Unevaluated strings literals

Document #: P2361R0
Date: 2021-04-14
Project: Programming Language C++
Audience: SG-16, EWG
Reply-to: Corentin Jabot <corentin.jabot@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.

Proposal

We propose than 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.

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.

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-literals*s).

Allowing prefixes and encode all strings using that prefix

his 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.

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 [1]. 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-literal*) ;
`static_assert` (*constant-expression* , *constant-expression*) ;

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

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.

Wording

Relative to N4885 + P2314 applied.

String literals [lex.string]

// [...]

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.

An *unevaluated string-literal* is a *string-literal* which shall have no `encoding-prefix`. An *unevaluated string-literal* which contains a *numeric-escape-sequence* or a *conditional-escape-sequence* is ill-formed. Each *universal-character-name* and each *simple-escape-sequence* in an *unevaluated-string-literal* is replaced by the member of the translation set it denotes. *unevaluated-string-literal* shall not contain the null character.

🔗 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` `(` *unevaluated-string-literal* `)` ;

The `asm` declaration is conditionally-supported; its meaning is implementation-defined. 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. This document specifies the semantics for the *unevaluated-string-literal*s `"C"` and `"C++"`. Use of a *unevaluated-string-literal* other than `"C"` or `"C++"` is conditionally-supported, with implementation-defined semantics. [*Note:* Therefore, a linkage-specification with a *unevaluated-string-literal* that is unknown to the implementation requires a diagnostic. — *end note*] [*Note:* It is recommended that the spelling of the *unevaluated-string-literal* 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.

🔲 Preprocessing directives [cpp]

🔲 Preamble [cpp.pre]

preprocessing-file:
group$_{opt}$
module-file

module-file:
pp-global-module-fragment$_{opt}$ *pp-module group*$_{opt}$ *pp-private-module-fragment*$_{opt}$

pp-global-module-fragment:
`module` ; *new-line group*$_{opt}$

pp-private-module-fragment:
`module` : `private` ; *new-line group*$_{opt}$

group:
group-part
group group-part

group-part:
control-line
if-section
text-line
`#` *conditionally-supported-directive*

control-line:
`# include` *pp-tokens new-line*
pp-import
`# define` *identifier replacement-list new-line*
`# define` *identifier lparen identifier-list*$_{opt}$ `)` *replacement-list new-line*
`# define` *identifier lparen* `...` `)` *replacement-list new-line*
`# define` *identifier lparen identifier-list* `,` `...` `)` *replacement-list new-line*
`# undef` *identifier new-line*
`# line` *pp-tokens new-line*
`# error` *pp-tokens*$_{opt}$ *new-line*
`# pragma` *pp-tokens*$_{opt}$ *new-line*
`#` *new-line*

if-section:
if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
`# if` *constant-expression new-line group*$_{opt}$
`# ifdef` *identifier new-line group*$_{opt}$
`# ifndef` *identifier new-line group*$_{opt}$

elif-groups:
elif-group
elif-groups elif-group

elif-group:
`# elif` *constant-expression new-line group*$_{opt}$

else-group:
`# else` *new-line group*$_{opt}$

endif-line:
`# endif` *new-line*

text-line:
pp-tokens$_{opt}$ *new-line*

conditionally-supported-directive:
pp-tokens new-line

lparen:
a `(` character not immediately preceded by whitespace

identifier-list:
identifier
identifier-list `,` *identifier*

replacement-list:
pp-tokens$_{opt}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

A *preprocessing directive* consists of a sequence of preprocessing tokens that satisfies the following constraints: At the start of translation phase 4, the first token in the sequence, referred to as a *directive-introducing token*, begins with the first character in the source file (optionally after whitespace containing no new-line characters) or follows whitespace containing at least one new-line character, and is

- a `#` preprocessing token, or
- an `import` preprocessing token immediately followed on the same logical line by a *header-name*, `<`, *identifier*, *unevaluated-string-literal*, or `:` preprocessing token, or
- a `module` preprocessing token immediately followed on the same logical line by an *identifier*, `:`, or `;` preprocessing token, or
- an `export` preprocessing token immediately followed on the same logical line by one of the two preceding forms.

The last token in the sequence is the first token within the sequence that is immediately followed by whitespace containing a new-line character. [1]hus, preprocessing directives are commonly called "lines". These "lines" have no other syntactic significance, as all whitespace is equivalent except in certain situations during preprocessing (see the `#` character string literal creation operator in **??**, for example). [*Note:* A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro. — *end note*]

[*Example:*

```
#                       // preprocessing directive
module ;                // preprocessing directive
export module leftpad;  // preprocessing directive
import <string>;        // preprocessing directive
export import "squee";  // preprocessing directive
import rightpad;        // preprocessing directive
import :part;           // preprocessing directive

module                  // not a preprocessing directive
```

[1] T

```
;                       // not a preprocessing directive

export                  // not a preprocessing directive
import                  // not a preprocessing directive
foo;                    // not a preprocessing directive

export                  // not a preprocessing directive
import foo;             // preprocessing directive (ill-formed at phase 7)

import ::               // not a preprocessing directive
import ->               // not a preprocessing directive
```

— *end example*]

A sequence of preprocessing tokens is only a *text-line* if it does not begin with a directive-introducing token. A sequence of preprocessing tokens is only a *conditionally-supported-directive* if it does not begin with any of the directive names appearing after a `#` in the syntax. A *conditionally-supported-directive* is conditionally-supported with implementation-defined semantics.

At the start of phase 4 of translation, the *group* of a *pp-global-module-fragment* shall contain neither a *text-line* nor a *pp-import*.

When in a group that is skipped, the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

The only whitespace characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the directive-introducing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other whitespace characters in translation phase 3).

The implementation can process and skip sections of source files conditionally, include other source files, import macros from header units, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

[*Example:* In:

```
#define EMPTY
EMPTY   #   include <file.h>
```

the sequence of preprocessing tokens on the second line is *not* a preprocessing directive, because it does not begin with a `#` at the start of translation phase 4, even though it will do so after the macro `EMPTY` has been replaced. — *end example*]

🔲 Conditional inclusion [cpp.cond]

defined-macro-expression:
`defined` *identifier*
`defined (` *identifier* `)`

h-preprocessing-token:
any *preprocessing-token* other than >

h-pp-tokens:
h-preprocessing-token
h-pp-tokens h-preprocessing-token

header-name-tokens:
unevaluated-string-literal
< *h-pp-tokens* >

has-include-expression:
`__has_include (` *header-name* `)`
`__has_include (` *header-name-tokens* `)`

has-attribute-expression:
`__has_cpp_attribute (` *pp-tokens* `)`

The expression that controls conditional inclusion shall be an integral constant expression except that identifiers (including those lexically identical to keywords) are interpreted as described below [2]ecause the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, etc. and it may contain zero or more *defined-macro-expression*s and/or *has-include-expression*s and/or *has-attribute-expression*s as unary operator expressions.

A *defined-macro-expression* evaluates to `1` if the identifier is currently defined as a macro name (that is, if it is predefined or if it has one or more active macro definitions, for example because it has been the subject of a `#define` preprocessing directive without an intervening `#undef` directive with the same subject identifier), `0` if it is not.

The second form of *has-include-expression* is considered only if the first form does not match, in which case the preprocessing tokens are processed just as in normal text.

🔲 Line control [cpp.line]

~~The *string-literal* of a `#line` directive, if present, shall be a character string literal.~~ 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.

[2] B

A preprocessing directive of the form

line *digit-sequence* " ~~*s-char-sequence*~~$_{opt}$ *unevaluated-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]

A unary operator expression of the form:

_Pragma (*unevaluated-string-literal*)

is processed as follows: The *unevaluated-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 Peter Brett who offered valuable feedback on this paper!

References

References

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

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