Why I'm slowly becoming a fan of C++ Contracts

Klaas van Gend · 040coders.nl · 29 October 2026

... and ... WTF happened last week?

SIOUX TECHNOLOGIES

What's wrong with `assert`?

```
using Callable = std::function<void()>;

void putOnUIThread(Callable action)
{
    assert(action != nullptr);
    asyncInvoke(ThreadId::UI, std::move(action));
}
```

Assume `action` is `nullptr`.

Quiz: *What happens if you run the above code?*

- *Debug?*
- *Release?*

The answer depends on the build

Build	`assert` does
Without `NDEBUG`	Checks; a failure aborts
With `NDEBUG`	Expands away; the expression is not checked

Quiz: *And if the failure only happens in production?*

- The check that helped in testing may be gone.
  The API requirement is still undocumented at the declaration.

What if the API said what it requires?

```
void putOnUIThread(Callable action)
    pre (action != nullptr)
{
    asyncInvoke(ThreadId::UI, std::move(action));
}
```

The caller is responsible for meeting the precondition.

Quiz: *Is this better than silently doing nothing?*

- **Yes:** a null action is a caller bug.
  The *contract* documents it in the interface
  and reports the violation.
  Better: a `std::stacktrace` points to the caller.

Contracts: three places to state a condition

buffer.h:

```
class Buffer {
public:
    double at(std::size_t index) const
        pre (index < size())                  // caller's duty
        post (value: value == data_[index]); // my promise
};
```

buffer.cpp:

```
double Buffer::at(std::size_t index) const {
    contract_assert(data_ != nullptr);       // my own sanity check
    return data_[index];
}
```

- `pre` / `post` go on the *declaration*: they are part of the **interface**
- `contract_assert` goes in the *body*: it checks the **implementation**

Which one belongs where?

```
double Buffer::at(std::size_t index) const
    pre (index < size())
    post (value: value == data_[index])
{
    contract_assert(data_ != nullptr || size() == 0);
    return data_[index];
}
```

Quick quiz

- Could a caller violate this condition?
- Could the implementation get this condition wrong?

Four evaluation semantics

Semantic	Condition	Violation handler	Then
`ignore`	No	No	Continue
`observe`	Yes	Yes	Continue, if the handler returns
`enforce`	Yes	Yes	Terminate
`quick-enforce`	Yes	No	Terminate immediately

Quiz: *Which one reports a violation and then carries on?*

- `observe`

Same source, different behavior

```
int divide(int numerator, int denominator)
    pre (denominator != 0)
{
    return numerator / denominator;
}
```

- `ignore` : no runtime check
- `observe` : handler is called; execution *may* continue
- `enforce` / `quick-enforce` : a violation terminates

The semantic is implementation-defined in C++26.

The violation handler?

```
#include <contracts>
#include <cstdio>
#include <print>

void handle_contract_violation(const std::contracts::contract_violation& v) {
    std::println(stderr, "{}:{}: violated `{}`",
                 v.location().file_name(), v.location().line(), v.comment());
}
```

- If handler replacement is supported, define this function; no registration needed.
- `comment()` and `location()` are implementation-defined; the comment may be empty.
- `v` also offers `kind()`, `semantic()` and `is_terminating()`.
- If the handler returns: semantic `observe` continues; semantic `enforce` terminates.

Quiz: *May the handler throw?*

- Yes — bypassing `enforce` termination. For `pre` / `post`, it is as if the function body exits with the exception. If it escapes a `noexcept` handler or checked function, `std::terminate` is called.

What if a predicate throws?

An exception escaping a `pre`, `post` or `contract_assert` predicate is a **contract violation**, not an ordinary exception propagation.

Semantic	What happens?
`ignore`	Predicate is not evaluated
`observe`	Handler runs; if it returns, the exception is discarded and execution continues
`enforce`	Handler runs; if it returns, the program terminates
`quick-enforce`	Program terminates without calling the handler

Quiz: *What happens if the handler rethrows the original exception?*

- With `throw;`, it escapes as if the function body exited (`pre` / `post`), or from the assertion statement (`contract_assert`).
  This bypasses `enforce` termination.

Contracts can be checked at compile time too

```
constexpr int twice(int value)
    pre (value >= 0 && value < 1000)
{
    return 2 * value;
}

static_assert(twice(21) == 42);
static_assert(twice(-5) == -10);
```

Quiz: *Which line fails to compile?*

- The second one — *if* its precondition is evaluated.

Not just `assert` with new spelling

C-style `assert` macro	C++26 Contract assertion
Removed by preprocessing with `NDEBUG`	Still parsed when ignored
May hide ill-formed code in Release	Predicate must be valid C++
Usually lives in the body	`pre` / `post` are part of the declaration
Project-specific macro conventions	Standard syntax for tools to inspect

> An ignored contract is still part of the program's source and interface.

Why did this take so long?

2004	Separate blocks	`in { ... } do { ... }`
2018	C++20 draft	`[[expects audit: x > 0]]`
2019	*Contracts* are removed before C++20 is finalized	
2022	Attribute-like syntax	`[[pre: f(a) < a]]`
2025	C++26 draft	`pre (x >= 0)`

Twenty+ years of arguments...

- and the design still had opponents...

What changed my mind?

I used to think:

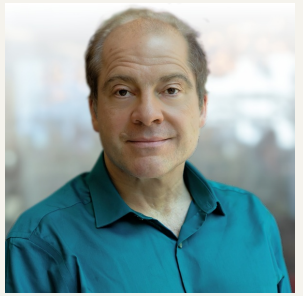
> “A camel is a horse designed by committee.”

Then I heard John Lakos explain the *journey toward backward-compatible, UB-safe C++*.

accu on sea

John Lakos' abstract

youtube



The key shift for me:

- **Contracts are not only about aborting.**
- **They also make API intent explicit.**

Why I'm becoming a fan

What does this function expect?

```
Image crop(Image const&, Rect);
```

Now the requirement is part of the interface:

```
Image crop(Image const& image, Rect area)
    pre (area.fits_inside(image.bounds()));
```

- *Callers* can see the requirement where they use the API.
- *Documentation tools* can display it beside the signature.
- *Analysis tools* can consume a condition instead of guessing from comments.

Comments can drift. Code is harder to ignore.

```
/// @note `count` must be positive
int average(int sum, int count);
```

The comment can get stale.

A *contract* is parsed with the declaration:

```
int average(int sum, int count)
    pre (count > 0);
```

This is useful documentation and a possible check.

But *not* a proof that every call is correct.

Contracts are great!

Why do people disagree?

Same feature, different priorities

Common ground before the disagreement

Both camps:

- Want contracts in C++.
- Know the C++26 design is incomplete.
- Agree that hardened library checks must not report failure and then continue into the operation they protect.

But what does “safety” mean here?

Two kinds of safety

- **Functional safety:** the whole system behaves safely in its real context.

  ```
  CHECK(door_is_locked());
  ```

- **Language safety:** memory safety, type safety, avoiding undefined behavior.

  ```
  CHECK(index < values.size());
  ```

Contracts may help with *correctness* and *functional safety*
without guaranteeing *language safety*.

Two preconditions, two different failures

```
double discounted(double price, double percent_off)
    pre (percent_off >= 0 && percent_off <= 100);
```

- If ignored, a bad discount may give a wrong —but still defined— price.

```
int average(int sum, int count)
    pre (count > 0)
{
    return sum / count;
}
```

- If ignored with `count == 0`, the division has *undefined behavior*.

The central question

Should the *build* be allowed to switch off either check?

The adoption argument

- If checks are always on, teams may omit them or remove them to meet performance goals.
- A configurable check can be deployed where its cost is acceptable.

The guarantee argument

- If a check may be ignored, it cannot reliably prevent undefined behavior.
- A declaration can look like a guarantee even when the build decides not to check it.

What can the author guarantee in C++26?

```
void resize(Image& image, std::size_t width)
    pre (width > 0);
```

> C++26 has no standard way to *require* this contract check in every build.

Labels and mandatory contracts have been proposed for later work.

They are not part of this C++26 design.

Is a contract just another `assert`?

```
int calls = 0;

void enter()
{
    contract_assert(++calls == 1);
    do_work();
}
```

Quiz: *Is changing `calls` inside the predicate safe?*

- **NO**. Contract checks may be skipped or evaluated more than once.
  Predicates should not rely on side effects or exactly-once evaluation.
  - similar to `assert` ?!?

The predicate itself can be dangerous

```
int add_100(int value)
{
    return value + 100; // signed overflow is undefined
}

int use(int value)
    pre (add_100(value) > value);
```

If the predicate has undefined behavior, the compiler may reason from the defined cases and remove the check

- even with a terminating semantic.

A check cannot automatically make its own expression safe.

More objections, fit to one slide

- **Separate builds:**
  inline code may be compiled with different contract settings; optimizer assumptions matter.
- **Exceptions:**
  if evaluating a predicate throws, the exception is treated as a contract violation.
- **One handler:**
  the program has one violation handler, not a handler per library.
- **`const` in predicates:**
  can change overload resolution, but does not prevent mutation through a pointer.

These are concrete design questions, not reasons to dismiss either side.

The strongest case *against* shipping now

> The default behavior becomes a compatibility commitment.

- The design has no portable always-check mode.
- The predicate can itself have undefined behavior.
- Declaration-level contracts have less real-world experience than body assertions.
- Missing features could constrain future designs.

“Add it later” is not cost-free once programs depend on today's defaults.

The strongest case *in favor of* shipping now

> A useful first version can create the experience needed to improve it.

- Contracts make requirements visible in declarations now.
- Configurable checking may encourage teams to add checks they would otherwise omit.
- Existing assertion migrations and library-hardening work offer some relevant experience.
- Some extensions may be added later without changing the basic syntax.

“Wait for a complete design” can also mean waiting indefinitely.

How much experience is enough?

Evidence for the design

- Implementations and experiments exist.
- One industrial report describes migration across about 2.2 million lines and 17 000+ assertions.
- Library-hardening modes already exercise several checking behaviors.

Important limit

- Most experience is with `contract_assert` replacing assertion macros.
- The newer declaration-level `pre` / `post` use is less tested.
- Some runtime costs are still unmeasured.

What did C++26 leave for later?

- Contracts on virtual functions
- Source-level control of evaluation semantics
- Capturing old values in postconditions
- User-defined violation messages

The camps agree these are gaps.

They disagree on whether the first version is a good foundation or a premature commitment.

But it's in the standard now?

C++26 was agreed on, right?

How is the C++ standard made?

- Every change starts as a **paper** (`Nnnnn` or `Pnnnn`) that *anyone* may write and submit.
- ISO WG21 **meets** about 3x/year (some in person, some online or hybrid),
  - plus teleconferences in between.
- **Attending:** through your **national body** (in the Netherlands: NEN)
  - or by invitation — there is no open ticket sale.

Decisions: paper → study group (SG21 for contracts) → evolution / library evolution → wording review → **plenary vote** by consensus.

Contracts: touched by **280 papers***, as of 23 October 2026 — more than 20 years of work.

From WG21 vote to national ballot

- **23–28 March 2026, Croydon, UK:**
  the WG21 plenary votes to send the C++26 draft to ISO for the DIS ballot.
  - **114 in favor, 12 opposed, 3 abstain**
  - The draft itself is a paper too: the editors' working draft, N5050.

- **1 June 2026:** editors publish N5050, the final C++26 draft, after editorial review.
- The convener transmits it to ISO, and the **National Bodies** vote on it.
  - The ballot **concluded on 22 October 2026**.

What does the DIS ballot decide?

The **Draft International Standard** ballot asks the *National Bodies* whether the draft can proceed toward publication.

Approval needs both:

1. At least **two-thirds** of votes cast by voting (P-)members in favor.
2. No more than **one-quarter** of all votes cast negative.

This is a vote on the **whole C++26 draft**, not a contracts-only referendum.

- So far, the ballot was always very positive.

Ballot quiz: can a large majority still fail?

The committee sent C++26 to ballot with strong support

- 114 in favor, 12 opposed, 3 abstained.

Quiz: *What if 9 of 22 national bodies vote "No"?*

- Is that within the one-quarter limit?
- What if only 12 of 20 voting members vote Yes?

The result: C++26 DIS failed

Condition	Needed	Result
Yes votes among voting members	At least 2/3	12 of 20 = 60% ✗
No votes among all bodies voting	At most 1/4	9 of 22 = 41% ✗

Both thresholds were missed.

Contracts were the focus of the recorded opposition, but the ballot rejected the whole draft.

So, what happens next?

After a failed DIS ballot, the process allows:

- A revised draft and another enquiry ballot
- A revised committee draft for comments
- A technical specification route
- Cancellation of the project

As of 26 October 2026, the next route had not been decided.

> The vote is a setback, not the end of the contracts discussion.



Where I stand

I, Klaas, want contracts in C++.

- A precondition on the declaration is valuable API documentation.
- A contract can be checked where the build enables checking.
- Machine-readable intent is useful to people **and tools**.

And I understand the opposition:

- Build-dependent checking is not a guarantee.
- The predicate and deployment model have real hazards.
- The feature needs more implementation and usage experience.

C++26 contracts: what should happen?

1. **Ship it as is:** keep contracts in C++26 unchanged.
2. **Fix it first:** revise the contracts design, then ballot again.
3. **Take it out, aim for C++29:**
   remove contracts from C++26 and continue the work for the next standard.
4. **Take it out, publish a TS:**
   remove from C++26 and release as a technical specification (TS) to gain experience.

- On Oct 29, approx 35 people in the 040coders audience voted roughly 0%, 20%, 40%, 40%, respectively.

A few recent sources to keep exploring

Papers and talks

- P2900R14: the C++26 contracts proposal
- N5050: the final* C++26 draft
- P3506R0: P2900 Is Still Not Ready for C++26 (Microsoft)
- P4428R0: Contract concerns and where they are addressed
- P4433R0: Do not remove Contracts from C++26

- John Lakos at ACCU on Sea (video)

Mateusz Pusz' blog posts

- C++26 contracts: how they work
- Where both sides agree
- Who decides whether a check runs?
- What can go wrong inside a predicate?
- Experience and committee process
- The DIS ballot result