Andrei Zissu

C++ Contracts
A Meaningfully Viable Product

Who am I?

Andrei Zissu

Morphisec – preventive cybersecurity + anti-ransomware

- WG21 member for the last 3 years
- Active in SG21
  - Not one of the main authors of the Contracts proposal (P2900)

What This Talk Is About

- A moderately deep dive into contracts in C++26
- With some detail on considerations, controversies and paths not taken
- I'll strive to make the terminology easily accessible
  - Not always stick to the official terms
- I'll try not to focus on theory, but rather on contracts machinery
  - But will sometimes present the important parts
  - Including relevant P2900 guiding principles, presented in proper context
- I will be borrowing heavily from P2900 (the C++ contracts proposal, previously known as the contracts MVP) and P2899 (the rationale paper)
  - Occasionally other papers will also be mentioned and possibly borrowed from
- Not nearly enough time to cover everything I would have wanted to

Some quick terminology

Precondition Specifiers

```
1   auto div(auto x, auto y)
2       pre(y != 0)
3   {
4       return x/y;
5   }
6
7   int main()
8   {
9       return div(1, 0);
10  }
```

Output of x86-64 gcc (contracts natural syntax) (Compiler #1)

A ▾ Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 139
  contract violation in function div<int, int> at /app/example.cpp:2: y != 0
  [assertion_kind: pre, semantic: enforce, mode: predicate_false, terminating: yes]
  terminate called without an active exception
  Program terminated with signal: SIGSEGV
```

- “Precondition” is a plain language term, while “precondition specifier” refers to a syntactic construct.

Postcondition Specifiers

```
1   auto plus(const unsigned x, unsigned y)
2       post(r: r >= x)
3   {
4       return x - y;
5   }
6
7   int main()
8   {
9       return plus(1, 1);
10  }
```

Output of x86-64 gcc (contracts natural syntax) (Compiler #1)

A ▾ Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 139
  contract violation in function plus at /app/example.cpp:2: r >= x
  [assertion_kind: post, semantic: enforce, mode: predicate_false, terminating: yes]
  terminate called without an active exception
  Program terminated with signal: SIGSEGV
```

- The term “postcondition” is used in everyday language, whereas “postcondition specifier” denotes a specific syntactic element.
- *r* is directly naming the function return value – we’ll cover this later
- Also keep the const parameter in mind for later…

Function Contract Specifiers

- The collective term for all pre and post specifiers (for preconditions and postconditions)
- I will often refer to them simply as pre and post

Assertion Statements (a.k.a contract_assert)

```cpp
auto plus(unsigned x, unsigned y)
{
    auto ret = x - y;
    contract_assert(ret >= x);
    return ret;
}

int main()
{
    return plus(1, 1);
}
```

Output of x86-64 gcc (contracts natural syntax) (Compiler #1)

A ▾ Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 139
  contract violation in function plus at /app/example.cpp:4: ret >= x
  [assertion_kind: assert, semantic: enforce, mode: predicate_false, terminating: yes]
  terminate called without an active exception
  Program terminated with signal: SIGSEGV
```

Contract Assertions

- The collective term for the syntactic construct specified by function contract specifiers (pre and post) and assertion statements
- Yes, this includes pre and post, while contract_assert is referred to only as “assertion statement”
- For convenience I will be referring to them mostly in short as “CA” – short for “contract assertion” (which is a bit of a mouthful)

Contract Predicates

- The C++ code inside a contract assertion
- Must be contextually convertible to bool
- We can usually afford to mix and match the terms “contract assertion” and “contract predicate”
- But occasionally they do have important distinguishing differences
- Like in ignore semantics – CA is “evaluated”, predicate is not

Some Bikeshedding History
(or whatever you choose to name it)

Contract Assertions

- Started off as CCA (Contract-Checking Annotation)
- Renamed since some committee members didn’t think the term “annotations” is appropriate
- Still being informally used (myself included)
- Also proposed: CAA (Contract-Assertion Annotation) - didn’t catch on
- I proposed "contract clauses”

contract_assert

- Other languages use “assert”
- But that would conflict with the existing C assert macro
- So pending a future solution, we needed another name
- Over 40 alternatives were collected and explored in P2961R2 - *A natural syntax for Contracts*

Bikeshedding – 1st Wheel

contractassert	mustexpr	dyn_assert	musthold	asrtexpr	stdassert
truexpr	co_assert	ccassert	contract_assert	std_assert	dyn_check
mustbetrue	assertexpr	assertion_check	cppassert	dynamic_assert	cca_assert
assrt	runtime_assert	_Assert	xpct	assert_check	Assert2

Bikeshedding – 2nd Wheel

cpp_assert	affirm	__assert	assess	insist	asrt
cassert	aver	posit	enforce	audit	claim
ass	must	confirm	assertion	ensure	chk
verify	expect	check			

- Finalists - contract_assert and assertexpr
- contract_assert won the day
- I voted against assertexpr as it resembles constexpr, which is used only in declarations
- Given that contract_assert is currently a statement and not an expression (see later) I’m particularly glad we chose this name

Syntactic restrictions, and how they came to be

Multiple Declarations – because we love IFNDR!

- Pre and Post must always be declared on function first-declarations
- First declaration - “a function from which no other declaration is reachable”
- i.e. 1st declaration found by compiler in current TU (a.k.a .cpp file)
- Otherwise it’s a “redeclaration”
  - That includes function body when defined separately from first declaration
- Term used but not defined by the standard – until P2900 that is
- Contracts repetition in function redeclarations is allowed (not mandatory, since compilers don’t need them) for several reasons:
  - User friendliness
  - Functions declared in multiple headers – we wouldn’t want include order to matter (never feed the include hell trolls!)
  - Friend declarations may be instantiated in any order

Multiple Declarations – because we love IFNDR!

- Repeated contract ill-formed if different from one on first declaration
- Equivalence is determined mostly by token equality (like ODR)
- But allows for different names for function and template parameters
- As long as the contract specifications do refer to the same entities
- Redeclarations in same TU cannot contain lambdas - would automatically render them “not the same” due to different type for each lambda

Multiple Declarations – because we love IFNDR!

- Well-formed renaming:

```
1   struct C
2   {
3       double div(double x)
4           pre(x != 0);
5   };
6
7   double C::div(double y)
8       pre(y != 0)
9   {
10      return y;
11  }
12
13  int main()
14  {
15      return C{}.div(1);
16  }
```

Multiple Declarations – because we love IFNDR!

- Ill-formed renaming:

```
1   struct C
2   {
3       double div(double x, double y)
4           pre(y != 0);
5   };
6
7   double C::div(double x, double y)
8       pre(x != 0)
9   {
10      return x/y;
11  }
12
13  int main()
14  {
15      return C{}.div(1, 1);
16  }
```

```
Output of x86-64 gcc (contracts natural syntax) (Compiler #1)
A  Wrap lines  Select all
<source>:8:11: error: mismatched contract condition in declaration
    8 |     pre(x != 0)
      |         ~~^~~~
<source>:4:15: note: previous contract here
    4 |         pre(y != 0);
      |             ~~^~~~
Compiler returned: 1
```

Semantics

Constification
– our favorite bowel non-movement

- Officially called “implicit const-ness”
- Often auto-corrected to “constipation”
- Main motivation:
  - Catch bugs like “assert(my_map[“universal_answer”] == 42)”
  - Avoid contracts producing side effects, whether inadvertently or intentionally

Principle 6: No Destructive Side Effects

Contract assertions whose predicates, when evaluated, could affect the correctness of the program should not be supported.

Constification
– our favorite bowel non-movement

- Effect:
- External entities referenced by CAs are treated similarly to data members in const member functions
  - *decltype* still reports the original constness - same as in const member functions
- Implicit const is shallow (same as any other const in C++)
  - Objects pointed to by pointers are **not** constified
  - Objects referred to by reference **are** constified
- Globals were originally not included in constification – this was changed later to improve teachability

```
int global = 0;

void f(int x, int y, char *p, int& ref)
  pre((x = 0) == 0)             // error: assignment to const lvalue
  pre((*p = 5))                 // OK
  pre((ref = 5))                // error: assignment to const lvalue
  pre((global = 2))             // error: assignment to const lvalue
{
  int* gp = &global;
  contract_assert((gp = nullptr)); // error: assignment to const lvalue
  contract_assert((*gp = 6));      // OK
}
```

Constification
– our favorite bowel non-movement

- Controversial opt out - *const_cast*
  - Must be used with utmost care, as the outcome is UB if the object was originally const

```
int g(int i, const int j)
  pre(++const_cast<int&>(i))     // OK (but discouraged)
  pre(++const_cast<int&>(j))     // undefined behavior
  post(++const_cast<int&>(i))    // OK (but discouraged)
  post(++const_cast<int&>(j))    // undefined behavior
{
  int k = 0;
  const int l = 1;
  contract_assert(++const_cast<int&>(k));   // OK (but discouraged)
  contract_assert(++const_cast<int&>(l));   // undefined behavior
}
```

Constification
– our favorite bowel non-movement

- Objections:
  - Modifies overload resolution, therefore contracts may actually be evaluating different code

```
struct X {};
bool p(X&) { return true; }
bool p(const X&) { return false; }

void my_assert(bool b) { if (!b) std::terminate(); }

void f(X x1)
  pre(p(x1))          // fails
{
  my_assert(p(x1));   // passes

  X x2;
  contract_assert(p(x2)); // fails
  my_assert(p(x2));       // passes
}
```

Constification
– our favorite bowel non-movement

- Objections:
  - Modifies overload resolution, therefore contracts may actually be evaluating different code
    - Counter - such overloads are bugs on their own
      - Counter counter (also given in other cases) - contracts must be useful with real world code, not educate programmers
        - Counter counter counter - contracts should indeed also encourage correct programming practices
    - Counter – constification is worth its shortcomings:
      - Side effects in contracts create a different program – rendering such contracts potentially meaningless/harmful
      - May help discover latent side effects in existing assert statements

Constification
– our favorite bowel non-movement

- Objections:
  - Shallow const
    - Constification is incomplete and inconsistent
    - In particular, pointers and references are inconsistent with each other
    - Counter - pointer/reference inconsistency is an existing language feature
    - Counter - automatic deep const would be very complex to specify and implement

Constification
– our favorite bowel non-movement

- Objections:
  - Constification blocks some common non-const usages, such as logging and std::map::operator[]
    - Counter - blocking std::map::operator[] is actually a good thing, as it can indeed modify the map; std::map::at() can be used instead
      - Not so with things like logging though
    - Counter - const_cast as a controversial escape hatch
    - Other proposed escape hatches: *mutable* (applied to CA), *operator noconst* (applied to expression in CA)

```
// Legacy API, check_valid doesn't modify x but isn't const-qualified
struct X;
bool check_valid(X& x);

void f(X& x)
  pre(check_valid(x)); // Ill-formed due to implicit const on x

void g(X& x)
  pre(noconst(x).is_valid()); // OK: noconst(x) treats x as non-const for this expr
```

Postconditions

Postconditions- Referring to the Result Object – what is a name?

- Post may specify a name for the return value, valid only within the post itself
- Works even when the function returns an unnamed temporary - no other way in the language to do this for all objects
- Although it is feasible for user-provided constructors, but only where those are available
- That name will capture the result object “on the fly”

Postconditions - Referring to the Result Object – what is a name?

- Similar to references but not really one, also compared to structured bindings
- Main difference is that decltype doesn't see it as a reference

```cpp
#include <type_traits>

int f() post(r: std::is_reference_v<decltype(r)>)
{
    int x;
    int& ref = x;
    static_assert(std::is_reference_v<decltype(ref)>);
    return 42;
}

int main()
{
    f();
    return 0;
}
```

Output of x86-64 gcc (contracts natural syntax) (Compiler #1)

Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 139
   contract violation in function f at /app/example.cpp:3: std::is_reference_v<decltype(r)>
   [assertion_kind: post, semantic: enforce, mode: predicate_false, terminating: yes]
   terminate called without an active exception
   Program terminated with signal: SIGSEGV
```

Postconditions- Referring to the Result Object – what is a name?

```
struct S {
  S();
  S(const S&) = delete; // noncopyable, nonmovable
  int i = 0;
  bool foo() const;
};

const S f()
  post(r: const_cast<S&>(r).i = 1)   // OK (but discouraged)
{
  return S{};
}

const S y = f();       // well-defined behavior
bool b = f().foo();    // well-defined behavior
```

- const_cast, although discouraged, is actually safe in this case - result object being captured on the fly is not yet assigned to the call site target variable, therefore it cannot yet be const

Referring to Parameters in Postconditions – wait, isn't const meaningless when passing by value?

- TLDR: by-value function parameters which are ODR-used (a.k.a consumed) in a post CA - must be const, in all the function declarations

```
void f(int i) post ( i != 0 );          // error: i must be const.

void g(const int i) post ( i != 0 );
void g(int i) {}                        // error: missing const for i in definition

void h(const int i) post (i != 0);
void h(const int i) {}
void h(int i);                          // error: missing const for i in redeclaration
```

Referring to Parameters in Postconditions – wait, isn't const meaningless when passing by value?

- Motivation (also in P2521 - 3.13):
- CAs are "captured" by reference
- Post CAs are intended to work on initial parameter values
- Contracts are impossible to reason about if based on mutating parameters
- Must be consistent – ill-formed if by-value parameters ODR-used in post are not const in all the function declarations
- No such new restriction on other parameters – only those ODR-used in post
- With future contract capture clauses (not in C++26) function parameters won't have to be const – since intent will be explicit

Evaluation and Contract-Violation Handling

Point of Evaluation – who said it's a single point?

function call

init params

pre

eval func body

contract_assert (when reached by control flow)

init result object (this means actual caller context object for non-trivially copyable returned types)

destruct local vars

post

destruct params → bind result object

- Pre/post see function parameters, but not local variables
- Result object is initialized before post (thus post can see it), but is not yet considered bound
  - Caller context const (if present) doesn't apply yet (meaning const_cast of result object in post is safe from UB)
- Post is only evaluated on normal exit (no try-catch in pseudo-code)
  - Evaluation when exiting due to exception may be possible in the future
- Post is evaluated only after local variables destruction - since those might influence what post is checking

Evaluation Semantics and selection thereof – because nothing is pre-ordained

- Checking semantics
  - *Observe*
  - Terminating semantics (would have been better named *enforcing* semantics)
    - *Enforce*
    - *Quick_enforce* (was “Louis semantic”)
- *Ignore* semantic
  - Unlike C assert, it ODR-uses and must be well-formed

Evaluation Semantics and selection thereof – because nothing is pre-ordained

- Upon violation
  - At run time
    - *Observe*
      - Invoke violation handler
      - Continue execution (upon normal return from violation handler)
    - *Enforce*
      - Invoke violation handler
      - Terminate (upon normal return from violation handler)
    - *Quick_enforce*
      - Terminate immediately
  - At compile time
    - *Observe* – issue diagnostic (a.k.a warning)
    - *Enforce* and *quick_enforce* – make program ill-formed
  - *Ignore* – doing nothing in both cases

Evaluation Semantics and selection thereof – because nothing is pre-ordained

- Selection method of contract semantics is implementation-defined
- Because different implementations have different needs
- No specification of when this happens
- Any build stage
- Run time
  - Allowing dynamic contracts configuration
- Build modes (e.g. debug and release) are no longer required for this
  - But of course are still allowed
- Different semantics may be selected for different CAs in the same TU
  - Or even for the same CA at different evaluations!

Evaluation Semantics and selection thereof – because nothing is pre-ordained

- Implementations may offer any non-empty subset of the 4 semantics
- Meaning that offering ignore semantic alone is compliant, which entails:
  - Enforcing that CAs remain well-formed
  - ODR-use
    - Side effects possible, e.g. initializing static data members of class templates
- Chosen semantic intentionally undetectable from code at compile time
  - To avoid contracts changing observed behavior

Principle 3: Concepts Do Not See Contracts

The mere presence of a contract assertion on a function or in a block of code should not change the satisfiability of a concept, the result of overload resolution and SFINAE, the branch selected by `if constexpr`, or the value returned by the `noexcept` operator.

Elision and Duplication - I swear to perhaps check the truth, and check it again and again and again, so help me the semantic

- Any CA may be evaluated 0…N times
  - Possibly with different semantics each time
- Repetition is required for supporting different caller and callee contract semantics
  - Likely to affect mainly pre and post
  - Up to 2 evaluations should normally suffice,
    - Other than multiple repetitions to test for unwanted CA side effects
- Implementation requirements
  - Define an upper bound for repetition
- Implementation recommendations
  - Allow users to configure any number of repetitions
  - Make the default one single evaluation without repetitions

Elision and Duplication - I swear to perhaps check the truth, and check it again and again and again, so help me the semantic

- Elision can only happen if the compiler can prove that the predicate:
- Returns true – typically if any of the following holds:
  - It's evaluated at compile time
  - It's guaranteed by previous CAs (in checked terminating semantics)
    contract_assert(x>1);
    contract_assert(x>0);
- Cannot throw, longjmp or terminate

Elision and Duplication - I swear to perhaps check the truth, and check it again and again and again, so help me the semantic

- With elision the compiler may also generate an equivalent expression
- Which is only required to cover the originally well-defined cases
  - UB in original predicate is fair game
- Existing side effects may be ditched in the process,
- No new side effects may be introduced

Principle 6: No Destructive Side Effects

Contract assertions whose predicates, when evaluated, could affect the correctness of the program should not be supported.

```
int i = 0;
void f() pre ((++i, true));
void g() {
  f();  // i may be 0, 1, 17, etc.
}
```

Elision and Duplication - I swear to perhaps check the truth, and check it again and again and again, so help me the semantic

- Elision is all or nothing - either all side effects of a CA are dropped or none at all
- Take away: better not have side effects in CAs
  - Or at least don’t depend on them for business logic
- As-if rule - multiple successful evaluations are undistinguishable from a single one
- Number of evaluations may still affect total execution time
  - Is that observable behavior? – depends on context

The Contract-Violation Handler – you can’t avoid termination (or can you?)

Definition:

- A function named ::handle_contract_violation
- Single argument of type const std::contracts::contract_violation&
- Returns void
- May be noexcept (but definitely doesn’t have to be)

The Contract-Violation Handler – you can't avoid termination (or can you?)

```
class contract_violation
{
// no user-accessible constructor; cannot be copied, moved, or assigned to
public:
    const char* comment() const noexcept;                          // asserted expression
    contracts::detection_mode detection_mode() const noexcept;     // failed assertion / exception during evaluation
    exception_ptr evaluation_exception() const noexcept;           // only for exception during evaluation
    bool is_terminating() const noexcept;                          // is it being invoked with a terminating semantic?
    assertion_kind kind() const noexcept;                          // pre/post/assert
    source_location location() const noexcept;
    evaluation_semantic semantic() const noexcept;                 // ignore/observe/enforce/quick_enforce
};
```

The Contract-Violation Handler – you can't avoid termination (or can you?)

Properties:

- Implementations must provide default handlers
- Recommended to output the *contract_violation* info and to be *noexcept*
- Default handler cannot be directly called by user code
- But it can be called indirectly, via *std::contracts::invoke_default_contract_violation_handler*
- But only from custom violation handlers, since users have no way of creating *contract_violation* objects

The Contract-Violation Handler – you can't avoid termination (or can you?)

Properties:

- Implementations may allow the handler to be replaceable
- Replacement entails providing a same-name-and-signature function (inc. return type)
- For normal functions this would result in linker errors (ambiguous symbol)
- May have a different throwing specification than the default handler
- Hence the standard library provides only an implementation but not an includable declaration of the default handler
- Same as global operators new and delete
- Some may consider that an unacceptable security risk – that's ok

The Contract-Violation Handler – you can't avoid termination (or can you?)

Rationale:

- Libraries control what contracts they contain
- Application controls how contracts are handled
  - With the custom handler instrumental in their toolbox
- Installed at link time , because:
  - Different TUs and libs may be compiled at different times and with different toolchains
  - Too risky security-wise at run time
- Replacing the handler doesn't require recompiling the whole application (and libraries)
  - Contracts would be dead on arrival otherwise

The Contract-Violation Handler – you can't avoid termination (or can you?)

contract_violation main object requirements:

- May reside anywhere other than the heap
- Method of construction determined by implementation

Throwing Violation Handlers and noexcept(contract_assert(false)) – Are you calling me a liar?

- Contract violation handlers can do anything
- Including *throw* or *longjmp* instead of returning normally
- *Throw* and *longjmp* can circumvent *enforce* semantic - and that's a feature, not a bug!
- That's actually an intentional backdoor in P2900
  - Throwing violation handlers were introduced in view of systems which can't afford termination
  - After Bjarne brought this up in P2698R0
  - Likely buggy code following a failed CA will probably crash anyway
- Actually not necessarily a backdoor
  - The main guarantee of *enforce* can be viewed as not allowing execution of the immediately following code
  - Remember I said "terminating semantics" should have been named "enforcing semantics"?

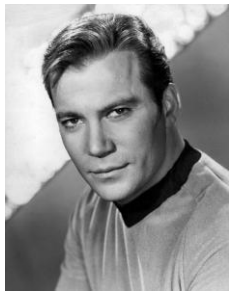
Throwing Violation Handlers and contract_assert(false) – are you calling me a liar?

- Compiler must assume any CA might throw (if throwing violation handlers are supported)
  - Since throwing specification is only known at link time
- Code bases might want to avoid making narrow-contract functions *noexcept*
- Since one day their CAs might throw
- And also to allow failure-based unitesting relying on exceptions
- Also known as the Lakos Rule

Throwing Violation Handlers and contract_assert(false) – are you calling me a liar?

- So then, what should *noexcept(contract_assert(false))* return?

> **Principle 1: Prime Directive**
>
> The presence or evaluation of a contract assertion in a program should not alter the correctness of that program (i.e., the property that evaluation of the program does not violate any provisions of its plain-language contract).

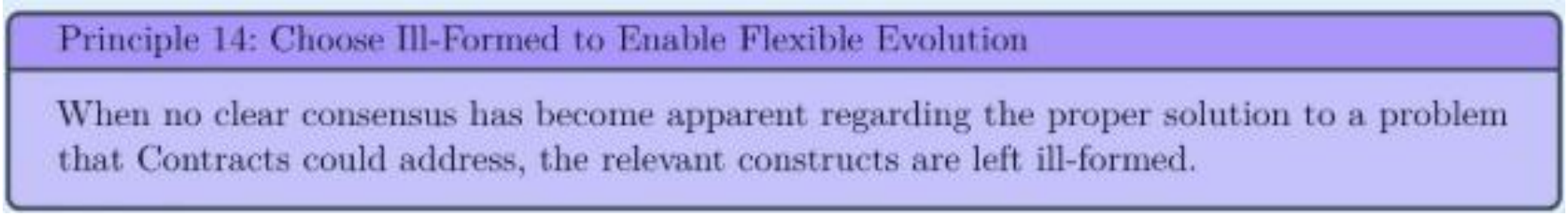


- Having contracts alter an exception specification would violate the prime directive
- But a previously non-throwing function/expression may now be throwing
- Depending on whether the installed violation handler is *noexcept*, which is unknown at compile time
- For which reason the mentioned expression must be assumed to be *true* (on platforms supporting throwing violation handlers)

Throwing Violation Handlers and contract_assert(false) – are you calling me a liar?

Rock and hard place:

- Tell the truth and violate the prime directive
- Or lie and be damned for all eternity
- And we did fight over this almost for all eternity… ∞
- And as in other similar cases we chose to avoid the land mine altogether
- Although this time not by making it ill-formed



Throwing Violation Handlers and contract_assert(false) – are you calling me a liar?

Rock and hard place:

- But rather with a little cheating
- We made *contract_assert* a statement rather than an expression
- Well, I lied a bit too - this does make *noexcept(contract_assert(false))* indirectly ill-formed
- It also restricts *contract_assert* somewhat – vs C assert which can also act as a sub-expression
- Which immediately-invoked lambdas help mitigate
- Instead of this:
  - const int j = (contract_assert(i > 0), i);
- You can write this:
  - const int j = ([i]{ contract_assert(i > 0); }(), i);
  - Which works since this works:
    - const int j = (void{}, i);

Throwing Violation Handlers and contract_assert(false) – are you calling me a liar?

But is it really cheating?

- Not if we redefine the noexcept operator
- To mean “no exceptions **in contract**”
- And that isn’t even lying
- Because UB can’t promise anything by its very definition
- Including that it won’t throw!

Final Word

- Contracts are complicated
- As is C++
- Which is why it took so long to have them delivered
- But you won’t normally care about all these details
- Like about most other C++ details
- But you can leverage them when necessary
- You already know what I’m going to say here…
- At the end of the day, contracts are a great tool
- Like C++, you know the drill…

Final Word

- Many thanks to the many contributors who made Contracts finally happen in C++26!
- And special thanks to my reviewers: Timur Doumler and Joshua Berne
- Which sent me so many comments that any remaining mistakes are definitely mine

Thank You!

Let’s get in touch!

andrziss@gmail.com

https://www.linkedin.com/in/andreizissu/

Questions?