The Joy of C++26 Contracts

(and Some Mythconceptions)

HERB SUTTER

CITADEL | Securities

Cppcon

The C++ Conference

2025

September 13 - 19

Disclaimer

Current status

- Contracts are part of draft C++26, with consensus in February
- We have two more meetings of technical fit-and-finish, including objections to contracts
- **Every concern will continue to get a full hearing**

This talk summarizes my best understanding of the design, expected best practices, FAQs, and major concerns

But (a) I could be wrong (and am happy to be wrong and learn),
and (b) you should treat the answers herein as provisional

Let me make myself a target…

Some experts strongly believe C++26 contracts are

beautiful and ready to go

not ready and shouldn’t ship

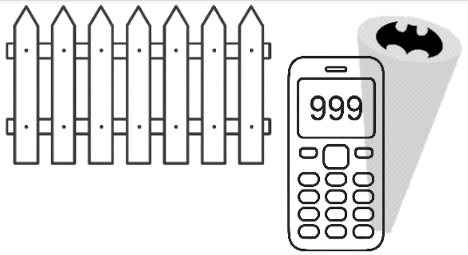
Q: How many of you think that I can offend both groups in two sentences?

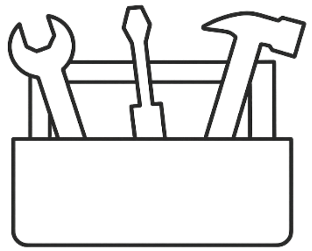
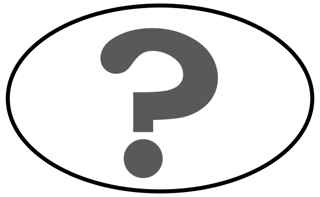
C++26 contracts have open questions and don’t yet have widespread deployment experience.

We “likely” won’t get a substantially better design than C++26 contracts, without wishing for magic linker improvements.

Roadmap

Motivation	Why defensive programming?
Overview	`pre`, `post`, `contract_assert`, `::handle_contract_violation()`
Best practices	"Effective C++26 Contracts"
Why ____?	Viable, ~minimal What each part enables
FAQs	Including open questions



Functional safety (vs. memory safety)

Functional safety: **Incremental confidence** that the system **meets its requirements**

- Primary aim of contracts
- Write one contract ⇒ get one check
- Incremental confidence: Depends on how many contracts you have (your own + in libraries)

Memory safety: **Guarantees** that the system is **free of certain classes of bugs**

- Build a certain way ⇒ get a guarantee (e.g., systematic language rules, systematic analysis rules)
- Contracts can **<u>also</u>** be mechanically generated for this, as an implementation detail
  Potential example: "Dereferencing a null pointer is checked by `contract_assert(pointer)`"
- Build-time guarantee: Automatically enforced by language or other build step (e.g., analyzer)

Example

Consider this code:

```
void my_function( Widget* w ) {
    if( is_gnarly(w) ) {
        write_to_gnarly_file( data.header(), w->to_string() );
    }
}
```

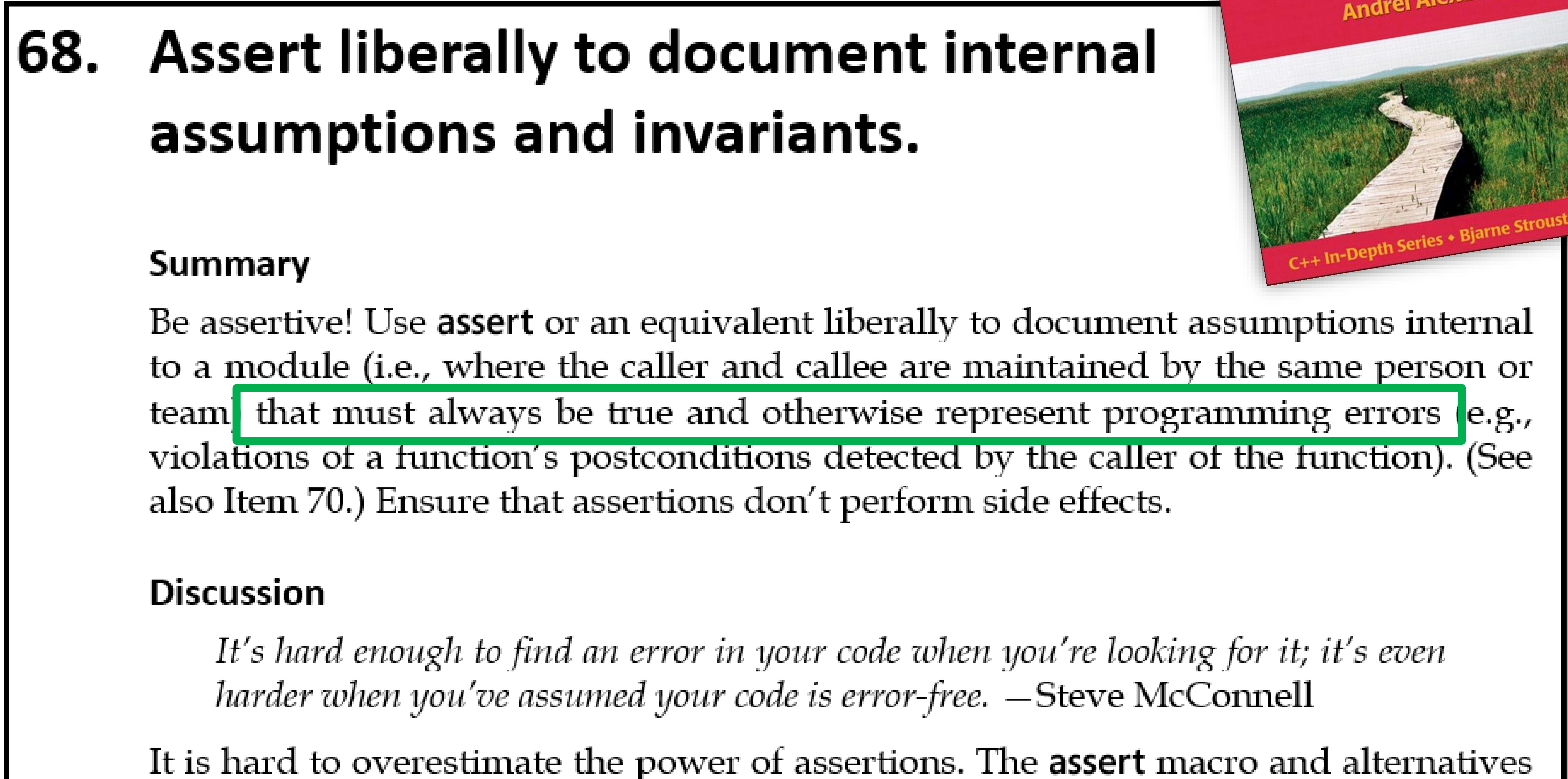
What could be better about this function? Discuss…

Example

Consider this code:

```
void my_function( Widget* w ) {
    assert(w);
    if( is_gnarly(w) ) {
        write_to_gnarly_file( data.header(), w->to_string() );
    }
}
```

Assert liberally



68. Assert liberally to document internal assumptions and invariants.

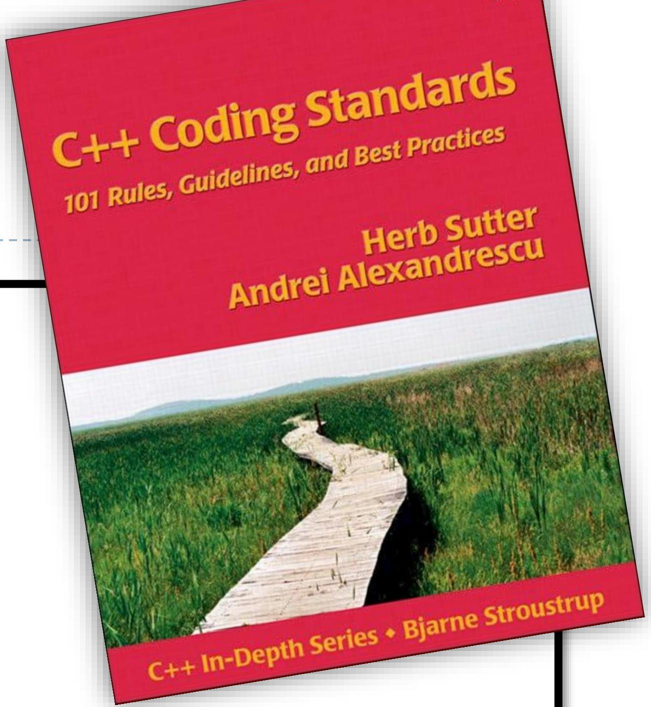
Summary

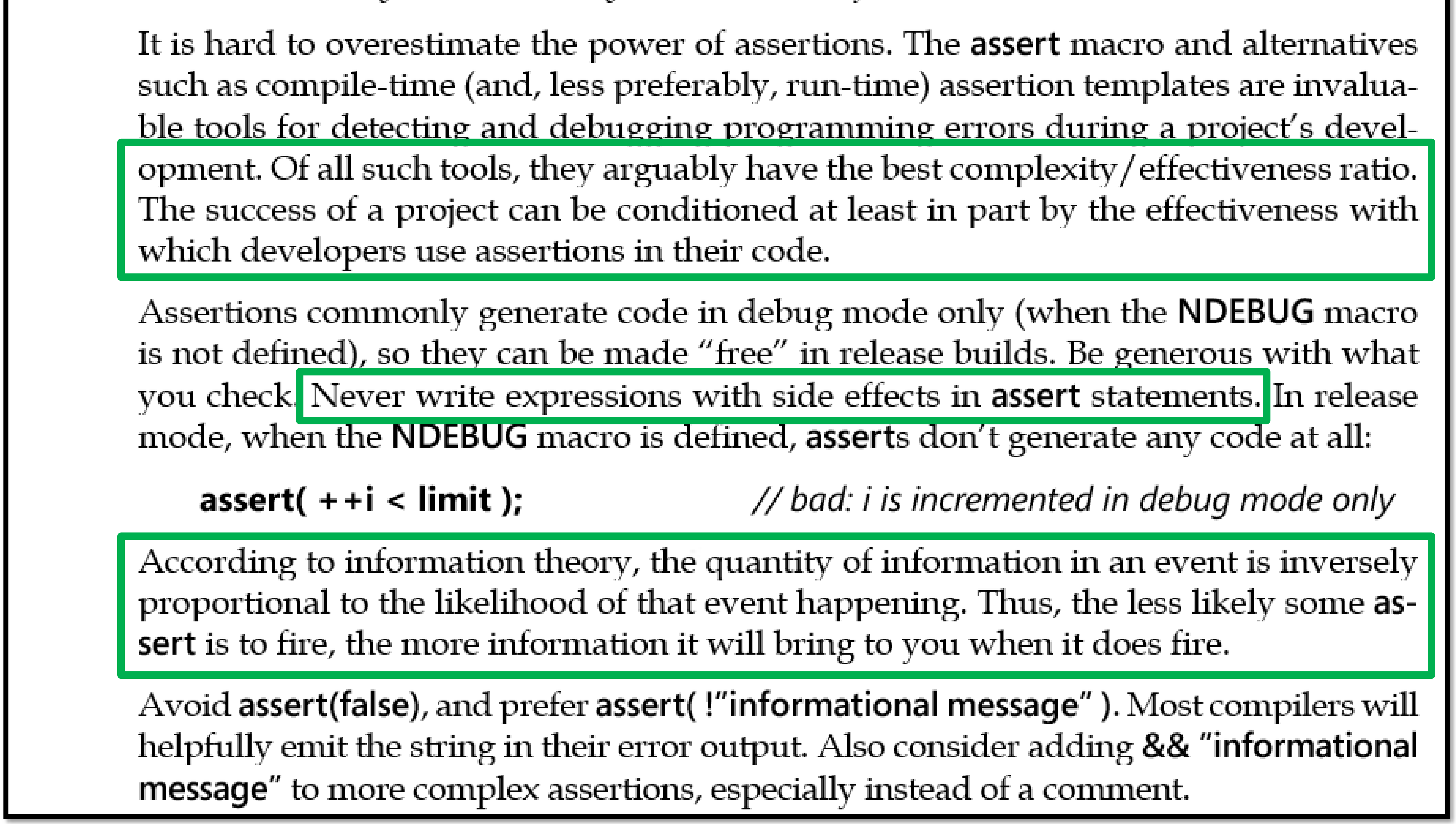
Be assertive! Use **assert** or an equivalent liberally to document assumptions internal to a module (i.e., where the caller and callee are maintained by the same person or team) that must always be true and otherwise represent programming errors (e.g., violations of a function's postconditions detected by the caller of the function). (See also Item 70.) Ensure that assertions don't perform side effects.

Discussion

It's hard enough to find an error in your code when you're looking for it; it's even harder when you've assumed your code is error-free. —Steve McConnell

It is hard to overestimate the power of assertions. The **assert** macro and alternatives


C++ Coding Standards
101 Rules, Guidelines, and Best Practices
Herb Sutter
Andrei Alexandrescu
C++ In-Depth Series • Bjarne Stroustrup



It is hard to overestimate the power of assertions. The **assert** macro and alternatives such as compile-time (and, less preferably, run-time) assertion templates are invaluable tools for detecting and debugging programming errors during a project's development. Of all such tools, they arguably have the best complexity/effectiveness ratio. The success of a project can be conditioned at least in part by the effectiveness with which developers use assertions in their code.

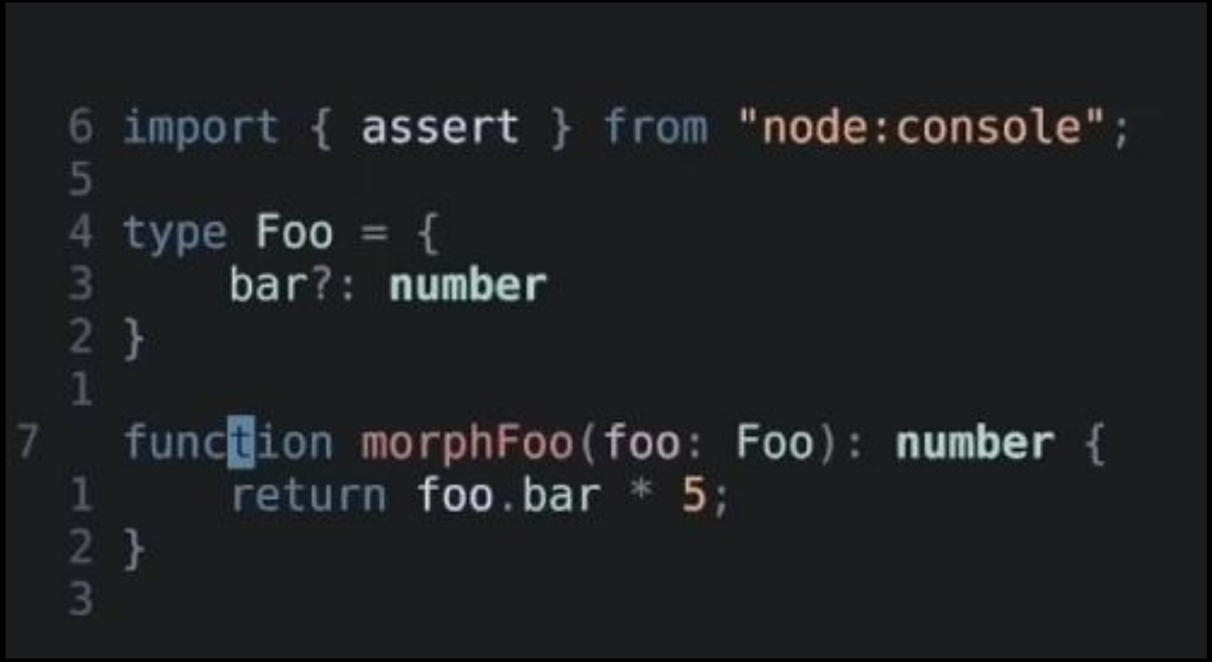
Assertions commonly generate code in debug mode only (when the **NDEBUG** macro is not defined), so they can be made "free" in release builds. Be generous with what you check. Never write expressions with side effects in **assert** statements. In release mode, when the **NDEBUG** macro is defined, **assert**s don't generate any code at all:

```
    assert( ++i < limit );                // bad: i is incremented in debug mode only
```

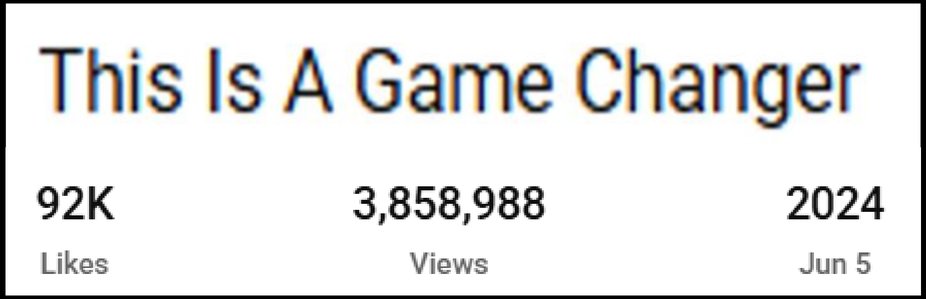
According to information theory, the quantity of information in an event is inversely proportional to the likelihood of that event happening. Thus, the less likely some **assert** is to fire, the more information it will bring to you when it does fire.

Avoid **assert(false)**, and prefer **assert(!"informational message")**. Most compilers will helpfully emit the string in their error output. Also consider adding **&& "informational message"** to more complex assertions, especially instead of a comment.

A YouTube short I recently sent to the C++ committee…



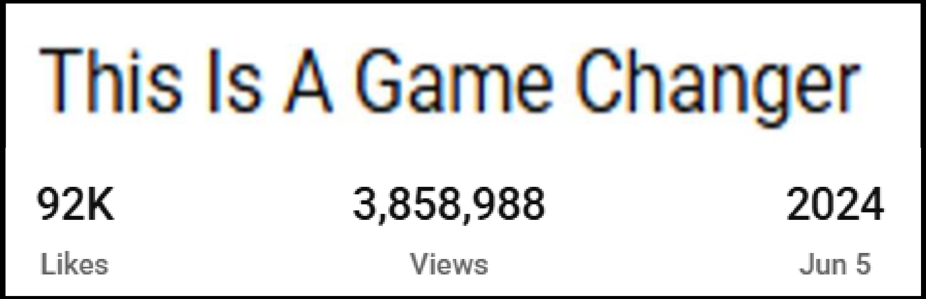
```
import { assert } from "node:console";

type Foo = {
    bar?: number
}

function morphFoo(foo: Foo): number {
    return foo.bar * 5;
}

```



This Is A Game Changer

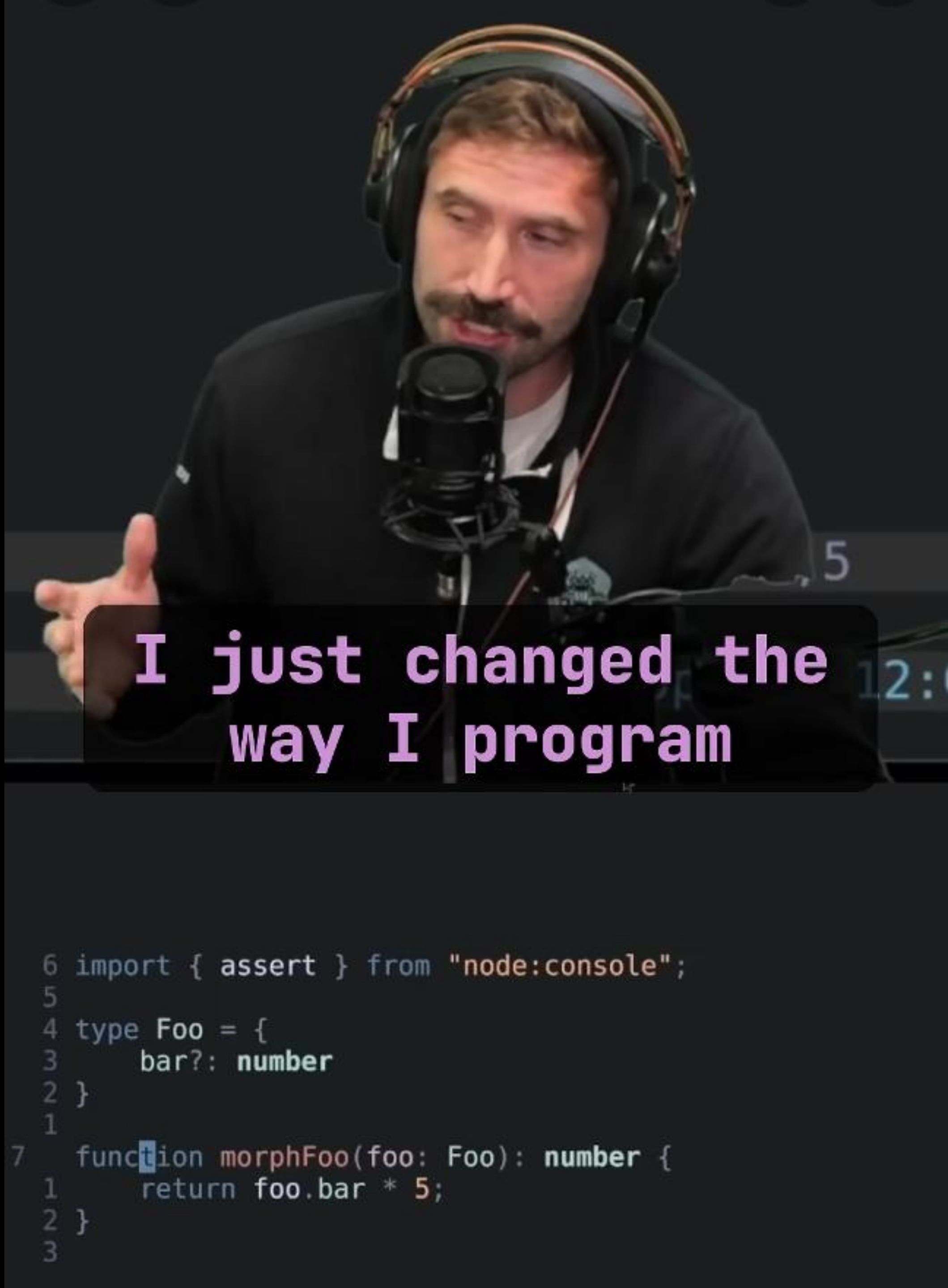
92K	3,858,988	2024
Likes	Views	Jun 5

A YouTube short I recently sent to the C++ committee…



This Is A Game Changer

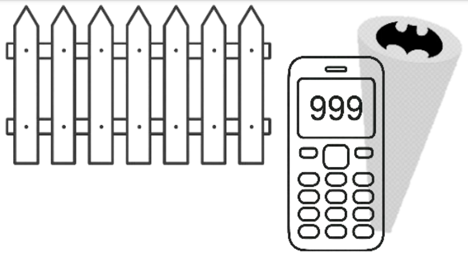
92K	3,858,988	2024
Likes	Views	Jun 5

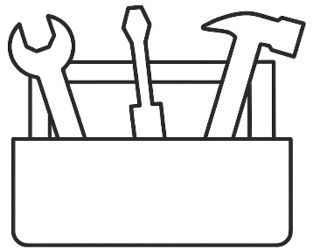


```
import { assert } from "node:console";

type Foo = {
    bar?: number
}

function morphFoo(foo: Foo): number {
    return foo.bar * 5;
}
```

Roadmap

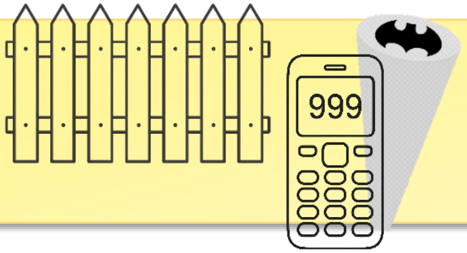
Motivation	Why defensive programming?
Overview	`pre`, `post`, `contract_assert`, `::handle_contract_violation()`
Best practices	"Effective C++26 Contracts"
Why ____?	Viable, ~minimal What each part enables
FAQs	Including open questions

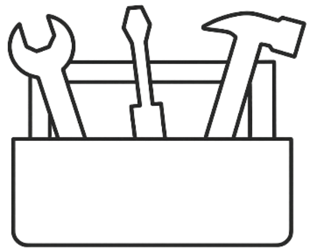
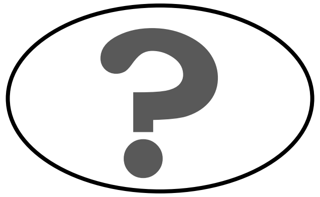




Roadmap

Motivation	Why defensive programming?
Overview	`pre`, `post`, `contract_assert`, `::handle_contract_violation()`
Best practices	"Effective C++26 Contracts"
Why ____?	Viable, ~minimal What each part enables
FAQs	Including open questions



A C***GPT prompt

prompt #2: “show me the chatgpt logo with amusing little red devil horns”

“Give an example C++ function that should use assert() in three different ways”

```
double calculateAverage(const std::vector<int>& numbers) {
  assert(!numbers.empty() && "Input vector must not be empty.");
  for (int num : numbers) {
    assert(num >= 0 && "All numbers must be non-negative.");
    sum += num;
  }
  double average = static_cast<double>(sum) / numbers.size();
  assert(average <= 100.0 && "Average exceeds maximum expected value.");
  return average;
}
```

Corrected + using C++26 contracts

In current draft C++26, we can write:

```
auto calculateAverage(std::span<int const> numbers) -> double
  pre(!numbers.empty() && "Input vector must not be empty.")
  post(ret: 0.0 <= ret && ret <= 100.0 && "Average outside expected range.")
{
  auto sum = 0;
  for (int num : numbers) {
    contract_assert(0 <= num && num <= 100 && "Numbers must be  in (0, 100).");
    sum += num;
  }
  return static_cast<double>(sum) / numbers.size();
}
```

The four horsemen (semantics)

Standard semantic	Check (evaluate)?	Call violation handler?	Terminate?
ignore	—	—	—
observe	✓	✓	—
enforce	✓	✓	✓
quick_enforce	✓	—	✓

Covers all four basic combinations

`::handle_contract_violation`

Optionally, you can supply a global handler (invoked only for observe or enforce)

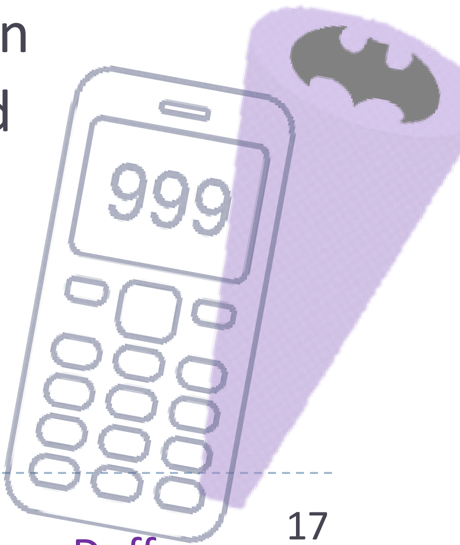
```
::handle_contract_violation( std::contracts::contract_violation const& ) noexcept_opt;
```

Parameter information

`location()`	QoI: for a precondition, should be the call site source_location
`comment()`	QoI: could be the pretty-printed predicate
`kind()`	`pre`, `post`, `contract_assert`
`semantic()`	`ignore`, `observe`, `enforce`, `quick_enforce` (but open-ended)
`detection_mode()`	predicate was false vs. predicate threw an exception
`is_terminating()`	whether a *standard* terminating semantic was used

Like all global replacement functions, provided:

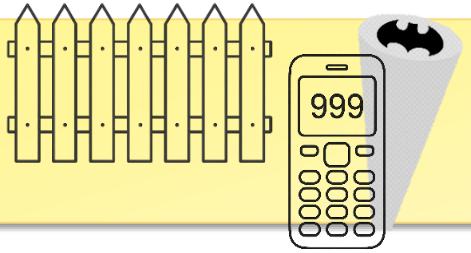
- By the whole program owner
- At link time (avoiding security issues)

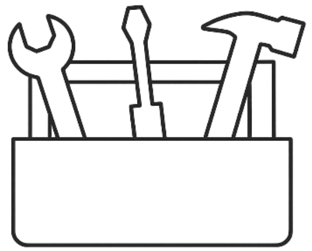


London, 1937
Don’t be Thomas Duffy

Roadmap

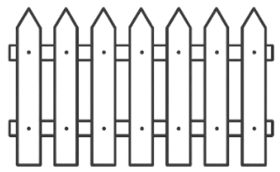
Motivation	Why defensive programming?
Overview	`pre`, `post`, `contract_assert`, `::handle_contract_violation()`
Best practices	"Effective C++26 Contracts"
Why ____?	Viable, ~minimal What each part enables
FAQs	Including open questions

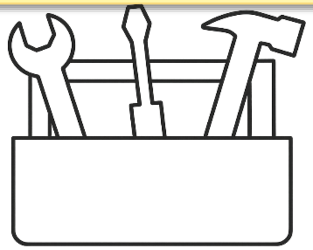


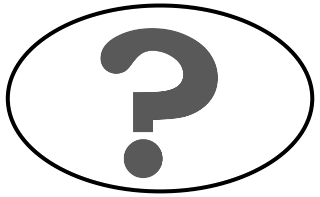


Roadmap

Motivation	Why defensive programming?	
Overview	`pre`, `post`, `contract_assert`, `::handle_contract_violation()`	
Best practices	"Effective C++26 Contracts"	✓ ✓ ☐
Why ____?	Viable, ~minimal What each part enables	
FAQs	Including open questions	







Interlude: Program bugs vs run-time errors

Say we're about to perform `vec[index]`

If index is **computed by the program**, a bad index is a **bug in the program code**

```
contract_assert(
  0 <= index && index < vec.size()
  && "our code computed a bad index"
);
```

The caller generally can't handle this or retry or anything sensible (short of a restart, which we'll discuss soon…)

The program is just wrong and needs to be fixed

If index is **read from input**, a bad index is a **user/data error**

```
if (index < 0 || index >= vec.size()) {
  /* error: "dear user, go fix your data" +
     report error/exception to the caller */
}
```

The caller can handle it (back out, retry, whatever is sensible)

assertions are not for run-time input validation

Item 0: Never use contracts/assertions for program logic

Example: Never use contracts for run-time input validation

Example: Never use contracts for normal program behavior

If you need control flow, use “if”

“The first thing to know about assertions … is they are redundant.

If you’re writing something that’s not redundant, don’t use an assertion.

This is a basic fact of information theory:

There is no error [here, program bug] *detection without redundancy.”*

— Lisa Lippincott (CppCon 2024)

Item 1: Avoid side effects in contract predicates

(Corollary of Item 0)

Bad: `contract_assert( ++x > 10 );` `// don't do this`

Just like all assertions

- C assert: Predicate will be executed zero or one times
- C++26: Predicate will be **executed zero or more times**

Good news:

- ✓ Most of these will be compile-time errors!
- Contract predicates treat variables as const — similar to lambda capture

Item 2: Avoid splitting compound conditions
For short-circuit eval you need to write the && !

```
Good: contract_assert( p && p->foo() );          // do this
Bad:  contract_assert( p );
      contract_assert( p->foo() );               // don't do this
```

With observe semantic:

- The second test is not protected by the first, and might be UB
- Not too surprising: You only get short-circuit evaluation for && when you write &&

Good news:

- ✓ Even in observe, the first check will be checked and tell you what you did wrong

Item 3: Use throwing violation handlers “judiciously”

Q: Can it be reasonable to install an exception-throwing violation handler?

A1 (default) Throwing handlers are mainly to **optimize testing contracts themselves**

A2 (advanced) Yes in a **specific scenario**, but be aware of **tradeoffs**

Scenario	We want to “enforce” contracts in production, but **can’t tolerate termination** Typical idea: “Failed subcomponent” unwind-and-restart
Beware	Injecting an exception during stack unwinding (or in noexcept fn) **causes termination** Nearly all code could be transitively called during stack unwinding (or in noexcept fn)
Suggestion	Run with `observe` + a violation handler like this *(disclaimer: experimental!)*

```
::handle_contract_violation( /*…*/ ) {
  /// anything else ///
  if (!uncaught_exception()) {
    throw restart_component_exception();
  }
}
```

Prior art: Ada, Eiffel, Java, and other languages throw for assertion violations, but don’t have C++’s unwinding-termination

Item 4: Understand whether/how build modes combine

Recall existing issue: “Debug” and “Release” builds aren’t link-compatible

- NDEBUG on/off is often a “benign” ODR violation, but…
- … _DEBUG (which affects ABI) and NDEBUG often vary together in real builds

Existing best practices

Best, no ODR issues:	Build everything in the same mode (including NDEBUG)
“Benign” ODR violation:	Test NDEBUG (incl. assert) in .cpp files only, not in inline/template .h code

Contracts offer an improvement, while still in today’s “TUs + linkers” world

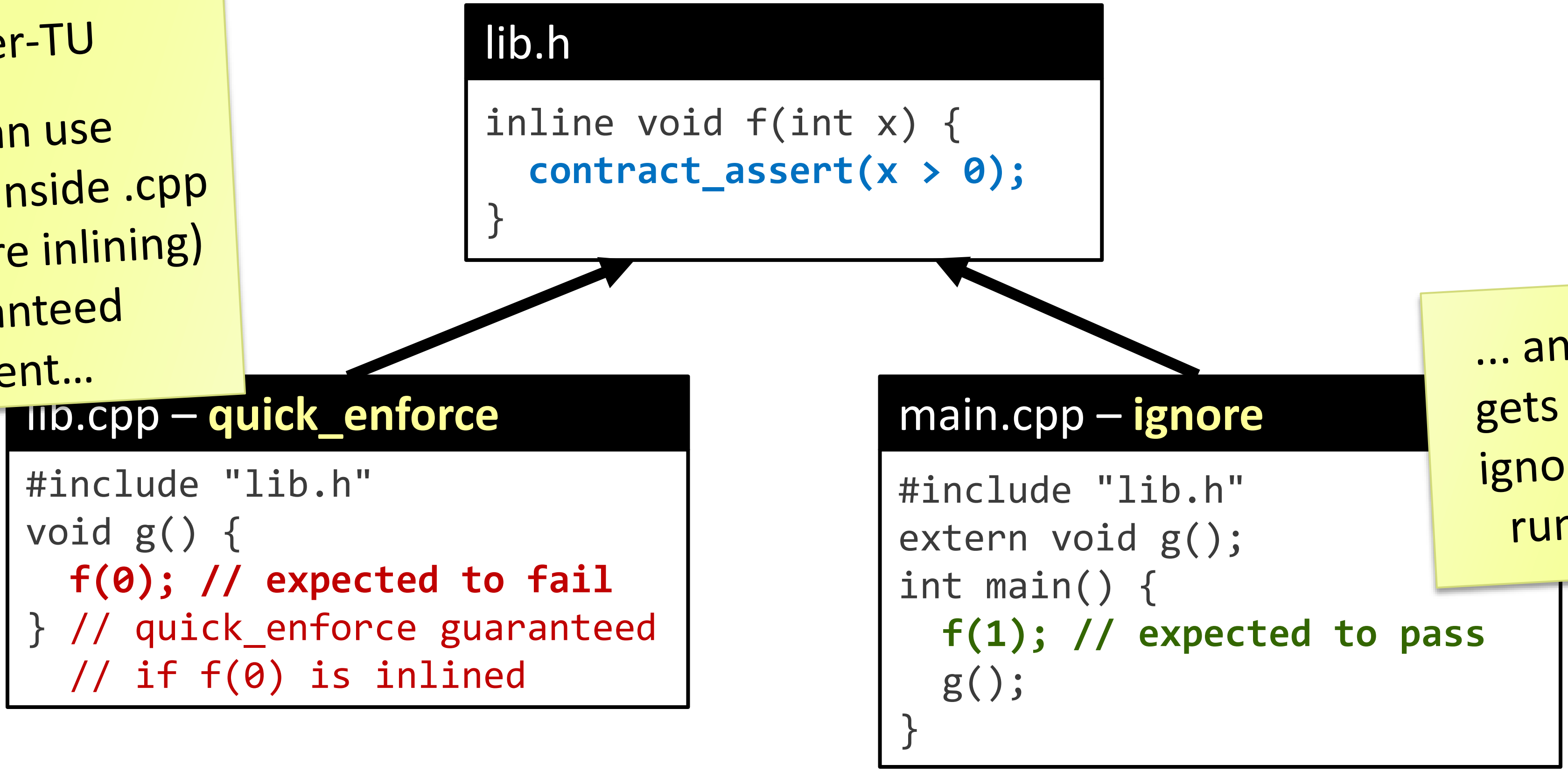
- Deliberately designed to allow mixing semantics (per-TU) without ABI/link incompat
- The current GCC and Clang contracts implementations allow linking TUs built with any configuration (without “benign” ODR violations = better than C assert)
- The standard allows platforms to document restrictions on if/when/how semantics mix

Item 4: Understand whether/how build modes combine

Consider this example (P3835)

You can guarantee local semantic per-TU

So lib.cpp can use contract_assert inside .cpp bodies (or ensure inlining) to get guaranteed enforcement…



lib.h

```
inline void f(int x) {
  contract_assert(x > 0);
}
```

lib.cpp – quick_enforce

```
#include "lib.h"
void g() {
  f(0); // expected to fail
} // quick_enforce guaranteed
  // if f(0) is inlined
```

main.cpp – ignore

```
#include "lib.h"
extern void g();
int main() {
  f(1); // expected to pass
  g();
}
```

… and main.cpp gets guaranteed ignore with zero run-time cost

if whole program has two copies of f(), linker picks one

Item 4: Understand whether/how build modes combine

To know whether you can have a program that combines TUs built with different semantics, check your compiler/platform docs

- Expectation: You probably can, but you still have to know about inlining/templates

If yes, a translation unit author (library or mainline) can get guaranteed semantics (checking, ignoring) for contract assertions **in their translation unit**:

- Code that appears in the .cpp file
- Code that appears in a .h file that you ensure is inlined (incl. templates)
- That's an important restriction you can't always meet, so be aware

Item 5: Understand C++26's v1.0 limitations

No custom error messages

- For static_assert similarly, we had to add them later

No contracts on virtual functions or function pointers

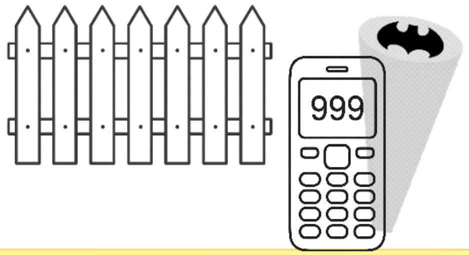
- Some people really want this urgently

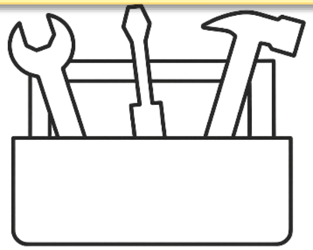
No contract groups/labels – e.g., a group for bounds violations, and a way to always enforce those

- Today: Per-TU semantics today (see Item 4), single global violation handler

Roadmap

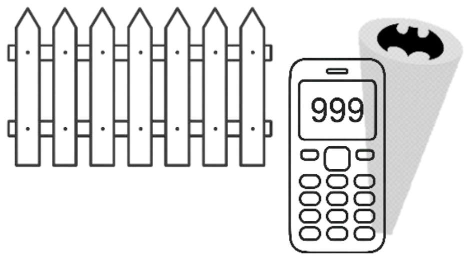
Motivation	Why defensive programming?	
Overview	`pre`, `post`, `contract_assert`, `::handle_contract_violation()`	
Best practices	"Effective C++26 Contracts"	✓ ✓ ☐
Why ____?	Viable, ~minimal What each part enables	
FAQs	Including open questions	

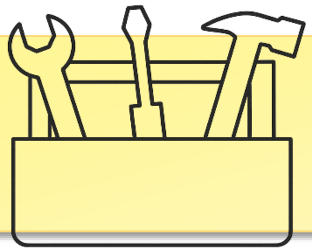




Roadmap

Motivation	Why defensive programming?
Overview	`pre`, `post`, `contract_assert`, `::handle_contract_violation()`
Best practices	"Effective C++26 Contracts"
Why ____?	Viable, ~minimal What each part enables
FAQs	Including open questions





Minimal: Addresses key requirements that assert() doesn't

Contracts	
pre / post	Major advance: Express assertions **on declarations**, lights up **analysis**
contract_assert	Better than a macro… 'nuff said
Semantics	
ignore	Essential
observe	Enables **checking in production**, and testing new predicates
enforce	Essential, for testing
quick_enforce	Enables **enforcing in production** with absolute min. size & perf impact
Violation handler	
default	Typically prints source file/line/col, message, and similar
and replaceable	Enables hooking into **your existing logging/alerting** mechanisms
… at link time	Avoids security issues of run-time replacement

Belt and suspenders…

Shift left

Bring more bug detection to test time, pre-merge

Safety net

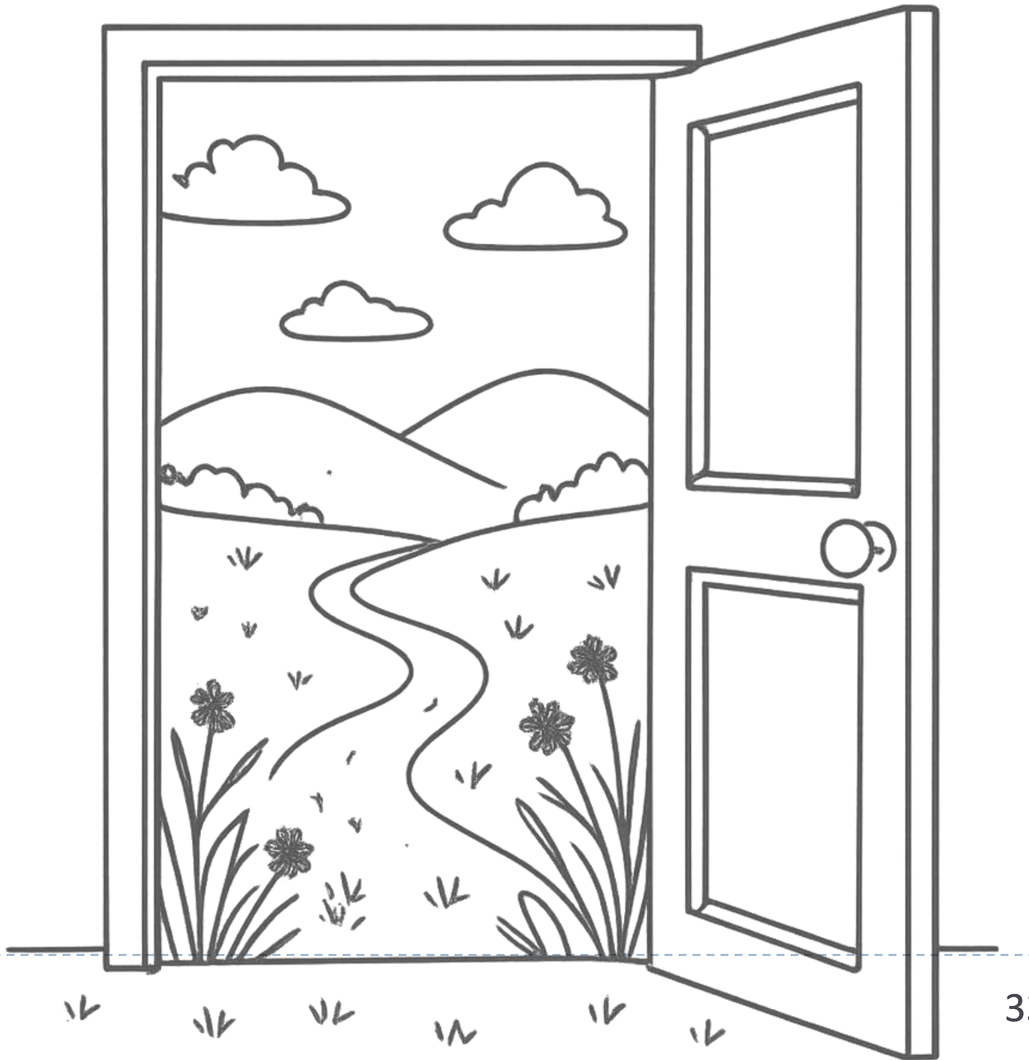
Also enable bug detection always, including production

Viable: Works, and doesn’t close doors

Guiding principle: Don’t close doors

Open doors include:

- Custom error messages
- Virtual functions
- Function pointers
- Labels/groups



Example: `noexcept_enforce`/`observe` (in GCC dev branch)

As a future extension example:

A semantic that makes throwing violation handlers nonthrowing

"The breakthrough realization, the epiphany, and the stroke of genius there is that instead of trying to cater to two different crowds by making the violation handler noexcept or not, we can serve two crowds with the same violation handler.

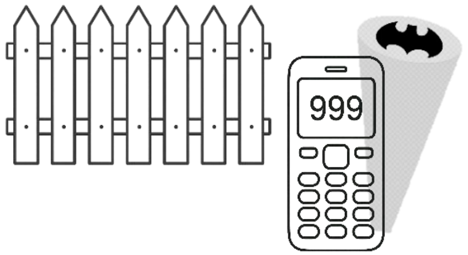
If it throws, the authors of components that don't want exceptions to escape get what they want. The authors of components that want exceptions to escape get what they want. The different decisions can be mixed in the same binary, and one violation handler does what both audiences want.

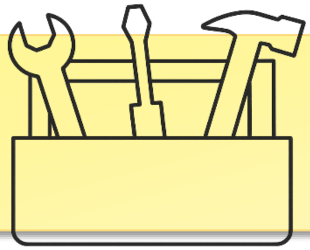
Author A compiles with noexcept_enforce, author B compiles with enforce. The same violation handler works for both. No linker magic, no tricks, just different compilation of the violation handler calls. Done in the wrapper functions we already have."

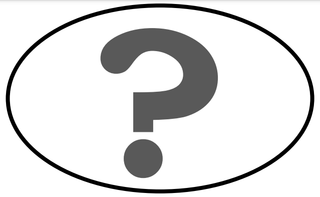
— Ville Voutilainen

Roadmap

Motivation	Why defensive programming?
Overview	`pre`, `post`, `contract_assert`, `::handle_contract_violation()`
Best practices	“Effective C++26 Contracts”
Why ____?	Viable, ~minimal What each part enables
FAQs	Including open questions

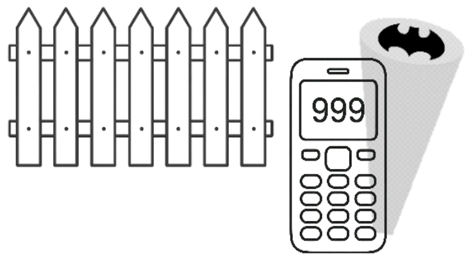


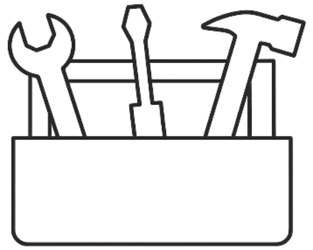




Roadmap

Motivation	Why defensive programming?
Overview	`pre`, `post`, `contract_assert`, `::handle_contract_violation()`
Best practices	"Effective C++26 Contracts"
Why _____?	Viable, ~minimal What each part enables
FAQs	Including open questions





Some quick FAQs

Q Do C++26 contracts have lots of implementation-defined behavior?

A They're things we usually make implementation-defined
(e.g., the standard doesn't specify /Compiler --switches, or build modes)

Q Are C++26 contracts hard to implement?

A GCC and Clang are complete, in code review, targeting next versions

Q Is it true that C++26 contracts don't help static analysis?

A

Catching Bugs Early: Validating C++ Contracts with Static Analysis

Thursday @ 15:15 MDT
Mike Fairhurst, GitHub CodeQL (Microsoft)
Peter Martin, Product Security (Bloomberg)

More quick FAQs

Q Is there a portable default behavior?

A C++26 says… ‘Enforce by default, print contract_violation information by default’ + “rate-limited for potentially repeated violations of observed contract assertions”

Current GCC message:

contract violation in function T ret(int) [with T = double] at mypath/myfilename:76: a > 1 [assertion_kind: pre, semantic: observe, mode: predicate_false, terminating: no]

Q Do contracts have a flaw where writing UB in code protected by a contract can silently eliminate another contract check by time-travel optimization?

A Not anymore, fixed for C++26

In a nutshell: “observable checkpoint,” treat a contract predicate like I/O

S’mores quick FAQs

Q What if I try to write a pre/post on a virtual? … a function pointer?

A Compile-time error in C++26; likely future extension post-C++26

C++11: `->auto` for lambdas … C++14: `->auto` for all functions

Q Isn’t making objects implicitly const in predicate checks weird?

A So far: Found actual bugs, and no realistic problem examples

Not novel in C++: Similar to implicit const in C++ lambda capture

Q Why aren’t C++26 contracts a drop-in replacement for [my assertion library]?

A1 A language feature beats macros, and won’t always work exactly like a macro

A2 Some examples break because they are *already const-incorrect* (see previous)

Open questions

1a A predicate that throws is converted to a contract violation

1b (related) Implementation and deployment experience on non-Itanium ABIs

Q: Are those **good semantics**?

Q: Is the run-time **cost acceptable**? Need to measure.
(*The hopeful conjecture: If most predicates are trivially non-throwing, we can elide most notional injected try/catches?*)

Suggested extension: Evaluation semantics that don't translate predicate exceptions into contract violations

These suggested extensions can likely be added post-26, e.g., using an additional semantic or a "labels" grouping feature

2 We can't yet write functions that don't exhibit undefined behavior when preconditions are violated

Suggested extension: A way to express a precondition check that can never be ignored, no matter which compiler flags are used (example: `pre<quick_enforce> (index < size())`)

More open questions

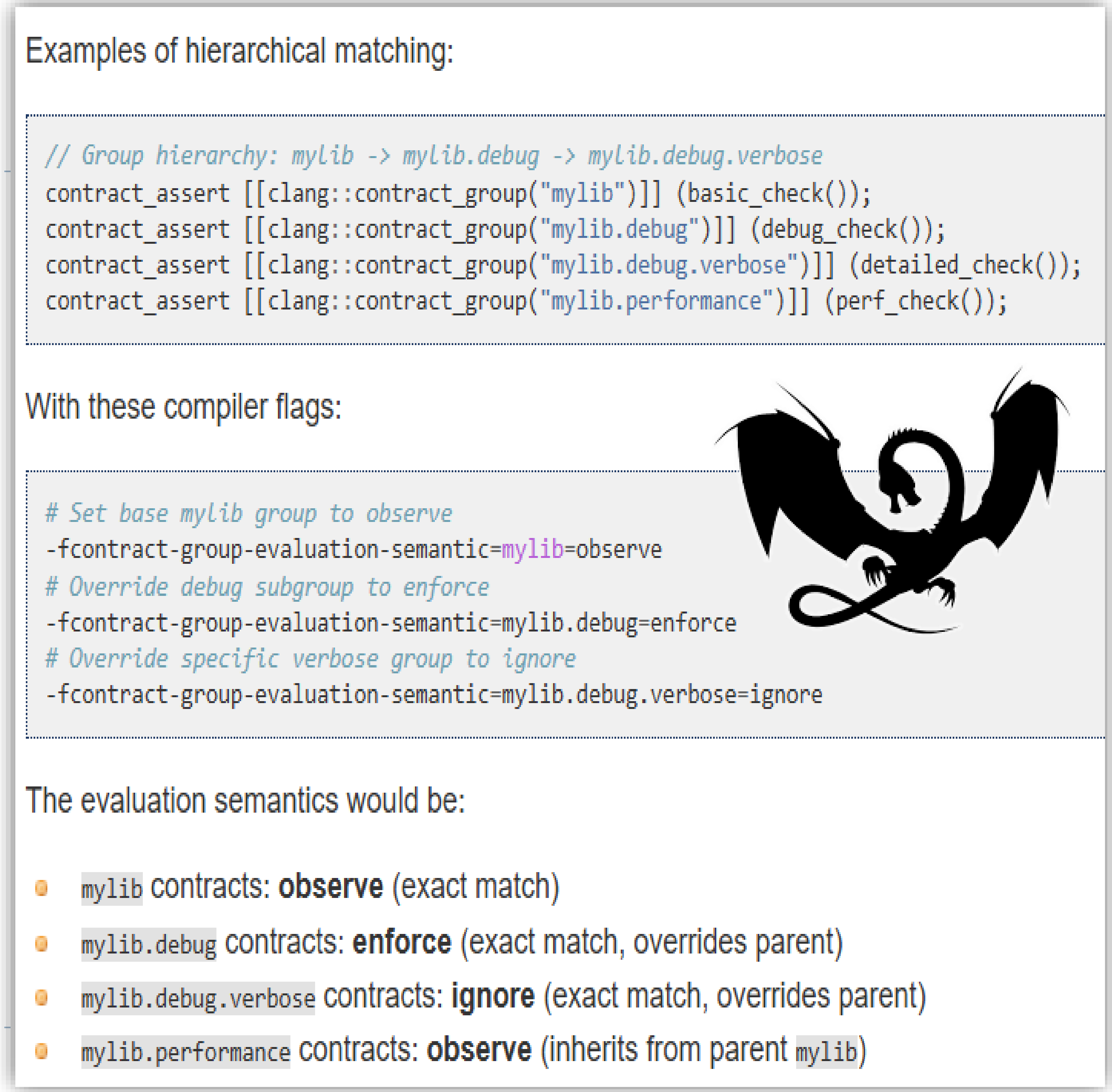
3 **Should we hold v1.0 from the standard till we can add { groups/labels, virtuals } ?**

Groups: to allow grouping of contracts with a similar purpose, or tied to a component

Virtuals: with substitutability?

Note the tension:
"v1.0 must be strictly minimal" +
"but add one feature I need"

Clang is providing groups as a vendor extension →



Examples of hierarchical matching:

```
// Group hierarchy: mylib -> mylib.debug -> mylib.debug.verbose
contract_assert [[clang::contract_group("mylib")]] (basic_check());
contract_assert [[clang::contract_group("mylib.debug")]] (debug_check());
contract_assert [[clang::contract_group("mylib.debug.verbose")]] (detailed_check());
contract_assert [[clang::contract_group("mylib.performance")]] (perf_check());
```

With these compiler flags:

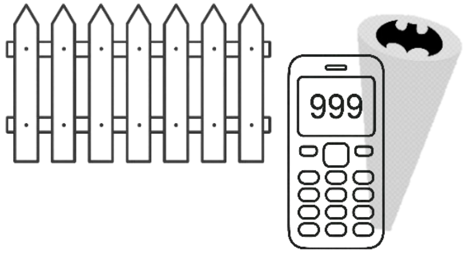
```
# Set base mylib group to observe
-fcontract-group-evaluation-semantic=mylib=observe
# Override debug subgroup to enforce
-fcontract-group-evaluation-semantic=mylib.debug=enforce
# Override specific verbose group to ignore
-fcontract-group-evaluation-semantic=mylib.debug.verbose=ignore
```

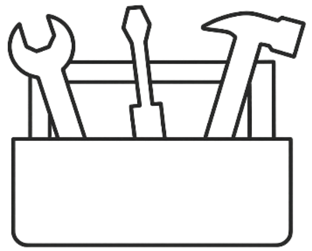
The evaluation semantics would be:

- `mylib` contracts: **observe** (exact match)
- `mylib.debug` contracts: **enforce** (exact match, overrides parent)
- `mylib.debug.verbose` contracts: **ignore** (exact match, overrides parent)
- `mylib.performance` contracts: **observe** (inherits from parent `mylib`)

Roadmap

Motivation	Why defensive programming?
Overview	`pre`, `post`, `contract_assert`, `::handle_contract_violation()`
Best practices	“Effective C++26 Contracts”
Why ____?	Viable, ~minimal What each part enables
FAQs	Including open questions





The Joy of C++26 Contracts

(and Some Mythconceptions)

HERB SUTTER

CITADEL | Securities

Cppcon
The C++ Conference

2025

September 13 - 19