shared(Args&&... args) {
    auto *rawptr = new ControlBlockAnd<T>(std::forward<Args>(args)...);
    return std::shared_ptr<T>::From(rawptr);
}
```



Yes, we can! Any decent library will implement this optimization.

make_shared can even be optimized

```
template<class T, class... Args>
std::shared_ptr<T> make_shared(Args&&... args) {
    auto *rawptr = new ControlBlockAnd<T>(std::forward<Args>(args)...);
    return std::shared_ptr<T>::From(rawptr);
}
```



Stephan T. Lavavej says: `make_shared` can even do this! The "we know where you live" optimization.

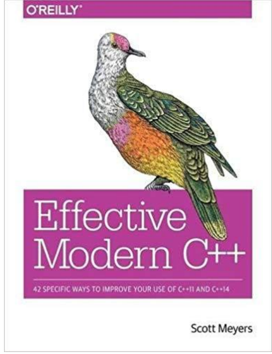
Conclusion: Use `make_shared`

- `make_shared` and `make_unique` wrap raw `new`, just as `~shared_ptr` and `~unique_ptr` wrap raw `delete`.
- If you ***never*** touch raw pointers with your hands, then you never need to worry about leaking them.
- `make_shared` can be a performance optimization!

By the way, `unique_ptr<T>` is implicitly convertible to `shared_ptr<T>`...

```
std::shared_ptr<Widget> sptr = std::make_unique<Widget>();
```

EMC++ Item 20



Use **`std::weak_ptr`** for

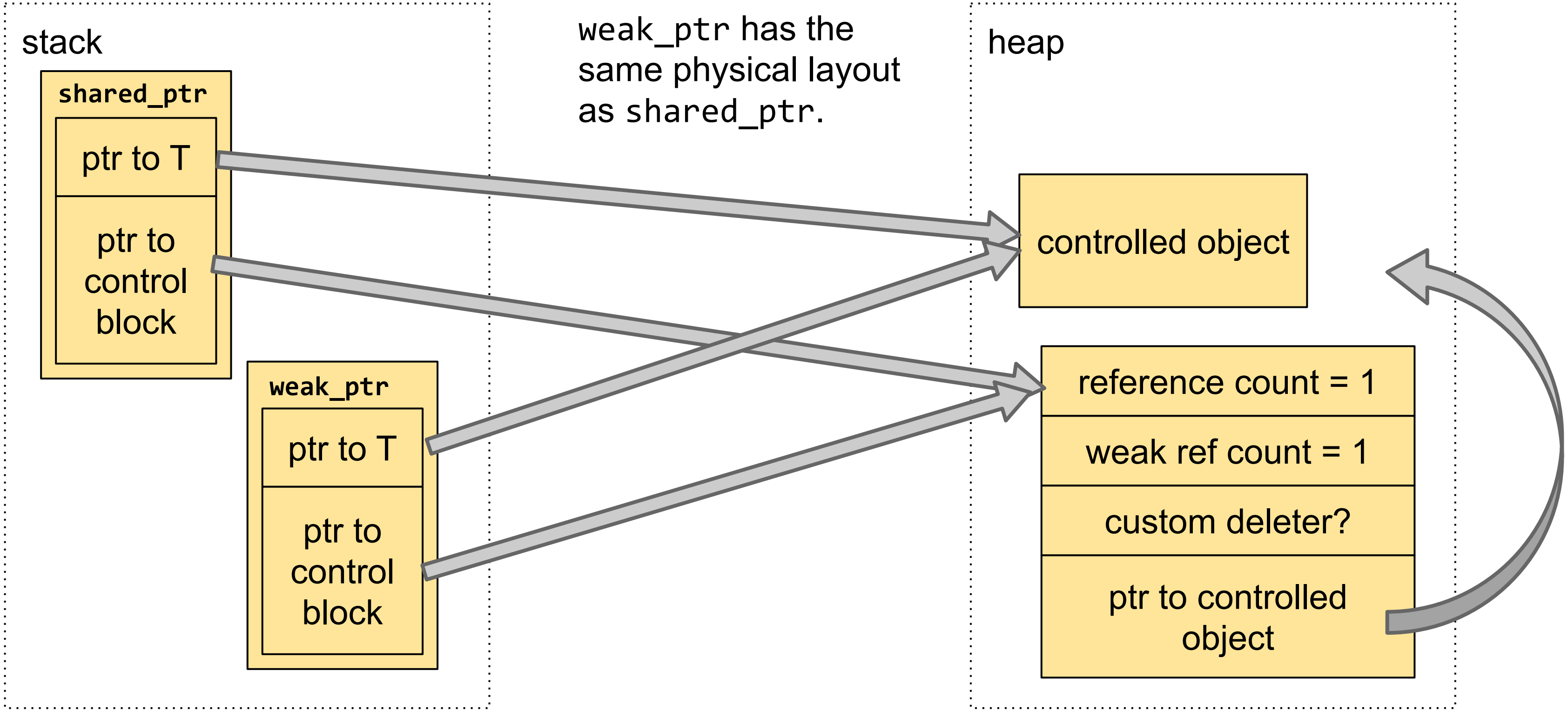
`shared_ptr`-like pointers

that can dangle

(and <u>know</u> when they are dangling)

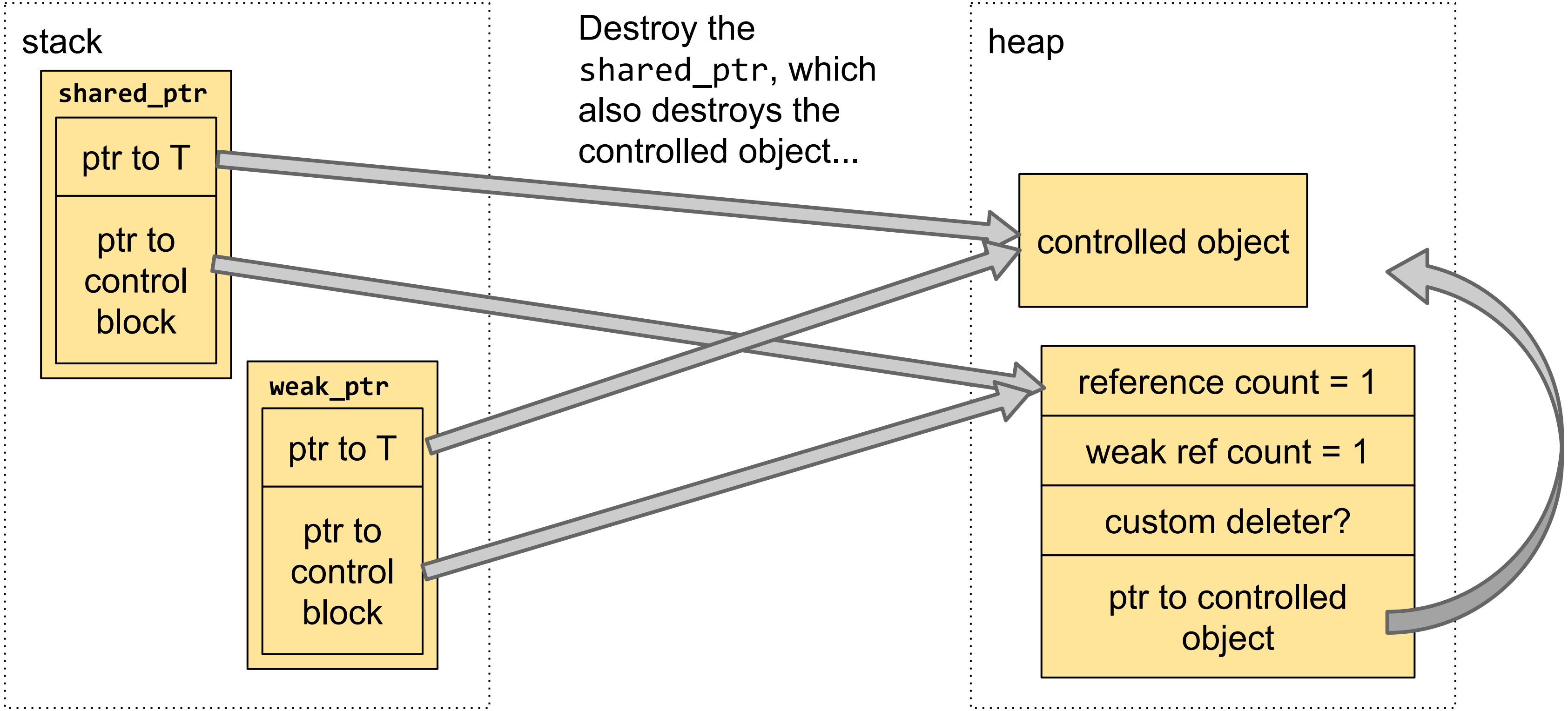
std::weak_ptr<T>

weak_ptr has the same physical layout as shared_ptr.



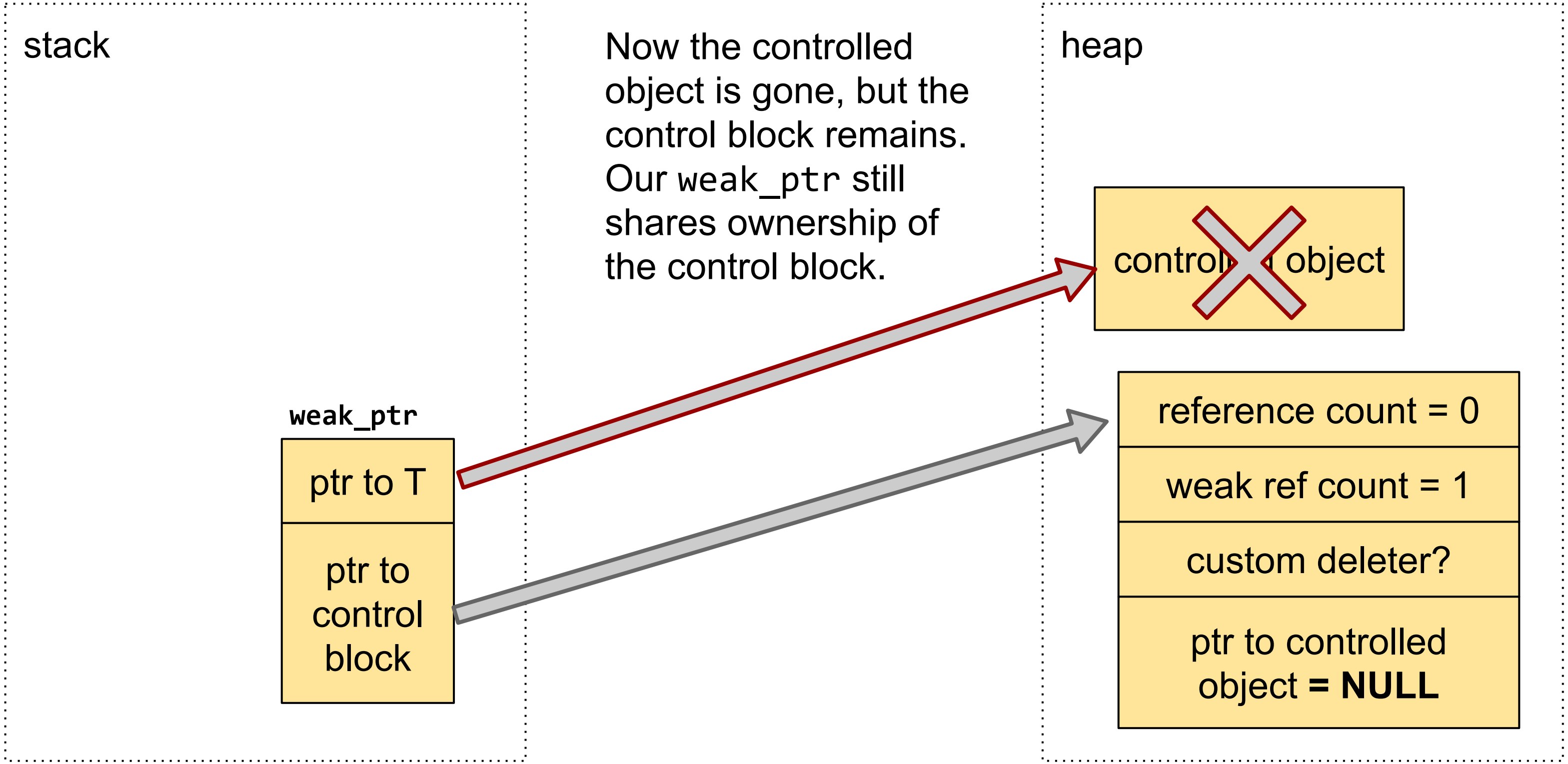
std::weak_ptr<T>

Destroy the `shared_ptr`, which also destroys the controlled object...



`std::weak_ptr<T>`

Now the controlled object is gone, but the control block remains. Our `weak_ptr` still shares ownership of the control block.



You can't dereference a `weak_ptr`

You can only convert it to a `shared_ptr`.

- Think of a `weak_ptr` as a "ticket for a `shared_ptr`."
    - If you hold a `weak_ptr`, you're entitled to receive a `shared_ptr` (if the controlled object still exists at all, of course).
    - If you hold a `shared_ptr`, you're certainly entitled to get a `weak_ptr`!
- The "redeem a ticket" operation can be spelled in two ways: explicit type-conversion, or `wptr.lock()`.
- The "get a ticket" operation can be spelled only as an explicit conversion. (Consider writing your own `unlock()` function?)

You can't dereference a weak_ptr

You can only convert it to a shared_ptr.

```
void recommended(std::weak_ptr<T> wptr) {
  std::shared_ptr<T> sptr = wptr.lock();
  if (sptr != nullptr) {
    use(sptr);
  }
}

void not_recommended(std::weak_ptr<T> wptr) {
  try {
    std::shared_ptr<T> sptr { wptr };  // call the explicit constructor
    use(sptr);
  } catch (const std::bad_weak_ptr&) {}
}
```

Idiomatic “redeem a ticket” one-liner

C++ has always supported variable declarations inside `if` conditions.

This is one of the rare times it’s a good idea to use that feature.

```
if (auto sptr = wptr.lock()) {
    use(sptr);
}
```

The other case is:

```
if (RedWidget *p = dynamic_cast<RedWidget*>(widgetptr)) {
    use_red_widget(*p);
}
```

What if a `Widget` is its own ticket?

Earlier, I said...

- Think of a `weak_ptr` as a “ticket for a `shared_ptr`.”
    - If you hold a `weak_ptr`, you’re entitled to receive a `shared_ptr` (if the controlled object still exists at all, of course).
    - If you hold a `shared_ptr`, you’re certainly entitled to get a `weak_ptr`!

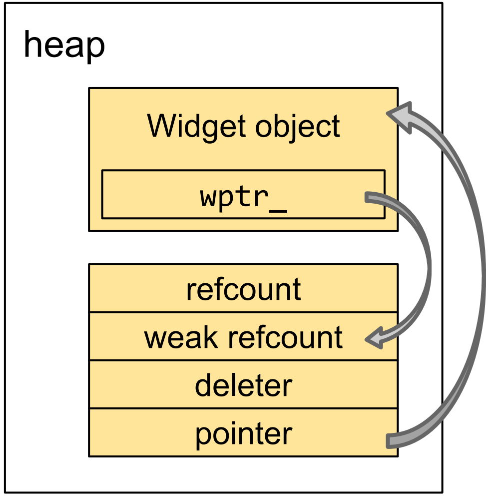
What if you want the rule to be “If you hold a ***raw*** pointer, you’re entitled to receive a `shared_ptr`”?

Can we make that work?

What if a `Widget` is its own ticket?

Can we make that work? Yes. We just need to store a ticket (`weak_ptr`) somewhere that the `Widget` itself can get to it.

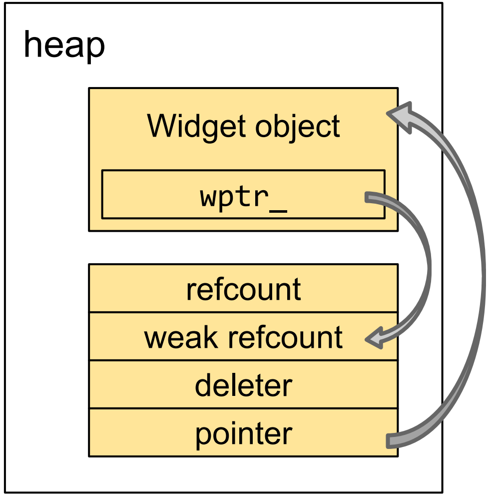
```
class Widget {
    std::weak_ptr<Widget> wptr_ = ???;
public:
    std::shared_ptr<Widget> shared_from_this() {
        return wptr_.lock();
    }
};
```



What if a `Widget` is its own ticket?

Pull out the `weak_ptr` into a special base class which is known to the STL implementation. Whichever constructor of `shared_ptr` first creates the control block also fills in the `weak_ptr`.

```
class Widget :
    public std::enable_shared_from_this<Widget> {
    ...
};

Widget *p = new Widget();
assert(p->shared_from_this() == nullptr);

std::shared_ptr<Widget> s1(p);
assert(p->shared_from_this() == s1);
```



Curiously recurring template pattern

Notice that `enable_shared_from_this`:

- Cannot be implemented "in user-space," because it must conspire with the implementation of `shared_ptr`'s constructor
- Uses the Curiously Recurring Template Pattern (CRTP)
    - The interface's signature uses type `Widget`, so `Widget` must be a parameter to the base class template — even though `Widget` ***derives from*** that base class!

```
template<class T>
class enable_shared_from_this {
    shared_ptr<T> shared_from_this();  // ...
};

class Widget : public std::enable_shared_from_this<Widget> { ...
};
```

Why might a `Widget` want to be its own ticket?

This `run()` function passes `listener` off to `Acceptor::async_accept()`, and then immediately returns. The lambda must participate in shared ownership of the `Listener` object, to keep it from being prematurely destroyed.

```
class Listener;

void on_accept(std::shared_ptr<Listener>, std::error_code);

void run(std::shared_ptr<Listener> listener) {
    listener->acceptor().async_accept(
        listener->socket(),
        [listener](std::error_code ec) {
            on_accept(listener, ec);
        }
    );
}
```

```
run(std::make_shared<Listener>(...));
```

Why might a `Widget` want to be its own ticket?

Rewrite the code in OO style. Now `Listener::run()` receives only a raw pointer — **`this`**. To participate in the `Listener`’s ***lifetime***, it needs to regain access to the control block. So we use `shared_from_this`.

```
class Listener : public std::enable_shared_from_this<Listener> {

    void on_accept(std::error_code);

    void run() {
        acceptor_.async_accept(socket_,
            [self = shared_from_this()](std::error_code ec) {
                self->on_accept(ec);
            }
        );
    }
};
```

```
std::make_shared<Listener>(...)->run();
```

In conclusion

- Use `std::unique_ptr` for unique ownership
- Use `std::shared_ptr` for reference-counting
- Don’t touch raw pointers with your hands
- Pass pointers by value, not by reference
- `std::weak_ptr` is a ticket for a `shared_ptr`
- Use `enable_shared_from_this` when an object is its own ticket

Questions?