Unevaluated strings

Document #:	P2361R3
Date:	2021-10-09
Programming Language C++	
Audience:	EWG

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>
	Aaron Ballman <aaron.ballman@gmail.com>

Abstract

*string-literal*s can appear in a context where they are not used to initialize a character array, but are used at compile time for diagnostic messages, preprocessing, and other implementation-defined behaviors. This paper clarifies how compilers should handle these strings.

Motivation

A *string-literal* can appear in `_Pragma`, `asm`, `extern`, `static_assert`, `[[deprecated]]` and `[[nodiscard]]` attributes...

In all of these cases, the strings are exclusively used at compile time by the compiler, and are as such not evaluated in phase 6. This means they should not be converted to the narrow encoding or any literal encoding specified by an encoding prefix (L, u, U, u8).

Their encoding should therefore not be constrained or otherwise specified, except that these strings can contain any Unicode characters.

This proposal aim to identify contexts in which strings are not evaluated so that they can be handle consistently by compilers.

Revisions

R3

- Improve wording by not making *unevaluated-string* preprocessing token as preprocessing tokens should not be context dependent. Fix the wording of `#line` and `_Pragma` accordingly.
- Append null-terminator during evaluation of string-literals to make it clear that *unevaluated-string* are not null-terminated.
- Adapt the grammar of *literal-operator-id*
- Adapt the wording of `extern` to clarify that the likage specification denotes unicode characters.

- Allow numeric escape sequences in asm statements.

R2

- *unevaluated-string-literal* to *unevaluated-string*.
- Add a note about not disallowing non-printable characters
- Add a note about *unevaluated-string* not being expressions.
- Fix typos.
- Improve wording.

Proposal

Unevaluated string literals can appear in

- `_Pragma`
- `#line` directives
- `[[nodiscard]]` and `[[deprecated]]` attributes
- `extern` linkage specifications
- `asm` statements
- `static_assert`
- literal operator

We propose that in all of these cases:

- No prefix is allowed
- The string is not converted to the execution encoding.
- `universal-character-name` and `simple-escape-sequence` (except `\0`) are replaced by the corresponding Unicode codepoints, and other escape sequences are ill-formed (except in asm statements, see below).

This last point is important. Because the encoding the compiler will convert these strings to is not known, and because UCNs can represent any Unicode characters, numeric-escape-sequences have no use beyond forcing the compiler to contend with invalid code units in diagnostic messages.

All of these changes are breaking changes. However, a survey of open source projects tend to show that none of the restrictions added impact existing code.

This proposal does not specify how unevaluated strings are presented in diagnostic messages.

Non printable characters and escape sequences

This proposal does not attempt to restrict further the characters allowed in unevaluated strings. In particular, they may contain all matter of space, control characters, invisible characters and alert. The handling of these characters in diagnostic messages is left as quality of implementation, mostly for simplicity. The alternative would be to only allow graphic characters (General_Category L, M, N, P, S + spaces).

Alternative considered

Allowing and ignoring any prefix

This is arguably the status quo. The issue is that it is hard to teach. Users should be able to expect for example that L"X" is always in the wide execution encoding. It could be argued that `"foo"` not being in the narrow-encoding is also confusing, however, there is precedence for that in headers names (which are already not *string-literal*s).

Allowing prefixes and encode all strings using that prefix

This is both implementer- and user-hostile. It would force users to use any of u, u8, U on all of their `static_assert` which contain non-ASCII characters as it is the only way to obtain a portable encoding. It has the advantage of being mostly consistent (all strings except those in headers names would be encoded using the encoding associated with their prefix) but would break existing code using non-ASCII characters in `static_assert` and attributes and litter C++ code with these prefixes, which seems to be a net negative.

`asm` statements

Several people in SG22, as well as an implementer raised concerns about banning numeric escape sequences in asm statement as supposedly an implementation could do "something" here. Given the implementation-defined nature of `asm` statements, we decided to only preclude encoding prefixes in `asm` statements, and allow numeric escape sequences, whose behavior would remain implementation-defined and possibly inconsistent with the behavior of string literals in other contexts.

An alternative approach outside of the scope of this paper would be to modify the specification of asm statements to allow a well-balanced token sequences, like for attributes parameters. This would better match existing practices.

Compilers survey

_Pragma

In `_Pragma` directives, the standard specifies that the `L` prefix is ignored. In C, all encoding prefixes are ignored. This divergence is highlighted in CWG897 [2]. MSVC does not support

`_Pragma(L"")`. Only Clang supports other prefixes in `_Pragma`.

Out of the 90 millions lines of code of the 1300+ open source projects available on vcpkg, a single use of that feature was found within clang's lexer test suite, for a total of 2000 uses of `_Pragma`. Similarly, the only uses of `_Pragma (u8"")`, `_Pragma (u"")`, `_Pragma (U"")`, etc were found in Clang's test suite (both because these are valid C and because neither GCC nor Clang are conforming, only `L""` is described as valid by the C++ standard).

Attributes

Clang does not support strings with an encoding prefix in attributes, other compilers accept them.

`static_assert`

All compilers support strings with an encoding prefix in static assert. MSVC appears to convert the string to the encoding associated with that prefix before displaying it, producing mojibake if a string cannot be represented in the literal encoding. The following diagnostics are emmited by MSVC with `/execution-charset:ascii`:

```
static_assert(false, "Your code is on 🔥");

<source>(1): warning C4566: character represented by universal-character-name
'\u00F0'  cannot be represented in the current code page (20127)
<source>(1): warning C4566: character represented by universal-character-name
'\u0178'  cannot be represented in the current code page (20127)
<source>(1): warning C4566: character represented by universal-character-name
'\u201D' cannot be represented in the current code page (20127)
<source>(1): warning C4566: character represented by universal-character-name
'\u00A5' cannot be represented in the current code page (20127)
<source>(1): error C2338: ????


static_assert(false, u8"Your code is on 🔥");
<source>(1): error C2002: invalid wide-character constant
```

`extern` & `asm`

No compiler support strings with an encoding prefix in extern and asm statements.

`#line`

GCC and Clang do not support encoding prefix in `#line` directives.

Future direction

This proposal does not prevent supporting constant expression in `static_assert` or attributes in the future; we can imagine the following grammar:

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

Those may make `static_assert(true, u8"foo");` valid again as `u8"foo"` would be a valid constant expression.

Implementability

This proposal requires implentations to keep around a non-encoded string for diagnostic purposes. This has recently come up in a clang patch to support EBCDIC as the literal encoding. To support diagnostics in this context, especially on a non-EBCDIC platform the original sequence of characters must be retained. This proposal offers a well-specified, portable mechanism to solve this problem.

Wording Challenges

Strings are handled in phase 5 and 6 before the program is parsed, which might force us to have a ”reversal” of these phases. *string-literal* and *unevaluated-string-literal* only differ by the context in which they may appear.

It is important to note that *unevaluated-string*, by vertue of not being evaluated, are not C++ expressions. They are purposefully left out of the *literal* grammar. Not being literal, and not being expressions, *unevaluated-string* do not have a value category.

Previous works

P2246R1 [1] removes wording specific to attributes mandating that diagnostic with characters from the basic characters are displayed in diagnostic messages, which was not implementable.

Wording

[Editor’s note: The wording is relative to N4885 + P2314R2 [3] applied]

[lex.phases] Phases of translation

[Editor’s note: Modify ”[lex.phases]/p1.6” as follow]

6. Adjacent *string-literal*s are concatenated ~~and a null character is appended to the result as specified in [lex.string]~~.

🔲 String literals [lex.string]

[Editor's note: Modify "[lex.string]" as follow]

[...]

~~In translation phase 6 [lex.phases], after adjacent *string-literal*s are concatenated, a null character is appended to the result.~~

Evaluating a *string-literal* results in a string literal object with static storage duration. Whether all *string-literal*s are distinct (that is, are stored in nonoverlapping objects) and whether successive evaluations of a *string-literal* yield the same or a different object is unspecified. [*Note:* The effect of attempting to modify a *string-literal* is undefined. — *end note*]

String literal objects are initialized with the sequence of code unit values corresponding to the *string-literal*'s sequence of *s-char* s (for a non-raw string literal) and *r-char* s (for a raw string literal) in order as follows:

- The sequence of characters denoted by each contiguous sequence of *basic-s-char* s, *r-char* s, *simple-escape-sequence* s, and *universal-character-name* s is encoded to a code unit sequence using the *string-literal*'s associated character encoding. If a character lacks representation in the associated character encoding, then:
  - If the *string-literal*'s *encoding-prefix* is absent or `L`, then the *string-literal* is conditionally-supported and an implementation-defined code unit sequence is encoded.
  - Otherwise, the *string-literal* is ill-formed.

  When encoding a stateful character encoding, implementations should encode the first such sequence beginning with the initial encoding state and encode subsequent sequences beginning with the final encoding state of the prior sequence. [*Note:* The encoded code unit sequence can differ from the sequence of code units that would be obtained by encoding each character independently. — *end note*]
- Each *numeric-escape-sequence* that specifies an integer value v contributes a single code unit with a value as follows:
  - If v does not exceed the range of representable values of the *string-literal*'s array element type, then the value is v.
  - Otherwise, if the *string-literal*'s *encoding-prefix* is absent or `L`, and v does not exceed the range of representable values of the corresponding unsigned type for the underlying type of the *string-literal*'s array element type, then the value is the unique value of the *string-literal*'s array element type `T` that is congruent to v modulo 2^N, where N is the width of `T`.
  - Otherwise, the *string-literal* is ill-formed.

When encoding a stateful character encoding, these sequences should have no effect on encoding state.

- Each *conditional-escape-sequence* contributes an implementation-defined code unit sequence. When encoding a stateful character encoding, it is implementation-defined what effect these sequences have on encoding state.
- A code unit of value 0 (representing the NULL character) is appened to the result.

[Editor's note: Add after "[lex.string]/p10"]

🔲 Unevaluated strings [lex.string.unevaluated]

unevaluated-string:
string-literal

An *unevaluated-string* shall have no *encoding-prefix*.

Each *universal-character-name* and each *simple-escape-sequence* in an *unevaluated-string* is replaced by the member of the translation set it denotes. An *unevaluated-string* which contains a *numeric-escape-sequence* or a *conditional-escape-sequence* is ill-formed.

An *unevaluated-string* is never evaluated and its interpretation depends on the context in which it appears.

[Editor's note: "translation set" is defined in P2314R2 [3] in [lex.phases]]

🔲 Declarations [dcl.dcl]

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

[...]

In a *static_assert-declaration*, the *constant-expression* shall be a contextually converted constant expression of type `bool`. If the value of the expression when so converted is `true`, the declaration has no effect. Otherwise, the program is ill-formed, and the resulting diagnostic message shall include the text of the *unevaluated-string* ~~*literal*~~, if one is supplied, except that characters not in the basic source character set are not required to appear in the diagnostic message. [*Example:*

```
static_assert(sizeof(int) == sizeof(void*), "wrong pointer size");
```

— end example]

The `asm` declaration [dcl.asm]

An `asm` declaration has the form

asm-declaration:
attribute-specifier-seq$_{opt}$ `asm` `(` *string-literal* `)` `;`

The `asm` declaration is conditionally-supported; its meaning is implementation-defined. The encoding prefix of the *string-literal* shall be absent. The optional *attribute-specifier-seq* in an *asm-declaration* appertains to the `asm` declaration. [*Note:* Typically it is used to pass information through the implementation to an assembler. *— end note*]

Linkage specifications [dcl.link]

All functions and variables whose names have external linkage and all function types have a *language linkage*. [*Note:* Some of the properties associated with an entity with language linkage are specific to each implementation and are not described here. For example, a particular language linkage might be associated with a particular form of representing names of objects and functions with external linkage, or with a particular calling convention, etc. *— end note*] The default language linkage of all function types, functions, and variables is C++ language linkage. Two function types with different language linkages are distinct types even if they are otherwise identical.

Linkage between C++ and non-C++ code fragments can be achieved using a *linkage-specification*:

linkage-specification:
`extern` *unevaluated-string-~~literal~~* `{` *declaration-seq*$_{opt}$ `}`
`extern` *unevaluated-string-~~literal~~* *declaration*

The *unevaluated-string-~~literal~~* indicates the required language linkage as a sequence of translation set characters.

This document specifies the semantics for the ~~*string-literals*~~ language linkages `"C"` and `"C++"`. Use of a ~~*string-literal*~~ language linkage other than `"C"` or `"C++"` is conditionally-supported, with implementation-defined semantics. [*Note:* Therefore, a linkage-specification with a ~~*string-literal*~~ language linkage that is unknown to the implementation requires a diagnostic. *— end note*] [*Note:* It is recommended that the spelling of the ~~*string-literal*~~ language linkage be taken from the document defining that language. For example, `Ada` (not `ADA`) and `Fortran` or `FORTRAN`, depending on the vintage. *— end note*]

Every implementation shall provide for linkage to the C programming language, `"C"`, and C++, `"C++"`. [*Example:*

```
complex sqrt(complex);          // C++ language linkage by default
extern "C" {
    double sqrt(double);          // C language linkage
}
```

— end example]

// [...]

🞐 Attributes [dcl.attr]

🞐 Deprecated attribute [dcl.attr.deprecated]

The *attribute-token* `deprecated` can be used to mark names and entities whose use is still allowed, but is discouraged for some reason. [*Note:* In particular, `deprecated` is appropriate for names and entities that are deemed obsolescent or unsafe. *— end note*] It shall appear at most once in each *attribute-list*. An *attribute-argument-clause* may be present and, if present, it shall have the form:

(*unevaluated-string~~-literal~~*)

[*Note:* The *unevaluated-string~~-literal~~* in the *attribute-argument-clause* can be used to explain the rationale for deprecation and/or to suggest a replacing entity. *— end note*]

🞐 Nodiscard attribute [dcl.attr.nodiscard]

The *attribute-token* `nodiscard` may be applied to the *declarator-id* in a function declaration or to the declaration of a class or enumeration. It shall appear at most once in each *attribute-list*. An *attribute-argument-clause* may be present and, if present, shall have the form:

(*unevaluated-string~~-literal~~*)

A name or entity declared without the `nodiscard` attribute can later be redeclared with the attribute and vice-versa. [*Note:* Thus, an entity initially declared without the attribute can be marked as `nodiscard` by a subsequent redeclaration. However, after an entity is marked as `nodiscard`, later redeclarations do not remove the `nodiscard` from the entity. *— end note*] Redeclarations using different forms of the attribute (with or without the *attribute-argument-clause* or with different *attribute-argument-clause* s) are allowed.

A *nodiscard type* is a (possibly cv-qualified) class or enumeration type marked `nodiscard` in a reachable declaration. A *nodiscard call* is either

- a function call expression that calls a function declared `nodiscard` in a reachable declaration or whose return type is a nodiscard type, or
- an explicit type conversion (**??**, **??**, **??**) that constructs an object through a constructor declared `nodiscard` in a reachable declaration, or that initializes an object of a nodiscard type.

Recommanded: Appearance of a nodiscard call as a potentially-evaluated discarded-value expression is discouraged unless explicitly cast to `void`. Implementations should issue a warning in such cases. [*Note:* This is typically because discarding the return value of a nodiscard call has surprising consequences. *— end note*] The *unevaluated-string~~-literal~~* in a

`nodiscard` *attribute-argument-clause* should be used in the message of the warning as the rationale for why the result should not be discarded.

🔲 User-defined literals [over.literal]

literal-operator-id:
`operator` ~~*string-literal*~~ *unevaluated-string* *identifier*
`operator` *user-defined-string-literal*

The ~~*unevaluated-string* or~~ *user-defined-string-literal* in a *literal-operator-id* shall have no *encoding-prefix* and shall contain no characters other than the implicit terminating `'\0'`. The *ud-suffix* of the *user-defined-string-literal* or the *identifier* in a *literal-operator-id* is called a *literal suffix identifier*. Some literal suffix identifiers are reserved for future standardization; see **??**. A declaration whose *literal-operator-id* uses such a literal suffix identifier is ill-formed, no diagnostic required.

🔲 Preprocessing directives [cpp]

[...]

🔲 Line control [cpp.line]

The *string-literal* of a `#line` directive, if present, shall ~~be a character string literal~~ satisfy the semantic constraints of an *unevaluated-string* [lex.string.unevaluated].

The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 while processing the source file to the current token.

A preprocessing directive of the form `# line` *digit-sequence new-line*
causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). If the digit sequence specifies zero or a number greater than 2147483647, the behavior is undefined.

A preprocessing directive of the form

`# line` *digit-sequence* ~~*" s-char-sequence$_{opt}$ "*~~ *string-literal* *new-line*

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the ~~character~~ string literal.

A preprocessing directive of the form

`# line` *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `line` on the directive are processed just as in normal text (each identifier currently defined

as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.

🔲 Pragma operator [cpp.pragma.op]

The *string-literal* of a Pragma operator shall satisfy the semantic constraints of an *unevaluated-string* [lex.string.unevaluated].

A unary operator expression of the form:

`_Pragma (` *string-literal* `)`

is processed as follows: The *string-literal* is *destringized* by ~~deleting the `L` prefix, if present,~~ deleting the leading and trailing double-quotes ~~,replacing each escape sequence `\"` by a double-quote, and replacing each escape sequence `\\` by a single backslash~~. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

[*Example:*

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ( "listing on \"..\\listing.dir\"" )
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)

LISTING( ..\listing.dir )
```

— *end example*]

Acknowledgments

Thank you to Masayoshi Kanke and Peter Brett who offered valuable feedback on this paper!

References

[1] Aaron Ballman. P2246R1: Character encoding of diagnostic text. https://wg21.link/p2246r1, 1 2021.

[2] Daniel Krügler. CWG897: _pragma and extended string-literals. https://wg21.link/cwg897, 5 2009.

[3] Jens Maurer. P2314R2: Character sets and encodings. https://wg21.link/p2314r2, 5 2021.

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4885