API Structures and Techniques

Learnings from Code Review

Ben Deane / CppCon / 2025-09-16

Frontmatter

The code you will see is *real* and is being used in production projects.
It's available under BSL-1.0.

Here it's simplified for slides. Assume all the "decorators" are where you'd put them.
Any errors in translation are mine, and you may point them out.

- Atkinson Hyperlegible (`https://www.brailleinstitute.org/freefont/`)
- Berkeley Mono (`https://usgraphics.com/products/berkeley-mono`)
- modus-operandi (`https://github.com/protesilaos/modus-themes`)

Hello, I'm Ben

I used to work in games. Then I worked in finance. Now I'm a Principal Engineer at Intel working on Power Management Firmware.

I work with Michael Caisse, Luke Valenty, and an awesome team.

I strive for good code.

"Learnings from Code Review"

All code is bad, and we humans copy it.

- We don't realize it's bad.
- It works.
- We are used to it.
- We are inured to the pain points in our domains.

It's really difficult to step back from the implementation.

The Codebase Environment

- Embedded, bare metal, C++20
- 5 years ago most of the programmers were writing C
- Firmware implementation is often over-specified
- 95% IO, almost no number crunching

  - Contemporary compiler front-end
  - Use *all* of the STL! (At compile time!)

Opportunities for Abstraction

We want to get away from:

- magic numbers
- magic operations
- having to think about state

We want abstraction, in the best sense.

Start with the Small Stuff

Quite literally the small stuff... dealing with bits!

We have to deal with bits all the time.

Embedded code is full of bit masks, bit calculations, etc.

Converting sizes

You have N bytes. You need to compute
the storage size of an array of (say) `uint32_t`s.

How many times have you written something like this?

```
const auto dword_size = (N + sizeof(uint32_t) - 1) / sizeof(uint32_t);
```

Or perhaps more likely (and with less portability):

```
const auto dword_size = (N + 3) / 4;
```

Converting sizes

That kind of thing is rife, littering the code. Instead, a general solution:

```
const auto sz = sized<T>{N}.template in<U>();

// with convenience aliases:
// sizedXX -> sized<std::uintXX_t>
```

Converting sizes

That kind of thing is rife, littering the code. Instead, a general solution:

```
const auto sz = sized<T>{N}.template in<U>();

// with convenience aliases:
// sizedXX -> sized<std::uintXX_t>
```

```
const auto sz = sized8{42}.in<std::uint32_t>();
```

Converting sizes

That kind of thing is rife, littering the code. Instead, a general solution:

```
const auto sz = sized<T>{N}.template in<U>();

// with convenience aliases:
// sizedXX -> sized<std::uintXX_t>
```

```
const auto sz = sized8{42}.in<std::uint32_t>();
```

```
// if you know the size at compile-time, a UDL is super-terse
const auto sz = 42_z8->z32;
```

Bitwise operations

It's 2025, and I still see example code for interacting with hardware that does manual bit shifting, masking, etc.

This is not typically:

- easier to read than a good abstraction
- very safe (unless you make it unreadable with `static_cast`s)

Making a bit mask

Fields in registers are typically specified with *inclusive* bit ranges.

```
constexpr auto msb = 7;
constexpr auto lsb = 3;
auto msb_mask = (1 << (msb+1)) - 1;
auto lsb_mask = (1 << lsb) - 1;
auto mask = msb_mask - lsb_mask;  //0b1111'1000
```

So far, so good... but these are all `int`s.

Integer promotion

```
auto x = 'a' - 'A';
```

What is the type of `x`?

UB shifts

```
constexpr auto msb = 31;
auto msb_mask = (1 << (msb+1)) - 1;
```

Oh dear.

If you want a bit mask, make one

```
auto ct_mask = bit_mask<std::uint8_t, 7, 3>();
auto rt_mask = bit_mask<std::uint8_t>(7, 3);
```

- Get the type you want
- More readable code
- Safe in all* cases

Bit packing and unpacking

Here I typically see the same lack of abstraction.

```
std::uint16_t hi = ...;
std::uint16_t lo = ...;
std::uint32_t value = (hi << 16) | lo;
```

Bit packing and unpacking

Actually of course, I see this.

```
auto value = static_cast<std::uint32_t>(
               (static_cast<std::uint32_t>(hi) << 16)
             | static_cast<std::uint32_t>(lo));
```

How about an abstraction?

Again, no reason not to.

```
auto value = bit_pack<std::uint32_t>(hi, lo);
```

And we can unpack, too.

```
auto [hi, lo] = bit_unpack<std::uint16_t>(value);
```

Of course this works for all sizes

```
auto value = bit_pack<std::uint32_t>(0x12, 0x34, 0x56, 0x78);
auto [b3, b2, b1, b0] = bit_unpack<std::uint8_t>(0x1234'5678);
```

In fact, this use case exists

```
auto const [a, b, c] = bit_destructure<8, 24>(0x1234'5678u);
assert(a == 0x78u);   // bits [0-8)
assert(b == 0x3456u); // bits [8-24)
assert(c == 0x12u);   // bits [24-32)
```

We can split on arbitrary bit boundaries if we want.

This hole goes arbitrarily deep

Taking this further, we end up with
field and message structures overlaid on underlying data.

```
using field1 = field<"f1", std::uint32_t>::located<at{0_dw, 7_msb, 0_lsb}>;
using field2 = field<"f2", std::uint32_t>::located<at{0_dw, 15_msb, 8_lsb}>;
using message_defn = message<"msg", field1, field2>;
```

But that's another talk.

Enum bit flags

How many people do this?

Enum bit flags

How many people do this?

- define an `enum` (or `enum struct`) with power-of-two values

Enum bit flags

How many people do this?

- define an `enum` (or `enum struct`) with power-of-two values
- overload bitwise operators

Enum bit flags

How many people do this?

- define an `enum` (or `enum struct`) with power-of-two values
- overload bitwise operators
- maybe jump through a bunch of hoops (use macros!)

Enum bit flags

How many people do this?

- define an `enum` (or `enum struct`) with power-of-two values
- overload bitwise operators
- maybe jump through a bunch of hoops (use macros!)
- ...convince ourselves that we have more type safety?

Enum bit flags

How many people do this?

- define an `enum` (or `enum struct`) with power-of-two values
- overload bitwise operators
- maybe jump through a bunch of hoops (use macros!)
- ...convince ourselves that we have more type safety?

I think maybe we've been doing it wrong.
(And there are multiple talks and libraries that help us do it wrong, even.)

Have we been doing it wrong?

Is everyone OK with this?

```
0b1'0000 | 0b0'0111 == 0b1'0111
```

Have we been doing it wrong?

Is everyone OK with this?

```
16 | 7 == 23
```

Have we been doing it wrong?

Is everyone OK with this?

```
S | N == V
```

Have we been doing it wrong?

Is everyone OK with this?

```
Pidgey | Squirtle == Ekans
```

Have we been doing it wrong?

Everyone's OK with this though?

```
enum E { A = 1, B = 2, C = 4 };
E x = A | B;
```

I think we're confused

I think we've been confusing labels with numbers.
Which is very easy to do.

The register value is still a `std::uint32_t`.

What we actually want is to label the bits with human-readable names.
But let's not pretend those *are* the values.

Why are we confused?

Why have we been doing this?

Maybe just because `std::bitset` doesn't offer a good interface.

But if we want to treat a register like a bitset, let's do that.

Why not just use a bitset?

And let's use the interface we want.

```
enum struct E { A, B, C, MAX };

auto bs = bitset<E::MAX>{place_bits, E::A, E::B};
assert(bs[E::B]);
bs.set(E::C);

register_write(bs.to<std::uint32_t>());
```

Now this is better type safety.

And all we needed was a conventional `MAX`.

From bits to bytes

OK, dealing with bits is sorted for now.

Sometimes we need to deal with bytes.

Let's examine a use case.

Heterogeneous tables

I have a table (byte array? dword array?) of data. It is to be *interpreted*.

Sometimes I want to read a `uint8_t`,
sometimes a `uint16_t`,
sometimes a `uint32_t`.

Naïve code is *almost certain* to have UB here.

Heterogeneous tables

```
unsigned char *table = <hardware address>;

auto type = *table++;
std::uint32_t value{};
switch (type) {
  case THING_1: // read a uint8_t
    value = *table++;
    break;
  case THING_2: { // read a uint16_t
    auto p = reinterpret_cast<std::uint16_t*>(table);
    value = *p;
    table += 2;
    break;
  }
  // etc...
}
```

I mean, this kind of thing happens. all. the. time.

Undefined behaviour

It's *really difficult* to avoid undefined behaviour. According to P1839:

"It is *impossible* in C++ to directly access the object representation of an object (i.e. to read its underlying bytes), even for built-in types such as int."

Basically:

- there's no getting around using `memcpy` or `bit_cast`.
- the mental model of objects being backed by byte arrays is *wrong*.

How about `byterator`

Like an iterator, except designed to peek/read/write any integral* type.

```
auto i = byterator{std::begin(table)};
auto type = i.readu8();
std::uint32_t value{};
switch (type) {
 case THING_1:
   value = i.readu8();
   break;
 case THING_2:
   value = i.readu16();
   break;
}
```

* trivially copyable?

Things `byterator` can do

`peek`, `read` and `write` in 8-, 16- and 32-bit variations:

```
auto v8 = i.peeku8();    // read value without advancing
v8 = i.readu8();         // read and advance
i.writeu8(v8);           // write and advance
```

Control over the return type:

```
// read 8 bits, return it as a 16-bit value
auto val = i.read<std::uint8_t, std::uint16_t>();
```

This is particularly useful for strong typing with enums:

```
enum struct E : std::uint32_t { A, B, C };
auto val = i.read<std::uint8_t, E>();
```

Let's talk about optionality

Let's talk about `std::optional`.

What's it used for?

What's wrong with it?

```
auto x = std::optional<std::uint32_t>{};
```

`std::optional` uses space

```cpp
using X = std::optional<std::uint32_t>;
static_assert(sizeof(X) == 2 * sizeof(std::uint32_t));
```

Because of alignment, adding a `bool` in to a structure
with an integral type typically doubles its size.

But... a lot of the time, we don't use the entire bit-space of the type.
Why not put the sentinel value into the type?

A different `optional`

We want to specify a tombstone value for the type.

We can do this for a specific instance:

```
using X = optional<int, tombstone_value<-1>>;
static_assert(sizeof(X) == sizeof(int));
```

A different `optional`

Or we can use `tombstone_traits`:

```
enum struct E : std::uint8_t { /* ... values ... */ };

template <> struct tombstone_traits<E> {
    // "-1" is not a valid value
    constexpr auto operator()() const {
        return static_cast<E>(std::uint8_t{0xffu});
    }
};

using X = optional<E>;
static_assert(sizeof(X) == sizeof(E));
```

Characteristics of this `optional`

No exception use.
Accessing a disengaged optional returns the tombstone value.

`tombstone_traits` are default-provided for:

- pointer types (`nullptr`)
- floating-point types (`infinity`)

(The tombstone for floating point types is not `NaN`. Why?)

`optional` extras

Multi-argument `transform`:

```
// S is a struct with tombstone_traits that contains an integer value

auto opt1 = optional<S>{17};
auto opt2 = optional<S>{42};
auto opt_sum = transform(
    [](S const &x, S const &y) { return S{x.value + y.value}; },
    opt1, opt2);
```

From `optional` to `cached`

A primary use case for `optional` is to defer construction.

Similarly, `cached` implements the "construct-on-use" semantic.

```
auto c = cached{[] { return expensive_computation(); }};
```

The interface very is similar to `optional`:

- `has_value`
- (`explicit`) conversion to `bool`

`cached`

But - accessing a disengaged `cached` object causes it to be created.

```
auto c = cached{[] { return expensive_computation(); }};

auto x = *c; // expensive computation happens
```

And there are two ways to recompute if needed:

```
c.reset();   // c is now disengaged
// ...
auto x = *c; // c is recomputed

c.refresh(); // c is recomputed immediately
```

Lots of things are only cached once...

If you're using a `cached` object but you never `reset` or `refresh` it...

...we call that `latched`.

```
auto c = latched{[] { return expensive_computation(); }};
auto x = *c; // expensive computation happens

// no reset function, no refresh function
```

`latched` is purely for deferred computation.

Time to deal with Time

Let's talk about dealing with time.

Scenario

Your microcontroller has a 32-bit register that continually counts up in microseconds.

2^{32} µs is nearly 72 minutes.

2^{31} µs is nearly 36 minutes.

So we see "half-range" checks

```
template <unsigned_integral T>
auto half_max() -> T {
  T x{};
  T y = ~x;
  return y/2;
}
```

This code is fine (ish)...

But then someone refactors it...

So we see "half-range" checks

```
template <unsigned_integral T>
auto half_max() -> T {
  T x{};
  // T y = ~x; // "helpful refactor"
  return ~x/2;
}
```

Hmmm... oh dear.

Our old ~~friend~~ enemy integer promotion pokes its nose in.

We don't want to see this

We don't want refactorings causing accidental bugs.

Whether that is because of:

- integer promotions/type shenanigans
- signed overflow
- other errors

We want this abstracted into a type.

`rollover_t`

That type is `rollover_t`.

```
// uint8_t is very small but makes an easy example
auto x = rollover_t<std::uint8_t>{};
```

`rollover_t` supports all the "normal" arithmetic operations, mod 2^n

`rollover_t`

But what about comparisons?

This is what `rollover_t` is really for: a rolling window where half the bit-space is greater than a value, and half is less.

This behaviour is easy* to implement.

However, the comparison operators on `rollover_t` are *deleted*.

* not easy, but possible

Deleted comparisons

Imagine a 3-bit counter. In this scenario:

- `0 < 1`
- `1 < 2`
- …
- `6 < 7`
- `7 < 0`

Because the values roll over, comparisons are necessarily *non-transitive*.

So instead, we spell comparison differently: `cmp_less`.

We don't want to outlaw danger

You know what you want to do. You know what you need to do.

Sometimes you need to do dangerous things.

But not accidentally.

I want you and your code reviewers to be informed and deliberate.

But we don't want dangerous interfaces

There's another problem with naïve time interfaces.

```
struct timer {
  auto now() -> time_point_t;
  auto run_at(std::invocable auto, time_point_t) -> void;
};
```

```
auto timeout = timer.now() + 5ms;
timer.run_at(do_something, timeout);
```

But there's a problem

```
auto timeout = timer.now() + 5ms;
// timeout is 5ms in the future
// but now we get interrupted
// now we go to sleep... zzz...
// ...and we wake up some 40 minutes later
// now timeout is ~30 mins in the future
timer.run_at(do_something, timeout); // wait a long time
```

Any interface that allows us to compute a timeout has this risk.

And it's incredibly difficult to reason about.

A Different Interface

Dealing with time points is inherently dangerous,
so let's deal exclusively with durations.

```
struct timer {
  auto run_after(std::invocable auto, duration_t) -> void;
};
```

...which is what we wanted to express anyway.

Let's not be wrangling timeouts manually.

A Different Interface

Or better, a senders-based interface:

```
auto sndr = time_scheduler{5ms}.schedule()
          | then(do_something);
```

Or:

```
auto sndr = time_scheduler{}.schedule()
          | then(do_something)
          | periodic(5ms);
```

Control over computing timeouts is "under the hood" where danger can be limited.

We want *one place* to reason about interrupt safety.

ThSarefad ety

Let's talk about thread safety.

Or, in the embedded domain, interrupt safety.

We have one tool

We can turn interrupts off and on again.

And this is cheap to do.

So we basically have a critical section ability.

`call_in_critical_section`

It's a basic abstraction, but still useful.

```
auto call_in_critical_section(std::invocable auto f);
```

call_in_critical_section

In fact, it's slightly more... we often want to call a function only when a predicate is (or becomes) true.

```
auto call_in_critical_section(std::invocable auto f,
                              std::predicate auto ...preds) {
  while (true) {
    auto lock = interrupt_disable_raii_type{};
    if ((... and preds())) {
      return f();
    }
  }
}
```

Bridging the platform gap

OK, so this works on platform...

...but how do we test it?

If this is really raising the level of abstraction,
we should be able to run desktop tests with it.

The problem

Almost all of our work involves handling concurrency.

The mechanism of concurrency on hardware is interrupts.

This is *fundamentally different* from the mechanism(s) available on desktop tests.

The problem is: how to test things in a way that works.

Low-level concurrency primitives

We basically have one primitive:

- a "global mutex" or critical section: we can turn off interrupts

There is no parallelism, but there is concurrency.

We don't have `std::mutex`. Or any other "standard" constructs.
We have this critical section function & RAII type.

Problem: interference

In one piece of code...

```
{
  call_in_critical_section(use_shared_data_a);
}
```

In another piece of code...

```
{
  call_in_critical_section(use_shared_data_b);
}
```

Two unrelated pieces of code, two unrelated pieces of data.
But still the "lock" is global and contending.

Problem: brittle correctness 1

In one piece of code...

```
{
  // overlook use of critical_section
  accidentally_use_shared_data();
}
```

In another piece of code...

```
{
  // overlook use of critical_section
  accidentally_use_shared_data();
}
```

And this can succeed accidentally because of interrupt priorities and the lack of parallelism (today).

Problem: brittle correctness 2

Implementation details of `critical_section` leak.

```
auto one_function() {
  call_in_critical_section(another_function);
}

auto another_function() {
  call_in_critical_section(do_something);
}
```

Because turning off interrupts is like a global *recursive* mutex.

This is incompatible with desktop testing

We *could* implement `critical_section` with a single, global, recursive mutex.
But that wouldn't be much good.

What we really want is:

- an interface that can be used identically on platform and on desktop
- whose implementation can be selected appropriately
- that forces reasoning about concurrency and won't accidentally succeed

Injecting concurrency

`call_in_critical_section` is a global, one-function API:
we'll use the Global API Injection Pattern.

```
template <typename...> inline auto injected_policy = standard_policy{};

template <typename Uniq = decltype([]{}), typename... DummyArgs,
          std::invocable F>
auto call_in_critical_section(F &&f) -> decltype(std::forward<F>(f)()) {
    auto &p = injected_policy<DummyArgs...>;
    return p.template call_in_critical_section<Uniq>(std::forward<F>(f));
}
```

Injecting concurrency

The `Uniq` template parameter is different at every call site unless the caller passes it.

```
struct my_queue {
  struct mutex;

  auto push(int v) {
    call_in_critical_section<mutex>(...);
  }

  auto pop() {
    call_in_critical_section<mutex>(...);
  }
};
```

This means callers use "specific mutexes" to protect *their* data.

The desktop (test) policy

The default for tests just uses a `std::mutex`.

```
struct standard_policy {
  template <typename> static inline std::mutex m{};

  template <typename Uniq, std::invocable F>
  static auto call_in_critical_section(F &&f)
      -> decltype(std::forward<F>(f)()) {
    std::lock_guard l{m<Uniq>};
    return std::forward<F>(f)();
  }
};
```

The platform policy

The platform policy turns interrupts on/off around the call.

```
struct platform_policy {
  template <typename Uniq, std::invocable F>
  static auto call_in_critical_section(F &&f)
      -> decltype(std::forward<F>(f)()) {
    interrupt_disable_raii_type int_dis{};
    return std::forward<F>(f)();
  }
};
```

(We saw this before.)

This works pretty well

Desktop tests can now better ensure correctness.

- No more lazy works-by-accident code.
- No relying on recursive locking working.
- We can (and do) run TSan on the real code!

Being able to run TSan is a massive step up.

Atomics

Turning off interrupts is not the only way
to achieve interrupt-safe data.

We also have atomic variables.

But there's a problem...

The problem with `std::atomic`

What do you understand by this?

```
std::atomic<int> x{42};
++x;
```

The problem with `std::atomic`

What do you understand by this?

```
std::atomic<int> x{42};
++x;
```

An atomic instruction?

The problem with `std::atomic`

`std::atomic` really isn't well-built for embedded devices.

Many embedded devices do have atomic capability,
but it may be very limited.

`std::atomic` suffers from a confusion of interface and implementation.
(If not, ask yourself why does `is_lock_free` exist?)

Problems with `std::atomic` (1)

The platform supports atomic exchange only on 32-bit alignment.

```
std::atomic<bool> b{};

if (not b.exchange(true)) {
  // I got here first!
}
```

Oh no, I forgot to align the variable!

I get a processor exception.

Problems with `std::atomic` (2)

Exchange has more constraints.

```
alignas(4) std::atomic<bool> b{};
bool something_else{};

if (not b.exchange(true)) {
  // I got here first!
}
```

Now it's aligned - no exception.

But oh no, an `atomic<bool>` is only 8 bits,
and the instruction works on 32 bits, clobbering the next three bytes.

Problems with `std::atomic` (3)

The platform supports *only* atomic exchange.

```
std::atomic<int> i{};

if (i++ == 0) {
  // I got here first!
}
```

There's no single instruction that does `fetch_add`;
instead we call some assembly function provided by the toolchain.

What I want

`atomic` has a meaning for the *interface*, not the *implementation*.

I don't care if it uses "atomic instructions".

I care that it uses the best (most efficient) mechanism
that's available on the platform for any particular operation.

So that's what I have

Once again, using the Global API Injection Pattern.

```
template <typename...> inline auto atomic_policy = standard_policy{};
```

```
template <typename... DummyArgs, typename T>
[[nodiscard]] auto atomic_exchange(T &t, T value) -> T {
    auto &p = atomic_policy<DummyArgs...>;
    return p.exchange(t, value);
}
```

The standard policy uses standard `atomic` functions (intrinsics).

A more fitting `atomic<T>`

Because `atomic` just means how we access the variable.

```
template <typename T> class atomic {
  auto exchange(T t) -> T {
    return atomic_exchange(value, t);
  }

  T value;
};
```

The platform policy uses `exchange`, but otherwise uses an interrupt lock.

Atomic type overrides

I can also inject type overrides for `atomic`.

```
template <typename T> struct atomic_type {
    using type = T;
};

template <> struct atomic_type<bool> {
    using type = std::uint32_t;
};
```

This makes using `atomic` safer.

No need to remember to align things for `exchange`.

Well, now we have atomic...

You know what would be really useful?

Well, now we have atomic...

You know what would be really useful?

`atomic_bitset`

`atomic_bitset`

`bitset` + `atomic`

This one construct would improve a ton of code that currently does janky things tracking completions, or similar.

Why `atomic_bitset` and not `atomic<bitset>`?

A few reasons:

- simpler atomic type customization
- some operations are removed

But overall, an `atomic_bitset` looks like a `bitset`.

```
enum struct Bits { ZERO, ONE, TWO, THREE, MAX };
auto bs = atomic_bitset<Bits::MAX>{all_bits};
bs.set(Bits::ZERO);
bs.reset(Bits::ZERO);
bs.flip(Bits::ZERO);
auto bit_zero = bs[Bits::ZERO];
```

Some limitations

`atomic_bitset` has some limitations because of its atomic nature.

- no binary operations (semantics require higher-level locking)
- no shift operations (not part of `std::atomic` yet/no instructions)
- underlying must be a single element (not an array)
- no iteration (would mean doing unbounded operations under locks)

(We have to somewhat acknowledge that people still confuse implementation and interface when it comes to `atomic`.)

Logging (just a taste)

I don't have time to cover logging: it's a whole talk in itself.

But here's a taste.

```
INFO("Hello {}, {} is an {}", "CppCon", 42, int);
```

What is C++ Good for?

"C++ is best suited for applications demanding high performance and efficiency, such as game engines, finance, operating systems, and embedded systems."

If you ask this question, the top answers are all performance, performance, performance.

What is C++ Best at?

Raw performance? Not really.

C++ doesn't have a monopoly on performance.

Top-end performance is *always* finicky and hardware-sensitive.
Always requires attention to details outside of the language.

Any (compiled) language can get within about 20% of FORTRAN,
if written as if it were FORTRAN.

What is C++ Best at? IMO

It's a combination. And the whole is more than the sum of the parts.

- layout control and low-level access (interfacing with hardware)
- metaprogramming (writing code at compile time so that runtime is efficient)
- abstraction & composition (building up the right structure/affordances)

All of these things come together to make good C++.

Closing thoughts

To make code better, think like a library developer.
We are always writing libraries.

Step away from the problem. Find a fresh perspective.
"We've always done it that way" isn't a good argument.

Relentless decomposition and abstraction are good things in design.
(You can choose the implementation abstraction level separately.)

https://github.com/intel/compile-time-init-build
https://github.com/intel/cpp-std-extensions
https://github.com/intel/cpp-baremetal-concurrency