Abstract

_string-literals_ 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

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
- header-name
- [[nodiscard]] and [[deprecated]] attributes
- extern linkage specifications
- asm statements
- static_assert

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.

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 \textit{string-literals}).

\textbf{Allowing prefixes and encode all strings using that prefix}

This is both implementer- and user-hostile. It would force users to use any of \texttt{u}, \texttt{u8}, \texttt{U} on all of their \texttt{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 \texttt{static\_assert} and attributes and litter C++ code with these prefixes, which seems to be a net negative.

\textbf{Compilers survey}

\_Pragma

In \_Pragma directives, the standard specifies that the \texttt{L} prefix is ignored. In C, all encoding prefixes are ignored. This divergence is highlighted in \texttt{CWG897} \cite{2}. MSVC does not support \_Pragma(\texttt{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 (\texttt{u8""}), \_Pragma (\texttt{u""}), \_Pragma (\texttt{U""}), etc were found in Clang's test suite (both because these are valid C and because neither GCC nor Clang are conforming, only \texttt{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.

\textbf{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 \texttt{/execution-charset:ascii}:

\begin{verbatim}
static_assert(false, "Your code is on 🔥");
\end{verbatim}

\begin{verbatim}
<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 '\u0178' cannot be represented in the current code page (20127)
\end{verbatim}
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 implementations 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 virtue 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]. *Adjacent unevaluated-string* s *are concatenated*.

![](lex.charset)

**Character sets**

[Editor's note: Modify “5.3.2” as follow]

A *universal-character-name* designates the character in ISO/IEC 10646 (if any) whose code point is the hexadecimal number represented by the sequence of *hexadecimal-digit* s in the *universal-character-name*. The program is ill-formed if that number is not a code point or if it is a surrogate code point. Noncharacter code points and reserved code points are considered to designate separate characters distinct from any ISO/IEC 10646 character. If a *universal-character-name* outside the *c-char-sequence*, *s-char-sequence*, or *r-char-sequence* of a *character-literal* or *string-literal*, *unevaluated-string* (in either case, including within a *user-defined-literal*) corresponds to a control character or to a character in the basic source character set, the program is ill-formed.

![](lex.pptoken)

**Preprocessing tokens**

[Editor's note: Modify “5.4 Preprocessing tokens” as follow]
Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a literal, or an operator or punctuator.

A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: header names, placeholder tokens produced by preprocessing `import` and `module` directives (import-keyword, module-keyword, and export-keyword), identifiers, preprocessing numbers, character literals (including user-defined character literals), string literals (including user-defined string literals and unevaluated string), preprocessing operators and punctuators, and single non-whitespace characters that do not lexically match the other preprocessing token categories. If a ‘ or a “ character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by whitespace; this consists of comments, or whitespace characters (space, horizontal tab, newline, vertical tab, and form-feed), or both. As described in ??, in certain circumstances during translation phase 4, whitespace (or the absence thereof) serves as more than preprocessing token separation. Whitespace can appear within a preprocessing token only as part of a header name or between the quotation characters in a character literal or string literal.

### String literals

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

```plaintext
string-literal:
    encoding-prefix_opt " s-char-sequence_opt "
    encoding-prefix_opt R raw-string
unevaluated-string:
    " s-char-sequence_opt "
    R raw-string
s-char-sequence:
    s-char
    s-char-sequence s-char
s-char:
    basic-s-char
    escape-sequence
    universal-character-name
```
basic-s-char:
    any member of the basic source character set except the double-quote ",", backslash \, or new-line character

raw-string:
    " d-char-sequence_opt ( r-char-sequence_opt ) d-char-sequence_opt "

r-char-sequence:
    r-char
    r-char-sequence r-char

r-char:
    any member of the source character set, except a right parenthesis ) followed by
    the initial d-char-sequence (which may be empty) followed by a double quote ".

d-char-sequence:
    d-char
    d-char-sequence d-char

d-char:
    any member of the basic source character set except:
    space, the left parenthesis (, the right parenthesis ), the backslash \, and
    the control characters
    representing horizontal tab, vertical tab, form feed, and newline.

[...]

A string-literal or an unevaluated-string that has an R in the prefix is a raw string literal. The
d-char-sequence serves as a delimiter. The terminating d-char-sequence of a raw-string is the
same sequence of characters as the initial d-char-sequence. A d-char-sequence shall consist of
at most 16 characters.

In translation phase 6, adjacent string-literals are concatenated. If both string-literals have
the same encoding-prefix, the resulting concatenated string-literal has that encoding-prefix. If
one string-literal has no encoding-prefix, it is treated as a string-literal of the same encoding-
prefix as the other operand. If a UTF-8 string literal token is adjacent to a wide string literal
token, the program is ill-formed. Any other concatenations are conditionally-supported with
implementation-defined behavior.Adjacent unevaluated-strings are concatenated.

[ Note: This concatenation is an interpretation, not a conversion. Because the interpretation
happens in translation phase 6 (after the string literal contents have been encoded in the
string-literal's associated character encoding), a string-literal's initial rawness has no effect on
the interpretation or well-formedness of the concatenation. — end note ]

[...]

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.

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

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

Preamble

```
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

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

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-literals* "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:

```c
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:
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]
control-line:
    # include pp-tokens new-line
    pp-import
    # define identifier replacement-list new-line
    # define identifier (paren identifier-list_opt ) replacement-list new-line
    # define identifier (paren . . . ) replacement-list new-line
    # define identifier (paren 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. Thus, 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
;          // 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:
```cpp
#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]**

```cpp
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 ( identifier )`
  `__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 because 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-expressions and/or has-include-expressions and/or has-attribute-expressions 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

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.

A preprocessing directive of the form

   # line digit-sequence "s-char-sequence-opt" unevaluated-string 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

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:

```c
#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:

```c
#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

References


