Contracts for C++: User-defined Diagnostic Messages

Document #:	P3099R3
Date:	2026-07-15
Project:	Programming Language C++
Audience:	EWG, LEWG

Reply-to:	Timur Doumler <papers@timur.audio>
	Peter Bindels <dascandy@gmail.com>
	Joshua Berne <jberne4@bloomberg.net>

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

1 Motivation **2**

2 Discussion **3**

2.1 Syntax 3

2.2 Semantics of the user-defined string 4

2.3 Default contract-violation handler 4

2.4 User-defined contract-violation handler 5

2.5 Empty string vs. no string 6

2.6 Constant evaluation 6

3 Implementation experience **7**

4 Wording changes **7**

Revision History

R3 (July 2026 mailing):

- Improved wording after CWG review of [P3423R1]
- Added implementation experience

R2 (May 2026 mailing):

- Updated Section 2.2 “Semantics of the user-defined string”
- Updated Section 2.6 “Constant evaluation”; specified behaviour for unevaluated contexts, discarded contexts, SFINAE, and trial evaluation
- Acknowledged overlap with [P3423R1]
- Rebased wording on C++26 DIS

R1 (December 2025 mailing):

- Added section “Empty string vs. no string”

R0 (October 2025 mailing):

- Original version

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; the message would typically be printed by the default contract-violation handler as part of the failed predicate expression. However, C++ can do better. Beyond improving the syntax, we can make the message string available 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, separately from the predicate expression, 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 [P3400R4] 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 [P3400R4] 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]

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

We therefore propose Option C. It strikes the most favourable balance for immediate usability. Future integration with labels (Option A) remains possible. [P3400R4] proposes a mechanism for allowing labels to manipulate and/or provide the message for a contract assertion, and we could potentially make Option C syntactic sugar for adding a label that returns that initial message 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`.

2.2 Semantics of the user-defined string

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. To avoid duplication in wording, we factor the concept of *diagnostic message* out of the existing definition of `static_assert`, and reuse that concept in both places, `static_assert` and contract assertions.

Note that another paper, [P3423R1], proposes to apply the same upgrade from string literals to compile-time generated diagnostic strings to other language constructs that accept such a parameter: `[[nodiscard]]`, `[[deprecated]]`, and `=delete`. Our proposed wording, which refactors *'static_assert'-message* into a new shared *diagnostic-message*, provides a foundation for that paper to easily leverage the same refactoring.

Option C seems attractive for contract assertions in particular, as it would allow runtime information, such as the value of runtime variables, to be included in the diagnostic message, aiding debugging. 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 thus 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 `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.

In line with the specification of `comment()`, we specify that `message()` returning the diagnostic message is a recommended practice, not a normative requirement. This is necessary to continue to enable builds where diagnostic messages are completely stripped out to remain conforming, motivated by the security requirements of some users to not have any human-readable information about their source code embedded in their binaries.

[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.

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 Option C2 avoids.

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 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.

In compile-time contexts where the contract assertion is not actually constant-evaluated — such as unevaluated contexts, discarded contexts, and trial evaluation[3] — contract violations do not occur, and thus no compile-time diagnostics are triggered. Note that this is consistent with other proposals in this space, in particular [P2758R5].

Importantly, this proposal does not change the constant-evaluation rules for contract assertions[4] in any way. It adds only the ability to indicate that an additional message should be part of that diagnostic, with no effect on when and under what conditions the compiler issues the diagnostic due to a contract violation.

Consequently, if the diagnostic message specified by the user does not meet the requirements for constant evaluation, this must result in a hard compiler error: we do not want the diagnostic message

[3] Speculative constant evaluation that the compiler performs to determine whether a variable could be constant-initialized; see [expr.const], paragraph 27.5 and footnote 65.

[4] See [P2900R14] Section 3.5.12; for a more detailed technical description, see [P2894R2].

to be part of the immediate context when all other parts of a contract assertion are not.

3 Implementation experience

User-defined diagnostic messages have been implemented in branches of GCC and Clang that are available on Compiler Explorer: `https://godbolt.org/z/7P1EWzEdq`. Both implementations provide the full set of functionality specified in this paper when the command-line flag `-fcontracts-p3099` is present. The feature is also available with the flag `-fcontracts-p3850` that enables prototype implementations of many of the papers described in the overall plan in [P3850R1].

Reusing (and slightly refactoring) the existing support already present for handling `static_assert` messages was fairly straightforward in both compilers. In GCC we reused its `cexpr_str` compile-time string extraction and routed both `static_assert` and contract assertions through a single new `cp_parser_diagnostic_message` parser, while in Clang we reused the validation already performed for `static_assert` messages. The remaining work was mostly threading the message through the contract AST, redeclaration checking, and violation construction. Redeclaration sameness is checked on the *extracted text* rather than the expression structure, so that `"hello"` and a `string_view` yielding `"hello"` compare equal.

Both compiler branches implement a shared ABI (eventually to be proposed as part of the Itanium ABI). The message text is a field of the `__cxa_contract_data_block` emitted for each violation, stored as a `const char*` in read-only data rather than in any compiler-specific representation. The layout of these objects is dynamic: the space for the extra pointer is used only when there is a message present on the contract assertion. Because both forks agree on this layout, a violation handler built with either compiler can read messages produced by assertions compiled with the other. This composes with [P3400R4], whose `compute_message` facet can transform or replace the message before it reaches the handler.

In the Clang implementation, we made the use of the pre-existing vendor-specific attribute `[[clang::contract_message(...)]]` and the presence of a diagnostic message proposed here mutually exclusive.

4 Wording changes

Note: The wording originally proposed in this paper was reused for [P3423R1] as it also had a need to refactor the diagnostic message from `static_assert`. That wording had some amount of core wording review and was significantly restructured. The shared parts of updated wording which resulted from that review are now proposed in this paper.

These wording changes are relative to the C++29 working draft with git hash 14ed707b, last modified on Wed, 17 Jun 2026 14:18:34 +0100.

6 Basics **[basic]**

6.10 + *a* Diagnostic messages **[basic.message]**

Add a new subclause 6.10+a[basic.message] after 6.10.3.4[basic.start.term]

6.10+a Diagnostic messages [basic.message]

diagnostic-message:
unevaluated-string
constant-expression

1 A *diagnostic-message* is a user-supplied string-like construct 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 such that

(1.1) — the expression M`.size()` is implicitly convertible to the type `std::size_t`, and

(1.2) — the expression M`.data()` is implicitly convertible to the type "pointer to `const char`".

When the *diagnostic-message* is not an unevaluated operand,

(1.3) — M`.size()` shall be a converted constant expression of type `std::size_t` and let N denote the value of that expression,

(1.4) — M`.data()`, implicitly converted to the type "pointer to `const char`", shall be a core constant expression and let D denote the converted expression,

(1.5) — for each i where $0 \leq i < N$, D`[`i`]` shall be an integral constant expression, and

(1.6) — the text of the *diagnostic-message* is formed by the sequence of N code units, starting at D, of the ordinary literal encoding (5.3.1[lex.charset]).

[*Note 1*: An immediate invocation (7.7.5[expr.const.imm]) that is a potentially-evaluated subexpression (6.10.1[intro.execution]) of the *diagnostic-message* is evaluated at the point where the *diagnostic-message* appears. — *end note*]

Note

The structure above is new, but the bullets have all been moved (with minimal changes) from [dcl.pre].

6.11 Contract assertions [basic.contract]

6.11.1 General [basic.contract.general]

Add a new paragraph after 6.11.1[basic.contract.general], paragraph 2:

2+a Each contract assertion has an optionally supplied *diagnostic-message* (6.10+a[basic.message]). The *diagnostic-message*, if any, is evaluated at the point of the declaration or statement it is part of.

6.11.3 Contract-violation handler [basic.contract.handler]

Modify section 6.11.3[basic.contract.handler], paragraph 2:

2 *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.

7 Expressions [expr]

7.7 Constant evaluation [expr.const]

7.7.7 Further definitions [expr.const.defns]

Modify section 7.7.7[expr.const.defns], paragraph 1:

1 An expression or conversion is *manifestly constant-evaluated* if it is

(1.1) — a *constant-expression*, or

(1.2) — the condition of a constexpr if statement (8.5.2[stmt.if]), or

(1.3) — the expression corresponding to a *consteval-block-declaration* (9.1[dcl.pre]), or

(1.4) — an immediate invocation, or

(1.5) — the result of substitution into an atomic constraint expression to determine whether it is satisfied (13.5.2.3[temp.constr.atomic]), or

(1.6) — the initializer of a variable that is usable in constant expressions or has constant initialization (6.10.3.2[basic.start.static]).[64]

[*Example*:

```
X<std::is_constant_evaluated()> x;                    // type X<true>
int y;
const int a = std::is_constant_evaluated() ? y : 1;   // dynamic initialization to 1
double z[a];                                           // error: a is not usable
                                                       // in constant expressions
const int b = std::is_constant_evaluated() ? 2 : y;   // static initialization to 2
int c = y + (std::is_constant_evaluated() ? 2 : y);   // dynamic initialization to y+y

constexpr int f() {
  const int n = std::is_constant_evaluated() ? 13 : 17; // n is 13
  int m = std::is_constant_evaluated() ? 13 : 17;       // m can be 13 or 17 (see below)
  char arr[n] = {};                                     // char[13]
  return m + sizeof(arr);
}
int p = f();                                           // m is 13; initialized to 26
int q = p + f();                                       // m is 17 for this call; initialized to 56
```

— *end example*]

[*Note 1*: Except for a ~~*static_assert-message*~~*diagnostic-message* that is itself specified as an unevaluated operand, a manifestly constant-evaluated expression is evaluated even in an unevaluated operand (7.2.3[term.unevaluated.operand]). — *end note*]

---

[64] Testing this condition can involve a notional evaluation of its initializer, with evaluations of contract assertions using the ignore evaluation semantic (basic.contract.eval), as described above.

8 Statements [stmt]

8.9 **Assertion statement** **[stmt.contract.assert]**

Modify subclause 8.9[stmt.contract.assert]:

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

9 **Declarations** **[dcl]**

9.1 **Preamble** **[dcl.pre]**

Modify subclause 9.1[dcl.pre], paragraph 1:

1 Declarations generally specify how names are to be interpreted. Declarations have the form

declaration-seq:
declaration declaration-seq$_{opt}$

declaration:
name-declaration
special-declaration

name-declaration:
block-declaration
nodeclspec-function-declaration
function-definition
friend-type-declaration
template-declaration
deduction-guide
linkage-specification
namespace-definition
empty-declaration
attribute-declaration
module-import-declaration

special-declaration:
explicit-instantiation
explicit-specialization
export-declaration

block-declaration:
simple-declaration
asm-declaration
namespace-alias-definition
using-declaration
using-enum-declaration
using-directive
static_assert-declaration
consteval-block-declaration
alias-declaration
opaque-enum-declaration

nodeclspec-function-declaration:
attribute-specifier-seq$_{opt}$ *declarator* ;

alias-declaration:
`using` *identifier attribute-specifier-seq*$_{opt}$ = *defining-type-id* ;

sb-identifier:
`...`$_{opt}$ *identifier attribute-specifier-seq*$_{opt}$

sb-identifier-list:
sb-identifier
sb-identifier-list , *sb-identifier*

structured-binding-declaration:
attribute-specifier-seq$_{opt}$ *decl-specifier-seq ref-qualifier*$_{opt}$ [*sb-identifier-list*]

simple-declaration:
decl-specifier-seq init-declarator-list$_{opt}$;
attribute-specifier-seq decl-specifier-seq init-declarator-list ;
structured-binding-declaration initializer ;

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

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

consteval-block-declaration:
`consteval` *compound-statement*

empty-declaration:
;

attribute-declaration:
attribute-specifier-seq ;

[*Note 1*: *asm-declaration*s are described in 9.11[dcl.asm], and *linkage-specification*s are described in 9.12[dcl.link]; *function-definition*s are described in 9.6[dcl.fct.def] and *template-declaration*s and *deduction-guide*s are described in 13.7.2.3[temp.deduct.guide]; *namespace-definition*s are described in 9.9.2[namespace.def], *using-declaration*s are described in 9.10[namespace.udecl] and *using-directive*s are described in 9.9.4[namespace.udir]. — *end note*]

Remove a paragraph from and modify subclause 9.1[dcl.pre], paragraphs 12-14:

~~12~~ ~~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~~

(~~12.1~~) — ~~the expression M`.size()` is implicitly convertible to the type `std::size_t`, and~~

(~~12.2~~) — ~~the expression M`.data()` is implicitly convertible to the type "pointer to `const char`".~~

13 In a *static_assert-declaration*, the *constant-expression* E is contextually converted to `bool` and the converted expression shall be a constant expression (7.7.3[expr.const.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* (6.10+a[basic.message]) is an unevaluated operand (7.2.3[term.unevaluated.operand]). 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 $0 \leq i < N$, 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 (5.3.1[lex.charset]).~~

14 *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.

[*Example 1*:

```
static_assert(sizeof(int) == sizeof(void*), "wrong pointer size");
static_assert(sizeof(int[2]));          // OK, narrowing allowed

template <class T>
void f(T t) {
  if constexpr (sizeof(T) == sizeof(int)) {
    use(t);
  } else {
    static_assert(false, "must be int-sized");
  }
}

void g(char c) {
  f(0);             // OK
  f(c);             // error on implementations where sizeof(int) > 1: must be int-sized
}
```

— *end example*]

9.4 Function contract specifiers [dcl.contract]

9.4.1 General [dcl.contract.func]

Modify section 9.4.1[dcl.contract.func]:

function-contract-specifier-seq:
function-contract-specifier *function-contract-specifier-seq*$_{opt}$

function-contract-specifier:
precondition-specifier
postcondition-specifier

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 section 9.4.1[dcl.contract.func], paragraph 5:

5 A *function-contract-specifier-seq* S_1 is the same as a *function-contract-specifier-seq* S_2 if S_1 and S_2 consist of the same *function-contract-specifier*s in the same order. A *function-contract-specifier* C_1 on a function declaration D_1 is the same as a *function-contract-specifier* C_2 on a function declaration D_2 if

(5.1) — their predicates P_1 and P_2 would satisfy the one-definition rule (6.3[basic.def.odr]) if placed in function definitions on the declarations D_1 and D_2, respectively, except for

— renaming of the parameters of f,

— renaming of template parameters of a template enclosing , and

— renaming of the result binding (9.4.2[dcl.contract.res]), if any,

and, if D_1 and D_2 are in different translation units, corresponding entities defined within each predicate behave as if there is a single entity with a single definition, and

(5.2) — both C_1 and C_2 specify a *result-name-introducer* or neither do~~.~~,

(5.2+a) — both C_1 and C_2 specify a *diagnostic-message* with the same text or neither do.

If this condition is not met solely due to the comparison of two *lambda-expression*s that are contained within P_1 and P_2, no diagnostic is required.

[*Note 1*: Equivalent *function-contract-specifier-seq*s apply to all uses and definitions of a function across all translation units. — *end note*]

[*Example 1*:

```
bool b1, b2;

void f() pre (b1) pre ([]{ return b2; }());
void f();                       // OK, function-contract-specifiers omitted
void f() pre (b1) pre ([]{ return b2; }()); // error: closures have different types.
void f() pre (b1);              // error: function-contract-specifiers only partially repeated

int g() post(r : b1);
int g() post(b1);       // error: mismatched result-name-introducer presence

namespace N {
  void h() pre (b1);
  bool b1;
  void h() pre (b1);    // error: function-contract-specifiers differ according to
```

```
                                    // the one-definition rule (6.3[basic.def.odr]).
}
```

— *end example*]

15 **Preprocessing directives** [cpp]

15.12 **Predefined macro names** [cpp.predefined]

Modify subclause 15.12[cpp.predefined], Table 22 [cpp.predefined.ft]:

Table 1: Table 22 — Feature-test macros [tab:cpp.predefined.ft]

Macro name	Value
⋮ *20 rows elided* ⋮	
`__cpp_constinit`	`201907L`
`__cpp_contracts`	`202502L`
`__cpp_contracts_message`	`xxxxxxL`
`__cpp_decltype`	`200707L`
`__cpp_decltype_auto`	`201304L`
⋮ *56 rows elided* ⋮	

17 **Language support library** [support]

17.3 **Implementation properties** [support.limits]

17.3.2 **Header `<version>` synopsis** [version.syn]

Modify section 17.3.2[version.syn], paragraph 2:

2 Each of the macros defined in `<version>` is also defined after inclusion of any member of the set of library headers indicated in the corresponding comment in this synopsis.

[*Note 1*: Future revisions of this document might replace the values of these macros with greater values. — *end note*]

```
                                    ⋮ 92 lines elided ⋮
  // <deque>, <queue>, <stack>, <string>
#define __cpp_lib_contracts                     202502L // freestanding, also in <contracts>
#define __cpp_lib_contracts_message             xxxxxxL // freestanding, also in <contracts>↩
#define __cpp_lib_copyable_function             202306L // also in <functional>
#define __cpp_lib_coroutine                     201902L // freestanding, also in <coroutine>

                                    ⋮ 224 lines elided ⋮
```

17.10 **Contract-violation handling** [support.contract]

17.10.1 **Header <contracts> synopsis** **[contracts.syn]**

Modify section 17.10.1[contracts.syn], paragraph 1:

1 The header <contracts> defines types for reporting information about contract violations (6.11.2[basic.contract.eval]).

```
// all freestanding
namespace std::contracts {

  enum class assertion_kind : unspecified {
    pre = 1,
    post = 2,
    assert = 3
  };

  enum class evaluation_semantic : unspecified {
    ignore = 1,
    observe = 2,
    enforce = 3,
    quick_enforce = 4
  };

  enum class detection_mode : unspecified {
    predicate_false = 1,
    evaluation_exception = 2
  };

  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;
    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;
  };

  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

17.10.3 **Class `contract_violation`** **[support.contract.violation]**

Add a new paragraph after 17.10.3[support.contract.violation], paragraph 8:

```
const char* message() const noexcept;
```

8+a *Returns*: An implementation-defined NTMBS in the ordinary literal encoding (5.3.1[lex.charset]), or a null pointer.

8+b *Recommended practice*: The string returned should be the *diagnostic-message* of the violated contract assertion, if one is supplied; otherwise it should be a null pointer.

Acknowledgments

We thank Anton Polukhin and Corentin Jabot for reviewing earlier revisions of this paper and providing valuable feedback.

Claude (Anthropic) was used for editorial assistance during the preparation of this paper, as well as significant parts of the prototype implementations.

Bibliography

[N4433] Michael Price, "Flexible static_assert messages", 2015
http://wg21.link/N4433

[P2741R3] Corentin Jabot, "user-generated static_assert messages", 2023
http://wg21.link/P2741R3

[P2758R5] Barry Revzin, "Emitting messages at compile time", 2025
http://wg21.link/P2758R5

[P2894R2] Timur Doumler, "Constant evaluation of Contracts", 2024
http://wg21.link/P2894R2

[P2900R14] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2025
http://wg21.link/P2900R14

[P3088R1] Timur Doumler and Joshua Berne, "Attributes for contract assertions", 2024
http://wg21.link/P3088R1

[P3400R4] Joshua Berne, "Controlling Contract-Assertion Properties", 2026
http://wg21.link/P3400R4

[P3423R1] Yihe Li, "Extending User-Generated Diagnostic Messages", 2025
http://wg21.link/P3423R1

[P3460R0] Eric Fiselier, Nina Dinka Ranns, and Iain Sandoe, "Contracts Implementors Report", 2024
http://wg21.link/P3460R0

[P3819R0] Peter Bindels, Joshua Berne, Timur Doumler, Iain Sandoe, and Eric Fiselier, "Remove evaluation_exception() from contract-violation handling for C++26", 2025
http://wg21.link/P3819R0

[P3850R1] Timur Doumler and Joshua Berne, "A proposed plan for extending Contracts in C++29", 2026
http://wg21.link/P3850R1