OPTIMISING FOR CHANGE

BEN DEANE / CPPNORTH / JULY 2023

TEST SLIDE

Here is some code. (About as much and as dense as it *may* get.)

```
template <typename F, typename... Args>
constexpr auto curry(F &&f, Args &&...args) -> decltype(auto) {
  if constexpr (std::invocable<F, Args...>) { // call function if possible
    return std::invoke(std::forward<F>(f), std::forward<Args>(args)...);
  } else { // capture args in recursive call
    return [f = std::forward<F>(f),
            ...args = std::forward<Args>(args)]<typename Self, typename... As>(
               this Self&& self, As &&...as) -> decltype(auto) {
      return curry(std::forward_like<Self>(f),
                   std::forward_like<Self>(args)...,
                   std::forward<As>(as)...);
    };
  }
}
```

Inline code words look `like this`.
This is the about the lowest text will be.

FRONT MATTER

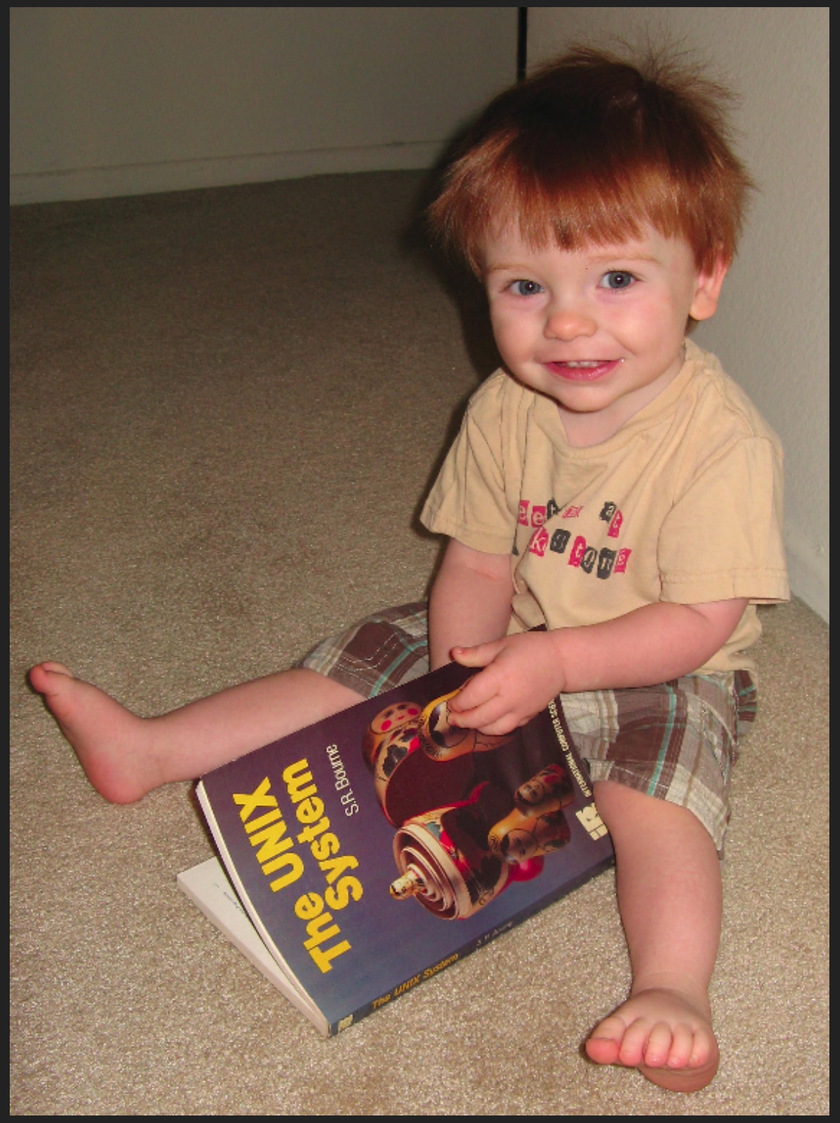
Move closer if you need to.

Please save questions until the end (or hallway track).

No AI was used in the creation of this talk.

All code is platonically correct but may have errors in translation because we all live in the cave.

ABOUT ME



I worked in games
for a long time.

I have worked in finance.

I now work in embedded.

MORE ABOUT ME

MORE ABOUT ME

I am optimistic and naïve.

MORE ABOUT ME

I am optimistic and naïve.

I am selfish and exacting.

MORE ABOUT ME

I am optimistic and naïve.

I am selfish and exacting.

I am lazy.

MORE ABOUT ME

I am optimistic and naïve.

I am selfish and exacting.

I am lazy.

…when it comes to computers, anyway.

WHAT DO WE MEAN BY OPTIMIZATION?

Very often, we mean optimizing for runtime speed.

Most of the rest of the time, we mean optimizing for memory.

Very occasionally, we mean something else.

THE NATURE OF SOFTWARE

"People think of software like plasticine;
it's more like quick-set concrete."

-- Unknown, paraphrased

TECHNICAL OPTIMIZATION IS EASY...

…as long as you can change the code.

The ability to change the code is paramount. Everything else is secondary.

- incorrect code can be fixed
- slow code can be sped up
- new requirements can be accommodated

OPTIMIZING FOR CHANGE

What does that mean? It means writing code that is:

- flexible: minimizing points of necessary change
- malleable: maximizing the ability to change
- verified: maximizing the confidence in change

1: FLEXIBILITY

The ability to minimize the amount of change in code when required, or

minimizing the places where code has to change.

SMALL SCALE FLEXIBILITY

Many times, code reviews are an opportunity to point out small-scale changes that increase flexibility.

EXAMPLE: VARIABLE DECLARATION

```
void func(range auto const& r) {
  for (int i = 0; i < r.size(); ++i) {
    // ...
  }
}
```

What happens when we compile with warnings?

EXAMPLE: VARIABLE DECLARATION

```
warning: comparison of integers of different signs:
  'int' and 'std::array::size_type' (aka 'unsigned long') [-Wsign-compare]
```

OK, so how do we fix that?

EXAMPLE: VARIABLE DECLARATION

```
// old
int i = 0;
// new
std::size_t i = 0;
```

In my experience, this is the most likely "fix"
that someone applies to a conversion warning.

EXAMPLE: VARIABLE DECLARATION

Which of these would you recommend?

```
std::size_t i{};
auto i = std::size_t{};
auto i = 0uz;
auto i = typename Array::size_type{};
```

Any of these work better. (And of course there are more options.)

THIS HAPPENS A LOT

Hopefully anyone in a senior position is doing code reviews.
Potentially a lot of code reviews.

Almost every time I've considered "why do I recommend this" the answer turns out to be because it's easier to maintain in the face of possible future change.

Even (especially) when I recommend habits I've picked up and
I'm not quite sure why until I really think.

EXAMPLE: ITERATOR/POINTER ARITHMETIC

```
// old
auto p = base + offset;
// new
auto p = std::next(base, offset);
```

What is only a pointer today may change tomorrow.
(A pointer is an iterator, anyway.)

So why constrain ourselves to random access? Get in the habit of using `next`, `prev`, `distance`, `advance`.

EXAMPLE (FROM BEFORE)

```
// old
for (int i = 0; i < r.size(); ++i) { ... }
// new
for (auto i = 0uz; i < std::size(r); ++i) { ... }
```

Because C-style arrays don't have a `size` member.

EXAMPLE: MEMCPY

```
// old
std::memcpy(dest, src, n);
// new
std::copy_n(src, dest, n);
```

`copy_n` does everything that `memcpy` does, and more.

EXAMPLE: FIXED RETURN TYPE

```
template <typename T>
struct Decorator {
  auto some_func() -> std::uint32_t {
    return obj.some_func();
  }

  T obj;
};
```

This is a very common one. Even if "we know" what `T::some_func` returns, why put that knowledge into the wrapper class?

EXAMPLE: FIXED RETURN TYPE

```
template <typename T>
struct Decorator {
  auto some_func() -> decltype(auto) {
    return obj.some_func();
  }

  T obj;
};
```

The chances that this wrapper will have to change are lowered.
(This is not only for templates.)

EXAMPLE: DURATIONS

Fixing the type of durations is very common.
We humans love to think in base-10 fractions of seconds.

```
constexpr auto ticks_to_us(auto ticks) {
  return ticks * 1'000'000 / 32'768;
}
```

Why not use the "natural" units instead?

```
using rtc_tick = std::chrono::duration<
  std::int64_t, std::ratio<1, 32'768>>;
```

ALGORITHMS

```
for (auto c : chars) {
  if (c == 'A') {
    return true;
  }
}
return false;
```

Why do we recommend algorithms? One reason: they increase flexibility.

Why should I care if chars is a `string`, a `vector<char>`, or something else?
It's just a `range`; I can use `find_if`.

THE STL DOESN'T CHANGE WHEN YOUR PROGRAM NEEDS TO CHANGE

Often, making code more flexible means making it more generic.

In C++ we are lucky to be able to do this without any runtime performance penalties.

But: don't go away thinking this is just about templates.

FUNCTIONS

```
auto load_city() {
  // step 1: load landscape and geometry

  // step 2: load game state

  // step 3: reticulate splines
}
```

Why do we recommend splitting up functions?

It minimizes future change if code is delineated clearly with a good interface.

A function is the basic unit of reusable code.

"I DON'T WANT TO KNOW"

- not fixing return types
- replacing loops with algorithms
- using `std::next` and friends
- breaking functions apart

These are all manifestations of a larger principle: "I Don't Want to Know"

"I DON'T WANT TO KNOW"

Under various circumstances, I don't want to know:

- how
- when/where/who
- even (some forms of) what

I need to know only:

- what do I need to provide?
- what can I expect?

MORE EXAMPLES

- `std::visit`
- error handling with exceptions
- monadic operations on `std::optional`/`std::expected`
- passing a lambda function

PASSING A LAMBDA FUNCTION

To some extent, turns code "inside out".

Caller doesn't need to know the return type.
Callee doesn't need to know the condition(s)/operation(s).

Sometimes, callee doesn't need to fix the return type either!

DENY ALL KNOWLEDGE

The less that functions, classes etc know about each other, the less they need to change when their surroundings change.

In the end, all roads lead to Rome. We know this already.

- Good declarative style: say what, not how
- Good OO style: tell, don't ask
- Good functional style: write total functions
- Good generic style: interfaces, not implementations
- *Good style in general*

LIMITING CHANGE LOCI

A table-driven approach puts everything in one place.
When things change, they only need to change in one place.

```
struct tcp_release {
  auto operator()() const {
    using namespace sml;
    return make_transition_table(
      *"established"_s + event<release>           / send_fin  = "fin wait 1"_s,
       "fin wait 1"_s  + event<ack> [ is_valid ]              = "fin wait 2"_s,
       "fin wait 2"_s  + event<fin> [ is_valid ] / send_ack  = "timed wait"_s,
       "timed wait"_s  + event<timeout>                       = X
    );
  }
};
```

TABLE-BASED APPROACHES

Especially good for (but of course not limited to) metaprogramming.

```
using special_forms = list<
  //   name       function       parsers...
  form<"lambda", make_lambda,  parse_formals, parse_body>,
  form<"define", make_binding, parse_var,     parse_expr>,
  form<"quote",  quote,        parse_expr>,
  form<"if",     if_form,      parse_expr,    parse_expr, parse_expr>>;
```

A table-based approach also enables an open-ended set of queries.

With concepts, table-querying metafunctions that automatically turn non-existence into falsity are easy to write.

ANOTHER WAY TO IMPROVE FLEXIBILITY

Delete code. (While retaining functionality.)

Any reasonably sized project has multiple pieces of code that do the same job.

- Find them.
- Make them generic enough to be the same.
- Remove all but one.
- The number of future change loci is now one.

BOTTOM-UP IMPLEMENTATION

Flexibility often proceeds from building small pieces
and composing them together.

Again, a function is the basic unit of reusable code.
Either a (runtime) function of values, or a (meta) function of types.

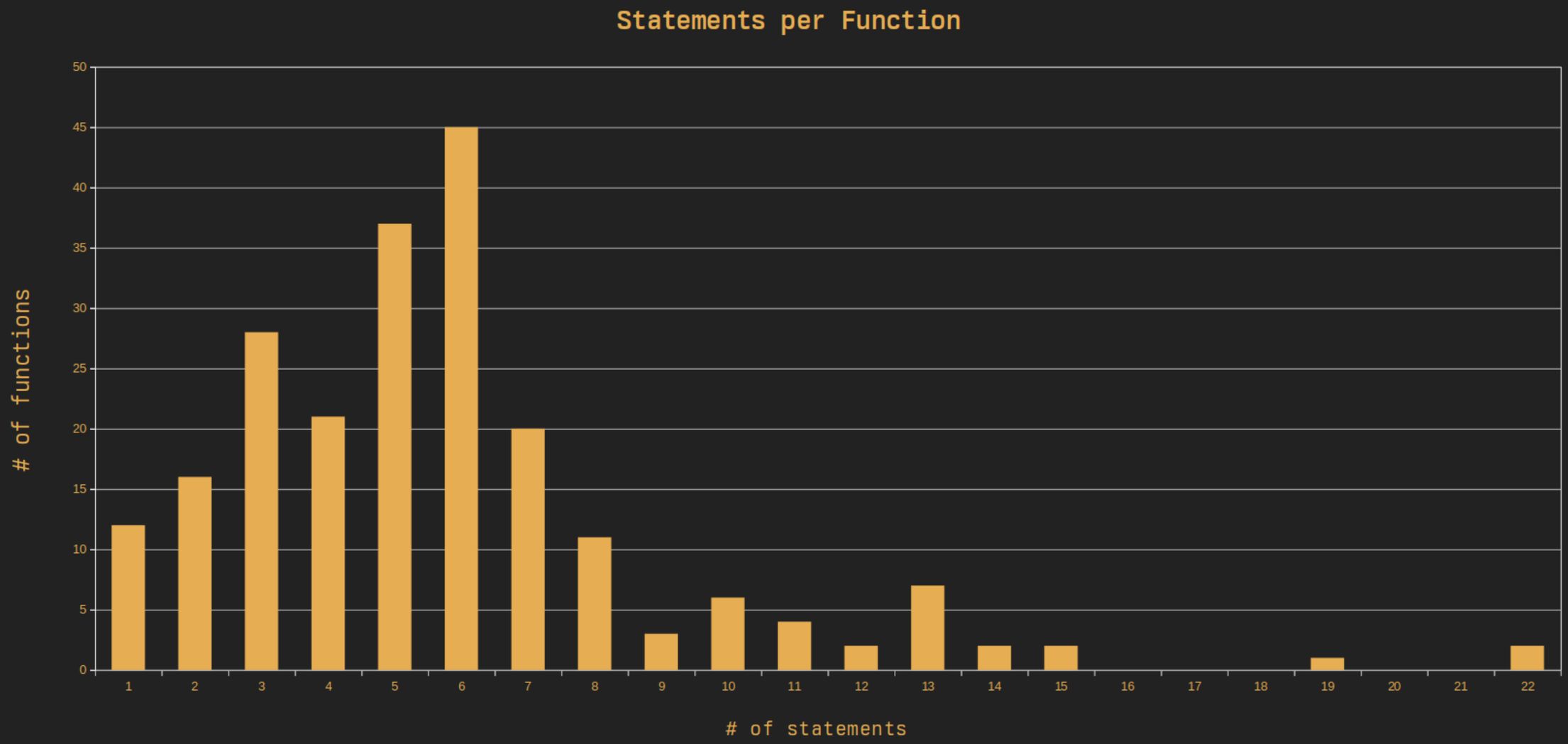
QUESTION

How large is large?

How many lines of code makes a large function?

How many lines of code makes a large file?

MAKE FUNCTIONS SMALL



SMALL-SCALE FLEXIBILITY

Does it matter that much?

Yes!

Because code gets copied everywhere, and quickly.

2: MALLEABILITY

Maximizing the ability to change.

IS CODE REUSABLE?

In the 1990s/2000s, OO's big claim was reusability.

That claim foundered. Why?

C++ OO IS BASED ON SIMULA

Simula is from 1967.

Ole-Johan Dahl and Krysten Nygaard (apologies for pronunciation) invented what we (in C++) call "object orientation".

They weren't solving the problem of reusability.

C++/SIMULA OO TO A SMALLTALKER?

"It is not hard to imagine what the world would look like to someone who had been raised by contractors and who had never used any drill other than a Hole Hawg.
Such a person, presented with the best and most expensive hardware-store drill, would not even recognize it as such. [They] might instead misidentify it as a child's toy, or some kind of motorized screwdriver.
If a salesperson or a deluded homeowner referred to it as a drill, [the Hole Hawg user] would laugh and tell them that they were mistaken--they simply had their terminology wrong."

-- Neal Stephenson, In the Beginning was the Command Line

BUT C++ DOES HAVE REUSABLE CODE

We have generic programming.

We have templates and concepts.

We have something like "real OO"; just at compile time, not runtime.

GOOD INTERFACES

Identify basis functions; write other code (free functions) in terms of them.

The difference between a member function and a free function is that a member function is allowed to break and restore invariants.

LOTS OF WAYS TO DO COMMON THINGS

"Have many variants of the simple algorithms. [...] You're going to be doing lots of different kinds of copies. You're going to be doing lots of different kinds of finds.
Have just a whole collection of them."

-- Sean Parent, C++ Seasoning

LOTS OF WAYS

- `find`, `ranges::find_last`
- `find_if`, `find_if_not`
- `ranges::find_last_if`, `ranges::find_last_if_not`
- `search`, `find_end`
- `search_n`

And this is just the standard, and just the single-range find type algorithms…

We have a *lot* of ways to do things.

LARGER SCALE FLEXIBILITY?

Malleability often translates to "larger scale flexibility."
Because decisions often age poorly.

aka "Life comes at you fast."

Q. HOW MANY ARE THERE?

We know that there are only a few numbers in programming.

Zero, one, two, or N.

And I'm not sure about zero, one and two…

WHAT ABOUT SINGLETONS?

Some things really are singletons, right?

- logging system?
- memory manager?
- things corresponding to hardware devices?

(I work in embedded.)

RULE OF THUMB

The decision over whether to have only one of something
should not affect its implementation.

Unfortunately, if you don't keep this rigorously in mind, before you know it,
you are constrained to having *only* one *because of* the implementation.

Decide how many you want at the top level only.

HYPOTHETICAL: SPORTSBALL

You're writing a sports game for a major franchise.
The kind of game that has the year in the title.

You only need one ball, right?
The game's only ever played with one ball.

So you make the ball a singleton.

HYPOTHETICAL: EMBEDDED SYSTEMS

You're writing (really) low level code. Clues:

- you use the term "register" to mean "memory location"
- your organization doesn't use the word "software"
- your data is usually measured in bits, not bytes
- the software (sorry, firmware) spec proceeds directly from the hardware spec

You *know* that there's only {one, two, eight} of these. Because it's in hardware!

MATHS VS IO

All code does only two things:

- mathematical calculations (sometimes)
- IO (always)

One secret to change-health is ruthlessly decoupling these two things.

(Functional languages can put IO in monads to "turn it into Maths",
but eventually they *still need some magic* to actually perform the IO.)

DECOUPLING BEHAVIOUR

```
LOG("I want to log {} and {}, "
    "but I don't care what the logging implementation is",
    42, 1729);
```

I want this code to work everywhere it needs to.

I want to be able to tailor the logging implementation.

I want to be able to test (easily).

NOT JUST LOGGING

This is a common issue for any "global" API.

- logging functions
- timing functions
- concurrency primitives
- file/networking functions
- access to hardware
- etc

The answer: almost always some form of dependency injection.

DEPENDENCY INJECTION

At runtime...

```
struct interface {
  virtual auto do_thing() const -> void = 0;
};

struct client {
  client(interface& i) : iface{i} {}
  auto do_thing() const -> void { iface.do_thing(); }

private:
  interface& iface;
};

struct impl : interface {
  auto do_thing() const -> void final { ... }
};
```

DEPENDENCY INJECTION

At compile time…

```
template <typename T>
concept interface = requires(T t) {
  { t.do_thing() } -> std::same_as<void>;
};

template <interface I>
struct client {
  client(I& i) : iface{i} {}
  auto do_thing() const -> void { iface.do_thing(); }

private:
  I& iface;
};

struct impl {
  auto do_thing() const -> void { ... }
};
```

DEPENDENCY INJECTION

At link time…

```
extern auto do_thing() -> void;

// link with platform_a.lib for that implementation
// link with platform_b.lib for the other implementation
// link with test.lib for test implementation

// or equivalently:
// - conditional compilation (#defines)
// - conditional TU inclusion with the build system
// - careful header inclusion
```

ALL THESE METHODS HAVE DRAWBACKS

I've done all of these in the past. I'm not happy with any of them.

- may introduce runtime overhead
- may impose rigidity on implementors
- may involve pass-through (runtime/compile-time) arguments
- or may involve turning class hierarchies inside-out
- involve linking something (statically or dynamically)

(I'm not really that bothered that the top-level interface is often a macro, though)

A PARTICULAR PITFALL

A common pitfall of DI (unless done carefully) is "carrying dependencies".

```
struct rt_intermediate_class { // doesn't use interface
  rt_intermediate_class(interface& i) : child{i} {}

  rt_leaf_class child; // actually uses interface
};
```

```
template <typename Interface>
struct ct_intermediate_class { // doesn't use Interface
  ct_leaf_class<Interface> child; // actually uses Interface
};
```

Frequently, classes pass arguments through to lower levels, but have no other use for them. Leaf classes depend on things, therefore anything that uses leaf classes has to know about their dependencies.

WHAT TO DO?

I want to write code like this:

```
// in arbitrary places, use the interface

#include <log.hpp>
LOG("I want to log {} and {}, "
    "but I don't care what the logging implementation is",
    42, 1729);
```

And then:

```
// in one place, specify the implementation

#include <log_impl.hpp>
```

THE INTERFACE

```
// the default impl does nothing
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)...);
}

#define LOG(...) log(__VA_ARGS__)
```

THE IMPLEMENTATION

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

THE SELECTION

```
// log to stdout
template <> inline auto log_config<> = my_logger{};
```

Or in tests:

```
// log to a string I can check
template <> inline auto log_config<> = test_logger{};

// (if I even need to check output; most tests
// can get by with the null logger)
```

AN IMPORTANT DETAIL

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

The return type of this function is important.

It cannot depend on the specialized type of `log_config`.
But it can depend on the `Args` if needed.

INJECTION AT THE BASE

Upsides:

- The default can control whether it's an error *not* to specialize
- Easy to add concepts to the interface
- Can provide several alternatives that model the interface
- No coupling, no carrying dependencies
- Use sites depend on the interface only
- Client provides a single specialization to select the implementation

Downsides:

- Beware ODR violations (but as easy to wrangle as the other methods)
- Works best for smallish interfaces (but interfaces ought to be smallish)

POSSIBLE VARIATIONS

If we need to pass extra, explicit, parameters, we can do that.
If we want to add a concept, we can do that.

```
template <typename C>
concept loggable_config = requires (C& c, int i) {
  { c.template log<level::Error>("{}", i) } -> std::same_as<void>;
};

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

3: VERIFICATION

Maximizing the confidence in change.

WRITING C++ THE RIGHT WAY

(according to Jason Turner, paraphrased)

CI. Tests. Warnings. Warnings as errors. Multiple compilers.
Static analysis. Runtime analysis. Fuzzing. etc.

We are in a golden age of C++ tooling.
Use all the tools you have at your disposal.

THIS CATCHES REAL BUGS

```
template <typename T>
constexpr bool is_lazy_format_string_with_args_v = [] {
    if constexpr (is_lazy_format_string_v<T>) {
        return T::has_args;
    } else {
        return false;
    }
};
```

In many codebases there is so much low-hanging fruit here.

UNPOPULAR OPINION TIME

If you want to optimize for change:

- use the debugger less
- use types and tests more
- wear the hair shirt

I have made this journey of learning. Now I can't go back.
No saint like a reformed sinner, right?

THE DEBUGGER IS FOR DEBUGGING

If I am using the debugger, by definition I don't understand what the code is doing.

This is a not a good situation if I wrote the code.

The debugger lies.

- it tells us only about the assembly
- we're not only programming the hardware

SCARY EXAMPLE

```
using PMF = auto (*)() -> void;

struct A {}; // defined
using A_PMF = auto (A::*)() -> void;

static_assert(sizeof(A_PMF) == sizeof(PMF));
```

This code compiles without warnings on MSVC++.

(As it may; *this* is OK.)

SCARY EXAMPLE

```
using PMF = auto (*)() -> void;

struct A;    // declared
using A_PMF = auto (A::*)() -> void;

static_assert(sizeof(A_PMF)  > sizeof(PMF));
```

This code also compiles without warnings on MSVC++, prior to VS 2019 v16.10.

C5243 is off by default (even with /W4).

TYPES AND TESTS

Types provide up-front verification

- universal quantification over data
- correctness by construction

Tests provide continuing verification over time

- encoding of requirements
- protection against unforeseen changes
- confidence at scale

Nobody is going to step through your code in the debugger after the day you do. But without type and test guardrails, they might well change its behaviour accidentally.

SHIFT EVERYTHING LEFT

Remember, I'm exacting and lazy.

I want the computer to tell me what's wrong:

- as early as possible
- as quickly as possible
- with a clear resolution path

When the code leaves my machine, test should find no bugs.

TESTING CAVEAT

Testing at the right level is key.

Whatever you make easy to do will be done. So:

- make it easy to write tests at the correct level
- make sure that tests bind only the behaviour you want

TWO SPECIFIC RECOMMENDATIONS...

…to level-up your testing.

If you are using GoogleTest: look at GUnit

- `https://github.com/cpp-testing/GUnit`
- 100% back-compat with GTest/GMock
- Modern C++ niceties for QOL
- Plus BDD testing with Gherkin

TWO SPECIFIC RECOMMENDATIONS...

...to level-up your testing.

If you are already doing example-based testing:

- Start property-based testing
- RapidCheck is probably the most mature C++ solution
- `https://github.com/emil-e/rapidcheck`
- We need more C++ growth in this area

PBT FTW: EXAMPLE

```
auto compute_index_value() -> bool {
  int value = /* nested computations */;
  return value;
}

auto find_free_index() {
  return /*computations involving */ compute_index_value();
}

int result = find_free_index();
```

This code compiles without warnings. It runs, too.
It passes typical example-based tests.

IF YOU THINK TESTING IS HARD...

You're right. But *it should work*.

So the best time to start practising is today.

"INJECTION AT THE BASE" PATTERN FOR TESTING

(I discovered/formulated the pattern in puzzling out how to test logging.)

`Q:` How do you test e.g. writing to `stdout`?

`A:` (traditional method) By fiddling with pipes and file redirects?
(See e.g. `fmtlib`'s `gtest_extra` code, and many other libs.)

`A:` (injection at the base) By abstracting the low-level interface and injecting it.
Replace the IO directly, it's C++ all the way down.
And the interface can be compartmentalized and tested piecemeal.

CROSS-PLATFORM TESTING

We want to:

- abstract the platform
- minimize the amount of platform-specific code
- therefore maximize the value from "desktop" tests

Valuable testing provides maximal "reality" for minimal cost.
It's IO vs maths again. So let's minimize the IO.

CASE STUDY: CONCURRENCY

My target is a microcontroller.

I don't have threads. I don't have mutexes.
(I don't have half the standard library).

I want to write flexible, malleable, verified code. How?

ABSTRACTION IS KEY

"Being abstract is something profoundly different from being vague: by abstraction - from what is uncertain or should be left open - one creates a new semantic level on which one can again be absolutely precise."

-- Edsger W Dijkstra, EWD656

CASE STUDY: CONCURRENCY

I have:

- interrupts
- the C++ memory model
- the notion of a global mutex/critical section

I also have:

- assumed single-threadedness
- lack of local reasoning

THE PATTERN THAT KEEPS ON GIVING

```
template <typename...>
inline auto concurrency_policy = default_concurrency_policy{};

template <typename... DummyArgs, typename F, typename... Args>
auto call_in_critical_section(F&& f, Args &&...args)
    -> std::invoke_result_t<F, Args...> { // return type can rely on f(args...)
  return concurrency_policy<DummyArgs...>
    .call_in_critical_section(std::forward<F>(f),
                              std::forward<Args>(args)...);
}
```

One basic thing I can do is to abstract code that turns interrupts on and off to instead run a function under a global mutex, RAII-style.

DESKTOP IMPLEMENTATION

```
inline std::mutex global_mutex{};

struct default_concurrency_policy { // desktop/test policy
  template <typename F, typename... Args>
  auto call_in_critical_section(F&& f, Args &&...args) const
    -> std::invoke_result_t<F, Args...> {
    std::lock_guard l{global_mutex};
    return std::forward<F>(f)(std::forward<Args>(args)...);
  }
};
```

And I can provide a test implementation that can use standard constructs.

I CAN GO FURTHER

Problem: platform code assumes single-threadedness and a single global lock.

But why should it? This isn't future-proof. It's relying on unseen knowledge.

On desktop, this wouldn't be an issue because we'd have multiple mutexes.

SIMULATING MULTIPLE MUTEXES

```
template <typename Mutex = decltype([]{}),
          typename... DummyArgs, typename F, typename... Args>
auto call_in_critical_section(F&& f, Args &&...args)
    -> std::invoke_result_t<F, Args...> {
  return concurrency_policy<DummyArgs...>
    .template call_in_critical_section<Mutex>(std::forward<F>(f),
                                              std::forward<Args>(args)...);
}
```

Let's introduce another template argument that is optional.
If not specified, it will be different for every call site.

DESKTOP IMPLEMENTATION

```
template <typename>
inline std::mutex global_mutex{}; // different for every type

struct default_concurrency_policy {
  template <typename Mutex, typename F, typename... Args>
  auto call_in_critical_section(F&& f, Args &&...args) const
      -> std::invoke_result_t<F, Args...> {
    std::lock_guard l{global_mutex<Mutex>};
    return std::forward<F>(f)(std::forward<Args>(args)...);
  }
};
```

Each different type argument will get a different mutex.

USAGE

```
struct queue {
  struct my_mutex; // local type (no need to define)
  ring_buffer<std::uint32_t, capacity> storage{};

  auto push(std::uint32_t value) -> std::uint32_t& {
    call_in_critical_section<my_mutex>(
      [&] -> std::uint32_t& { return storage.push_back(value); });
  }
);
```

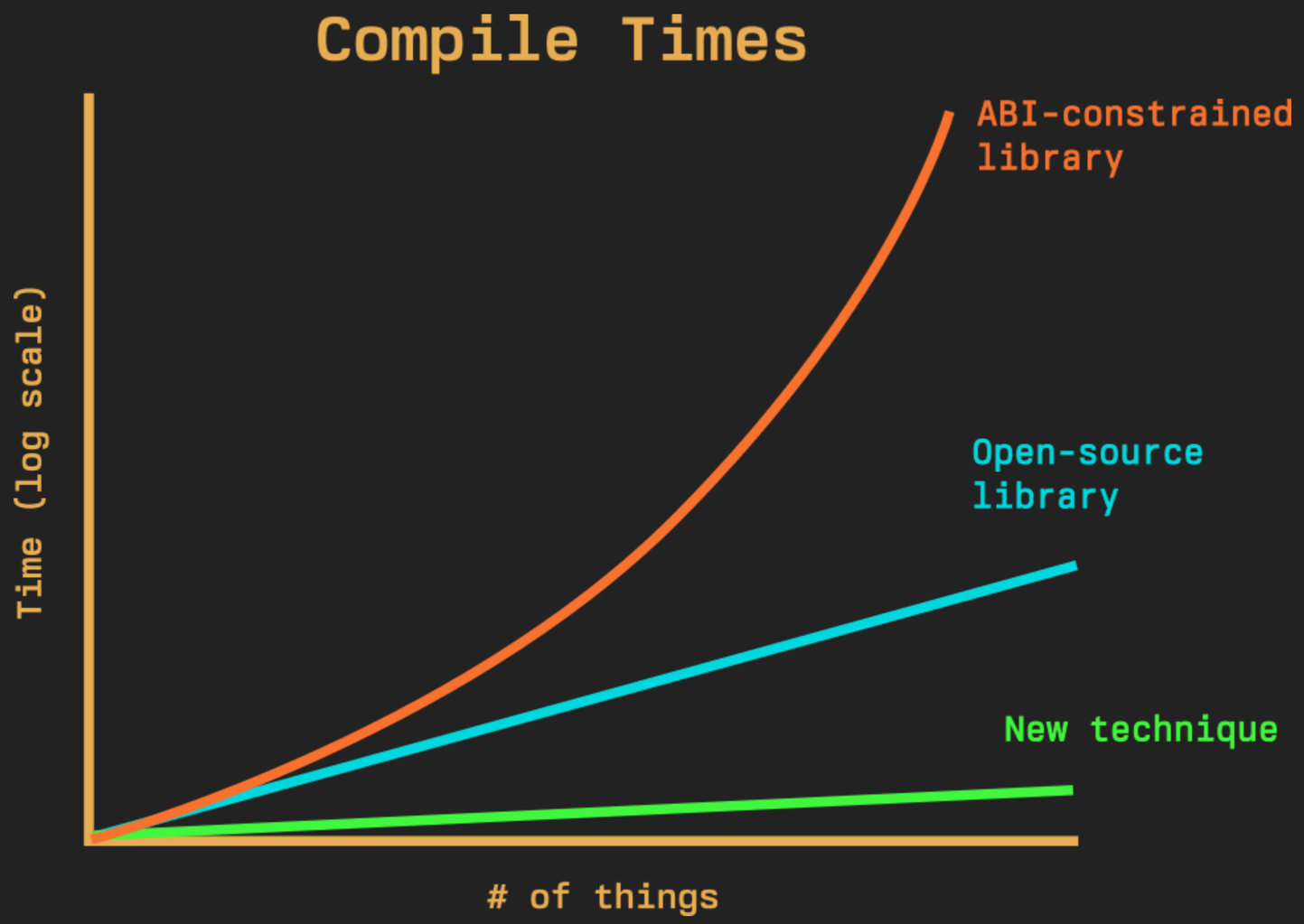
Now the platform code can properly specify the "mutex" that will protect its data.

ABSTRACTION BENEFITS

The code is "real" now.

- desktop tests can run with the same calling code
- platform code has no cost incurred
- code is more correct in the future
  - not relying on a single global mutex
  - no "accidental" correctness because of this
  - author must reason about data protection
- tests can implement interrupts with real threads
  - and use TSan!

BUT BEN, <PERF WORRIES>



VERIFICATION: OUTSIDE THE CODE

One interpretation of "verified" is "tested".

But another is "used".

Our code gets used every day, and the way it is used
is either friendly or not-so-friendly to change.

DON'T WRITE CHECKLISTS

Checklists are for pilots and surgeons.
Not for devops or QA folks.

If you can write a checklist, you can automate it.
Instruct a computer to do it, not a human.

It's a short step from a checklist that has 3 steps to one that has 43.

OPTIMIZE FOR MAINTENANCE

For me, this is a hard-learned lesson.

Characteristics of maintenance-sympathetic software:

- run all the time
- run in an ecosystem (homogeneous or heterogeneous)
- boot quickly
- designed to be turned off
- fail quickly and safely, restart automatically and gracefully
- designed without much config

If you're writing this kind of software, there are so many benefits from doing this even when you "don't have to".

HYPOTHETICAL SITUATION

You're working on a network game service.
It supports tens of millions of players and more than a few titles.

At first, game teams probably don't like you very much.
You have to say no a lot. But for good reason.

- No, I can't give you a function that does that, but you can achieve the same effect with these building blocks…
- No, my code can't depend on your protocol. I know nothing about that, by design.
- No, we can't introduce that feature like that, it would mean downtime for all the other games…

CHANGE HAPPENS

If you're successful at all, your product will have to react to change.

The landscape a year after launch looks very different.

And a year goes by in the blink of an eye…

DEVELOP A CHANGE MINDSET

For some years now, my title has varied, but my job has been the same.

Turning technical problems into business decisions.

Anticipating change means that the answer to "can we do X?" is almost always yes.
And here's what it will take…

SOME GOOD NEWS

Dealing with change is so difficult (and so poorly handled) that anything you can do to improve it is likely to turn out well.

Small considerations pay large dividends over time.

Optimizing for change is a great thing you can do for your career.

It's hard - but like anything, gets easier with practice.