Declarative Style Evolved

Declarative Structure

Ben Deane / C++Now / 2025-04-30

Frontmatter

No AI/LLM was used in the creation of this talk.

Code is simplified for slides; may have some errors in translation.
Assume everything like `[[nodiscard]]` and `constexpr` exists where you'd put it.

- Body font: Atkinson Hyperlegible Next
  `https://www.brailleinstitute.org/freefont/`
- Code font: Berkeley Mono
  `https://usgraphics.com/products/berkeley-mono`
- Code theme: 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.

Why Declarative Structure?

Because we want ΔS to be negative.

We can't win of course. But we can put up a fight.

Declarative style & structure is the best way I've found to approach that.

This talk is...

...somewhat about declarative structure.
But not *only*.

Also about structures that work well *with* a declarative style.

And in general, about effective ways to decouple *interfaces* from *implementations* to enable efficient declarative APIs.

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%, if written appropriately.

What is C++ Best at? IMO

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

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)

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)

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)

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++.

Taking Best Advantage of This

Take (you decide the temperature):

If I'm writing C++ and *not* using these tools to the fullest,
I'm not getting the most out of C++.

Defending Against Myself

"...our intellectual powers are rather geared to master static relations and ... our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."

– Edsger Dijkstra
(EWD 215 - A Case Against the GOTO Statement)

https://www.cs.utexas.edu/~EWD/transcriptions/EWD02xx/EWD215.html

I Have Cultivated an Allergy

I'm mildly allergic to statements. Especially:

- selection statements (`if`, `switch`)
- iteration statements (`for`, `while`, `do`)
- jump statements (`break`, `continue`, `goto`)

GOTO is not the only thing I consider harmful.

Selection Statements: Two Places Only

First, at the "leaf" level: inside a class that embodies choice.

```
template <typename NodeList>
auto topological_sort(auto& graph) -> std::optional<NodeList>;
```

Selection Statements: Two Places Only

Second, at the top level: a configuration choice.

```
if (config.log_filename) {
  // use file output
} else {
  // use std::cout
}
```

Decomposition & Recomposition

The real goal here is malleability.
Things change.

We want the ability to switch things around:
to *break things down* to component parts and *recombine them* in new ways.

To do this, we need good, clean interfaces.

And we need each thing tested and verified separately.

The Code Base(s) I Work In

Have the following features:

- heavy use of strong, named types
- composition-oriented
- metaprogramming (type- & value-based)
- highly constrained
- >95% IO, <5% maths

I spend a lot of my time observing how people solve problems,
and building up abstractions to help.

To Pre-empt the (Boring) Question

Q. How are compile times?
A. *Acceptable*.

Unit tests compile and run in human time (seconds);
full builds take ~20 minutes.

I want a productive, iterative workflow;
if there are compilation time regressions, we just fix them.

Testing

Declarative implies compositional.

Does this make testing easier?

What does it mean for tests to be declarative?
Is this desirable?

Testing at the Right Level

Unit tests test *exactly* and *only* what is desired.

Binding more behaviour than necessary means extra brittleness.

Testing in isolation requires relentless decoupling.

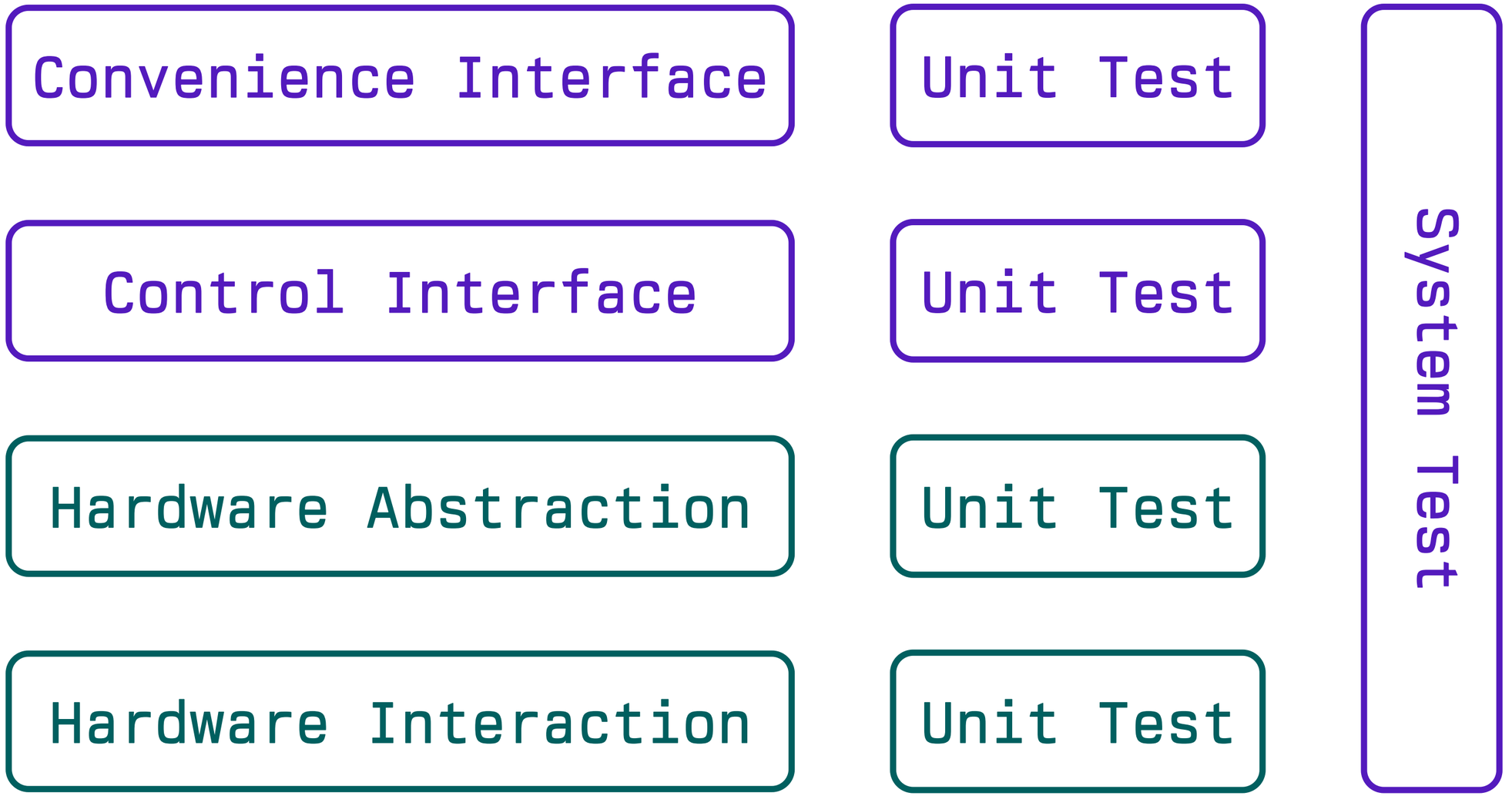
Often we are victims of too-easy testing at the wrong level.

If it's Easy to Test at the Wrong Level

- Things that should be unit tested are instead bound in system tests.
- Tests become brittle because they bind too much.
- Tests pull in more dependencies than they need to.
- Tests are slower to run than they need to be.
- CI time grows and desktop testing becomes more painful.
- API structure suffers.
- Problems are hard to trace back to sources.

Whatever is *easy* to do, people *will* do. *A lot*.

Test Levels


Convenience Interface
Unit Test
Control Interface
Unit Test
Hardware Abstraction
Unit Test
Hardware Interaction
Unit Test
System Test

Testing Should Be Procedurally Easy

Not only in C++, but in the build system too.
It should be this easy (or *easier*):

```
add_unit_test(my_test CATCH2
  FILES my_test.cpp
  LIBRARIES my_lib)
```

And with that, you get automatically:

- test binary with ctest integration (of course)
- all the warnings the project uses
- sanitizers: ASan, UBSan, TSan, ...
- valgrind, coverage, etc, etc

Testing Should Be Procedurally Easy

All kinds of tests should be just as easy to declare.

- fuzz tests
- cucumber/gherkin tests
- mutation tests
- compilation failure tests
- property tests

You could even support multiple unit test frameworks.

Quality Should Be Procedurally Easy

And as with tests, so with the code in general.
It should be automatic and axiomatic to get:

- linting (clang-tidy, black/mypy/etc, ...)
- formatting (clang-format, cmake-format, black/etc, ...)
- coverage

And CI runs with multiple compilers & versions.

Make It Easy To Do the Right Thing

Global variables are a fact of life.
Especially in embedded work.

Tests for such systems tend to be loaded with setup/teardown boilerplate.

Make it *hard* to do the *lazy* thing.

- tests are random by default
- turning off randomness gives a noisy warning

Make It Noisy To Do the Wrong Thing

You can mark a test with `NORANDOM`, but I'm not going to let you forget it.

```
message(WARNING
  "${name} is set to NORANDOM: \
  unrandomized tests are not best practice. \
  You did a bad thing, and you should feel bad. \
  You have incurred the wrath of Jason Turner. \
  ಠ_ಠ \
  But your sins will be absolved if you remove NORANDOM.")
```

Make The Desktop Workflow Efficient

It should be easy to wrangle build targets.

A one-line declaration means you get:

- a conventionally-named target to *build*
- a conventionally-named target to *run*
- a conventionally-named target for *quality checks*

As well as the correct situation of that target with respect to the others.
i.e. roll-up targets for building, running, linting, etc.
Also make it easy to run top-level targets *for changes only*.

Corollary: Use the Command Line, Simply

CI uses the command line.

The distance between CI and desktop development should be *as small as possible*.

If things are easy on the command line,
they ought to be easy for <IDE of choice>.

Know how to configure your IDE/editor!

Using Mocks

In my opinion, mocks are usually a code smell.

There, I said it.

Mocks Smell Bad Most of the Time

They encourage (do nothing to dissuade) large interfaces.

```
class MockCallback {
  public:
    MOCK_METHOD(void, init, ());
    MOCK_METHOD(void, init,
                (std::size_t irq_number, std::size_t priorityLevel));
    MOCK_METHOD(void, run, (std::size_t irq_number));
    MOCK_METHOD(void, status, (std::size_t irq_number));

    MOCK_METHOD(void, write, (int fieldIndex, uint32_t value));
    MOCK_METHOD(uint32_t, read, (int fieldIndex));
};
```

We even have frameworks that can auto-generate mocks.

And now we have LLMs that can generate oodles of boilerplate.

Mocks Smell Bad Most of the Time

They lead to excessive boilerplate.

Median test length (with mocks): 22 lines
Median test length (refactored): 6 lines

Mocks Smell Bad Most of the Time

Most frameworks enshrine poor OO practices.

"Want to test something?
Make an Abstract Base Class so that it can be mocked.
This Is The Way."

(this_is_fine.jpg)

Mocks Smell Bad Most of the Time

They introduce temporal confusion.

```
constexpr auto &manager = BasicBuilder::manager;

EXPECT_READ(packet_avail_en_field_t).WillRepeatedly(Return(1));
EXPECT_READ(rsp_avail_en_field_t).WillRepeatedly(Return(0));

EXPECT_READ(packet_avail_sts_field_t).WillOnce(Return(1));
EXPECT_WRITE(packet_avail_sts_field_t, 0);

EXPECT_CALL(callback, run(33)).Times(1);

manager.run<33>();
```

When we reason about tests, don't we think "arrange; act; assert"?
Not "arrange-assert; act".

Mocks Smell Bad Most of the Time

What it might look like instead:

```
enable_field_t<33'1>::value = true;
status_field_t<33'1>::value = true;

auto m = interrupt::manager<config_shared, test_nexus>{};
m.run<33_irq>();

CHECK(not status_field_t<33'1>::value);
CHECK(flow_run<flow_33_1>);
CHECK(not flow_run<flow_33_2>);
```

Are Mocks that Bad?

"But Ben, you're using mocks wrongly. I can show you how to use them right."

OK sure, I'm "holding it wrong".

And so is practically every team I've worked on.
So at some point we should consider that maybe they're just
a poor solution for most tests.

Are Mocks that Bad?

"But Ben, you're using mocks wrongly. I can show you how to use them right."

OK sure, I'm "holding it wrong".

And so is practically every team I've worked on.
So at some point we should consider that maybe they're just
a poor solution for most tests.

(Is this a metaphor for C++ safety?)

The Somewhat Surprising Point That Emerges

Tests are the one thing that should be imperative.

They should be so simple that imperative is natural.

Arrange; Act; Assert.
Just a few lines.

It's declarative *structure* that makes this possible.

An Evolution of Logging

A case study in growing a system.

A discovery of declarative techniques.

https://github.com/intel/compile-time-init-build

How It Started

I mean, it's not 1995. We had a modern* logging solution.

```
 1: template <typename FN, typename LN, typename Msg>
 2: void log(FN filename, LN lineNumber, Level level, Msg msg) {
 3:   auto formattedMsg = msg.args.apply([&](auto... args){
 4:     return fmt::format(msg.str.value, args...);
 5:   });
 6:
 7:   output_line(filename, lineNumber, level, formattedMsg);
 8: }
 9:
10: #define LOG(LEVEL, MSG, ...) log(__FILE__, __LINE__, LEVEL, \
11:                                  FormatHelper{MSG ## _sc}.f(__VA_ARGS__))
12:
13: #define INFO(...) LOG(log_level::INFO, __VA_ARGS__)
14: // similarly for other levels: TRACE, WARN, ERROR, etc
```

*contemporary

How It Started

```
template<typename CharT, CharT... chars>
struct string_constant {
  constexpr static CharT storage[sizeof...(chars)]{chars...};
  constexpr static std::string_view value{storage, sizeof...(chars)};
};

// Name is a string_constant type
TRACE("flow.start({})", Name{});
```

- compile-time formatting where we can
- runtime formatting otherwise

And a good time was had.

Then One Day

Luke: "Let's make logging better! Let's accelerate it! Let's use the MIPI SyS-T spec!"

So we do some work: now we have 2 logging backends (`fmt` and `mipi`).

```
#ifdef CIB_LOG_MIPI_CATALOG
#include <log/catalog/mipi_encoder.hpp>
#else
#include <log/fmt/logger.hpp>
#endif
```

And they each have a `log` function.

OK, How Do We Test That?

The binary back end ends up at a function:

```
LogDestination::log_buf(data, size);
```

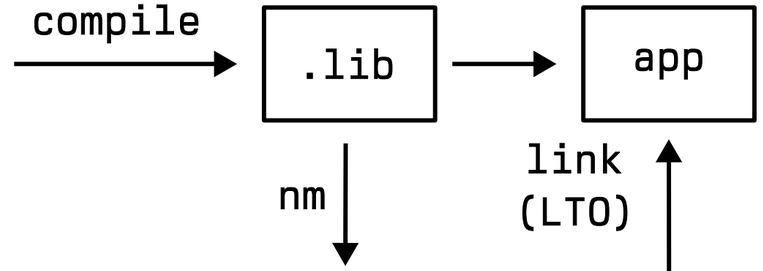
So we can write a test destination:

```
template <log_level Level, auto... ExpectedArgs>
struct test_log_buf_destination {
    template <typename... Args>
    static auto log_buf(std::uint32_t *buf, std::uint32_t size) {
        REQUIRE(size == 1 + sizeof...(ExpectedArgs));
        CHECK(*buf++ == expected_header(Level));
        CHECK(((ExpectedArgs == *buf++) and ...));
    }
};
```

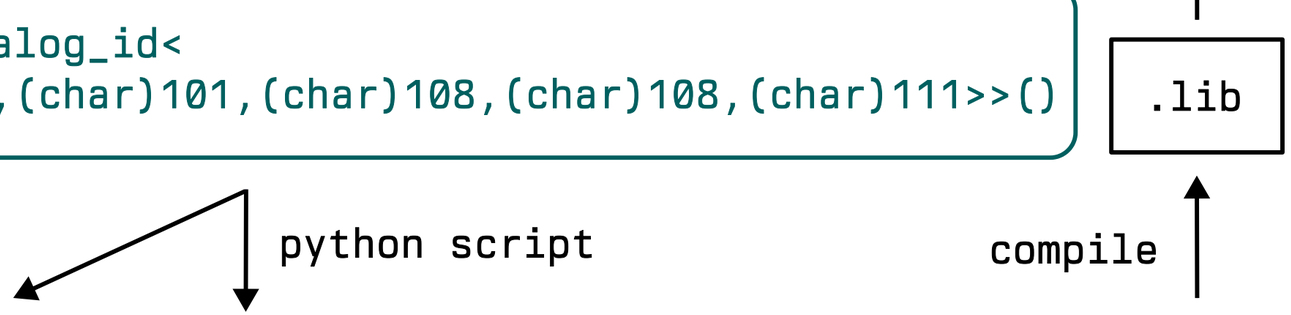
How the Guts of Logging Work

```
LOG("Hello");
// type: sc<char,'H','e','l','l','o'>

template <typename>
auto catalog_id() -> std::uint32_t;
```




```
U unsigned int catalog_id<
  sc<char,(char)72,(char)101,(char)108,(char)108,(char)111>>()
```




```
{
  "message": "Hello",
  "id": 1337
}
```

```
template <>
auto catalog_id<sc<char,72,101,108,108,111>>()
  -> std::uint32_t {
  return 1337;
}
```

Well, now

We have two logging backends. How to declare which to use?

```
#ifdef CIB_LOG_MIPI_CATALOG
#include <log/catalog/mipi_encoder.hpp>
#else
#include <log/fmt/logger.hpp>
#endif
```

Well, now

We have two logging backends. How to declare which to use?

```
#ifdef CIB_LOG_MIPI_CATALOG
#include <log/catalog/mipi_encoder.hpp>
#else
#include <log/fmt/logger.hpp>
#endif
```

(Yeah, I don't like macros much either.
But just wait until you see what's coming...)

How To Configure Logging

conditional compilation	💔
link different implementations	💔
build system shenanigans	💔
runtime polymorphism	💔
compile-time polymorphism	❤️

I've tried all these methods... I didn't like most of them.

We'll Use the Global API Injection Pattern

This thing is just the ticket for things like logging.

It lets us keep things inside C++.

Logger Interface

```
struct null_logger {
  auto log(auto&&...) const noexcept -> void {}
};

template <typename...>
inline auto log_config = null_logger{};

template <typename... DummyArgs, typename... Args>
auto log(Args &&...args) -> void {
  log_config<DummyArgs...>.log(std::forward<Args>(args)...);
}
```

`DummyArgs...` will always be an *empty* pack, but it's nevertheless used to select the specialization of `log_config`.

Brief Aside: Using Packs

There are a few ways to orchestrate packs.

```
// after an explicit arg pack
template <typename... Xs, typename... Ts>
auto f();
// f<void> => Xs... = [void], Ts... == []

// in between two deduced arguments
template <typename X, typename... Ts, typename Y>
auto f(X, Y);
// f(1,2) => X == int, Ts... == [], Y == int
```

Both of these will result in empty packs.

And the empty packs can still be used to select specializations.

Using Packs

To separate explicit and deduced parameters:

```
template <typename... Ts, typename... Xs>
auto f(Xs...);
// f<void>(1) => Ts... == [void], Xs... == [int]
```

And we could also prepend explicit parameters here.

So we have a few options. </aside>

Logger Implementation

```
// one actual impl logs to stdout
struct fmt_logger {
  template <typename... Args>
  auto log(Args &&...args) const -> void {
    fmt::print(std::forward<Args>(args)...);
  }
};

// another writes to hardware
struct binary_logger {
  template <typename... Args>
  auto log(Args &&...args) const -> void {
    // in its own inimitable way
  }
};
```

We implement loggers in the usual way.

Logger Selection

```
template <> inline auto log_config<> = fmt_logger{};
```

Here's the crux.

Every TU *must* see the *same specialization* to avoid ODR violations.

But that's OK: this is a global API. We only want one logger.*

Naturally production will use the hardware logger, and tests can use the fmt logger.

Testing Logs

Actually, most tests don't care about logging.
So there's no need for them to specialize the variable template.

```
TEST_CASE("callback matches message", "[callback]") {
  constexpr auto id_match = msg::equal_to<id_field, 0x80>;
  auto callback = msg::callback<"cb", msg_defn>(id_match, [] {});
  CHECK(callback.is_match(std::array{0x8000ba11u, 0x0042d00du}));
}
```

They run quietly; the null logger is used by default.

Testing Logs

Tests that *do* care can inject a test logger and check the log buffer.

```
std::string log_buffer{};

template <>
inline auto log_config<> =
    logging::fmt::config{std::back_inserter(log_buffer)};

TEST_CASE("callback logs mismatch", "[callback]") {
  constexpr auto id_match = msg::equal_to<id_field, 0x80>;
  auto callback = msg::callback<"cb", msg_defn>(id_match, [] {});

  callback.log_mismatch(std::array{0x8100ba11u, 0x0042d00du});
  CAPTURE(log_buffer);
  CHECK(log_buffer.contains("cb - F:(id (0x81) == 0x80)"));
}
```

Life is Good Again

We declare a specialization for whichever logger we want.

We use the various `LOG` macros for different levels.

Life is good, we carry on...

No Rest for the Wicked

Day 47. "Oh, by the way...

...we need secure logs as well."

Life lesson: premature pluralization is seldom a mistake.

Adding secure logs

This is no problem for the injection pattern;
we can flavour the specializations with a type.

```
template <typename Flavor, typename... DummyArgs, typename... Args>
auto log(Args &&...args) -> void {
  log_config<Flavor, DummyArgs...>.log(std::forward<Args>(args)...);
}
```

```
template <>
inline auto log_config<regular_flavor_t> = ...;

template <>
inline auto log_config<secure_flavor_t> = ...;
```

Adding secure logs

The calling code is getting worse, though...

```
#define CIB_INFO(...) \
    CIB_LOG(default_flavor_t, level::INFO, __VA_ARGS__)

#define CIB_SECURE_INFO(...) \
    CIB_LOG(secure_flavor_t, level::INFO, __VA_ARGS__)
```

The macros are undergoing a combinatorial increase. Eeehhh...

I guess we can convince ourselves (for now)
that we probably want these for convenience anyway.

Still No Rest

Day 73. "So about the MIPI SyS-T spec...

...it allows for log modules. Yeah, we want those."

What now?

This is getting out of hand.

Am I going to add `CIB_SECURE_INFO_WITH_MODULE` and all its brethren?

Aaaaaaaaaaahhh.... no thanks.

Am I going to squeeze in more positional parameters to the logging code?

No. I have a better idea. Let's be declarative.

Let's Declare the Module

A module mostly corresponds to a scope.
Log module structure follows code structure.

- namespace
- class
- function

So let's declare the module at the scope we're in,
and let the logging code pick it up.

Let's Declare the Module

Define a default, for use if no other is declared.

```
template <ct_string S> struct module_id_t {};
using cib_log_module_id_t = module_id_t<"default">;
```

Define a way to declare an overriding module in whichever scope we're in.

```
#define CIB_LOG_MODULE(S) \
  using cib_log_module_id_t = [[maybe_unused]] module_id_t<S>;
```

And then have the existing macros use `cib_log_module_id_t`.

Declaring the Module

Logs from the same place in the code pretty much want to use the same module.

```
namespace wibble {
  CIB_LOG_MODULE("wibble");

  auto wobble() {
    CIB_INFO("normal: wobble");
  }
  auto wubble() {
    CIB_ERROR("emergency: wubble");
  }
}
```

Once More, Life's Good

We don't have a lot of code gumming up log sites.

We have declarative log modules.

We have good test structure, easy injection of implementation.

We do have some constraints:
namely that "log metadata" (level, module) is known at compile time.
But that's fine for us.

Now, Another Request

The MIPI SyS-T back end has several different message formats it can send.

Different message formats have different tradeoffs.

- A `short32` message is only 32 bits, but doesn't include any metadata.
- A `catalog` message is bigger, but includes metadata.

Up until now, the back end made a decision about which message to send based on the amount of data. (Optimising for message size.)

But we want to control this sometimes.

Restructuring the Binary Logger

The logger does the following:

- Anything that can be formatted at compile time, is.
- The remaining format string is converted to an ID.
- The back end sends a packet with the ID and the remaining (runtime) format arguments.

```
int x = 42;
LOG("Hello {} {}", std::bool_constant<true>, x);
```

The Binary Logger

```
LOG("Hello {} {}", std::bool_constant<true>{}, 42);
```

```
ID 1337: "Hello true {}"
```

```
int: 42
```

```
packet :: {
  header :: uint32_t
  ID :: uint32_t
  data :: int[1]
}
```

The Binary Logger

The binary logger does three separate things:

- build packets
- assign IDs
- send packets

So let's make those things separate!

We Already Use a Declarative Module ID

Let's extend this to a declarative environment.

An environment is a compile-time key-value store that supports `query`.

```
template <auto Query, auto Value> struct property {
    using query_t = std::remove_cvref_t<decltype(Query)>;

    [[nodiscard]] consteval static auto query(query_t) noexcept {
        return Value;
    }
};
```

The simplest form of which is a *property* (one key-value pair).

Environments Self-Compose

Put together several environments, and you get... an environment.

```
template <typename... Envs> struct env {
  template <valid_query_over<Envs...> Q>
  consteval static auto query(Q) {
    using I = boost::mp11::mp_find_if_q<boost::mp11::mp_list<Envs...>,
                                        has_query<Q>>;
    using E = nth_t<I::value, Envs...>;
    return Q{}(E{});
  }
};
```

So *property* really exists for simplicity of interface.

And Queries Look Like This...

```
constexpr inline struct get_module_t {
  template <typename T> requires true // more constrained
  consteval auto operator()(T &&t) const
    -> decltype(std::forward<T>(t).query(*this)) {
    return std::forward<T>(t).query(*this);
  }

  consteval auto operator()(auto &&) const { // fallback
    return ct_string{"default"};
  }
} get_module;
```

Some of them don't need fallbacks: it depends whether we want to provide a default.

Populating the Log Environment

The default environment is empty.
We can populate values by extending it (with a bit of metaprogramming).

```
using cib_log_env_t = env<>;

using new_env = extend_env_t<cib_log_env_t, get_module, "module">;
```

Module declarations now work by extending the scoped environment.

```
CIB_LOG_MODULE("wibble");
// using cib_log_env_t = extend_env_t<cib_log_env_t, get_module, "wibble">;
```

Environments for Everything

We can use the logging environment to customize everything.

It can be viewed as an alternative, declarative method of passing arguments.

- log level (severity)
- log flavour (default/secure)
- log module
- packet builder (for binary backend)
- other...

There's no runtime cost to putting things into the environment,
so it is decoupled from the selected backend.

An Hourglass-Style Pattern

This environment use turns out to be a classic "hourglass-style" pattern.

```
// top of the hourglass: client code
constexpr inline struct get_builder_t {
  ...
} get_builder;

CIB_LOG_ENV(get_builder, packet_builder{});

CIB_INFO("It's big, it's heavy, it's wood!");
```

The client code can define queries (and their values)
that the library code doesn't need to know about.

An Hourglass-Style Pattern

The library code uses a builder and a writer that are customized.

```
// middle of the hourglass: library code
template <typename Writer> struct log_handler {
  template <typename Env, /* ... */>
  auto log(/* args... */) {
    auto builder = get_builder(Env{});
    // ...
    write(builder.build(/* ... */);
  }
};
```

This code really just deals with log/module ID assignment.
Otherwise, it uses the packet builder & writer from the client.

An Hourglass-Style Pattern

The client also provides the log packet builder and writer.

```
// bottom of the hourglass: client code
struct packet_builder {
  auto build(/* args... */) { ... }
};

struct log_writer {
  auto write(/* data... */) { ... }
};
```

Declarative Customization

Client code can customize any part of this:

- custom queries
- custom writer
- custom packet builder
- whole custom logger if you like

This is nice, but it's also particularly important for logging.

We want logging to be cheap and ubiquitous,
and it's particularly sensitive to inlining decisions.

The Binary Logger Evolved

The binary logger does three separate things:

- build packets (customizable)
- assign IDs (the essence of what the library provides)
- send packets (customizable)

And they're now separated.

Of course there are suitable defaults in the library:
out of the box, it provides MIPI SyS-T logging.

Dealing with the Unexpected

Arthur: "You know, it's at times like this, when I'm trapped in a Vogon airlock with a man from Betelgeuse, and about to die of asphyxiation in deep space that I really wish I'd listened to what my mother told me when I was young."

Ford: "Why, what did she tell you?"

Arthur: "I don't know, I didn't listen."

One of These Things...

...is not like the others.

- `CIB_TRACE`
- `CIB_INFO`
- `CIB_WARN`
- `CIB_ERROR`
- `CIB_FATAL` (hint: it's this one)

Sometimes, things go *irretrievably* wrong...

CIB_FATAL == log + panic

```
// inside CIB_FATAL...
// s :: { compile-time string, runtime arg tuple }

CIB_LOG_ENV(get_level, level::FATAL);
log<cib_log_env_t>(__FILE__, __LINE__, s); // log message

apply([](auto &&...args) {
  panic<s.str>(FWD(args)...);                // panic
}, s.args);
```

We Already Have a Pattern For This

Global API?

Especially a small (one function!) API?

Need to test it?

Sounds like a job for the Global API Injection Pattern.

`panic` is Also Injected Declaratively

```
struct default_panic_handler {
  template <ct_string>
  static auto panic(auto &&...) noexcept -> void {}
};

template <typename...> inline auto panic_handler = default_panic_handler{};

template <ct_string S, typename... DummyArgs, typename... Args>
auto panic(Args &&...args) -> void {
    panic_handler<DummyArgs...>.template panic<S>(std::forward<Args>(args)...);
}
```

Once again, `DummyArgs...` will always be an empty pack, used to select the specialization of `panic_handler`.

Logging: The Final Flourishes

There are just a couple more nice-to-haves that make logging call sites better.

Logging: The Final Flourishes

There are just a couple more nice-to-haves that make logging call sites better.

(Remember the interesting macros I mentioned earlier?)

Logging: Compile-time Values

`constexpr` values have a nasty habit of becoming runtime values when you pass them to functions...

```
// This string is entirely formatted at compile-time
CIB_INFO("The answer is: {}", CX_VALUE(42));

// This string has a runtime format argument
CIB_INFO("What do you get if you multiply six by nine? {}", 42);
```

`CX_VALUE` prevents that.

(And solves the problem that some perfectly usable `constexpr` values can't be passed as template parameters.)

Logging: Type Names

Also, it's convenient to treat types and values the same.

```
constexpr auto you = 6;
CIB_INFO("You are number {}", CX_VALUE(you));
CIB_ERROR("I am not a {}, I am a free man!", CX_VALUE(decltype(you)));
```

Where logging a type means logging the type's name.

Let's Change Gears Now

Let's talk about composing asynchronous functions.

Asynchronous Functions

AKA senders.

Asynchronous Functions

AKA senders.

(I see we're still bad at naming things)

Senders Turn This...

```
send_message();

recv_callback();

process_response();
```

Code that deals with concurrency is intermixed with business logic.

These three functions are in physically different places in the code.

They marshal data that's in different places.

...Into This

```
send_message()
| recv_callback()
| process_response();
```

Code that deals with concurrency is gone. The business logic remains.

It's brought together into one place.

The data is in the same place, flowing through the computation.

Separating Concerns

Senders don't care about the *mechanism* of concurrency.

The *same code* can run with:

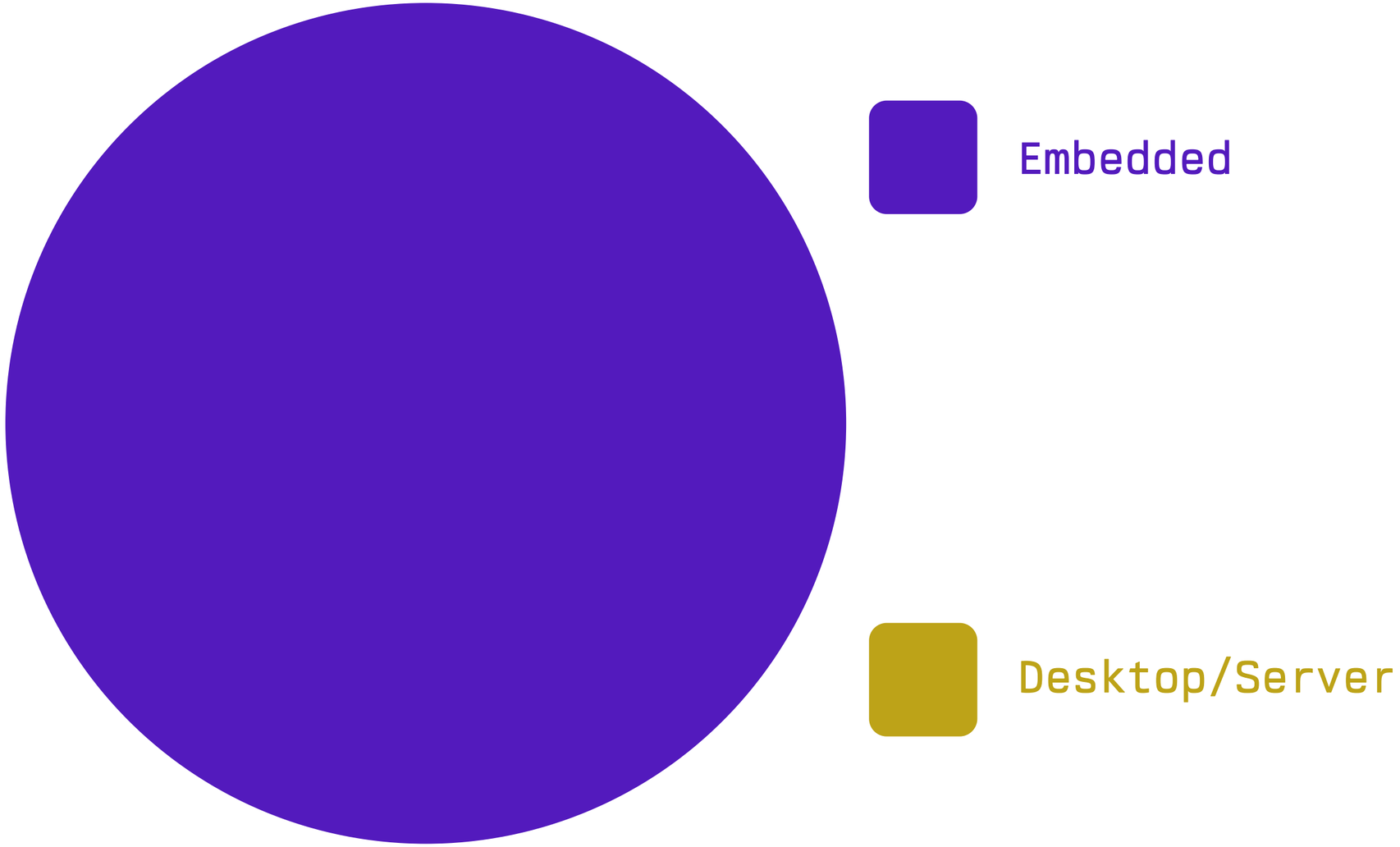
- threads
- coroutines
- interrupts
- synchronous calls
- etc.

The Real Power of Senders

Anyone* can write an async framework that makes good use of large-scale concurrency & parallelism.

The *real* trick is in making that same framework support very small, constrained devices, at least as efficiently as hand-written code.

Processors in the World (Approx)



Useful Glue: the Trigger Scheduler

A *priority* scheduler represents running code on a *priority* interrupt.

A *timer* scheduler represents running code on a *timer* interrupt.

A *trigger* scheduler represents running code on *any other* interrupt.

Useful Glue: the Trigger Scheduler

```
auto s = trigger_scheduler<"example">{};
auto sndr = start_on(s, just(42));

int var{};
start(connect(sndr, receiver{[&] (auto i) { var = i; }}));

run_triggers<"example">();
// now var == 42
```

The CIB callback library attaches a named callback to an ISR and that callback runs the associated `trigger_scheduler`.

Trigger Scheduler + `incite_on`

The `trigger_scheduler` really shines when it's used with `incite_on`.

```
auto s = send_msg()
  | incite_on(trigger_scheduler<"callback">{})
  | let_value([] (auto response) { ... });
```

And `run_triggers` allows values to be passed.

```
// inside the ISR
auto rsp = read_response_registers();
run_triggers<"example">(rsp);
```

`incite_on`

`incite_on` is a useful combinator.

Note - this would be wrong:

```
auto s = send_msg()
  | continue_on(trigger_scheduler<"callback">{});
```

Hence `incite_on` for the use case where the upstream sender *causes* the downstream scheduler to run.

(This could also be done – carefully – with `when_all` but `incite_on` expresses it better.)

`groov`

`https://github.com/intel/generic-register-operation-optimizer`

`groov` is an open-source (Boost license) library
that uses senders as an interface to register operations.

(Note: in this context, "register" means "memory location with special semantics.")

groov Declarations

```
struct bus {
  template <auto...>
  static auto write(auto addr, auto value) -> sender auto;

  template <auto>
  static auto read(auto addr) -> sender auto;
};

using F = groov::field<"field", std::uint32_t, 0, 0>;
using R = groov::reg<"reg", std::uint32_t, 0, groov::w::replace, F>;
using G = groov::group<"group", bus, R>;
```

Fields are inside registers.

Registers are in groups.

A bus implements read and write.

Bus Types

Buses' `read` and `write` functions return senders.

So buses can use synchronous MMIO to access registers,
or can do asynchronous reads/writes over a link.

Fields and registers also have write functions and masks
that a bus can use to optimize access.

- read-only
- write 1 to clear
- etc

Optimized Operations

Operations can be optimized by the bus in some situations.

For example:

- if the underlying fabric supports byte enables.
- if some fields are read-only.
- if there are different semantics for offset writes (e.g. RP2040).

Read-Modify-Write

(When it can't be optimized by the bus.)

```
auto s = just(grp / "reg"_r)
       | groov::read
       | then([](auto spec) {
           spec["reg"_r] ^= 0xff;
           return spec;
         })
       | groov::write;
```

Is it Efficient?

Many registers are accessed through a simple MMIO bus.

```
struct bus {
  template <auto Mask, auto...>
  static auto write(auto addr, auto value) {
    return just_result_of([=] {
      *addr = (*addr & ~Mask) | value;
    });
  }
  template <auto>
  static auto read(auto addr) {
    return just_result_of([=] {
      return *addr;
    });
  }
};
```

What we want is for this style of access to optimize to just a load or a store.

Ben, IS IT EFFICIENT?

```
std::uint32_t data = 0xa5;
using R = groov::reg<"reg", std::uint32_t, &data>;

auto s = just(grp / "reg"_r)
       | groov::read
       | then([](auto spec) {
           spec["reg"_r] ^= 0xff;
           return spec;
         })
       | groov::write;
start_detached_unstoppable(s);
```

Ben, IS IT EFFICIENT?

```
std::uint32_t data = 0xa5;
using R = groov::reg<"reg", std::uint32_t, &data>;

auto s = just(grp / "reg"_r)
       | groov::read
       | then([](auto spec) {
           spec["reg"_r] ^= 0xff;
           return spec;
         })
       | groov::write;
start_detached_unstoppable(s);
```

```
mov dword ptr [rip + data], 90
ret
```

Correct By Inspection

A senders-based interface offers:

- declarative computation
- code and dataflow in one place
- composition as a first-class citizen
- low-level implementation independence
- async or sync choice
- as efficient as possible

Changing Gears Again

Let's talk more about concurrency, and how to test it.

My Problem

Almost all of my job involves handling concurrency.

The mechanism of concurrency on hardware is interrupts.

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

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

Low-level Concurrency Primitives

I basically have only two primitives I can use:

- a "global mutex" or critical section: I can turn off interrupts
- atomic `std::uint32_t` supporting `exchange`

There is no parallelism, but there is concurrency.

I don't have `std::mutex`. I have `std::atomic` in limited form.

Problem: Interference

In one piece of code...

```
{
  critical_section cs{};
  use_shared_data_a();
}
```

In another piece of code...

```
{
  critical_section cs{};
  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() {
  critical_section cs{};
  another_function();
}

auto another_function() {
  critical_section cs{};
  do_something();
}
```

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

This is Incompatible With Desktop Testing

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

What I 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

Another one-function API:
it's time for the Global API Injection Pattern again.

```
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));
}
```

(Note: the return type is not dependent on `DummyArgs...`)

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 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)();
  }
};
```

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.

Another Problem: `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. (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`

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`.

Bringing it Together

Declarative interfaces for the build system.

```
add_unit_test(my_test CATCH2
  FILES my_test.cpp
  LIBRARIES my_lib)
```

```
$ ninja -C build run_my_test
ninja: Entering directory `build'
[1/1] Generating my_test.passed
Randomness seeded to: 146929539
===============================================================================
All tests passed (28 assertions in 12 test cases)
```

Bringing it Together

Declarative interfaces for test injections.

```
std::string log_buffer{};
template <>
inline auto log_config<> =
  logging::fmt::config{std::back_inserter(log_buffer)};

TEST_CASE("test log output", "[suite]") { ... }
```

Bringing it Together

A declarative interface for asynchronous and synchronous code alike.

```
auto s = just(grp / "reg"_r)
  | groov::read
  | then([](auto spec) {
      spec["reg"_r] ^= 0xff;
      return spec;
    })
  | groov::write;
```

Bringing it Together

Declarative interfaces for dealing with hardware limitations.

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

Conclusions

Testing at the right level is key.
Declarative abstractions also extend to the build system.

Logging, asynchrony, concurrency and atomics are
all usefully improved by the right low-level interfaces.

Injecting global APIs is a really useful technique which enables effective desktop testing even while supporting embedded platforms.

Code that is physically close makes for easier reasoning.
Good declarative interfaces help this without sacrificing performance.

Epilogue: `CX_VALUE`

One common interface to log compile-time values and types.

```
CIB_INFO("{} has type {}", CX_VALUE(42), CX_VALUE(decltype(42)));
```

```
INFO: 42 has type int
```

I want to do this

```cpp
auto x = CX_VALUE(42);          // 42
auto y = CX_VALUE("hello"sv); // std::string_view{"hello"}
auto z = CX_VALUE(int);         // std::type_identity<int>{}
```

So that the interface is uniform.

Note:

- `x` uses a structural value
- `y` uses a non-structural value
- `z` uses a type
- The argument is always known at compile-time

Fairly obvious that this must be a macro

OK, let's write the skeleton.

```
#define CX_VALUE(...)
  [] {
    if constexpr(/* detect */) {
      // __VA_ARGS__ is a type
    } else {
      // __VA_ARGS__ is a value
    }
  }()
```

Immediately-invoked lambdas are great for macro encapsulation!

Problem 1: detect type/value

```
1: template <auto> constexpr auto is_type() -> std::false_type;
2: template <typename> constexpr auto is_type() -> std::true_type;
3:
4: // and then the expression
5: decltype(is_type<__VA_ARGS__>())::value;
```

This works for structural values and for types, but not for non-structural values!

We can't pass a `std::string_view` as an NTTP.

Problem 1: detect type/value

OK, let's make something we *can* pass as an NTTP.

```
struct from_any {
    template <typename... Ts> constexpr from_any(Ts const &...) {}
    constexpr operator int() const { return 0; }
};

// and now
decltype(is_type<from_any(__VA_ARGS__)>())::value
```

`from_any` is a class (not a template!) that is implicitly constructible from anything.

Bonus: it implicitly converts to `int`. So we don't actually need C++20 structural type rules: an `int` can be an NTTP!

Wait, how does that work?

Consider the cases:

```
// this is a value of type from_any constructed from 42
from_any(42)

// this is a value of type from_any constructed from a string_view
from_any("hello"sv)

// this is a function type taking int and returning from_any
from_any(int)
```

Wait, how does that work?

Which leads to:

```
template <int> constexpr auto is_type() -> std::false_type;
template <typename> constexpr auto is_type() -> std::true_type;

is_type<from_any(42)>()          // false_type
is_type<from_any("hello"sv)>() // false_type
is_type<from_any(int)>()         // true_type
```

First problem solved! We have differentiated types and values.
(including non-structural values)

Next...

```
#define CX_VALUE(...)
  [] {
    if constexpr(decltype(is_type<from_any(__VA_ARGS__)>()))::value) {
      // __VA_ARGS__ is a type
    } else {
      // __VA_ARGS__ is a value
    }
  }()
```

Now we need to fill in each arm of the `if`.

Only one arm at a time will actually be used.

But! Both sides must be *syntactically well-formed* in all cases!

When it's a type?

We can use a similar trick to `is_type` to get the actual type out.

```
template <typename> struct typer;
template <typename T> struct typer<from_any(T)> {
    using type = T;
};

template <int> constexpr auto type_of() -> void;
template <typename T> constexpr auto type_of()
                                    -> typename typer<T>::type;

// and then
using actual_type = decltype(type_of<from_any(__VA_ARGS__)>());
```

This will be `void` if we pass in values, otherwise it will be the type we pass in.

So far so good

```
#define CX_VALUE(...)
  [] {
    if constexpr(decltype(is_type<from_any(__VA_ARGS__)>())::value) {
      using actual_type = decltype(type_of<from_any(__VA_ARGS__)>());
      return std::type_identity<actual_type>{};
    } else {
      // __VA_ARGS__ is a value
    }
  }()
```

One side down, one side to go.

Now we need something that will return the value,
but *also* be syntactically well-formed if given a type.

Consider the following

Here's a curious construction:

```
(__VA_ARGS__) + special{};
```

If given a *value*, this is a binary plus expression.
We can overload operators on `special` to do what we need.

If given a *type*, this is a C-style cast of a unary plus expression.
We can make that well-formed.

We are almost done!

```
struct special {
  template <typename T>
  friend constexpr auto operator+(T &&t, special) {
    return t;
  }

  friend constexpr auto operator+(special) -> special;
  template <typename T> constexpr operator T() const;
};
```

Making the `special` class with operator overloads is easy.

And we're actually done

```
#define CX_VALUE(...)
  [] {
    if constexpr(decltype(is_type<from_any(__VA_ARGS__)>())::value) {
      using actual_type = decltype(type_of<from_any(__VA_ARGS__)>());
      return std::type_identity<actual_type>{};
    } else {
      return (__VA_ARGS__) + special{};
    }
  }()
```

This even works pre-C++20!

Interface achieved

```
auto x = CX_VALUE(42);          // 42
auto y = CX_VALUE("hello"sv); // std::string_view{"hello"}
auto z = CX_VALUE(int);         // std::type_identity<int>{}
```

Conclusions

Testing at the right level is key.
Declarative abstractions also extend to the build system.

Logging, asynchrony, concurrency and atomics are
all usefully improved by the right low-level interfaces.

Injecting global APIs is a really useful technique which enables effective desktop testing even while supporting embedded platforms.

Code that is physically close makes for easier reasoning.
Good declarative interfaces help this without sacrificing performance.