user-generated `static_assert` messages

Document #:	P2741R3
Date:	2023-06-16
Programming Language C++	
Audience:	EWG

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose that `static_assert` should accept user-generated string-like objects as their diagnostic message.

Revisions

R3

Expand the design section to explain how instantiation and recursion work, following BSI comments.

R2

Following SG-16 review:

- Remove support for `char8_t` for now.
- Improve wording

R1

- Expand discussion on the encoding of compile time expressions.
- Remove wording for volatile types as they cannot be constant-evaluated per [expr.const] (Thanks Jens!).

R0

Initial revision

Motivation

We propose that the message of a `static_assert` could be an arbitrary constant expression producing a sequence of characters, rather than be limited to string literals. This would allow

libraries doing work at compile time to be able to better diagnose the exact problem. This could be used, for example, to

- Explain why a formatting string in format is invalid
- Explain why as compile time regex in CTRE is invalid. In general, this would be helpful for any library that generates code at compile time, such as DSL generators (lexy), Unicode table generators, unit test frameworks, or any other library or DSL who needs to diagnose complex constraints.
- Better explain the constraints of compile-times APIs
- Reduce the reliance on the preprocessor and stringification of identifiers
- Avoid duplication in static assert messages

Without this proposal

```
template <typename T, auto Expected, unsigned long Size = sizeof(T)>
constexpr bool ensure_size() {
    static_assert(sizeof(T) == Expected, "Unexpected sizeof");
    return true;
}
static_assert(ensure_size<S, 1>());
```

```
error: static assertion failed due to requirement 'sizeof(S) == 1':
    Unexpected sizeof
static_assert(sizeof(T) == Expected, "Unexpected sizeof");
^             ~~~~~~~~~~~~~~~~~~~~~
note: in instantiation of function template specialization
    'ensure_size<S, 1, 4ULL>' requested here
static_assert(ensure_size<S, 1>());
^
note: expression evaluates to '4 == 1'
static_assert(sizeof(T) == Expected, "Unexpected sizeof");
~~~~~~~~~~~^~~~~~~~~~~
1 error generated.
Compiler returned: 1
```

With this proposal

```
static_assert(sizeof(S) == 1,
    std::format("Unexpected sizeof: expected 1, got {}", sizeof(S))); // *
```

```
error: static assertion failed due to requirement 'sizeof(S) == 1':
    Unexpected sizeof: expected 1, got 4
static_assert(sizeof(S) == 1,
^             ~~~~~~~~~~~~~~
note: expression evaluates to '4 == 1'
static_assert(sizeof(S) == 1,
~~~~~~~~~~~^~~~
1 error generated.
Compiler returned: 1
```

* `constexpr std::format` is not proposed in this proposal (and very much intentionally so this is but one building block). We should note however that `libfmt` has supported constexpr formatting since version 8.0.0, mid-2021.

This both simplifies the code and makes the diagnostic clearer.

Interaction with reflection

The capabilities presented here would be made more useful by reflection (P1240 [**?**]), notably for the ability to use the name of an instantiated template parameters in diagnostic messages, however, both features are independently useful and do not overlap and there is no need to tie these features together.

Community use cases

Here is a sampling of discussions of this facility online

- Stackoverflow - How to combine static_assert with sizeof and stringify?

  > Memory usage is quite critical in my application. Therefore I have specific asserts that check for the memory size at compile time and give a static_assert if the size is different from what we considered correct before. [..] The problem is that when this static_assert goes off, it might be quite difficult to find out what the new size should be. [...] It would be much handier if I could include the actual size.

  People replying suggest instead injecting a template parameter in the function enclosing the static_assert, which would be outputted by most compilers. The generated error message is not user-friendly.

  ```
  In instantiation of ‘void check_size() [with ToCheck = foo; long
      unsigned int ExpectedSize = 8ul; long unsigned int RealSize = 16ul’]:

  bla.cpp:15:22:   required from here
  bla.cpp:5:1: error: static assertion failed: Size is off!
  ```

- Stackoverflow - Better Message For ‘static_assert‘ on Object Size

  Similar use case.

- Display integer at compile time in static_assert()

  The user would like to express the following code:

  ```
  enum First
  {
      a,
      b,
      c,
      nbElementFirstEnum,
  };
  enum Second
  {
      a,
  ```

```
    b,
    c,
    nbElementSecondEnum,
};

static_assert(
First::nbElementFirstEnum == Second::nbElementSecondEnum,
"Not the same number of element in the enums."s + to_string(First::
    nbElementFirstEnum) + " "s + to_string(Second::nbElementSecondEnum);
```

There again, the suggestion is to surface these values as a template parameter, hoping the compiler would show enough content to surface them.

- Stack overflow - How to pass a not explicitly string literal error message to a static_assert?

  In this question, the user would like to reuse the same message in multiple static_assert and is reluctant to either copy-paste their code or use a macro. Alas, there is no better solution.

- Many other questions in stack overflow would require reflection to be fully solved: They are all more or less identical to this one: C++11 static_assert: Parameterized error messages

- That feature was previously requested and discussed on std-proposals here, and here.

Design

We proposed to allow a constant expression string as the second parameter of `static_assert`. That's it. In particular, we propose no way to construct a string, as these are orthogonal concerns that can be handled by reflection, making `std::format constexpr`, or by string interpolation (P1819r0 [4]), or simply by concatenating `std::string`s or using third-party libraries, or, a combination of some or all of the above. The question we are answering in this paper is: I have a string, can I use it as my `static_assert` message?

What is a string?

We do not think this core-language feature should be tied to a specific type or header like `std::string_view`. Indeed many user-defined types can be used to form and store a diagnostic message, and relying on details of the standard libraries are likely to be more complicated than note for implementers and users alike. Instead, we propose a definition of a string-like type that allows the support of `string` and `string_view`, as well as similar user-defined types. A compatible string-like type is a type that:

- Has a `size()` method that produces an integer
- Has a `data()` method that produces a pointer of character type such that
- The elements in the range `[data(), data()+size())` are valid.

This is consistent with how structured bindings and range for loops work.

Non-contiguous ranges

We only propose to support ranges that offer `size()` and `data()`. There are a few reasons for that.

Defining what a range is is slightly more involved - although we could reuse the definition used by ranged-base `for`, and it seem less motivated. ie, string-like are usually contiguous.

But the main reason is that implenters expressed some slight performance concerns. constant evaluating `operator++` and `operaor*` for each character at compile time is a lot less efficient than evaluating a pointer. It could even be faster to convert a list to string and then use it in a diagnostic message, than to use the list directly, as the later forces us to get in and out of the constant evaluation domain.

What if the expression producing the message is ill-formed?

The message-producing expression is intended to be always instantiated, but only evaluated if the assertion failed.

```
template <auto N>
consteval std::string_view oups() {
    static_assert(N, "this always fires");
    return "oups!";
}
void f() {
    static_assert(true, oups<false>());
}
```

[Compiler explorer]

`oups<false>` is instantiated even though the static_assert never triggers. This behavior is consistent with the rest of the language - except for discarded statements, and is motivated by the fact that if the message-producing is ill-formed, it would otherwise not be detected until the `static_assert` triggers. In effect, it could hide bugs in code whose purpose is to diagnose bugs, and this seems extremely user-hostile.

We should also note that in the example above both static_assert would trigger, but it is very much implementation defined whether an implementation could evaluate `oups<false>` despite it being ill-formed.

Could this thing recurse forever?

Consider:

```
template <auto N>
consteval std::string_view oups() {
    if constexpr(N == 0)
    return "oups";
```

```
    else
    static_assert(!N, oups<N-1>());
    return "oups!";
}

void f() {
    static_assert(false, oups<99999>());
}
```

[Compiler explorer]

This naturally falls into implementation limits. No message is likely to be produced as we cannot evaluate the terminal branch until bumping into the limits.

with a more reasonable limit, say

```
static_assert(false, oups<5>());
```

[Compiler explorer]

In this example, the static assert messages are produced from 1 to 5 (again, it is mostly QOI and context-dependent whether the user-generated message can be produced).

Encodings

Constant evaluation deals with literal encoding, which may not be UTF-8. As such, `static_assert` should allow both `char` and `char8_t` as messages and will need to convert both to the encoding of diagnostic messages. This is different from string literals in static assert which are not evaluated and converted directly from Unicode (likely UTF-8) to the encoding of diagnostic messages.

Support for `wchar_t`, `char16_t`, `char32_t` is not proposed, but would not be an issue.

Note however that, if an implementation did not have the ability to convert from the literal encoding to the diagnostic encoding, properly rendering non-basic characters in a sequence of `char` might be the most involved part of this proposal. same if we were to support `wchar_t`. Supporting any of the UTF encodings is however a non-issue.

Note that in the case of `char` and `wchar_t` there is very little room to treat these things as anything but strings in the (wide) ordinary literal encoding. character and string literals are in the literal encoding and so would be any expression that would concatenate or otherwise be composed from character and string literals in some way. Treating them in any other way would lead to mojibake (as it always does when interpreting a string in an encoding that is different from the one it is constructed with).

We shouldn't either try to evaluate these expressions using an encoding, as they may refer to variables already evaluated.

Should we want another encoding here (and we really should not, C++ has enough encodings as it is), we would need a new kind of literals and a new set of types from distinguishing them. Such type actually exist: `char8_t`!

In other words, given the following code:

```
constexpr string_view str = "Hello";
static_assert(false, str);
```

The abstract machine leaves no room to consider `str` as anything but as a string literal in the ordinary literal as it was already evaluated at the point the variable is used.

What if the string literal encoding cannot represent the diagnostic message in some environments?

P1854R3 [1] makes mojibake in literals ill-formed (as it already is on most platforms). It is still possible to produce mojibake using escape sequences, for example.

Given there are 2 possibles opportunities for losses here, namely that the ordinary literal cannot encode some text, and that the output of the compiler may have a different encoding to, only one of these could theoretically be remedied, I do not think there is much of a concern here: compilers will, as QOI, present any potential encoding issues in a way that is useful to users, escaping invalid code units for example. They already do so.

There isn't much value in trying to legislate more than that. If a `static_assert` fires, the program is ill-formed. If the diagnostic message has encoding errors, is it double ill-formed? Should we care about potential mojibake in `static_assert` that do not fire?

Should we expose the encoding of the compiler's output?

For completeness, as I was asked if i considered that: No

Knowing the execution encoding is useful as it allows users to perform conversions. Knowing the encoding of the compiler output does not allow to do anything the compiler cannot do on its own: Having established that strings are in the ordinary encoding, the compiler can transcode that and the user could not do it better. And we'd have to make constexpr all the existing text manipulations facility to do anything useful EVEN if somehow we could form strings in that theoretical compiler encoding, which we can't do, short of some magic conversion from an u8 string of some form. At this point, why did we not use u8 to begin with?

Maybe `std::format` cannot produce u8 and so we'd fail to support a motivating use case. Fair enough, but if we are concerned about that, we should add u8 support to `std::format` !

What about null-terminated strings?

Null-terminated strings are mostly useful to communicate with C libraries and systems and are rarely useful at compile time. While it would not be a huge effort to support them, maybe it's best to keep the design simple.

SG-16 expressed a slight preference to not support null-terminated string.

Alternatives

In the status quo, depending on the specific use case, macros can be used to stringify some arguments, or the `static_assert` can be lifted in a function template such that most compilers should print the value of this template parameters as part of the diagnostic message.

Neither of these solutions is really satisfying or complete.

Previous work

A similar capability was previously proposed in 2014 by N4433 [2]. At the time, the consensus was that it required too many pieces that did not exist then. As string and string_view are constexpr, `std::format` could be made constexpr (and the fmt lib already can create messages at compile times), and reflection is upon us, we think this feature could be immediately useful.

Future work

As `static_assert` is constantly evaluated, it cannot be used to diagnose, for example, unsatisfied preconditions on parameters and local variables. For that, we will need an additional facility composed of `consteval` functions, as proposed by P0596R1 [5] and P2758R0 [3].

Implementation

This feature was implemented in Circle and prototyped in Clang, with no difficulties.

Wording

[Editor's note: This wording assumes P2361R5 has been applied to the working draft] .

 Preamble [dcl.pre]

simple-declaration:
decl-specifier-seq init-declarator-list$_{opt}$;
attribute-specifier-seq decl-specifier-seq init-declarator-list ;
attribute-specifier-seq$_{opt}$ *decl-specifier-seq ref-qualifier*$_{opt}$ `[` *identifier-list* `]` *initializer* ;

static_assert-message:
unevaluated-string
conditional-expression

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

empty-declaration:
;
attribute-declaration:
attribute-specifier-seq ;

[...]

Syntactic components beyond those found in the general form of *simple-declaration* are added to a function declaration to make a *function-definition*. An object declaration, however, is also a definition unless it contains the `extern` specifier and has no initializer[basic.def]. An object definition causes storage of appropriate size and alignment to be reserved and any appropriate initialization[dcl.init] to be done.

A *nodeclspec-function-declaration* shall declare a constructor, destructor, or conversion function. [*Note:* Because a member function cannot be subject to a non-defining declaration outside of a class definition[class.mfct], a *nodeclspec-function-declaration* can only be used in a *template-declaration* [temp.pre], *explicit-instantiation* [temp.explicit], or *explicit-specialization* [temp.expl.spec]. — *end note*]

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 `std::size_t`, and
- the expression M`.data()` is implicitly convertible to "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 is an unevaluated operand [expr.context] .

Otherwise, the *static_assert-declaration fails* ~~,~~ and ~~the program is ill-formed, and the resulting diagnostic message [intro.compliance] should include the text of the *string-literal*, if one is supplied~~.

- the program is ill-formed, and
- if the *static_assert-message* is a *conditional-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 [lex.charset].

Recommended practice:

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

[*Example:*

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

— *end example*]

Feature test macro

[Editor's note: In `[tab:cpp.predefined.ft]`, bump the value of `__cpp_static_assert` to the date of adoption] .

```
#define __cpp_static_assert 2023XX // date of adoption
```

References

[1] Corentin Jabot. P1854R3: Conversion to literal encoding should not lead to loss of meaning. https://wg21.link/p1854r3, 1 2022.

[2] Michael Price. N4433: Flexible static_assert messages. https://wg21.link/n4433, 4 2015.

[3] Barry Revzin. P2758R0: Emitting messages at compile time. https://wg21.link/p2758r0, 1 2023.

[4] Vittorio Romeo. P1819R0: Interpolated literals. https://wg21.link/p1819r0, 7 2019.

[5] Daveed Vandevoorde. P0596R1: Side-effects in constant evaluation: Output and consteval variables. https://wg21.link/p0596r1, 10 2019.

[N4892] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4892