The Smart Pointers I Wish I Had

Matthew Fleming
Pure Storage

“Fix” Lifetime Issues with this *One Weird Trick*

Developers Hate Him!

My Domain

- C++11 (though we can probably move to C++17 next year)
- Moderately multi-threaded (8 to 60 threads, one thread per core)
- Mostly lock-free / wait-free
- Almost entirely async (using a homegrown stackless coroutine system)
  - Hopeful we can switch much of this to C++20 coroutines
- 100+ developers, 1.3M LOC
  - Most not C++ experts, new to async, lock-free, flash storage, TMP, file systems, etc., etc.
  - Tools are crucial so they can focus on what, not how; i.e. think about their problem and not C++ or async or lock-free
- Low-ish latency flash storage

Click-Bait Title

- There is no Silver Bullet (Fred Brooks)
- We can’t fix everything
- Nothing is magic
- But we can make things better™
  - Many incremental improvements
  - (Better is sometimes in the eye of the beholder)
- “An Alternate Smart Pointer Hierarchy”
  - 2019 C++Now talk focusing more on owning pointers

Thesis 1

None of these coding tools/types are needed.

100% correct code (modulo exception safety) can be written with raw C++.

Each of these tools should add value.

Definitions

- Lifetime error: using a pointer/reference, when it no longer points to the intended target
  - Can cause memory corruption, scribbling, or it may silently just work!
- Double-free: the same address is freed twice
  - Can lead to other lifetime errors, allocator corruption, allocator crashes, and more
- Use-after-free: using a pointer after the memory has been freed
  - This is a lifetime error, but may also be a segfault

Using Tools to Detect Lifetime Issues

- `-fsanitize=address` is awesome
  - But it can't catch a timing race or rare early `delete` until it actually happens
- Static analysis is awesome
  - But tools don't know how long a function parameter will live
- Tests are nice
  - But they don't prove correctness, especially if they can't force rare timing paths
- Reading code is the only thing I really trust
  - And convincing ourselves it all works

Guideline 1: No Owning Raw Pointers

- Can we ban `delete` entirely and only use explicitly owning pointers?
  - Not always; atomic types, pre-C++14 lambdas may need raw, owning pointers
- We've been talking about this since 2011 or before
  - `std::unique_ptr<>`
- Fixes memory leaks due to missed `delete`
  - Makes it slightly harder to have a double-free

Guideline 1: No Owning Raw Pointers

All of these compile. Only one is likely to crash immediately.

```
1 auto foo = make_unique<int>(0);
2 delete foo.get();
3 delete &*foo;
4 delete &foo;
```

Guideline 1’: Almost No Owning Raw Pointers

- `delete` may not be a bug, but it should be rare
- Each of the (small number of) `delete`s in the code should be obviously right

Too Easy to Type the Wrong Thing?

```
-bool notify(error_obj *) {
+bool notify(error_obj * opt_e) {
+    if (opt_e) {
+        // Drop the error; this is another case where there's
+        // nothing to notify.
+        delete opt_e;   <--
+        return false;
+    }
     return operation_succeeded_;
 }
```

Too Easy to Type the Wrong Thing?

```
-bool notify(error_obj *) {
+bool notify(error_obj * opt_e) {
+    if (opt_e) {
+        // Drop the error; this is another case where there's
+        // nothing to notify.  This is a legacy API where we
+        // still pass ownership with a raw pointer. Pacify the
+        // style-checker by not using delete.
+        unique_ptr<error_obj> to_del(opt_e);
+        return false;
+    }
     return operation_succeeded_;
 }
```

Too Easy to Type the Wrong Thing?

```
-bool notify(error_obj *) {
+bool notify(observer_ptr<error_obj> opt_e) {
+    if (opt_e) {
+        // Drop the error; this is another case where there's
+        // nothing to notify.
+        // delete opt_e;                          <--
+        // unique_ptr<error_obj> to_del(opt_e);   <--
+        return false;
+    }
     return operation_succeeded_;
 }
```

Thesis 2

Explicit is better than implicit

Guideline 2: No Non-Owning Raw Pointers

- C++20 introduces `observer_ptr<>`
  - Definitely an improvement over raw non-owning pointers
  - IMO an 70% solution to the overall problem
- Unlike owning pointers (move only) we can convert 100% of non-owning pointers to `observer_ptr<>`
  - We can always get the raw pointer back for legacy APIs
  - Trivially copyable, assignable, etc. just like a `T*`

Thesis 3

These types work best when we have less code to change

I.e., just like raw pointers trivially flowed through our interfaces, smart pointers (owning and non-owning) should flow trivially

(except where they shouldn’t)

observer_ptr<> in Brief

```
template <typename any_t>
struct observer_ptr {
  any_t * raw_ptr = nullptr;

  explicit observer_ptr(any_t * p)
    : raw_ptr(p) {}
  // defaulted copy, move ctor
  // defaulted copy, move assign
  // convert from observer_ptr<U>

  any_t * release();
  void reset();
  explicit operator bool() const;

  any_t * get() const
    { return raw_ptr; }

  any_t * operator->() const
    { return raw_ptr; }
  any_t & operator*() const
    { return *raw_ptr; }

  // Plus more utility functions
  // that match unique_ptr
};
```

A Few Issues

- `observer_ptr<>` lacks implicit conversion from `unique_ptr` / `shared_ptr`
  - More calls to `.get()`
- Should we be able to trivially get a non-owning pointer from an owning one?
  - Tension between trivial use of new type and possibly incorrect code

```
1 void foo(unique_ptr<int> & in);
2 void foo(observer_ptr<int> in);
3
4 auto bar = make_unique<int>(0);
5 foo(bar);
```

`.get()` considered harmful?

- Life is better with explicit ownership, so we're moving in the right direction
- `.get()` makes it really easy to get back a raw pointer
  - Moving in the wrong direction again

- Design point: make it hard to subvert ownership
  - Even if it means more typing
  - Make obvious a potential ownership issue

`.get()` considered harmful?

Which one would you scrutinize more in a code review or when looking for a suspected memory bug?

```
1    legacy_func(foo.get());

2    legacy_func(&*foo); // Is &* is a code smell?

3    legacy_func(raw_pointer_ignoring_lifetime(foo));

4    legacy_func(foo.operator->());
```

A Few Issues

- Also missing implicit construction from raw pointer

```
void newer_func(observer_ptr<int> val);

// older code
int value;
// newer_func(&value); // doesn't compile
newer_func(observer_ptr<int>(&value));
```

Implicit Construction?

- Upside of no implicit construct from raw pointer
  - `foo * old_func_returning_ownership();`
  - This also can't implicitly become an `observer_ptr`
- Can't seem to have the best of both worlds:

```
observer_ptr(T * const & rhs) : raw_(rhs) {}

// observer_ptr(T * && rhs) = delete;
// explicit observer_ptr(T * && rhs) : raw_(rhs) {}
```

Raw References

Which Variables Can Be Modified?

```
do_thing_1(foo, bar);
do_thing_2(&foo, &bar);
```

Fixing(?) Raw References

- From Google's C++ style guide:

  Prefer using return values over output parameters: they improve readability, and often provide the same or better performance. If output-only parameters are used, they should appear after input parameters.

  Parameters are either input to the function, output from the function, or both. Input parameters are usually values or `const` references, while output and input/output parameters will be pointers to non-`const`.

Fixing(?) Raw References

- From Bloomberg's BDE Style guide:

  Inputs and parameters should be passed by value for fundamental, enumerated, and pointer types, and by non-modifiable reference otherwise. An argument must be passed by pointer if its address is retained beyond the end of the function call.

  Outputs must be passed by pointer, never by modifiable reference (except where required for consistency with an externally-defined API).

Which One Could Segfault?

```
foo.bar = 7;
```

Or

```
foo->bar = 7;
```

Which One Could Segfault?

```
foo = 7;
```

Or

```
*foo = 7;
```

Which One Could See Garbage Data?

```
cout << foo.bar << '\n';
```

Or

```
cout << foo->bar << '\n';
```

Non-Local Reasoning

- The farther away I need to look for an answer, the longer it takes to comprehend code

Async Changes Understanding of Lifetime

```
void slow_accum(foo & input,
                int & output) async {
    output = 0;
    for (auto const & elem : input) {
      co_await timeout(1ms);
      output += elem.size();
    }
}
```

Thesis 4

References aren’t awesome.

They’re a language necessity, but they make the calling code harder to reason about.

Reference vs pointer gives two ways to do one thing, and almost no difference between them.

Thesis 4a

Consistent code is great for lowering cognitive load.

Passing by-pointer to indicate modifications means consistency.

For async code, passing by-pointer or by value clarifies possible lifetime issues.

Apply Google's Rules...

```
// output may not be null
void slow_accum(foo const & input,
                int *       output) async {
    *output = 0;
    for (auto const & elem : input) {
      co_await timeout(1ms);
      *output += elem.size();
    }
}
```

Apply Smart Pointer Guidelines…

```
// output may not be null. (Where's contracts?)
void slow_accum(foo const &        input,
                observer_ptr<int> output) async {
    *output = 0;
    for (auto const & elem : input) {
      co_await timeout(1ms);
      *output += elem.size();
    }
}
```

Apply BDE Guidelines…

```
// input may not be null. output may not be null.
void slow_accum(foo const *        input,
                observer_ptr<int> output) async {
    *output = 0;
    for (auto const & elem : *input) {
      co_await timeout(1ms);
      *output += elem.size();
    }
}
```

Apply Smart Pointer Guidelines…

```
// input may not be null. output may not be null.
void slow_accum(observer_ptr<foo const> input,
                observer_ptr<int>       output) async {
    *output = 0;
    for (auto const & elem : *input) {
      co_await timeout(1ms);
      *output += elem.size();
    }
}
```

We've Lost Something

- C++ reference claims "never null", essentially by contract
  - `observer_ptr<>` makes no claim
- Would an `observer_ref<>` make sense?
  - "Best" of all the worlds?
  - Non-null by contract (assert, if you write the code)
  - Uses `operator->` instead of `.` for dereference
    - Makes issues of lifetime visible
  - Decision: allow construction from a reference?
    - Pro: more like C++ reference
    - Con: can hide a mutated parameter via non-const implicitly constructed `observer_ref<>` from a variable

Non-Owning Smart Pointers

- Feels like two kinds of “by pointer” inputs
  - `out<>`, `out_opt<>`, `in_out<>` for output arguments
    - Output args are generally not null, unless we explicitly mark them optional
    - C++ doesn’t let us make a write-only type like `out<>` should be
  - `borrowed_ptr<>` and `borrowed_ref<>` for inputs
    - Non-owning pointer
    - Preserve some pointer-versus-reference differences
    - By Contract, `borrowed_ref<>` may not be nullptr

Non-Owning Smart Pointers

	`borrowed_ref`	`T &`	`borrowed_ptr`	`T *`	`out`	`in_out`	`out_opt`
null?	No	No	Maybe	Maybe	No	No	Maybe
Async lifetime	IC	IC	IC	IC	FC	FC	FC
Default construct?	No	No	Yes	Yes	No	No	No

IC = Initial call unless documented; FC = Function completion

https://godbolt.org/z/dL13t0

Take 2

```
// `input` must have a lifetime longer than this async function
void slow_accum(borrowed_ref<foo const> input,
                out<int>                output) async {
    *output = 0;
    for (auto const & elem : *input) {
      co_await timeout(1ms);
      *output += elem.size();
    }
}

int count;
co_await slow_accum(&foo_container, &count);
```

Guideline 3

- Only pass by-value to an async function
  - (Pointers and non-owning smart pointers are values)
  - Function definition should use appropriate smart pointers to encapsulate meaning
  - Possibly desirable:
    - `template<typename T> using out = observer_ptr<T>;`
  - Or, use inheritance to define a full type with appropriate guarantees
- Be explicit about lifetime if the type isn't
  - `out<>` parameters have long enough lives
  - Borrowed parameters by default are not valid past initial invocation
  - `span<>` and `string_view` are also borrowed parameters!

Migrating Code

1. Fix owning pointers first
   a. Unique, shared, or alternate owning pointers are all better than raw
   b. Our codebase had many factory functions returning raw pointers, but captured into an owning pointer, so the transition was less painful
2. Consider scary functions like `raw_pointer_ignoring_lifetime()`
   a. It makes sketchy places stand out
   b. It's easy to search for, to start fixing legacy functions that should take an `observer_ptr<>`

Theses

1. 100% correct code can be written with raw C++; types should add value
2. Explicit is better than implicit
3. These types work best when we have less code to change
4. References have issues; consistent code lowers cognitive load

- Choices for smart pointer semantics can make code easier to read
  - Make passing as args easy
  - Makes dereferences visible
  - Makes modified arguments visible

Guidelines

1. No owning raw pointers (except when required)
2. No non-owning raw pointers
3. Pass by-value to async code; be explicit about lifetime

Appendix A

Double-free is still possible with owning smart pointers.

Multi-threaded races

Spot the bug:

```
void member_func() {
    ...
    // Early release so we don't hold memory longer
    // than necessary.
    buf_.reset();
    ...
}
```

Multi-threaded races

Thread 1

```
void smart_ptr::reset() {
    void * tmp = buf_.raw_pointer;
    // scheduled out for T2


    // T1 resumes
    buf_.raw_pointer = nullptr;
    delete tmp;      // Boom!
}
```

Thread 2

```
void smart_ptr::reset() {


    void * tmp = buf_.raw_pointer;
    buf_.raw_pointer = nullptr;
    delete tmp;


}
```

Appendix B

Why we may need an owning raw pointer.

No owning raw pointers?

```
struct stack_head {
    unique_ptr<any_t> head;
    size_t            count = 0u;
};

std::atomic<stack_head> head_;

void push(unique_ptr<any_t> && item) {

    stack_head upd = head_.load();
    do {
        item->next = cur.head;      <--
        upd.head = std::move(item); <--
        upd.count = old.count + 1;
    } while (!head_.cas(cur, upd));
}
```

- So many compiler errors, because `unique_ptr` is a move-only type.
- Also creates an unexpected free, if the CAS fails

No owning raw pointers?

```
struct stack_head {
    any_t * head;
    size_t  count = 0u;
};

std::atomic<stack_head> head_;

void push(unique_ptr<any_t> && arg) {
    any_t *    item = arg.release();
    stack_head cur = head_.load();
    do {
        item->next = cur.head;
        upd.head = item;
        upd.count = old.count + 1;
    } while (!head_.cas(cur, upd));
}
```

```
~foo() {
    for (any_t *cur = head_.load().head,
         *next = nullptr; cur;
         cur = next) {
        next = cur->next;
        delete cur;
    }
}
```

- Is `delete` always bad? Or just rare?