Contracts for C++:
User-defined diagnostic messages

Timur Doumler (papers@timur.audio)
Peter Bindels (dascandy@gmail.com)
Joshua Berne (jberne4@bloomberg.com)

Document #: P3099R1
Date: 2025-10-19
Project: Programming Language C++
Audience: EWG

Abstract

We propose an extension to C++26 Contracts that allows the user to specify a diagnostic message for a particular contract assertion. This functionality is frequently requested and already available in Clang via a vendor attribute. The proposed syntax is straightforward and consistent with existing language features such as `static_assert`.

Contents

Revision history

R1

Update for December 2025 mailing:

- Added section "Empty string vs. no string"

R0

Original version published in October 2025 mailing.

1 Motivation

A user-defined diagnostic message can provide additional information that can help developers more quickly understand why a particular assertion failed and how to resolve the issue. The ability to optionally provide such a message is valuable for any assertion facility, including contract assertions.

The C `assert` macro does not directly support an associated diagnostic message. However, the idiom `assert(expr && "Reason")` has become a common workaround, and many non-standard assertion facilities provide explicit support for diagnostic messages.

The same workaround as with C `assert` is possible with C++26 contract assertions today, and the string would typically be printed by the default contract-violation handler. However, C++ can do better. Beyond improving the syntax, we can expose the string to a user-defined contract-violation handler via the `std::contracts::contract_violation` object passed into it by the implementation. The handler can then display, log, or otherwise process the message in whichever way best suits the program and its environment.

Anecdotally, when a C++ compiler implementer first encountered the specification for C++26 Contracts, their immediate reaction was:

> *"So how do I add a custom message to an assertion here? I need that. Hm, you did not add this yet… where's the syntactic position for a vendor attribute so I can add this in my implementation?"*

Luckily, we had foreseen the need for such vendor attributes and had added [P3088R1] to the C++26 Contracts proposal, specifying the syntactic position for attributes appertaining to contract assertions (even though no such attributes were added in C++26). This enabled Clang to implement user-defined diagnostic messages on top of C++26 Contracts, and they are available as a vendor attribute today:

```
T& operator[] (size_t i)
  pre [[clang::contract_message("Out-of-bounds access")]] (i < size());
```

Now that we have some implementation experience with this vendor extension, including deployment experience in libc++ and the LLVM codebase ([P3460R0]), the time has come to propose standardising this feature for C++29.

2 Discussion

2.1 Syntax

We see three possible approaches for specifying the syntax for this feature:

A. Label syntax (consistent with the proposal [P3400R1] also targeting C++29):

```
T& operator[] (size_t i)
  pre <message("Out-of-bounds access")> (i < size());
```

B. Standard attribute (consistent with the Clang implementation):

```
T& operator[] (size_t i)
  pre [[message("Out-of-bounds access")]] (i < size());
```

C. Second argument (analogous to `static_assert`):

```
T& operator[] (size_t i)
  pre (i < size(), "Out-of-bounds access");
```

Option A requires no additional syntax apart from what [P3400R1] already provides, and is the most extensible. For example, two different labels could provide different messages that could be programmatically combined in different ways, another label could provide formatting options, etc. However, this syntax is somewhat noisy, it places the message before the predicate, even though the predicate is the primary information, and it makes for a more complicated interface containing an arbitrary multitude of messages.

Option B follows the existing practice in Clang. However, it is the most noisy syntax, places the message before the predicate, and represents functionality that does not meet our current litmus test for attributes as *ignorable* constructs ([dcl.attr.grammar]/7).

Option C is the shortest and simplest, which is important as we anticipate this feature to be used frequently. It is the only option that places the message *after* the predicate. It is also consistent with `static_assert` and thus should look and feel very familiar to C++ developers. Incidentally, it is also identical to the syntax used in D.[1]

We therefore propose Option C. It strikes the most favorable balance for immediate usability. Future integration with labels (Option A) remains possible: if we ever pursue the direction of specifying diagnostic messages with labels, Option C can retroactively become syntactic sugar for such labels without breaking any existing code already using the Option C syntax. We propose to apply this syntax extension to all three kinds of contract assertions: `pre`, `post`, and `contract_assert`.

[1] See https://dlang.org/spec/expression.html#AssertExpression

2.2 Compile-time vs. runtime strings

We can identify three options for what kind of strings to allow as the diagnostic message:

A. String literals only
B. Compile-time generated strings
C. Runtime-generated strings

Option A is the most conservative and matches what `static_assert` started out with in C++11. Option B matches the current specification of `static_assert`, which has been repeatedly extended ([N4433], [P2741R3]) with positive experience. We therefore propose Option B, applying the same rules and constraints to the string as `static_assert` already does, leveraging existing practice and avoiding duplication in wording.

Option C seems attractive as it would allow runtime information, such as the value of runtime variables, to be included in the diagnostic message. However, this would require the implementation to run user code after a contract violation has been identified but before the contract-violation handler has been invoked. The security implications of this are currently not fully understood. As with `evaluation_exception()`, which has the same implications ([P3819R0]), we recommend deferring Option C for now. It can be added on top of this proposal at a later time if it can be shown that the security risk is fully avoidable.

2.3 Default contract-violation handler

The current specification states that the output of the default contract-violation handler should be "the most relevant contents of the `std::contracts::contract_violation` object". We propose extending this to say that, if a diagnostic message is supplied, it should also be included in that output.

Since the behaviour of the default contract-violation handler is entirely implementation-defined, and the specification is only a recommended practice, we do not see a need to be more specific about *how* the message should be included in its output.

2.4 User-defined contract-violation handler

The diagnostic message must also be accessible to the user-defined contract-violation handler as this is a primary motivation to add the feature. This requires a modification of the `std::contracts::contract_violation` API. We considered three options:

A. Replace the string returned by `comment()` with the diagnostic message, if one has been supplied.
B. Include the diagnostic message in the string returned by `comment()`.
C. Do not modify the specification of `comment()`; instead, add a new property `message()` that returns the diagnostic message if one has been supplied.

Option A matches the current implementation in Clang and has the advantage that no changes to header `<contracts>` are required. However, it removes[2] the ability to retrieve the original compiler-generated `comment()` string, which is useful on its own: it will typically contain a human-readable representation of the violated predicate expression.

Option B likewise requires no changes to header `<contracts>` and has the additional advantage that only one function call is necessary to get a string that contains all the available information. This makes it easy to dump the entire information into a log file, and impossible to *forget* to include the user-defined string in that output. However, arguments about simplicity in the contract-violation handler are generally questionable as user-defined contract-violation handlers will be written very rarely (typically once per company).

On the other hand, Option B gives the user no portable way to use the compiler-generated and the user-defined messages *separately*. This may be desirable, for example if the author of the contract-violation handler wishes to dump the entire information into a log file but only include the user-defined diagnostic message for sending a JSON-formatted bug report.

We therefore propose Option C. It requires adding a new member function but does not modify any existing functionality, provides the cleanest API, and offers access to both the compiler-generated and the user-defined diagnostic message. This gives the author of the contract-violation handler the freedom to either use them separately or combine them into a single string in whichever way serves their users best.

2.5 Empty string vs. no string

With Option C for the `contract_violation` API, we also need to specify what `message()` should return when no diagnostic message has been supplied. We see two viable options:

C1. A null pointer
C2. An empty string

Option C1 preserves the semantic distinction between "an empty string has been supplied" and "no string has been supplied", which is lost with Option C2. This distinction can be significant. For example, receiving null as the return value for `message()` might indicate that the contract violation came from a translation unit built with C++26 and not C++29.

On the other hand, Option C1 runs the risk of crashing the program if the user decides to log the string returned by `message()` directly without checking for null, which is avoided with Option C2.

At first glance, the ease with which the author of a contract-violation handler might accidentally dereference a null pointer seems concerning. However, contract-violation

[2] Clang offers a workaround in the form of a macro that expands to the compiler-generated string. However, if the user wishes to combine this string with their own diagnostic message string, they must do so in every contract assertion (instead of once in the contract-violation handler). The macro also cannot work in all cases as the information required to generate the string is not always available at the stage of the preprocessor. Workarounds that do not involve macros are conceivable but the resulting user experience is still questionable.

handlers are typically written and debugged once, and a contract-violation handler that dereferences a null pointer would not last long in the real world. We therefore propose Option C1: we should be prioritising expressiveness and the preservation of user-supplied information over the ease of writing a contract-violation handler when given an otherwise even choice.

We also considered returning a `std::optional` from `message()`. However, this choice would not avoid the crash on unchecked dereference, while adding a much larger dependency on the Standard Library to the contract-violation handler API. Our design strives to avoid the latter; we decided against the use of `std::string_view` for C++26 Contracts partly for that reason.

2.6 Constant evaluation

When a contract violation occurs during constant evaluation, no contract-violation handler is called. Instead, the implementation issues a compile-time diagnostic. We follow the practice of `static_assert` and propose that if a user-defined diagnostic message has been supplied, the implementation should include its text in that compile-time diagnostic.

An interesting property of this proposal is that the functionality offered by contract assertions is now a strict superset of `static_assert`, because `static_assert(expr, string)` is equivalent to `contract_assert(expr, string)` evaluated with the *enforce* semantic during constant evaluation.

Furthermore, this proposal (at least partially) subsumes the utility for emitting compile-time diagnostics proposed in [P2758R5], since `contract_assert(false, string)` evaluated with the *observe* semantic during constant evaluation has the same effect.

3 Proposed wording

The proposed wording is relative to the current C++ working draft [N5014]. Add a new section "Diagnostic message strings" [basic.message] as follows:

> *diagnostic-message:*
> *unevaluated-string*
> *constant-expression*
>
> A *diagnostic-message* is a user-supplied string-like object for the purposes of specifying a diagnostic message. If a *diagnostic-message* matches the syntactic requirements of *unevaluated-string*, it is an *unevaluated-string* and the text of the *diagnostic-message* is the text of the *unevaluated-string*. Otherwise, a *diagnostic-message* shall be a *constant-expression M*, and the text of the *diagnostic-message* is determined as follows:
>
> – *M*`.size()` shall be a converted constant expression of type `std::size_t` and let *N* denote the value of that expression.

– *M*`.data()`, implicitly converted to the type "pointer to `const char`", shall be a core constant expression and let *D* denote the converted expression,
– for each *i* where 0 ≤ *i* < *N*, *D*[*i*] shall be an integral constant expression, and
– the text of the *diagnostic-message* is formed by the sequence of *N* code units, starting at *D*, of the ordinary literal encoding ([lex.charset]).

Modify [dcl.pre] as follows:

~~*static_assert-message:*~~
~~*unevaluated-string*~~
~~*constant-expression*~~

static_assert-declaration:
`static_assert (` *constant-expression* `) ;`
`static_assert (` *constant-expression* `,` ~~*static_assert-message*~~*diagnostic-message* `) ;`

~~If a *static_assert-message* matches the syntactic requirements of *unevaluated-string*, it is an *unevaluated-string* and the text of the *static_assert-message* is the text of the *unevaluated-string*. Otherwise, a *static_assert-message* shall be an expression *M* such that~~

~~– the expression *M*`.size()` is implicitly convertible to the type `std::size_t`, and~~
~~– the expression *M*`.data()` is implicitly convertible to the type "pointer to `const char`".~~

In a *static_assert-declaration*, the *constant-expression E* is contextually converted to `bool` and the converted expression shall be a constant expression ([expr.const]). If the value of the expression *E* when so converted is `true` or the expression is evaluated in the context of a template definition, the declaration has no effect and the ~~*static_assert-message*~~*diagnostic-message* ([basic.message]) is an unevaluated operand ([expr.context]). Otherwise, the *static_assert-declaration fails* and

~~–~~ the program is ill-formed~~, and~~
~~– if the static_assert-message is a constant-expression *M*,~~
~~– *M*`.size()` shall be a converted constant expression of type std::size_t and let N denote the value of that expression,~~
~~– *M*`.data()`, implicitly converted to the type "pointer to const char", shall be a core constant expression and let D denote the converted expression,~~
~~– for each i where 0 ≤ i < N, D[i] shall be an integral constant expression, and~~
~~– the text of the *static_assert-message* is formed by the sequence of *N* code units, starting at *D*, of the ordinary literal encoding ([lex.charset])~~.

Recommended practice: When a *static_assert-declaration* fails, the resulting diagnostic message should include the text of the ~~*static_assert-message*~~*diagnostic-message*, if one is supplied.

Modify [basic.contract.general] as follows:

Each contract assertion has a *contract-assertion-predicate* which is an expression of type `bool`. [*Note*: The value of the predicate is used to identify program states that are expected. — *end note*] Each contract assertion has an optionally supplied *diagnostic-message* ([basic.message]).

[...]

If a contract violation occurs in a context that is manifestly constant-evaluated ([expr.const]), and the evaluation semantic is a terminating semantic, the program is ill-formed. [*Note*: A diagnostic is produced if the evaluation semantic is observe ([intro.compliance]). — *end note*]

Recommended practice: The resulting diagnostic message should include the text of the diagnostic-message of the violated contract assertion, if one is supplied.

Modify [basic.contract.handler] as follows:

Recommended practice: The default contract-violation handler should produce diagnostic output that suitably formats the most relevant contents of the `std::contracts:: contract_violation` object, rate-limited for potentially-repeated violations of observed contract assertions, and then return normally. The diagnostic output should include the text of the *diagnostic-message* of the violated contract assertion, if one is supplied.

Modify [stmt.contract.assert] as follows:

assertion-statement :
`contract_assert` *attribute-specifier-seq*$_{opt}$ `(` *conditional-expression* `);`
`contract_assert` *attribute-specifier-seq*$_{opt}$ `(` *conditional-expression* `,` *diagnostic-message* `);`

Modify [dcl.contract.func] as follows:

precondition-specifier :
`pre` *attribute-specifier-seq*$_{opt}$ `(` *conditional-expression* `)` `;`
`pre` *attribute-specifier-seq*$_{opt}$ `(` *conditional-expression* `,` *diagnostic-message* `);`

postcondition-specifier :
`post` *attribute-specifier-seq*$_{opt}$ `(` *result-name-introducer*$_{opt}$ *conditional-expression* `);`
`post` *attribute-specifier-seq*$_{opt}$ `(` *result-name-introducer*$_{opt}$ *conditional-expression* `,`
diagnostic-message `);`

Modify [contracts.syn] as follows:

```
class contract_violation {
  // no user-accessible constructor
public:
  contract_violation(const contract_violation&) = delete;
```

```
    contract_violation& operator=(const contract_violation&) = delete;

    see below ~contract_violation();

    const char* comment() const noexcept;
    contracts::detection_mode detection_mode() const noexcept;
    exception_ptr evaluation_exception() const noexcept;
    bool is_terminating() const noexcept;
    assertion_kind kind() const noexcept;
    source_location location() const noexcept;
    const char* message() const noexcept;
    evaluation_semantic semantic() const noexcept;
  };
```

Modify [support.contract.violation] as follows:

> `const char* message() const noexcept;`
>
> *Returns*: The *diagnostic-message* of the violated contract assertion, if one is supplied; a null pointer otherwise.

Acknowledgements

We thank Anton Polukhin for reviewing an earlier version of this paper and providing valuable feedback.

Bibliography

[N4433] Michael Price: "Flexible `static_assert` messages". 2014-04-09

[N5014] Thomas Köppe: "Working Draft Programming Languages — C++". 2025-08-05

[P2741R3] Corentin Jabot: "User-generated `static_assert` messages". 2023-06-16

[P2758R5] Barry Revzin: "Emitting messages at compile time". 2025-02-11

[P3088R1] Timur Doumler and Joshua Berne: "Attributes for contract assertions". 2024-02-13

[P3400R1] Joshua Berne: "Controlling Contract-Assertion Properties". 2025-02-28

[P3460R0] Eric Fiselier, Nina Dinka Ranns, and Iain Sandoe: "C++ Contracts — Implementers Report". 2024-10-16

[P3819R0] Peter Bindels, Timur Doumler, Joshua Berne, Eric Fiselier, and Iain Sandoe: "Remove `evaluation_exception()` from contract-violation handling for C++26". 2025-09-02