Document number:	N4220
Date:	2014-10-10
Working group:	SG12
Reply to:	David Krauss (david_work at me dot com)
Revises:	N3882
References:	N4219

An update to the preprocessor specification (rev. 2)

1. Motivation and scope

The C preprocessor specification inherited by C++ uses undefined behavior to specify latitude for implementation differences. This technically allows one compiler to produce a defective executable from a particular program, while another fills the void with useful features. In theory this makes porting programs dangerous, but in practice consensus has been reached regarding the meaning of most programs invoking undefined behavior, and programmers have little reason to worry. By capturing the status quo in the standard, safety can be properly guaranteed, and the preprocessor may be more completely understood without referring to platform documentation.

Additionally, the preprocessor has not kept pace with C++11 lexical extensions (raw strings, user-defined literals, and new encoding prefixes), resulting in corner cases lacking specification.

This proposal updates the preprocessor specification to remove undefined behavior, missing specifications, and possible contradictions, and to better document implementation differences. It resolves CWG DR 268, 366, 897, 1436, 1625, 1698, 1709, 1718, and 1882. Issues related to universal-character-names are not addressed here, but in N4219.

Table of Contents

2. Principles

Recent versions of popular implementations are surveyed (GCC 4.9, MSVC 12, Clang 3.4, and EDG/ICC 13) and used to form a consensus. Normative changes and requirements of behavioral change to these platforms are to be minimized, with a preference to explicitly requiring documentation of features. Constructs supported without by-default diagnosis (as documented features, not merely working as such) by any platform will not become ill-formed.

Sometimes a conceptual model is provided by the current specification, but it may not be fully realized due to some implementation difficulty, evident in current practice or in specification as undefined behavior. These cases should be defined uniformly by the model and conditionally supported. This allows reasonable implementation strategies to be fully conforming, yet provides a language without conceptual "holes," that may be reasonably implemented by closely following the model.

In cases where no model is evident but the popular implementations follow some pattern, a simple model will be developed to standardize current practice.

As a last resort, in cases lacking a conceptual model or consensus, or where diagnosis may be infeasible, “ill-formed with no diagnosis required” is used in preference to undefined behavior. This clarifies that an implementation should not attempt any specific behavior besides diagnosis.

The objective is a properly standardized language where every well-formed program means the same thing across platforms, although not all platforms will immediately accept all well-formed programs. No new features are proposed except where already widely implemented, so implementation programming effort is kept to a minimum.

3. Issues of specified undefined behavior

Cases listed in this section are problematic because the standard specifies undefined behavior, which guarantees neither diagnostics nor safe use. Such a construct may be recommended by the documentation of one implementation yet quietly elicit a defective product from another implementation. These may be considered to be highest priority.

N3801 has proposed that all such cases be reclassified as diagnosable errors, but more care is taken here to respect existing practice and experience, and to minimize implementation work.

3.1. Unterminated literals

According to [lex.pptoken] §2.5/2, a single or double quote without a partner on the same line produces undefined behavior, rather than becoming its own token. Such unmatched single character tokens do not appear in the language grammar, and they render the program ill-formed unless in a macro argument which always is either ignored or stringized. This can be considered a conceptual model, and it is in these terms that the undefined behavior is specified.

Another reasonable behavior in the case of the double quote would be to accept embedded newlines in multi-line strings.

```
#define comment( x )
comment( ' )
comment( " )
```

GCC, Clang, EDG, and MSVC all complain of an unterminated macro invocation in this test, or an isolated test of either case, in addition to diagnosing newlines in both literals. The undefined behavior should be respecified as an error.

3.2. Interactions of the `defined` operator

According to [cpp.cond] §16.1/4, the `defined` operator must not be generated by macro replacement. The C++11 conceptual model does not suggest how it would be evaluated in such a case.

```
#define cheeseburger

#define can_haz( x, y ) defined x ## y
#if can_haz( cheese, burger )
#error generated defined works with external suppression
#endif

#define d( x ) defined x
#if d( cheeseburger )
#error "define x" suppresses argument expansion
#endif
```

GCC, EDG, and Clang all support generating the defined operator, but only if the user suppresses argument expansion using the concatenation operator. MSVC does not seem to usefully support generating the defined operator, nor does it ever diagnose the condition. Only GCC will produce a diagnosis in any case, and only if both the `-std` and `--pedantic` flags are set.

The means of evaluation of the operator varies, though. EDG treats `defined` as a preprocessing operator and substitutes the tokens `0L` or `1L` as appropriate during macro expansion. Clang and GCC delay `defined` evaluation until after macro expansion, although during macro expansion the operand token is deleted. These differences may be observed by passing the `defined` expression to the concatenation and stringize operators, and both behaviors are conforming.

Undefined behavior also results if the operand to `defined` is missing or not an identifier. All implementations already diagnose this condition.

These constraints should be sufficient to update C++11 to reflect existing practice:

1. If the `defined` operator appears in the replacement list of a macro after argument substitution and prior to rescanning, it is considered to be used, and its evaluation is conditionally-supported.

2. If a `defined` operator or its operand is the subject of a preprocessor operator (# or ##), the program is ill-formed, no diagnostic required.

3. A use of the `defined` operator is not conceptually replaced until the controlling expression is evaluated. (The program is forbidden to tell the difference; this is intended to give broader effect to the previous rule.)

4. If a use of the `defined` operator does not match one of the two described forms, the program is ill-formed.

The only implementation work is that MSVC should document and diagnose lack of conditional support.

3.3. Implementation-specific forms of `#include`

§16.2 gives implementations the freedom to define nonstandard forms of the `#include` directive; that is, there may be additional text after the header-name, or missing angle brackets may be tolerated. None of the reviewed implementations provided for the former according to their documentation, or allowed the latter.

```
#include <iostream> 42 // Implementation-specific syntax
#include iostream // Missing angle brackets
```

The undefined behavior may be respecified as an error with no ill consequences.

3.4. Directives inside macro argument lists

[cpp.replace] §16.3/11 allows an implementation to recognize directives within a macro invocation, by specification as undefined behavior.

```
#define nullary()
#define assert_empty(x) nullary(x)
assert_empty(
#define x // UB: directive inside argument list
x // OK if directive was executed before replacement finishes
)
```

This behavior is documented in the GCC preprocessor manual, §3.9, which also records that it was added to make function-like macro invocations behave more like function calls. Clang and GCC warn only under `--pedantic` mode, and EDG makes no diagnosis. MSVC treats the tokens normally, not as a directive, and makes no diagnosis. (Because of the hash token, though, any directive-like sequence must be either ignored or stringized.)

The surveyed implementations do not diagnose the condition and support is very widespread. A simple conceptual model is that executing directives appearing inside a macro argument list before the macro expansion be conditionally-supported. However, corner cases arise regarding the scope of macro definitions. GCC accepts a macro use syntactically before its definition, and also accepts the undefinition of a macro within its own invocation. The latter case is called "pathological" by the GCC manual, but the former is a consequence of the very motivation of the feature.

The basic principles needed to support the motivating case are:

1. Directives behave normally where possible. Each use of a directive is individually conditionally-supported.
2. A macro may be used within an argument list while it is in scope, between its `#define` and `#undef`, provided that the scope extends past the end of the argument list.
3. The implementation need not diagnose violations of #2 because of the difficulty in identifying the scope context of each macro replacement operation.

Note that there is no possibility of nested directives because a directive cannot contain a newline, and a macro invocation cannot contain a directive without using a newline.

To be sure, the motivating case in C++ is weak because inline functions uniformly supersede function-like macro interfaces. But, requiring the replacement of working support with error diagnoses does not really help anyone. The solution should probably be coordinated with C. The benefits of preprocessor undefined behavior expurgation are shared equally across languages.

The only implementation work is that MSVC should document and diagnose lack of conditional support.

3.5. Stringize failure

[cpp.stringize] §16.3.2/2 defines a failure condition for the stringize operator but specifies undefined behavior, not diagnosis.

```
#define s_lit( x ) # x
s_lit( \ ) // Stringizes to "\", which is not a valid token.
```

GCC and Clang helpfully delete the offending backslash, and emit an explanatory warning. The diagnoses from EDG and MSVC are the same as for an unterminated literal.

The undefined behavior should be respecified as an error. No implementation work is necessary, although Clang and GCC might promote the warning diagnosis to an error.

3.6. Concatenation failure

[cpp.concat] §16.3.3/3 requires that concatenation produce exactly one result token, but specifies undefined behavior, not diagnosis. However, because the associativity of the concatenation operator is undefined, correctly diagnosing a sequence of concatenations would require analyzing every subsequence, with quadratic complexity.

```
#define cat3(x,y,z) x ## y ## z
cat3( !, !, ! ) // Easy to diagnose
cat3( 1e, +, 3 ) // Harder to diagnose
```

Clang and GCC treat the easy case as an error. EDG provides a warning. MSVC does not diagnose it. None of the implementations complain of the invalid intermediate result of right-hand association, `+3`, in the hard case.

Existing practice would be best reflected by specifying concatenation to be left-associative. This provides a practical diagnosable rule to replace the undefined behavior. Only MSVC requires a diagnosis to be added to comply with this rule.

The GCC documentation notes that older GNU preprocessor versions were not reliably left-associative, so there may be C preprocessors in the wild that violate this rule. Divergence between C and C++ should be acceptable as left-associativity is only more specific, not more restrictive, it is ubiquitous, and it is simple (if not the simplest behavior) to implement.

No implementation work is required except that MSVC must diagnose the condition.

3.7. `#line` overflow

The `#line` directive is required to support natural numbers up to `(2<<31)-1`, but other numbers produce undefined behavior ([cpp.line] §16.4/3).

```
#line 5000000000 // undefined behavior
```

GCC and Clang do not diagnose overflow but wrap around using 32-bit `unsigned` modulo arithmetic. EDG does diagnose overflow as an error, and provides 64-bit `unsigned` arithmetic. MSVC issues a warning if the number does not fit into 24 bits, but provides 32-bit, two's complement arithmetic anyway, which may cause `__LINE__` to expand to a negative number.

In any case, the `#line` directive does not set an upper bound on the results obtained by `__LINE__`. Merely prepending a few gigabytes of empty lines to a valid program is enough to elicit inexplicable behavior from any implementation except EDG.

The only valid conceptual model is an implementation-defined maximum line number. Overflow of this quantity should be diagnosed as any other implementation limit such as those listed in Annex B. It appears that such diagnosis is not required, so normative specification can probably be removed and replaced with a non-normative suggested minimum limit. This would bring popular implementations into compliance.

3.8. Implementation-specific forms of `#line`

[cpp.line] §16.4/5 gives implementations the freedom to define nonstandard forms of the `#line` directive, in terms of undefined behavior. No such extensions have been documented among the surveyed implementations, but they diagnose the condition, so a diagnostic requirement would have no impact.

3.9. Redefining predefined macros

DR 1882.[1] Applying `#define` or `#undef` to a predefined macro name or `defined` produces undefined behavior ([cpp.predefined] §16.8/4). Behavior in practice varies widely, because `__LINE__` and `__FILE__` behave more like keywords than macros, while other macros, such as `__DATE__`, a user might wish to override. Furthermore, all popular implementations support the `__COUNTER__` macro, whose name is not reserved except to the library, but do not treat its redefinition differently from `__LINE__`.

More generally, the identifiers reserved to the library as macros under the umbrella of “for any use” ([global.names] §17.6.4.3.2) are often defined by implementations as extension keywords, regardless of use of library facilities ([constraints.overview] §17.6.4.1). This is one case where undefined behavior is appropriate. A user can expect nothing more from a program containing such an identifier, until the implementation’s documentation has been consulted. However, there is also no need to forbid such keywords from being defined by the implementation for the user.

[1] All defect reports mentioned here belong to CWG.

4. Issues of missing specification

These cases reflect rifts between the evolving C++ language and the preprocessor.

4.1. Generating a raw string

Raw string reversion seems not to apply within generated tokens, but it is not clear.

The preprocessor operators # and ## specify their results in terms of "spelling of each preprocessing token in the argument" and "the preceding preprocessing token is concatenated with the following preprocessing token," respectively. These suggest simple textual operations. Furthermore, if the result is not a valid preprocessor token, the behavior is undefined, which is consistent with blind text manipulation. However, all popular implementations in fact re-tokenize the concatenation using a re-run of phase 3. The only alternatives are to catch malformed preprocessing tokens during semantic analysis (at the expense of performance and error tolerance), or to attempt no diagnosis, and perhaps crash.

The `_Pragma` operator is specified directly in terms of processing a string through phase 3 to produce tokens.

Tokenizing, which can reasonably be considered synonymous with phase 3, is specified in terms of "the input stream" ([lex.pptoken] §2.5/3), which certainly corresponds to the output of phase 2, seemingly corresponds to the result of `_Pragma` destringizing, and perhaps corresponds to the character sequence being converted to a single preprocessing token by # or ##.

When a raw string token is formed by phase 3, transformations by phases 1 and 2 are reverted, retroactively. In practice, phases 1, 2, and 3 are coupled and run simultaneously, "reversion" involves disabling phases 1 and 2, and preprocessor operators run phase 3 as a subroutine on a string buffer initialized according to the specified text, to obtain object-oriented token representation. But, another plausible model would avoid string buffers in favor of references to the physical file text. In this case reverting phase 1 and 2 transformations during macro expansion would be more natural.

No popular implementation reverts the effects of phases 1 and 2 during phase 4. All implementations diagnose a problem only if the raw string is improperly delimited. The standard should reflect this behavior, but because it imposes some structural requirements upon the implementation, forming a raw string literal by concatenating an `R` with a non-raw literal, or by `_Pragma` destringizing, should be conditionally-supported. To be clear, support requires *not* reverting trigraphs and such.

4.2. C++11 string literals passed to `_Pragma`

DR 897. The `_Pragma` operator specification does not account for encoding prefixes besides `L`. This can easily be rectified by changing the literal `L` to the *encoding-prefix* production. Although common implementations do not yet support this, it should be easy enough to do, and furthermore it seems too trivial to merit a specification as conditionally-supported.

Raw string literal prefixes cannot simply be ignored. Raw strings are perfectly suited to quoting source code, and therefore `_Pragma` operands. An obvious conceptual model is that delimiters should be erased from a raw string token, and the result is considered already destringized. As a nontrivial change, this should be conditionally-supported.

The current verbiage "through phase 3" is also unclear as to whether phases 1 and 2 are included. To allow normal handling of extended characters in raw strings phase 1 is a necessity, and then phase 2 naturally follows.

4.3. Stringizing a raw string

DR 1709. Stringizing is commonly used to produce debugging and diagnostic messages from source code. Presently if a raw string contains a newline character it will produce an invalid string literal token when stringized. The obvious solution is to replace each newline character in any stringized string (or character) literal with the `\n` escape code. Because this naturally fits into the search-and-replace operations already performed by stringize, this should be a minor change for implementations, and does not require a specification as conditionally-supported.

4.4. Modern constant expressions in preprocessor expressions

DR 1436, 366. C and C++03 essentially defined integral constant expressions in terms the preprocessor could understand. C++11 moved beyond this and C++1y goes further. Presently the preprocessor is required to evaluate user-defined literals, support indexing string literals, and quietly convert floating-point values to integers, among other aberrations.

The proposed resolution in DR 1436 misses some tokens and assumes that lambda expressions will never be allowed, which may not remain true for long. (Without the `return` keyword, a lambda cannot produce a result, but it can still exist.) A modified approach is proposed based on non-terminal productions. Note that the comma operator is not allowed even within parentheses.

(DR 366 was already resolved, but the fix was later undone.)

4.5. Unbalanced `#if` in a header file

Preprocessing-file appears as a grammatical construct ([cpp] §16/1), but it is not used or constrained by anything. In the C standard from whence it came, it represents the contents of a header file (C11 §5.1.1.1). The effect of identifying preprocessing-file with a header inclusion is that the grammar requires each `#if` group to balance within the same file. Otherwise, the specification is ambiguous enough that preprocessing-file may be construed as what C calls a "preprocessing translation unit."

5. Issues of inconsistent specification

These cases are inherent defects of the current standard and may be resolved without changing its meaning.

5.1. Precedence of the stringize operator

As a unary, prefix operator, the natural way to read and to parse stringize is with higher precedence than concatenation. All popular implementations do it this way. The specification also only provides for replacing a parameter in the replacement list, using tokens from the argument list. There is no specification of how to perform stringize "after" concatenation, yet "The order of evaluation of # and ## operators is unspecified" ([cpp.stringize 16.3.2/2]).

Furthermore, with its variety of encoding-prefixes and ud-suffixes, C++11 suggests more applications for a combination of operators than previous dialects.

For example, as a prefix:

```
#define UCN_LAMBDA \u03bb
#define UCN_CHARCODE( UCN ) * U ## # UCN
char32_t lambda_code = UCN_CHARCODE( UCN_LAMBDA )
```

as a suffix:

```
constexpr int operator "" _hash ( char const *, std::size_t );
#define HASH_CONSTANT( SEL ) \
  constexpr int SEL ## _hashcode = # SEL ## _hash;
```

The phase 6 string literal concatenation rules are designed to allow these without token manipulations, by replacing such concatenation operators with a pair of empty quotes attached to the prefix or suffix. But many users will not be aware of, or may ignore, this alternative.

To standardize current practice, and ensure that the standard does not suggest ambiguity where behavior is unambiguously defined, the stringize operator should be given high precedence.

5.2. Lexing the ud-suffix after a raw string literal

The lexing policy [lex.pptoken] §2.5/3 says

> If the next character begins a sequence of characters that could be the prefix and initial double quote of a raw string literal, such as `R"`, the next preprocessing token shall be a raw string literal. … The raw string literal is defined as the shortest sequence of characters that matches the raw-string pattern
>
> *encoding-prefix*$_{opt}$ `R` *raw-string*

This does not provide for a ud-suffix after the raw-string, and furthermore a ud-suffix (or identifier) cannot be parsed as a "shortest sequence of characters" but needs max munch. The min munch specification is redundant anyway: the *r-char* production already excludes the termination sequence, so raw strings terminate the same way as other strings.

Removing the above-quoted sentences, the remainder of the paragraph defines raw string semantics, and it would be much better placed in [lex.string] among the other paragraphs relating to the contents of strings.

5.3. Line endings and encoding translation in raw string literals

Phase 1 not only translates trigraphs and universal-character-names, it encompasses source encoding translation, including line endings. The language in §2.5/3 "any transformations performed in phases 1 and 2 (trigraphs, universal-character-names, and line splicing) are reverted" is ambiguous as to whether that list is exhaustive, but it seems not to be.

Reverting *all* phase 1 transformations would effectively copy a section of the source file to the literal in binary mode. Raw strings are intended as a means of quoting text, not binary data. (Some, including myself, have gotten the impression that some degree of binary support exists.)

The list "trigraphs, universal-character-names, and line splicing" should be normative exclusively, and "any transformations in phases 1 and 2" should be struck.

5.4. Macro invocation spanning end of file

DR 1718. The potential span of a macro invocation is specified differently in [cpp.replace] §16.3/11 and [cpp.rescan] §16.3.4/1. The former says

> The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro.

The latter says

> Then the resulting preprocessing token sequence is rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.

This suggests that the initial scan is performed on the entire token sequence considering all files, but rescanning after one macro has been replaced is limited to one file.

So this TU is valid

```
// header.h
#define a()

a(

// source.c
#include "header.h"
) /* Initial rescanning begins here. */
```

But this change to the header would invalidate it.

```
#define a()
a() /* Initial rescanning begins here. */
a(
```

(Note that the `#include` directive has already been deleted before `a()` is defined, so the invocation cannot be considered to span the directive.)

The surveyed implementations limit a macro invocation to one source file. Any distinction between "initial scanning" and "rescanning" seems unintentional. Moreover [cpp.replace] should completely specify the potential span of an invocation.

6. Issues of ambiguous specification

Like section 3, the following cases reflect omissions or allowed implementation variance, but they differ in that the user is exposed to little risk.

6.1. Files ending in backslash

DR 1698. C++03 specified undefined behavior if a file did not terminate in a newline, which is necessary to complete a preprocessor directive (such as a header guard `#endif`). C++11 mostly fixed this by specifying that a terminating newline shall be added by phase 2 if not already present. However it does not specify whether phase 2 may form a new line splice upon adding this newline.

Clang, GCC, and EDG all diagnose a TU comprising a single backslash, so this behavior should be standardized. MSVC does not diagnose such a condition, and furthermore ICEs if a TU ends in a backslash-terminated `#if` directive.

6.2. Correct space insertion when stringizing

DR 1625. Some implementations introduce space between stringized tokens for textual clarity. The specification requires certain space characters, but does not forbid any except at the beginning and end. As the preprocessor may be used with the intention of generating strings of fixed, platform-independent length, this implementation-specific behavior should be explicitly mentioned by the standard.

This solution blesses the extreme case of introducing space between every pair of tokens. A user wishing to ensure that space does not appear between tokens may stringize the tokens separately and use phase 6 string literal concatenation. In any case, no normative change is proposed.

6.3. Meaning of "nested replacement"

(See DR 268 for an example test case. This issue has also been widely noted in C.)

In macro replacement, a sequence of tokens composing an invocation is replaced by a sequence of tokens as prescribed by [cpp.replace] §16.3. Invocations resulting from the replacement process are further replaced. An invocation token sequence may be formed from the replacement of two adjacent invocations, or the replacement of an invocation and the succeeding tokens. Thus invocations do not form a proper hierarchy, but a DAG. Thus confusion arises when the term "nested replacement" is used without further clarification in §16.3.4/2. To summarize the established arguments:

1. Considering the DAG, a replacement is nested only within those replacements contained by every route between itself and the root node.
2. Each replacement is owned by the first replaced token, which names the invoked macro. By ignoring the source(s) of the argument list, a hierarchy is obtained rather than a DAG.

3. This is a matter of implementation latitude. Real-world programs do not form a DAG that is not a tree anyway. The dilemma need not be resolved by the standard.

The surveyed C++ implementations all follow the behavior described by #1. This pattern is a natural result of a recursive replacement procedure, with the macros currently being replaced being represented as a FIFO stack mirroring the call stack. Each invocation is discovered during the rescanning of the innermost, fully-enclosing replacement sequence. "Nesting" perfectly describes the relationship of the function calls internal to the implementation.

Implementations of #2 exist, but apparently not among up-to-date C++ products. Materials published by CPPGM.com (C++ Grandmaster), a pseudo-academic organization, prescribe this behavior to students wishing to implement preprocessors. To form the hierarchy, a preprocessor must record the source replacement of each generated token in a persistent data structure.

Preprocessor metaprogramming has become viable as standardization has provided consistent results across different platforms. Recursion is a common wish for ambitious metaprogrammers, and #1 implies a weaker restriction on recursion. Moreover it is a topic of some practical interest.

Because the C++11 ecosystem is less fragmented than C, or previous generations of C++, and users have become more sophisticated over decades of practice, #3 does not apply as it once did. Now is a good opportunity to guarantee that future C++ compilers will continue to work as current ones do. As the change is purely a refinement, this does not harm cross-compatibility with C. Commitment to #1 does dictate that C may not bless #2 exclusively, but they could not do so anyway because of the preponderance of implementations of #1.

This proposal therefore recommends that the suggested resolution in DR 268 be adopted by C++, even if it is not by C. A small clarification to that text has been made in this proposal, because “once again available for replacement” negates “an occurrence of the macro's name will not result in further replacement” but not “even if it is later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.”

6.4. Signedness of character literals in controlling expressions

DR 925. C character literals always have type `signed int`, whereas in C++ they may potentially be unsigned. The difference may be observed by the preprocessor.

```
#if '0'-'1'>0
#   error Character literals are unsigned.
#endif
```

A simple solution is to require that `char` be signed during preprocessing, and hence have range identical to `std::intmax_t`. The solution does not seem to be appropriate because of common nonconformance. GCC and Clang incorrectly allow the signedness of `char` to be observed when compiling C, as specified by C++, and EDG does not allow the signedness to be observed, thus following the C specification even when compiling C++.

If both languages want to bless these popular existing implementations, a solution would be to allow the signedness of character literals to vary as their encoding already does. It is probably better to coordinate this with C than to change unilaterally, and no change is proposed here.

The problem probably never manifests seriously, and workarounds exist such as C++11/C11 Unicode literals, or simply adding or comparing the expression of unknown signedness to `0U`. (Following through with such fixes may require tricky unsigned modulo arithmetic, though.)

6.5. Value of the `__LINE__` macro

The terms "presumed line number" and "current token" are used without definition. It is uncertain how `__LINE__` used in an argument (or its expansion) should expand in a multi-line macro invocation. Assuming the current token is the location up to which phase 4 has parsed, during such an expansion it would be the outermost closing paren, and this is what GCC, EDG, and MSVC do. However, higher-quality output is obtained if `__LINE__` reflects the location of the token it most directly replaces in the macro invocation. This is what Clang does.

```
#include <cstdio>
#define id( ... ) __VA_ARGS__

int main() {
    std::printf( "%d %d", id(
        __LINE__, // Clang outputs 6, others 8.
        __LINE__ // Clang outputs 7, others 8.
    ));
}
```

(Furthermore, one token may be divided among several lines by line splicing.)

As a resolution, the term "line number" should be refined to unambiguously apply to all lines, "presumed line number" should be defined and used appropriately, and "current token" should be changed to some implementation-specific concept.

6.6. Consecutive paste (##) tokens

There is no rule against two consecutive paste tokens in the replacement list of a macro. It is ambiguous whether they are both operators, or if one is an operator and the other is an operand, i.e., that something will be concatenated with a ## token. The former interpretation acts the same as a single ## operator, and the latter interpretation usually, but does not always, result in a diagnosis. Such a macro should be treated as ill-formed upon definition. This issue produced crashes on GCC and Clang before special replacement-time detection was applied.

```
#define cat2hh(x, y) x ## ## y
cat2hh(,) // expands to either empty list or ##
cat2hh(a,b) // either expands to “ab” or diagnosed
```

7. One common feature requiring standardization

Most C++11 implementations are already noncompliant due to their implementation of this extension, and this will be exacerbated by required diagnosis of failed token pastes.

7.1. Comma preceding empty variable arguments

Since long before variadic macros were standardized, GCC has provided an extension to erase the comma before an empty list. Clang, MSVC, and EDG have followed suit. The latest evolution of the GCC extension requires the specific syntax `, ## __VA_ARGS__` and quietly accepts ill-formed programs. According to the standard, the variable argument matched by `...` is required to exist, and to be separated from its predecessor by a comma, even if it is empty. The extension is only applied if that final comma is missing. Otherwise, an empty variable argument causes the comma to be pasted to a placemarker token, or a non-empty variable argument results in a paste producing two tokens, according to undefined behavior.

Considering its widespread use and implementation, the extension is ready for standardization. MSVC and EDG currently clone an earlier variant of the extension which misinterprets a complying program, if an empty variable argument is preceded by a comma. Blessing the lowest-impact extension would require other implementations to behave exactly the same, whereas the status quo is noncompliance with a little variance. This status quo must be considered untenable.

A slight improvement is proposed to the latest edition of the extension: the ## operator does not suppress expansion of the `__VA_ARGS__` argument in the argument list context. This causes problems for macros for which the user may attempt to nest invocations, such as putting one `BOOST_SCOPE_EXIT` block inside another. Patches have been submitted to GCC and Clang.

8. Standardese

Changes are specified relative to N3797. Each of the following subsections provides verbiage for one of the previous sections. Where changes overlap, changes are accumulated over the sequence of subsections, as noted. The intent is to allow selective adoption of the higher-priority changes proposed in lower-numbered sections.

8.1. Changes to fix undefined behavior specifications

Modify [lex.pptoken] §2.5/2:

> If a `'` or a `"` character matches the last category, ~~the behavior is undefined~~ the program is ill-formed.

Replace [lex.name] §2.11/3:

> ~~In addition, some identifiers are reserved for use by C++ implementations and standard libraries (17.6.4.3.2) and shall not be used otherwise; no diagnostic is required.~~
>
> Unless otherwise specified, if an identifier begins with an underscore followed by a capital letter, or if it contains two consecutive underscores (__), the behavior is undefined. Such preprocessing tokens are reserved to the implementation for any use.

Modify [cpp.cond] §16.1/4:

> Prior to evaluation of subexpressions and the `defined` operator, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced ~~(except for those macro names modified by the defined unary operator)~~, just as in normal text, except that tokens forming `defined` operator expressions shall not be replaced. If the token `defined` is generated as a result of this replacement process, its evaluation is conditionally-supported. If the token `defined` or its operand is used as an operand to the # or ## operator, the program is ill-formed, no diagnostic required. If use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, ~~the behavior is undefined~~ the program is ill-formed.

Modify [cpp.include] §16.2/4:

> If the directive resulting after all replacements does not match one of the two previous forms, ~~the behavior is undefined~~ the program is ill-formed.

Modify [cpp.replace] §16.3/7:

> The identifier immediately following the `define` or `undef` is called the *macro name*. It shall not be the identifier `defined` [cpp.cond]. There is one name space…

Strike from [cpp.replace] §16.3/11:

> ~~If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the behavior is undefined.~~

Insert a new paragraph in [cpp.replace] §16.3:

> A sequence of preprocessing tokens within the list of arguments (after `(` and before the matching `)`) may form a preprocessing directive; its execution is conditionally-supported. If supported, directive execution and formation of the argument list shall occur

simultaneously, but a directive between an identifier token and a (token shall prevent identification of a macro invocation. If an identifier forming part of the argument list (not an embedded preprocessing directive) refers to a macro definition which is not in scope at its terminating) [cpp.scope], the program is ill-formed, no diagnosis required.

Modify [cpp.stringize] §16.3.2/2:

> If the replacement that results is not a valid character string literal, ~~the behavior is undefined~~ the program is ill-formed.

Modify [cpp.concat] §16.3.3/3. (This is superseded by section 8.2 below):

> If the replacement that results is not a valid preprocessing token, ~~the behavior is undefined~~ the program is ill-formed. The resulting token is available for further macro replacement. ~~The order of evaluation of ## operators is unspecified.~~ The ## operator groups left-to-right.

Modify [cpp.line] §16.4/2:

> The *line number* of ~~the current~~ a source line is ~~one~~ greater than the preceding presumed line number by the number of new-line characters read or introduced in translation phase 1 [lex.phases] while processing the source file ~~to the current token~~ between those two lines. The presumed line number of the initial line of a file is one. Every line number shall be between one and an implementation-defined limit.

Modify [cpp.line] §16.4/3:

> 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~~ the *presumed line number* of the subsequent source line to be 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.~~

Modify [cpp.line] §16.4/5:

> If the directive resulting after all replacements does not match one of the two previous forms, ~~the behavior is undefined~~ the program is ill-formed; otherwise, the result is processed as appropriate.

Strike from [cpp.predefined] §16.8/1. (This is superseded by section 8.4 below):

> `__LINE__`

The ~~presumed~~ line number (within the current source file) of the current source line (an integer literal).

Remove [cpp.predefined] §16.8/3:

~~The values of the predefined macros (except for `__FILE__` and `__LINE__`) remain constant throughout the translation unit.~~

Strike from [cpp.predefined] §16.8/4:

~~If any of the pre-defined macro names in this subclause, or the identifier `defined`, is the subject of a `#define` or a `#undef` preprocessing directive, the behavior is undefined.~~ Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

Strike from [global.names] §17.6.4.3.2/1:

~~Certain sets of names and function signatures are always reserved to the implementation:~~
~~— Each name that contains a double underscore `__` or begins with an underscore followed by an uppercase letter [lex.key] is reserved to the implementation for any use.~~

~~—~~Each name that begins with an underscore is reserved to the implementation for use as a name in the global namespace.

Add to [implimits] Annex B:

Line number of a source line [2 147 483 647].

8.2. Changes to add missing specifications

Modify [lex.pptoken] §2.5/3:

… this reversion shall apply before any *d-char*, *r-char*, or delimiting parenthesis is identified, but not while forming tokens as part of an operation invoked by translation phase 4.

Modify [cpp.include] §16.2/3:

… causes the replacement of that directive by a *preprocessing-file* comprising the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file …

Modify [cpp.concat] §16.3.3/3. (This supersedes section 8.1 above):

If the replacement that results is not a valid preprocessing token, ~~the behavior is undefined~~ the program is ill-formed. Prepending an `R` to a string literal to form a raw string literal is conditionally-supported. The resulting token is available for further macro replacement. ~~The order of evaluation of ## operators is unspecified.~~ The ## operator groups left-to-right.

Modify [cpp.pragma.op] §16.9:

A unary operator expression of the form

`_Pragma` (*string-literal*)

is processed as follows. ~~The string literal is destringized by deleting the L prefix~~ The encoding-prefix and `R` if present, and ~~deleting~~ the leading and trailing double-quotes are deleted from the string literal. If it is not a raw string literal, replace ~~replacing~~ each escape sequence `\"` by a double-quote, and replace ~~replacing~~ each escape sequence `\\` by a single backslash. The resulting sequence of characters is processed through translation phases 1, 2, and 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. White-space separating these tokens shall consist of space and horizontal tab characters, no diagnosis required. The original four preprocessing tokens in the unary operator expression are removed.

Modify [cpp.stringize] §16.3.2/2:

… (including the delimiting " characters). Furthermore, in a raw string literal each newline character is replaced with the escape sequence `\n`, and each extended character is replaced by an equivalent universal-character-name. If the replacement that results is not a valid character string literal, …

Modify [cpp.cond] §16.1/2:

Each preprocessing token that remains (in the list of preprocessing tokens that will become the controlling expression) after all macro replacements have occurred shall be in the lexical form of a token (2.7 [lex.token]). At the time of evaluation, the controlling expression shall not contain terminal symbols from syntactic categories ([syntax]) other than *integer-literal*, *character-literal*, *boolean-literal*, *multiplicative-expression*, *additive-expression*, *shift-expression*, *relational-expression*, *equality-expression*, *and-expression*, *exclusive-or-expression*, *inclusive-or-expression*, *logical-and-expression*, *logical-or-expression*, or *conditional-expression*.

8.3. Changes to resolve inconsistent specifications

Modify [cpp.stringize] §16.3.2/2:

~~The order of evaluation of # and ## operators is unspecified.~~ The # operator groups with higher precedence than the ## operator.

Strike from [lex.pptoken] §2.5/3:

— ~~If the next character begins a sequence of characters that could be the prefix and initial double quote of a raw string literal, such as `R"`, the next preprocessing token shall be a raw string literal. Between the initial and final double quote characters of the raw string, any transformations performed in phases 1 and 2 (trigraphs, universal-character-names, and line splicing) are reverted; this reversion shall apply before any *d-char*, *r-char*, or delimiting parenthesis is identified. The raw string literal is defined as the shortest sequence of characters that matches the raw-string pattern~~

~~*encoding-prefix*$_{opt}$ `R` *raw-string*~~

— ~~Otherwise, i~~If the next three characters …

Modify [lex.string] §2.14.5/2:

> A *d-char-sequence* shall consist of at most 16 characters. Between the initial and final double quote characters of the raw string, transformations performed in phases 1 and 2 forming trigraphs, universal-character-names, and line splices are reverted; this reversion shall apply before any *d-char*, *r-char*, or delimiting parenthesis is identified, but not while forming tokens as part of an operation invoked by translation phase 4.

Modify [cpp.replace] §16.3/4:

> There shall exist a `)` preprocessing token in the same source file that terminates the invocation.

Modify [cpp.replace] §16.3/4:

> A replacement list shall not have a `##` preprocessing token ~~shall not occur at the beginning or at the end of a replacement list~~ as its first or last token, and it shall not contain two such tokens consecutively, for either form of macro definition.

8.4. Changes to resolve ambiguous specifications

Modify [lex.phases] §2.2/1:

> 2. A source file that is not empty and that does not end in a new-line character, or that ends in a new-line character immediately preceded by a backslash character before any such splicing takes place, shall be processed as if an additional new-line character were appended to the file. This additional new-line shall not form part of a splice.

Modify [cpp.stringize] §16.3.2/2:

Each occurrence of white space between the argument’s preprocessing tokens becomes a single space character in the character string literal; other added whitespace is implementation-specific.

Replace [cpp.stringize] §16.3.2/2:

~~If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Further, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.~~

As long as the scan involves only preprocessing tokens from a given macro's replacement list, or tokens resulting from a replacement of those tokens, an occurrence of the macro's name will not result in further replacement, even if it is later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

Once the scan reaches the preprocessing token following a macro's replacement list — including as part of the argument list for that of another macro — occurrences of the macro's name are no longer marked as nonreplaceable upon insertion by replacement. [*Example:*

```
#define NIL(xxx) xxx
#define G_0(arg) NIL(G_1)(arg)
#define G_1(arg) NIL(arg)
G_0(42)                        // result is 42, not NIL(42)
```

The reason that `NIL(42)` is replaced is that `(42)` comes from outside the replacement list of `NIL(G_1)`, hence the occurrence of `NIL` within the replacement list for `NIL(G_1)` (via the replacement of `G_1(42)`) is not marked as nonreplaceable. *—end example*]

Modify [cpp.predefined] §16.8/1. (This supersedes section 8.1 above):

`__LINE__`

The ~~presumed~~ line number (within the current source file) of ~~the current~~ a source line containing some token currently undergoing macro replacement, but not from a replacement list (an integer literal).

8.5. Changes to add features

Modify [cpp.replace] §16.3/4:

Otherwise, there shall be ~~more~~ at least as many arguments in the invocation ~~than~~ as there are parameters in the macro definition (excluding the `...`).

Modify [cpp.replace] §16.3/12:

If there is a `...` immediately preceding the `)` in the function-like macro definition, then ~~the~~ any trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the *variable arguments*. The number of arguments so combined is such that, following merger, the number of arguments is not greater than one more than the number of parameters in the macro definition (excluding the `...`).

Insert a new paragraph in [cpp.replace] §16.3:

The concatenation operation denoted by the token sequence `, ## __VA_ARGS__` in the replacement list, not preceded by another ## token, shall be processed specially. It shall be replaced by a placemarker preprocessing token if the variable arguments and preceding comma are omitted. Otherwise, it is replaced as if the ## token had not existed in the replacement list. No concatenation occurs, and macros within the variable arguments may be expanded before their substitution.

Modify [cpp.replace] §16.3.1/2:

…replace it. If no argument existed to form the variable arguments, it shall be treated as an empty argument.

9. FAQ

Why leave so much implementation dependence, such as in generating the defined operator, and executing directives inside argument lists? The language should be specified in definite terms.

This proposal is designed to be easily adopted, without adding undue requirements for implementers. Conditional support may easily be strengthened to guaranteed support later. The sooner we start on this evolutionary path, the sooner convergence can happen. However, requiring convergence immediately may not be productive for platforms with user-bases that truly lack demand for obscure features.

For the particular cases mentioned, the proposed conditional support models might describe supersets of the functionality covered by the truly best conceptual models. Reconciling all the sufficiently-tolerant implementations and all the actual usage cases will require more work, and the normative adjustments will be non-trivial.

Why not remove all undefined behavior? In effect this proposal *adds* UB for extension keywords. Even feeding too many lines into the compiler may still produce nasal demons.

There is a time and place for UB. This proposal specifies it where it already exists in practice. If stamping out such instances is a good idea, we can have that debate and institute further cleanup. That process will be easier when reality is reflected by the published standard, or even just in the working draft. See previous Q&A.

Why support weird behaviors like generating the defined operator and directives inside argument lists? These do not belong in modern code.

Removing support for documented features is a non-starter. Platforms vendors support their users by allowing them to build source code into finished products, not by dictating how code should no longer be written. Also, preprocessor metaprogramming and build-mastering can be difficult fields, and the solution space should not be reduced without reasons that go beyond aesthetics.

Why not start with a clean slate? The original specification from 1989 was purposely loose. A uniform formal model would be superior to an evolutionary improvement.

The number of known defects is high, but not that high. This proposal does not preclude taking such a direction later, but it would be more effort. Some things remain intentionally non-portable, such as whitespace generated by the stringize operator and the value of the `__LINE__` macro, because the optimal or "most correct" or "best QOI" behavior would be a matter of debate. That debate can happen later; right now is an opportunity for immediate progress.

10. Acknowledgements

Richard Smith provided helpful feedback, particularly regarding reserved macros names.

Arthur O'Dwyer presented a good case for allowing nested macros to use `,##__VA_ARGS__`. Also he provided valuable perspective with a detailed review.

11. Revision history

N3882 — Initial revision.

N4220 — Disallow consecutive ## tokens in macro replacement list.
Pre-expand argument to `, ## __VA_ARGS__` overriding suppression by paste.
Require support for `, ## __VA_ARGS__`.
Do not litigate reserved macro names.
Forbid `defined` as a macro name separately from implementation reservations.
Specify that a header file is a preprocessing-file.
Improve conditional constant expression requirements.
Add TOC, FAQ, and acknowledgements.