An alternate smart pointer hierarchy

Matthew Fleming
Pure Storage

Pre-Summary

- Nothing here is magic
  - But maybe it gets us thinking along different lines
- Names can be important
- Nothing done with the smart pointers we’ve created is impossible to do with the standard smart pointers
  - But names are important

Outline

Motivation and A Brief History

October 2009

- Not sure what the shape of the s/w will be, but maybe:

```
{async} read_flash_page({addr}, {buffer}, {result});
{async} write_flash_page({addr}, {buffer}, {result});

{async} read_client_page({segment}, {index}, {buffer}, {result});
{async} commit_txn({participants}, {data}, {metadata});

{KV-store view} catalog_select({table}, {sequence range});
{stream} catalog_open({KV-store view}, {direction}, {bound tuples});
```

Resource management

- Not sure of the shape of the software, but we will have lots of kinds of resources
  - Buffers (zero-copy would be nice)
  - Database Tables
  - Async requests
  - RAID rebuild
- Idea: resources know how to release themselves
  - To a thread-local freelist? Using `free()`? A recycling pool?
  - A combination? E.g. recycle preallocated items and delete any overflow items? Or keep only N items in the recycling pool?
- The implementation of the resource class knows how it should be released
  - … and this won't always be via `free()` / `delete`

In the beginning...

- In C, there is no destructor
  - Need a new name for “delete” / “destroy” / “release”

```
struct foo {
    void (*dispose)(foo *);
    ...
};
```

What language are we using?

- Initial code development done in C
    - Because systems software is C
    - And one of the founders didn't like C++
    - … but another one did …
    - He who has a working implementation first, wins?
- Ended up with C++ but some leftover manually-implemented vtable-like code
    - Great Renaming in 2010 from `.c` to `.cpp`
    - If you know how the compiler implements a vtable, you can write C code that works with C++ objects
    - Which is a horrible thing to do, but it means you can move forward with product development without rewriting the world
    - The last instances of C code using vtable overlays was removed in 2013

Pivoting to C++...

- We already have a name for “dispose”
  - And months of software development that’s using it

```
struct disposable {
  private:
    virtual void dispose() = 0;

    template<typename any_t> friend struct owned;
};
```

Add a smart pointer…

```
template<typename itf_t>
struct owned {
    using element_type = itf_t;

    // … constructor, assignment, operator->(), etc.

    ~owned() {
        if (p_) {
            static_cast<disposable *>(p_)->dispose();
        }

    }
  private:
    element_type * p_ = nullptr;
};
```

Isn't that the same as...?

```
template<typename disp_t>
struct dispose_deleter {
    operator()(disp_t * ptr) { ptr->dispose(); }
};

template<typename any_t>
using owned = std::unique_ptr<any_t, dispose_deleter<any_t>>;
```

- Yes, semantically it's the same thing
    - But `std::unique_ptr<>` didn't exist in 2010
- And `owned<>` doesn't imply single ownership; the "unique" in `unique_ptr` implies it (even if it doesn't technically require it)
    - Names have meaning

A common style guideline

- Q: should types have both data members and pure virtual functions?
- A (common?) style is to have either a pure-virtual interface class, and (private) implementation class(es), or POD-like types with no virtual methods
  - A type with virtual functions is exposed as pure virtual, and is `disposable`
    - Hidden implementation classes are only visible as `owned<API-class>`
  - A type with no virtual functions typically does not inherit from `disposable`
    - `delete_ptr<>` to express ownership of this case if we need dynamic allocation
    - Does what you'd expect, can't override the deleting operation
    - We can add more safety-belts that `unique_ptr<>` doesn't have: e.g. `static_assert` that the type has a virtual destructor if it needs one
  - Though I hear pure virtual interface classes have fallen out of favor for some codebases

A common style guideline

- Is `shared_ptr<>` thread-safe?
  - [Louis Brandy's Curiously Recurring C++ Bugs at Facebook]()
  - If you have to ask, the answer is no
- If your type is multiply-owned, all interface functions should be thread-safe
  - Probably shouldn't be exposing raw data members and hoping consumers do the right thing
- But I, as the consumer, don't need to know
  - I have an `owned<>` declaring ownership
  - I have virtual methods and nothing else
  - My handle is valid until I'm done with it
  - When I release the resource, I don't care what happens to it
    - That's the resource's problem, not mine

Why didn't we remove `delete_ptr<>`?

- Very minor advantages (for us) over `std::unique_ptr<>`
  - Safety-belt for virtual destructor to prevent slicing
  - No `.get()` method makes it harder to subvert ownership

Some kind of asynchrony

```
struct flash_driver {
    virtual {async}
    read(flash_page addr, {target buffer}, {result code}) = 0;

    virtual {async}
    write(flash_page addr, {source buffer}, {result code}) = 0;

    virtual {async}
    erase(flash_block blk, {result code}) = 0;

    // ...
};
```

Create an async API

```
struct req : disposable {
    virtual void start(std::function<void()> on_finish) = 0;
};
```

- A “`request`” is effectively a manually-implemented coroutine
  - All captured data is an explicit member variable
  - Anything calling an async function must itself be async
  - It ends up being `req`s all the way down
- We have on the order of 1000 different `req`s in the code
  - TODO: explore coroutines and see if I can slowly convert

More APIs

```
struct byte_buffer : disposable {
    virtual std::span<std::byte> to_aperture() = 0;

    virtual void
    trim_buf(size_t length, size_t offset = 0u) = 0;
};

// Factory function
owned<byte_buffer> byte_buffer_new(size_t bytes);
```

Fill in some blanks...

```
struct flash_driver {
    virtual owned<req>
    read(flash_page addr, byte_buffer & target,
         device_command_result * ref_result) = 0;

    virtual owned<req>
    write(flash_page addr, owned<byte_buffer> && source,
          device_command_result * ref_result) = 0;

    virtual owned<req>
    erase(flash_block blk, device_command_result * ref_result) = 0;

    // ...
};
```

Design Goals

Goal #0

- Express ownership (and non-ownership) through the type system
- Auto-cleanup of owned resources via RAII
- Zero-overhead for non-owning pointers (after optimization)

Goal #1

- It should be hard to take ownership of a resource I'm not intended to own
  - Make it harder to create double-free or use-after-free bugs

```
std::unique_ptr<foo> bar = make_unique<bar>();
std::unique_ptr<foo> oops(bar.get());
```

Goal #1

- It should be hard to take ownership of a resource I’m not intended to own
  - Make it harder to create double-free or use-after-free bugs

```
void func(foo * ptr) {
    delete ptr; // Is this correct? It compiles…
}
```

Goal #1

- It should be hard to take ownership of a resource I’m not intended to own
  - Make it harder to create double-free or use-after-free bugs

```
void func(foo * ptr) {
    std::unique_ptr<foo> oops(ptr); // No raw delete!
}
```

Goal #1

- It should be hard to take ownership of a resource I’m not intended to own
    - Make it harder to create double-free or use-after-free bugs

```
void func(foo * ptr) {
    std::unique_ptr<foo> oops(ptr); // No raw delete!
}
```

```
void func(std::unique_ptr<foo> const & ptr);
void func(std::shared_ptr<foo> const & ptr);
// … how many more prototypes do I need for the same function,
// just to avoid the raw pointer?
```

Goal #1

- It should be hard to take ownership of a resource I'm not intended to own
  - Using references instead of pointers doesn't make it that much harder

```
void func1(foo * opt_ptr) {
    delete opt_ptr;
}
void func2(foo & val) {
    func1(&val);                // oops…
}
void func3(foo & val) {
    std::unique_ptr<foo>(&val); // oops...
}
```

Goal #1

- It should be hard to take ownership of a resource I’m not intended to own
    - `.get()` considered harmful?
    - C++20 [`std::pointer_traits<T>::to_address()`] I think is for fancy pointers, not smart pointers
        - And “to_address” isn’t a scary enough phrase

Goal #2

- It should be easy to construct a non-owning pointer from an owning pointer
  - Without a lot of extra typing
  - I.e. implicit constructors from owning to non-owning smart pointer

Goal #3a

- Eliminate raw pointers
  - A convention that “raw pointers are non-owning” is still problematic
    - Accidentally taking ownership or deleting
    - People forget conventions, the compiler doesn’t
    - Legacy or 3rd party code may still have raw pointers that also have ownership
  - Can’t eliminate 100% of raw pointers, but we can get close

Goal #3b

- Eliminate raw references
  - `const &` is safe from delete and pushing into a `unique_ptr<>`
  - Use of `foo.bar` implies contiguous memory to a C programmer; also implies we shouldn't think about lifetime
  - Passing `&foo` to a function makes it obvious at the call site that a parameter may be modified
    - C programmers can't get confused, since the `&` is visible
    - C++ programmers don't need to look up the prototype to know
    - But the semantic that we know the "thing to be filled in" is never `nullptr` is nice

Possible names

- If I don’t own a resource, I’m *borrowing* it (if it has lifetime control)
  - The borrowed view is only valid for the context of my stack frame
  - Need a copy to go async, because caller still owns the resource and can do anything she wants
    - Or document the API well (because sometimes performance is more important than perfect safety)
  - Objects with lifetimes control are always *owned* or *borrowed*
- Some non-ownership use-cases were for an *out* parameter
  - Because we manually implemented co-routines, but this comes up in other contexts too
  - I can assume these pointers are valid until the requested async operation completes
    - It makes no sense to ask a function to fill in some value, then delete the memory it’s still working on
    - Accidents will still happen, but perfection probably has too high a cost

Proposed solution

- borrowed<any_t> expresses a lack of ownership
    - a non-owning pointer, with operator->() and operator*() as expected
    - borrowed_ref<> for reference semantics (assigned on construction, never null)
        - 2 choices: assignable or not, null or not
        - Need for a non-assignable, optional borrowed? Or an assignable, never-null borrowed?
        - There are use-cases, but they're not common, and we have a lot of names already
    - … maybe we should have named it borrowed_ptr<> and borrowed_ref<>
        - But codebases evolve, and the idea of reference semantics on a borrowed<> wasn't there originally
    - We own the code for borrowed<> and owned<>; in theory we could add debug checks to the borrowed's dereference operators that the source memory is still good

Proposed solution

- `out<any_t>` also expresses a lack of ownership
  - With the expectation that the underlying memory has a “long enough” lifetime
    - As long as the synchronous function call, or until an async request completes
  - `in_out<>` to indicate we’re not just storing through this pointer, but possibly reading through it
  - `out_opt<>` to indicate an optional out parameter
    - Fun with names: does “opt_out” mean an optional out parameter, or that you’re opting out of some behavior
    - Do we need an `in_out_opt<>`? Could there be an optional in parameter?
  - Use of `out<>` should look identical to a raw pointer
    - I.e. I only need to change prototypes and member variables, no other code.
  - This is unrelated to the proposed `out_ptr`, which is meant for wrapping a pointer-to-pointer for legacy C APIs.
    - Could name ours `out_param<>`, `in_out_param<>`, etc
    - Naming things is hard!

Putting it together

```
owned<req> read(flash_page addr, byte_buffer & target,
                device_command_result * ref_result);

owned<req> write(flash_page addr, owned<byte_buffer> && source,
                 device_command_result * ref_result);


... = drv->read(addr, *target, &result);
... = drv->write(addr, std::move(buf), &result);
```

Putting it together

```
owned<req> read(flash_page addr, borrowed_ref<byte_buffer> target,
                out<device_command_result> ref_result);

owned<req> write(flash_page addr, owned<byte_buffer> && source,
                 out<device_command_result> ref_result);


... = drv->read(addr, target, &result);
... = drv->write(addr, std::move(buf), &result);
```

Implementation Details

Prototypes

```
namespace detail {
    struct reference {};
    struct pointer {};
}

// Forward declarations
template<typename any_t = disposable> struct owned;
template<typename any_t, typename tag_t = detail::pointer> struct borrowed;
template<typename any_t, typename tag_t> struct param_ptr;

// Aliases
template<typename any_t>
using borrowed_ref = borrowed<any_t, detail::reference>;

template<typename any_t> using out     = param_ptr<any_t,
detail::reference>;
template<typename any_t> using in_out  = param_ptr<any_t,
detail::reference>;
template<typename any_t> using out_opt = param_ptr<any_t, detail::pointer>;
```

owned<> **constructors**

```
// No copying is allowed
template<typename other_t>
owned(owned<other_t> const &) = delete;

owned(element_type * p = nullptr) : p_(p) {}

template<typename other_t>
owned(owned<other_t> && rhs) : p_(rhs.p_) {
    rhs.p_ = nullptr;
}
```

`owned<>` constructors

```
template<typename any_t>
explicit owned(any_t * const & p) : p_(p) {}

template<typename any_t>
/* implicit */ owned(any_t * && p) : p_(p) {}
```

- Not 100% safe (e.g. `std::move(ptr)`) but correct almost all the time
    - An rvalue pointer that expresses ownership is a memory leak if it's not put in an `owned<>`

```
owned<req> flash_driver_impl::read(page, buffer, res) {
    return new flash_read_req(*this, page, buffer, res);
}
```

owned<> **other ops**

```
element_type * operator->() const { PS_ASSERT(p_); return p_; }

element_type & operator*() const { PS_ASSSERT(p_); return *p_; }

explicit operator bool() const { return p_ != nullptr; }

template<typename any_t>
owned<itf_t> & operator=(owned<any_t> && rhs) {
    // Order ops to allow self-assign to not blow up
    element_type * p_next = rhs.p_;
    rhs.p_ = nullptr;
    element_type * p_prev = p_;
    p_ = p_next;
    if (p_prev) { static_cast<disposable *>(p_prev)->dispose(); }
    return *this;
}
```

Avoiding `.get()`

```
// Forward declare the function name, so we can friend it in each smart
// pointer implementation.
template<typename ptr_t>
ptr_t::element_type * raw_pointer_ignoring_lifetime (ptr_t const & ptr);

template<typename any_t> any_t *
raw_pointer_ignoring_lifetime(std::unique_ptr<any_t> const & ptr) {
    return ptr.get();
}

template<typename any_t> any_t *
raw_pointer_ignoring_lifetime(std::shared_ptr<any_t> const & ptr) {
    return ptr.get();
}

// (overloads for owned<>, borrowed<>, etc. after each class is defined)
```

Avoiding `.get()`

- Still possible to construct a raw pointer:
  - `auto * foo = &*smart_ptr;`
  - `&*` can be considered a code smell; can’t forbid it entirely but it should make a reader twitchy
    - (just like seeing `raw_pointer_ignoring_lifetime` should make a reader twitchy)
  - `owned<>::operator->()`
    - `foo = bar.operator->()` should make me even more twitchy

borrowed<> constructors

```
borrowed() : p_(nullptr) {
    static_assert(std::is_same_v<tag_t, detail::pointer>,
                  "borrowed_ref must be initialized on construction");
}

borrowed(std::nullptr_t) : p_(nullptr) {
    static_assert(std::is_same_v<tag_t, detail::pointer>,
                  "borrowed_ref cannot be nullptr");
}

// Our source isn't quite like this but this is logically how to make a
// borrowed<> from another smart pointer.
template<typename ptr_t, PS_REQUIRES(detail::is_smart_ptr_v<ptr_t>) >
borrowed(ptr_t const & ptr) : p_( raw_pointer_ignoring_lifetime(ptr) ) {
    PS_ASSERT(std::is_same_v<tag_t, detail::pointer> || !!p_);
}
```

borrowed<> **constructors**

```
template<typename other_t, other_tag_t>
borrowed(borrowed<other_t, other_tag_t> const & other ) : p_(other.p_) {
    static_assert(!std::is_same_v<tag_t, detail::reference> ||
                  std::is_same_v<other_tag_t, detail::reference>,
"Cannot make a reference from pointer without an explicit use of
operator*");

    PS_ASSERT(std::is_same_v<tag_t, detail::pointer> || !!p_);
}

template<typename other_t, other_tag_t>
borrowed(borrowed<other_t, other_tag_t> && other ) : p_(other.p_) {
    static_assert(!std::is_same_v<tag_t, detail::reference> ||
                  std::is_same_v<other_tag_t, detail::reference>,
"Cannot make a reference from pointer without an explicit use of
operator*");

    PS_ASSERT(std::is_same_v<tag_t, detail::pointer> || !!p_);
}
```

borrowed<> **constructors**

```
// borrowed_ref cannot be initialized from a pointer at all;
// we use SFINAE for this to help overload resolution.
template<typename any_tag_t = tag_t,
         PS_REQUIRES(std::is_same_v<any_tag_t, detail::pointer>) >
borrowed(any_t * const & p) : p_(p) {}

// An unnamed pointer probably shouldn't be borrowed<>, as it's likely a
// memory leak.
template<typename any_tag_t = tag_t,
         PS_REQUIRES(std::is_same_v<any_tag_t, detail::pointer>) >
explicit borrowed(any_t * && p) : p_(p) {}
```

`borrowed<>` constructors (deleted)

```
// Making a borrowed from a param_ptr<> is dodgy, as the two have different
// intended meanings.  We may discover someday that this is needed.

template<typename other_t, typename other_tag_t>
explicit borrowed(param_ptr<other_t, other_tag_t> const &) = delete;

template<typename other_t, typename other_tag_t>
explicit borrowed(param_ptr<other_t, other_tag_t> &&) = delete;
```

borrowed<> constructors (deleted)

```
// Prevent some accidents with an rvalue-reference to a smart pointer, since
// we know this would create a dangling reference.  For reasons I don't
quite
// understand, this needs to be an explicit constructor to match as
expected.

template<typename other_t>
explicit borrowed(owned<other_t> &&) = delete;

template<typename other_t>
explicit borrowed(std::unique_ptr<other_t> &&) = delete;

// … delete constructor for other smart pointer types
```

`borrowed<>` **comparisons**

```
bool operator==(borrowed const & rhs) const     { return p_ == rhs.p_; }
bool operator==(element_type const * rhs) const { return p_ == rhs; }
bool operator==(std::nullptr_t) const           { return p_ == nullptr; }
// Similarly, three comparisons for operator!=

// Free function overloads for operator== and operator!=, for different
// kinds of smart pointers using the detail::is_smart_ptr_v<ptr_t> helper.
// All smart pointers can be converted to borrowed, so it's the common type
// for pointer comparison.
```

- We do not support `operator<` and other comparisons because there should be no relation between two different managed resources.
    - May need this some day, but it would make me sad and I'd like to know the use case before it's supported

borrowed<> **comparisons**

```
namespace std {
    template<typename any_t, tag_t>
    struct common_type<any_t *, borrowed<any_t, tag_t>> {
        using type = borrowed<any_t>;
    };

    template<typename any_t, tag_t>
    struct common_type<borrowed<any_t, tag_t>, any_t *> {
        using type = borrowed<any_t>;
    };
}
```

param_ptr<> constructors

```
param_ptr() = delete;
param_ptr(param_ptr const &) = default;
param_ptr(param_ptr &&) = default;

param_ptr(std::nullptr_t) : p_(nullptr) {
    static_assert(std::is_same_v<tag_t, detail::pointer>,
                  "Cannot have a null mandatory parameter");
}

// We allow construction of a parameter from a raw pointer only.  This
forces
// callers to pass &foo for objects that could be modified by the function
// being called.
template<typename other_t> /* implicit */
param_ptr(other_t * p) : p_(p) {
    PS_ASSERT(std::is_same_v<tag_t, detail::pointer> || p_);
}
```

param_ptr<> constructors

```
template<typename other_t, other_tag_t> /* implicit */
param_ptr(param_ptr<other_t, other_tag_t> const & other) : p_(other.p_) {
    static_assert(std::is_same_v<tag_t, detail::pointer> ||
                  !std::is_same_v<other_tag_t, detail::pointer>,
"Implicit conversion from optional to mandatory parameter is forbidden”);
```

param_ptr<> other ops

```
// No comparison operations at all

// No assignment operations -- "parameters" should always be
// constructed once and never overwritten.

element_type * operator->() const { return p_; }
element_type & operator*() const  { PS_ASSERT(!!p_); return *p_; }

/* implicit */ operator element_type * () const { return p_; }

// Allow explicit casting to any other pointer type that's safe,
// just as though this was a raw pointer.
template<typename other_t> explicit operator other_t * () const {
    return static_cast<other_t *>(p_);
}
```

Add a helper...

```
template<typename impl_t, typename itf_t = disposable>
struct disposable_base : itf_t {
    static_assert(std::is_base_of_v<disposable, itf_t>);

    void disposable_dispose() {
        delete static_cast<impl_t *>(this);
    }

  private:
    void dispose() final override {
        static_cast<impl_t *>(this)->disposable_dispose();
    }
};
```

Hand-waved async flash page read

```
struct flash_read_req : disposable_base<flash_read_req, req> {
    borrowed_ref<flash_driver_impl>  parent_;
    flash_page const                 addr_;
    borrowed_ref<byte_buffer>        buf_;
    out<device_command_result> const ref_result_;
    std::function<void()>            on_complete_;
    dma_address                      dma_addr_;
    uint16_t                         cmd_slot_;
    owned<req>                       inner_;

    flash_read_req({ parent, addr, buf, ref_result })
        : parent_(parent), addr_(addr), buf_(buf), ref_result_(ref_result)
{}

    void start(std::function<void()> on_complete> override;
    void notify_dma_addr();
    Void notify_slot();
    void notify_done(hw_result_code code);
};
```

Hand-waved async flash page read

```
void flash_read_req::start(std::function<void()> on_complete)
{
    on_complete_ = on_complete;
    inner_ =
        parent_->dma_svc_->get_dma_address(buf_->to_aperture(), &dma_addr_);
    if (inner_) {
        inner_->start([this](){ notify_dma_addr(); });
    } else {
        notify_dma_addr();
    }
}
```

Hand-waved async flash page read

```
void flash_read_req::start(std::function<void()> on_complete)
{
    on_complete_ = on_complete;
    inner_ =
        parent_->dma_svc_->get_dma_address(buf_->to_aperture(), &dma_addr_);

    call<&flash_read_req::notify_dma_addr>(inner_);
}
```

Hand-waved async flash page read

```
void flash_read_req::notify_dma_addr()
{
    inner_ = parent_->hw_svc_->allocate_fpga_command_slot(&cmd_slot_);
    call<&flash_read_req::notify_cmd_slot>(inner_);
}

void flash_read_req::notify_cmd_slot()
{
    parent_->completion_table_.at(cmd_slot_) = this;

    // Set up FPGA registers to read from paddr into dma_addr_
    phys_addr paddr = parent_->geometry_->translate(addr_);

    // Our interrupt handler will call back into this req when it sees
    // a completion for the provided command slot.
}
```

Hand-waved async flash page read

```
void flash_read_req::notify_done(hw_result_code code)
{
    parent_->hw_svc_->release_fpga_command_slot(cmd_slot_);
    inner_.reset(); // Early release of FPGA command slot

    *ref_result_ = convert_hw_code(code);
    buf_->trim_buf(result_size_from_hw_code(code));
    on_complete_();
}

owned<req> flash_driver_impl::read(flash_page                 addr,
                                   borrowed_ref<byte_buffer>  target,
                                   out<device_command_result> ref_result)
{
    return new flash_read_req(*this, addr, target, ref_result);
}
```

Other allocation strategies

```
owned<req> flash_driver_impl::read(flash_page                 addr,
                                   borrowed_ref<byte_buffer>  target,
                                   out<device_command_result> ref_result) {
    size_t index = geometry_->linear_lun_index(addr);
    flash_read_req & ret = reserved_.at(index);
    PS_ASSERT(ret.is_free_);
    ret.is_free_ = false;
    ret.initialize(addr, target, ref_result);
    return owned<req>(&ret);
}

void flash_read_req::disposable_dispose() {
    PS_ASSERT(!is_free_);
    is_free_ = true;
    // Do nothing, we're pre-allocated.
}
```

Other allocation strategies

```
owned<req> flash_driver_impl::read(flash_page                addr,
                                   borrowed_ref<byte_buffer>  target,
                                   out<device_command_result> ref_result)
{
    owned<flash_read_req> ret =
        atomic_stack_.empty() ? nullptr : atomic_stack_.pop();
    if (ret) {
        ret->initialize(addr, target, ref_result);
    } else {
        ret.reset(new flash_read_req(*this, addr, target, ref_result));
    }
    return ret; // may need std::move to coerce the owned<req> move-ctor
}
```

Other allocation strategies

```
void flash_read_req::disposable_dispose() {
    if (parent_->atomic_stack_.sampled_size() > 30) {
        delete this;
    } else {
        parent_->atomic_stack_.push(this);
    }
}

void flash_read_req::disposable_dispose() {
    if (is_preallocated_) {
        parent_->atomic_stack_.push(this);
    } else {
        delete this;
    }
}
```

Multiple Owners

A simple reference counter

```
struct latched_refcounter {
    // 16 million refs should be enough for anyone...
    static uint32_t constexpr k_max_refs = 0x00ffffffu;

    latched_refcounter(uint32_t init = 1u) : refs_(init) {
        PS_ASSERT(init > 0u);
    }

    bool try_ref();
    void ref();
    uint32_t deref(uint32_t count = 1u);
  private:
    std::atomic<uint32_t> refs_;
};
```

A simple reference counter (cont'd)

```
bool latched_refcounter::try_ref()
{
    return atomic_try_update(&refs_, [](uint32_t * ref_v) -> bool {
        PS_ASSERT(*ref_v <= k_max_refs);
        if (*ref_v == 0u) {
            return false; // cannot increment once we hit 0
        }
        ++(*ref_v);
        return *ref_v <= k_max_refs;
    }, std::memory_order_relaxed);
}
```

A simple reference counter (cont'd)

```
void latched_refcounter::ref() {
    bool success = try_ref();
    PS_ASSERT(success);
}

uint32_t latched_refcounter::deref(uint32_t count = 1u) {
    // Need release semantics so any subsequent single-threaded
    // code (like a destructor) sees all updates done prior to
    // releasing the last ref.
    uint32_t old = refs_.fetch_sub(count,
                                   std::memory_order_acq_rel);
    PS_ASSERT(old >= count);
    return old - count;
}
```

A multiply-owned object

```
struct refable : disposable {
    // The implementation must be thread-safe and
    // guarantee atomicity of ref() and dispose() methods.
    //
    // Do not call ref__() directly; instead implementations should
    // use refable_mixin<> to provide a function ref().  A raw
    // pointer is required here for covariant return type.
    virtual refable * ref__() = 0;
};
```

A multiply-owned object (cont'd)

```
template<typename src_t, typename base_t = refable>
struct refable_mixin : base_t {
    template<typename target_t = src_t> owned<target_t> ref_as() {
        // … type checking and casting to target_t
    }

    owned<src_t> ref() {
        return static_cast<src_t *>(this->ref__());
    }
};
```

Add a helper…

```
template<typename impl_t, typename itf_t = refable_mixin<impl_t>>
struct refable_base : itf_t {
    // … static_assert that itf_t inherited from disposable
    explicit refable_base(uint32_t init = 1u) : refs_(init) {}

    refable * ref__() override;
    void refable_dispose_final();
  protected:
    latched_refcounter refs_;
  private:
    void dispose() final override;
};
```

Add a helper...

```
refable * refable_base::ref__() {
    refs_.ref();
    return static_cast<refable *>(this);
}

void refable_base::dispose() {
    if (refs_.deref() == 0u) {
        static_cast<impl_t *>(this)->refable_dispose_final();
    }
}

void refable_base::refable_dispose_final() {
    delete static_cast<impl_t *>(this);
}
```

A use of `refable`

```
struct byte_buffer : refable {
    virtual std::span<std::byte> to_aperture() = 0;

    virtual void
    trim_buf(size_t length, size_t offset = 0u) = 0;
};

// Factory function
owned<byte_buffer> byte_buffer_new(size_t bytes);
```

A use of `refable` (cont'd)

```
owned<req> read(addr, borrowed_ref<byte_buffer>, result);
```

- Want to read the data into a provided buffer (not just aperture)
  - … and resize the buffer if it wasn't correct, using `trim_buf()`
  - Here, the function needed access to the buffer itself

```
owned<req> write(addr, owned<byte_buffer> &&, result);
```

- May want to write the same buffer to multiple locations
  - So we `ref()` the buffer before handing off to each call to write
  - Might be nice to have a `std::span<std::byte const> to_const_aperture()` for the write case, and assert in to_aperture() if someone asks for a modifiable buffer when the buffer is multiply-owned

Hand-waving mirrored write

```
results_.reserve(mirrors_.size());

req_accum reqs;
for (auto const & mirror : mirrors_) {
    results_.emplace_back();
    reqs.push(drv->write(mirror, buf_->ref(), &results_.back());
}

inner_ = reqs.extract();
call<&mirrored_write_req::notify_writes>(inner_);
```

Ownership

```
owned<byte_buffer> buf_;
```

- Is this instance of a byte_buffer singly-owned? Multiply-owned?

```
owned<foo> member_;
```

- Is this instance of `foo` singly-owned? Multiply-owned?
  - In general I might know if `foo` is `refable` or just `disposable`. But I still know whether the API gives me a ref() function, not whether the implementation was ref'd

```
owned<> member2_;
```

- Is this instance of a disposable resource singly or multiply owned?

Why does it matter?

- The consumer should care about the handle in their hand, nothing else
  - Less non-local reasoning!
  - To be fair, the `byte_buffer` case is complex, since I probably shouldn't read into a buffer with more than one ref
    - Handing off a `borrowed<byte_buffer>` allows a consumer to take a ref (and I may not be aware of)
    - … on the other hand, we own all the software; we should have some idea what the consumers are doing, by contract
    - The type system doesn't let me say at compile-time whether we can only get `const` access to the data, without making some kind of `byte_buffer_const` type
    - … this is where owning all our code and not making a general computing platform is handy; we can play *slightly* looser with sharp corners like this

Cancellation helper

```
struct cancel_binding_impl : disposable {
    std::function<void()> cancel_;

    void dispose() override {
        cancel_();
    }

    owned<> init(std::function<void()> on_cancel) {
        on_cancel_ = on_cancel;
        return *this;
    }
};
```

Cancellation! (and more hand-waving!)

```
struct flash_read_req : refable_base<flash_read_req, req> {
    cancel_binding_impl cancel_;

    flash_read_req(params..., opt_out<owned<>> cancel_binding)
        : members(params...)
    {
        if (cancel_binding) {
            // Take a reference to ourself, so the req is valid until it’s
            // both disposed and the cancel binding has been reset.
            *cancel_binding = cancel_.init([this, self = ref()] {
                do_cancel();
                self.reset();
            });
        }
    }
};
```

Using cancellation (hand waving!)

```
owned<req> inner_;
owned<>    cancel_;

auto inner = drv->read(..., &cancel_);
inner_.reset(timeout_race_req_new(std::move(inner),
    [this](){ cancel_.reset(); }, 10_ms));

// If we time out before the read op finishes, we cancel it and will
// get a cancellation error returned to the continuation function.
inner_->start([this](borrowed<ps_err> opt_e) {
    if (opt_e) {
        // call caller's on_complete with the error
    } else {
        notify_read();
    }
});
```

Using cancellation

- The cancellation binding looks like any other `owned<>` resource
  - But it calls arbitrary code
  - RAII means we can't forget to release that ref on the underlying request
- We could expose some other kind of handle for the cancellation
  - But it would still need to always-happen on destruction
  - `owned<>` is convenient, and not a lie -- we do have some kind of ownership/handle, and the implementation must make it always safe to `.reset()`

Summary

Comparing names

- A custom deleter requires a mouthful of a name
  - `using ptr = std::unique_ptr<foo, foo_deleter>`
  - Why do I (the consumer of a smart pointer) care what the type of the deleter is?
    - I just want to relinquish ownership; it's a leaky abstraction if I know how that happens
    - But deleter is encoded in the type system; I can't not say it (except with a `using` alias)
- And possibly a mouthful at initialization time
  - `ptr val(new foo, instance-of-foo_deleter)`
  - Deleter with state requires a 16-byte `unique_ptr`
- And the deleter instance is specified separately from the pointer
  - … which makes it easier to get one wrong, especially during refactoring
- A `disposable` instance always knows how to delete itself
  - The implementation already had to know, but now we don't need to separately specify

Comparing names (cont'd)

- If ownership changes to shared ownership:
  - Large rototill of codebase from `unique_ptr` to `shared_ptr`
- … And do uses of a type really need to know the ownership model?
  - I should just know I have ownership, meaning I know my pointer is still valid
- None of this is game-changing
  - But if a `unique_ptr` isn't always unique, that's confusing
  - And if I didn't need to know unique or multiple ownership, why encode that in the type system?
  - And if I don't actually care how a resource is freed, why encode that in the type system?
  - And if I wanted control over how reference counting happened, `shared_ptr` doesn't let me have it
- This is an advantage of designing a code base for private use -- we can decide some use cases don't matter to us

Comparing names (cont'd)

- `unique_ptr` still has a place, for e.g. safe wrapper around `FILE *`
    - But `delete_ptr` is “better” for the base case
    - And if I had virtual functions I may as well be `disposable`
    - Leaving little room left in our code base for unique/shared ptr

Rolling your own

- If I roll my own (C++ thing) am I likely to like mine better?
  - If I did before standardization, there’s a large effort to replace for little benefit
  - I can better support my (likely) narrower usecase
    - E.g. better asserts, better checks, different kinds of flexibility
    - Better API (no `.get()`, no `operator<`, etc.)
    - Better performance?

Does language affect thought?

- Not really -- I can still *think* about things
  - But it's harder to express them
- If all the code I see is `owned<>` or POD, will I tend to think in terms of virtual APIs and factories, since that's all I can easily express?
  - Would that be bad?

Conclusion

- Names matter
  - “Ownership” feels like a more important concept than what kind of ownership
- General purpose languages may be solving different problems
  - POD types and pure-virtual APIs works for us
  - It may even work for you
  - I probably wouldn’t design a programming language around it, though
- This isn’t ground-breaking
  - Still Turing complete either way
  - But it kinda feels nice

Appendix

`retain_ptr<>` **and** `owned<>`

```
struct refable_retain_traits {
    using pointer = ???;
    static void increment(refable * p) { p->__ref(); }
    static void decrement(refable * p) { owned<>(p); }
};
```

- Not quite sure what to expose for `pointer` type alias
  - We worked pretty hard to eliminate `.get()` and raw pointers
    - I don't really want to give that back...
  - Exposing the `use_count()` is possible

[Link to 2018 version of `retain_ptr` proposal]()

Comparing sizes

	Object size	`unique_ptr<>` size	`owned<>` size
disposable	+ 8 bytes	8 or 16 bytes	8 bytes
Virtual destructor	+ 8 bytes	8 or 16 bytes	n/a
refable	+ 16 bytes	???	8 bytes

- Size isn’t everything, but systems software can care about this
  - We almost never have fewer pointers than objects, but we can easily have more pointers than objects

Comparing sizes

- Virtual methods imply we'd need a virtual destructor (+8 bytes)
    - Or inherit from disposable instead; it's the same +8 bytes
    - `owned<>` is always 8 bytes, regardless of what dispose() does
    - `unique_ptr<>` is 8 or 16 bytes, depending on implementation and custom deleter, and we can't use a customer deleter with `make_unique()`
    - `shared_ptr<>` is 16 bytes (plus reference block?), and we can't use a custom deleter with `make_shared()`
    - `weak_ptr<>` is probably possible to implement with owned<>, but we haven't needed it
        - … and `make_shared` + `weak_ptr` is equivalent to giving full ownership in terms of memory lifetime

owned_pod<>

- Terrible name (but nothing better occurred to me)

```
template<typename pod_t, typename disp_t = disposable>
struct owned_pod {
    // static assert pod_t is standard layout
    pod_t & operator*() { PS_ASSERT(owner_); return data_; }
    pod_t * operator->() { PS_ASSERT(owner_); return & data_; }
    // ... other access
    // reset operation to release owner and poison data_
  private:
    pod_t          data_;
    owned<disp_t> owner_;
};
```

`owned_pod<>`

```
template<typename any_t>
using owned_span = owned_pod<std::span<any_t>>;
```

- `owned_span<>` is an array-like object that knows its lifetime
  - Handy for passing ownership of vectors, spans, slices, etc in a uniform way
  - `owned_span<uint8_t>` is a generic way to refer to someone’s bag-of-bytes buffer

owned_pair<>

```
template<typename ift_t, typename disp_t = disposable>
struct owned_pair {
  private:
    itf_t *        p_;
    owned<disp_t> owner_;
};
```

- Expresses ownership of a generic pointer that isn’t already disposable
    - We’d like to remove most uses of this in our code, since it’s not really needed
- Can express a sub-view of some object, with lifetime tied to the global object

Callbacks: `notify_param`

```
using notify_f = void (*)(void * impl, borrowed<ps_err> opt_e);

struct notify_param {
    notify_f func = nullptr;
    void *   param = nullptr;

    notify_param(notify_f f, void * p) : func(f), param(p) {}
    notify_param() = default;

    void fire(borrowed<ps_err> opt_e = nullptr) {
        notify_f f = func; void * p = param;
        func = nullptr; param = nullptr;

        PS_ASSERT(!!f);
        f(p, e);
    }
};
```

Callbacks: notify_param

```
template<typename impl_t>
struct as_param {
    template<void (impl_t::*notify_func_t)(borrowed<ps_err>)>
    struct impl_notify {
        static void func(void * param, borrowed<ps_err> opt_e) {
            PS_ASSERT(!!param);
            (static_cast<impl_t *>(param)->*notify_func_t)(opt_e);
        }
    };

    template<void (impl_t::*notify_func_t)(borrowed<ps_err>)>
    static notify_f to_notify() {
        return &impl_notify<notify_func_t>::func;
    }
};
```

Callbacks: `notify_param`

```
template<typename impl_t>
class notify_impl
{
  protected:
    using param_t = as_param<impl_t>;

  public:
    template<void(impl_t::*notify_func_t)(borrowed<ps_err>)>
    notify_param to_notify()
    {
        return notify_param(param_t::template to_notify<notify_func_t>(),
            param_t::template to_param<notify_impl<impl_t>>(this));
    }
};
```

Preventing accidents

- Some code in C still called `p->dispose()` manually
  - We want `->dispose()` to not compile in C++ code

```
class no_dispose_base {
    void dispose();
};

template<typename itf_t>
struct no_dispose : itf_t, no_dispose_base {
  private:
    no_dispose() = delete; // No default constructor.
    // no copy or assign, either
};
```

```
no_dispose<itf_t> * operator->() const { return static_cast<no_dispose<itf_t> *>(p_); }
```

Preventing accidents

```
template<typename itf_t>
struct no_dispose : itf_t, no_dispose_base { … };
```

- struct foo_impl final : disposable_base<foo_impl> { … };
- struct foo_impl; owned<foo_impl> parent; ...

- Oops, this doesn't work for final classes or forward-declared classes
  - And Clang throws a warning when inheriting from a type with an expansion array
- Hence, mark `dispose()` private and let `owned<>` be a friend.
  - Fun fact: we only fixed `owned<>` like this in late 2018
    - But now we can `borrowed<>` all the things!