Preprocessing is never undefined

Replacing compile-time UB with IFNDR, and then diagnosing

Document #: P2843R1
Date: 2025-01-13
Project: Programming Language C++
Audience: Evolution Working Group
SG22 C Liaison

Reply-to: Alisdair Meredith
<ameredith1@bloomberg.net>

Contents

1 Abstract

This paper revises all specification of the C++ preprocessor and lexer that uses the term *undefined behavior* to use the more appropriate term — *ill-formed, no diagnostic required*. It then further analyzes each such ill-formed program to determine whether diagnostics should be required, or the program should become well-formed.

2 Revision history

R1: 2025 January (pre-Hagenberg mailing)

This revision has been a long time coming, due to work on other papers claiming priority. It now has a long list of updates, as preliminary revisions that were aimed at a number of previous meetings never reached publication.

- Updated introduction with rationale about UB during compilation
- Recorded the open Core issues in this area
- Added C liaison to the target audience
- Rebased after [P2621R2] was adopted at the Varna meeting
  - Dropped all discussion related to the paper that landed
  - Removed subsection on conflict resolution
  - Proposed wording rebased onto [N5001]
- Updated compiler conformance on the `defined` tests
  - note that all compilers now conform with the recommendations of this paper
- Completed initial analysis strengthening diagnostics on the remaining cases of UB to IFNDR
- Proposed bringing library prohibition on redefining keywords into the preprocessor specification
- Proposed completing the list of banned macro names with identifiers used in the preprocessor specification
- Proposed diagnosing ill-formed comments
- Added a section to wording on updating the open Core issues
- Added a summary by compiler for implementation changes if this paper is accepted

R0: 2023 May (pre-Varna mailing)

- Initial draft of this paper.

3 Introduction

Undefined behavior is a form of C++ specification that applies to the runtime behavior of a well-formed program and its inputs. Logically, there is no potential for undefined behavior within the preprocessor, which simply transforms source code before translation — although subsequent phases of translation might introduce undefined behavior when processing the source code.

More literally, undefined behavior is recognised as one of the worst things that can happen to our program, where we would much rather run into behavior that is implementation defined, unspecified, or even erroneous — all of which indicate a more contained set of behaviors for the program. However, for undefined behavior to occur, the program logic must take it down a path that ultimately leads to undefined behavior.

It is recognised that programs that are ill-formed, no diagnostic required are in an even worse state than programs that contain undefined behavior as, if an executable is created for that program, then the whole program has undefined behavior; simply running the program immediately invokes undefined behavior and the program can do literally *anything*.

Yet this paper proposes replacing undefined behavior with the notionally worse ill-formed, no diagnostic required program. This is still a good idea, as in all the cases where we propose such a change, the undefined behavior occurs during the act of translating the program, i.e., it is the act of compiling the program itself that has full freedom to trash your hard drive, emit nasal daemons, etc.

With the changes in this paper, it will formally be as safe to compile our programs as we always thought it was. Further, the ill-formed, no diagnostic requires phrasing naturally leads to the question “why is that ill-formed program not diagnosed?”, prompting a useful tightening of our specification.

4 Basic Proposal

A better formulation for all cases where the C++ Standard specifies undefined behavior in the preprocessor would be that the program is ill-formed, no diagnostic required. Such a change would have no effect on any implementations today, other than to remove an unexploited ability for the act of compiling code to have unexpected consequences, and serves as the basis for the follow up work below, making most of those cases either diagnosable, or well-defined.

4.1 Minimal wording

Make the following changes to the C++ Working Draft. All wording is relative to [N4986].

15.2 [cpp.cond] Conditional inclusion

10 Prior to evaluation, 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. If the token `defined` is generated as a result of this replacement process or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the ~~behavior is undefined~~ program is ill-formed, no diagnostic required.

15.3 [cpp.include] Source file inclusion

4 A preprocessing directive of the form

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

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text (i.e., 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~~ program is ill-formed, no diagnostic required. The method by which a sequence of preprocessing tokens between a `<` and a `>` preprocessing token pair or a pair of `"` characters is combined into a single header name preprocessing token is implementation-defined.

15.6.1 [cpp.replace.general] General

13 The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the ~~behavior is undefined~~ program is ill-formed, no diagnostic required.

15.6.3 [cpp.stringize] The # operator

2 A *character string literal* is a *string-literal* with no prefix. If, in the replacement list, a parameter is immediately preceded by a `#` preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument (excluding placemarker tokens). Let the *stringizing argument* be the preprocessing token sequence for the corresponding argument with placemarker tokens removed. Each occurrence of whitespace between the stringizing argument's preprocessing tokens becomes a single space character in the character string literal. Whitespace before the first preprocessing token and after the last preprocessing token comprising the stringizing argument is deleted. Otherwise, the original spelling of each preprocessing token in the stringizing argument is retained in the character string literal, except for special handling for producing the spelling of *string-literal*s and *character-literal*s: a `\` character is inserted before each `"` and `\` character of a *character-literal* or *string-literal* (including the delimiting `"` characters). If the replacement that results is not a valid character string literal, the ~~behavior is undefined~~ program is ill-formed, no diagnostic required. The character string literal corresponding to an empty stringizing argument is `""`. The order of evaluation of `#` and `##` operators is unspecified.

15.6.4 [cpp.concat] The ## operator

3 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token.

[*Note 1:* Concatenation can form a *universal-character-name*. *—end note*]

If the result is not a valid preprocessing token, the ~~behavior is undefined~~ program is ill-formed, no diagnostic required. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.

15.7 [cpp.line] Line control

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 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~~ program is ill-formed, no diagnostic required.

5 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~~ program is ill-formed, no diagnostic required; otherwise, the result is processed as appropriate.

15.11 [cpp.predefined] Predefined macro names

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~~ program is ill-formed, no diagnostic required. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

5 Reviewing each ill-formed constructions

For the benefit of the proposed wording in the following examples, we are assuming the UB to IFNDR change at the start of this paper has been applied, and so the context of this analysis is diagnosing or accepting as well-defined behavior that is ill-formed, no diagnostic required.

Note that paper [P2621R2] has already performed the same task for 5 [lex] so most of the remaining concerns are limited to subclauses of 15 [cpp]

5.1 Towards a better specification

When informally proposed as a paper for the Core Working Group, the chair expressed reservations about a paper that did nothing but change UB to IFNDR, and suggested that a paper that also tackled the "no diagnostic required" aspect would be most welcome. Hence, the rest of this paper will examine further changes to require diagnosing and rejected many invalid programs, or in some cases, accepting some undefined programs as well-defined.

There is a long history of trying to address the undefined nature of the preprocessor. The following review aims to resolve all of the issues highlighted by previous work.

5.1.1 Past proposals

— [N3801] Removing Undefined Behavior from the Preprocessor, Gabriel Dos Reis
— [N4220] An update to the preprocessor specification, David Krauss
— [N4858] Disposition of Comments for CD Ballot, ISO/IEC CD 14882, Barry Hedquist
— [P1705R1] Enumerating Core Undefined Behavior, Shafik Yaghmour
— [P2234R1] Consider a UB and IF-NDR Audit Scott Schurr

5.1.2 Open core issues

— [CWG2575] Undefined behavior when macro-replacing `defined` operator
— [CWG2576] Undefined behavior with macro-expanded `#include` directives
— [CWG2577] Undefined behavior for preprocessing directives in macro arguments
— [CWG2578] Undefined behavior when creating an invalid string literal via stringizing
— [CWG2579] Undefined behavior when token pasting does not create a preprocessing token
— [CWG2580] Undefined behavior with `#line`
— [CWG2581] Undefined behavior for predefined macros

5.2 Non-standard use of the `defined` operator

This is Core issue [CWG2575].

5.2.1 Using a macro defined as `defined`

Function-like operator	Keyword-like operator

`# define MACRO defined`
`# if MACRO(MACRO)`
`#   error accepted the macro definition`
`# endif`
	`# define MACRO defined`
	`# if MACRO MACRO`
	`#   error accepted the macro definition`
	`# endif`

All four popular compiler front ends accept this undefined code and “correctly” apply the `defined` operator that expands from the first use of `MACRO()`, and only Clang issues a warning that the behavior is undefined (since Clang 3.9).

clang	default
gcc	`Wextra`
EDG	(none)
MSVC	(none)

This paper recommends defining the behavior to be exactly what all of our current front ends do, and simply expand the macro for another iteration of phase 4. It is believed that this can be achieved simply by removing the sentence conferring undefined behavior, as the intended defined behavior should then proceed without calling further attention to it.

5.2.2 Ill-formed use of the `defined` operator

The next issue is the wording

> or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement

In the case of

Well defined	Undefined

`# if defined A + B`
`# endif`
	`# if defined +B  // + cannot start an identifier`
	`# endif`
	
	`# if defined 1  // 1 is not an identifier`
	`# endif`
	
	`# if defined    // no argument to the operator`
	`# endif`

the well-defined path sees tokenization where `A` is passed to the `defined` operator, and then combined into the *constant-expression* `(defined A) + B`.

However, the “undefined” path is more interesting. All Godbolt compilers agree that the `1` case is an error. Both gcc and Clang diagnose the other two cases that the known set of non-parenthetical cases are ill-formed.

	`+B`	`1`	“
clang	Error	Error	Error
gcc	Error	Error	Error
EDG	Error	Error	Error
MSVC	Error	Error	Error
—	—	—	—
C++23	UB	UB	UB
P2843	Error	Error	Error

A further example demonstrates how current implementation interpret the grammar around parentheses.

Well defined	**Undefined**

`# if defined A + B`
`# endif`
	`# if defined (A + B)`
	`# endif`
	
	`# if defined (+B)`
	`# endif`
	
	`# if defined (1)`
	`# endif`
	
	`# if defined ()`
	`# endif`

	`(A+B)`	`(+B)`	`(1)`	`()`
clang	Error	Error	Error	Error
gcc	Error	Error	Error	Error
EDG	Error	Error	Error	Error
MSVC	Error	Error	Error	Error
—	—	—	—	—
C++23	UB	UB	UB	UB
P2843	Error	Error	Error	Error

In this case, all compilers agree (according to their error messages) that a closing parenthesis is expected instead of the `+` for the first case, and all the other cases report an error expecting an *identifier*. None of the compilers take the path of undefined behavior that would consider all these cases as failing to lex as the operator, which would treat `defined` as a user-supplied funcion-like macro. There would be no impact on existing implementations to enforce this usage as a diagnosable error.

According to the current C++ Standard, C++23, the contents of the parentheses fail the grammar for an identifier, so `defined` is interpreted as a user-provided function-like macro, which is undefined behavior, or as violating both valid forms of the `defined` operator, which is also undefined behavior. It would be reasonable to argue all of these cases should be an error as there is no user-defined macro named `defined` in scope, but that is not the error any of these implementations are reporting, and as it would be UB to define such a macro, it is not clear that would be a helpful error message.

Given that all implementations already report an error in case, we have no qualms recommedning this case always be a diagnosable error.

Our final observation will be that, perhaps surprising some of us, the built-in comma operator cannot be used in a *constant-expression*.

Comma operator	Plus operator

`# if defined A, B`
`# endif`
 	
`# if defined (A), B`
`# endif`
 	
`# if (defined A), B`
`# endif`
 	
`# if defined (A, B)`
`# endif`
	`# if defined A + B`
	`# endif`
	
	`# if defined (A) + B`
	`# endif`
	
	`# if (defined A) + B`
	`# endif`
	
	`# if defined (A + B)`
	`# endif`

	`defined A, B`	`defined (A), B`	`(defined A), B`	`defined (A, B)`
clang	Error	Error	Error	Error
gcc	Pedantic	Pedantic	Pedantic	Error
EDG	Error	Error	Error	Error
MSVC	Error	Error	Error	Error
——	——————	——————-	——————-	——————-
C++23	Error	Error	Error	UB
P2843	Error	Error	Error	Error

In this case, all uses of the comma operator are invalid *constant-expressions* or invalid uses of the `defined` operator, although I believe that lexing should turn them all into invalid comma operators and not invoke the UB of a token sequence supplied to `deprecated`. Conversely, the equivalent cases for operator plus are all well-formed and clearly understood. Note that while most implementation already reject these programs, gcc does accepts all but the last form, and does not diagnose a concern unless requesting pedantic warnings.

There is one remaining test case that still exposes bugs in modern preprocessors.

```
# if (defined A, B)
# endif
```

The unusual issue here is that the comma operator should not be eligible to be a `constant-expression`, comprising two _assignment-expression_s. However, once we wrap that comma expression in parentheses, it becomes a regular *expression* and *is* eligible to be a *constant-expression*. As the `defined A` tokens should lexically consume the `defined` operator, I believe that there is no room for undefined behavior in this final test.

```
# if 0, 1
# endif

# if (0, 1)
# endif

# if defined A, B
# endif

# if (defined A, B)
# endif
```

	`0, 1`	`(0, 1)`	`defined A, B`	`(defined A, B)`
clang	Error	Ped-warn	Error	Ped-warn
gcc	Ped-warn	Ped-warn	Ped-warn	Ped-warn
EDG	Error	Error	Error	Error
MSVC	Error	Error	Warning	Error
C++23	Error	OK	Error	OK
P2843	Error	OK	Error	OK

This test, which entirely lacks undefined behavior, possibly shows the greatest variation among implementations — a reminder of how rarely we tread the dusty pages of the preprocessor specification.

I will ultimately file bug reports with the appropriate vendors, but am waiting on a resolution to this paper before churning the various compiler bug reporting systems.

5.2.3 Recommendation

The recommendation is that since all main compilers already support macro expansion into a `defined` operator, that behavior should become well-defined. Conversely, misuses of the `defined` operator syntax are diagnosed, so that should become normatively ill-formed.

Note that this paper is not recommending any changes where some compilers already disagree with the current standard — if all compilers were to disagree we might recommend changes to support more behavior.

5.2.4 Proposed wording

15.2 [cpp.cond] Conditional inclusion

10 Prior to evaluation, 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. If ~~the token `defined` is generated as a result of this replacement process or~~ use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the program is ill-formed ~~, no diagnostic required~~.

5.2.5 Resolve issue

Resolve [CWG2575] regarding undefined behavior when macro-replacing the `defined` operator.

5.2.6 Compatibility

The proposed changes to not make any currently valid programs ill-formed nor change their meaning. Hence, we suggest no Annex C wording is necessary, although there may be an impact on some implementations rejecting programs with specific undefined behaviors in C++23 as invalid in C++26.

5.3 Preprocessing directives in macro arguments

This is Core issue [CWG2577], regarding what happens if macro arguments are themselves subject to conditional compilation or other preprocessor directives. Consider the following example:

IFNDR

```
#define DECLARE_CONSTRUCTOR( CLASS               \
                           , TYPE                \
                           , PARAM)              \
  CLASS (TYPE PARAM);

struct Any {


   template <class T>
   DECLARE_CONSTRUCTOR( Any
#if defined __cpp_rvalue_references
                      , T &&
#else
                      , T const &
#endif
                      , arg_name
                      );




};
```

Well-formed

```
#define DECLARE_CONSTRUCTOR( CLASS               \
                           , TYPE                \
                           , PARAM)              \
  CLASS (TYPE PARAM);

struct Any {

#if defined __cpp_rvalue_references
   template <class T>
   DECLARE_CONSTRUCTOR( Any
                      , T &&
                      , arg_name
                      );
#else
   template <class T>
   DECLARE_CONSTRUCTOR( Any
                      , T const &
                      , arg_name
                      );
#endif

};
```

This example shows the ill-formed, no diagnostic required behavior, and the corresponding well-formed workaround. Currently, Microsoft rejects the IFNDR version, while Clang, EDG, and GCC accept that code without a diagnostic, not even a warning.

It should not be difficult to diagnose the ill-formedness if we were to mandate a diagnostic, and the existing compilers would be free to continue accepting such code as a conforming extension, as long as they issue a diagnostic. However, me may prefer the alternative resolution of accepting this code as well-formed, with the “obvious” meaning.

What about other preprocessing directives? The next example investigates `#line`.

```
#define DECLARE_CONSTRUCTOR( CLASS, TYPE, PARAM)  \
        CLASS (TYPE PARAM);

struct Any {

   template <class T>
   DECLARE_CONSTRUCTOR( Any
                      , T &&
                      , arg_name
#line 12345 "fooled_you.h"
                      );
};
```

Again, Clang, EDG, and GCC accept this ill-formed, no diagnostic required program, and even agree on how the line number changes. However, Microsoft rejects this code with a diagnostic. Is this another behavior we would prefer to make well-defined, rather than diagnosibly ill-formed? The positive use case is less clear.

How about the `#include` directive?

```
#define DECLARE_CONSTRUCTOR( CLASS, TYPE, PARAM)  \
        CLASS (TYPE PARAM);

struct Any {

   template <class T>
   DECLARE_CONSTRUCTOR(
#include "arguments.h"
                      );
};
```

Assuming the file `arguments.h` contains a well-formed list of three arguments, this program is still rejected and diagnosed as ill-formed on all four popular front ends. Should we make sure this form is always diagnosed as an error?

What about if we change the header file, `arguments.h`, to also include the closing bracket of the macro invocation?

```
#define DECLARE_CONSTRUCTOR( CLASS, TYPE, PARAM)  \
        CLASS (TYPE PARAM);

struct Any {

   template <class T>
   DECLARE_CONSTRUCTOR(
#include "arguments.h"

};
```

This program is similarly rejected by Clang. EDG, and MSVC, but is accepted by GCC! Mandating a diagnosable error for all use of `#include` inside a macro argument list would require a change of this one compiler, but is that a common enough case that it might break significant code? Is there a reason that this case is supported, but the plain argument list in the header is not?

Too many additional directives to review each individually, but we note that we have not considered:

- `#` *newline*
- `#define`
- `#error`
- `#pragma`
- `#undef`
- `#warning`
- `export`
- `import`
- `module`

In practice we have no strong use case to support any preprocessor directives in this context other than conditional inclusion:

- `#if`
- `#ifdef`
- `#ifndef`
- `#elif`
- `#elifdef`
- `#elifndef`

— `#else`
— `#endif`

However, once you allow for conditional inclusion, it is relatively easy to imagine use cases for `#error`, `#warning`, and even `#define` and `#undef` within the conditionally included blocks.

Rather than open up this case to deeper analysis, and given the relatively simple workaround for conditional inclusion given by the first example, the recommendation of this paper is to make all preprocessor directive ill-formed in this context, implicitly requiring a diagnostic, and saving users from worrying about special cases.

We expect such a change to loudly break programs that previously had undefined behavior — although conforming compilers would still be allowed to accept such code as long as they issued a diagnostic about using a non-standard extension.

5.3.1 Proposed wording

15.6.1 [cpp.replace.general] General

13 The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the program is ill-formed~~, no diagnostic required~~.

5.3.2 Resolve issue

Resolve [CWG2577] regarding undefined behavior for preprocessing directives in macro arguments.

5.3.3 Compatibility

The proposed changes to not make any currently valid programs ill-formed nor change their meaning. Hence, we suggest no Annex C wording is necessary, although there may be an impact on some implementations rejecting programs with specific undefined behaviors in C++23 as invalid in C++26.

5.4 Invalid syntax for #include

This is [CWG2576]. The program is ill-formed, no diagnostic required, if after all recursive application of macros, a #include directive does not match one of the two specific forms:

```
    # include <filename>
    # include "filename"
```

Testing the main four compiler front ends through Godbolt compiler explorer, we try the following program:

```
#define cstddef <cstddef>
#include cstddef        // well-defined

#define anotherdef "cstddef"
#include anotherdef     // well-defined

#include "cstdlib" ""   // MSVC only warns
#include "cstd""io"

#include
#include vector
#include <vector
#include "vector
#endif

#define A vector
#define B <ios
#define C tream>

#include A
#include B
#include B C            // only gcc accepts
```

5.4.1 Implementation experience

Almost all implementations diagnose the current undefined behavior as ill-formed. The two exceptions are commented in the test program above.

Given the directive #include "cstdlib" "" Microsoft does not perform string concatenation, but does discard the trailing string literal with a warning. All other compilers reject this directive, also (correctly) without performing string concatenation that would occur in phase 6 *after* include directives are processed in phase 4.

The other surprising implementation of undefined behavior is that gcc accepts turning adjacent macro expansions into a single token that can then be successfully interpreted as a header name and included. However, it might be argued that accepting this form is implementation defined according to the last sentence of 15.3 [cpp.include]p4

> The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined into a single header name preprocessing token is implementation-defined.

5.4.2 Implementation experience

To conform to this proposal, MSVC would have to reject its behavior of allowing a trailing string literal on a #include directive. The implementation could continue to accept existing programs that rely on this feature as a conforming extension, and already issues a diagnostic that meets the minimum requirements to do so.

5.4.3 Proposed wording

15.3 [cpp.include] Source file inclusion

4 A preprocessing directive of the form

```
# include pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text (i.e., 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 program is ill-formed~~, no diagnostic required~~. The method by which a sequence of preprocessing tokens between a `<` and a `>` preprocessing token pair or a pair of `"` characters is combined into a single header name preprocessing token is implementation-defined.

5.4.4 Resolve issue

Resolve [CWG2575] regarding undefined behavior for invalid forms of `#include`

5.5 The # operator

This is Core issue [CWG2578].

The issue that leads undefined behavior is for the # operator to expand to text that produces a result that is not a valid character string literal.

Consider the following simple example that uses a macro that will expand to a backslash character, which will escape the quote character that should delimit the end of the character string literal, so that the literal becomes invalid due to the lack a closing delimiter, the quote character in source.

```
// This code demonstrates compile-time UB
#define TO_TEXT(a) #a
#define TEXT(a) TO_TEXT(a)

#define BACKSLASH \\

int main() {
    const char *x = TEXT(BACKSLASH);
}
```

After phase 4 of preprocessing, this program turns into

```
int main() {
    const char *x = "\";
}
```

but there is no requirement to diagnose the error, as the macro expansion that occurs evaluating the TO_TEXT macro is undefined behavior (or ill-formed, no diagnostic required if our simple proposal is adopted).

An attentive reviewer might try to fix this code by ensuring that the character string literal is still terminated with code similar to

```
// This code is ill-formed and requires a diagnostic
#define TO_TEXT(a) #a
#define TEXT(a) TO_TEXT(a)

#define BACKSLASH \\

int main() {
    const char *x = TEXT(BACKSLASH)";
}
```

By adding a quote character after the invocation of TEXT do we close the unterminated character string literal? Unfortunately not, as even before we try to expand the TEXT macro there is an ill-formed attempt to create a string literal with only one of its two delimiting quote characters, so the TEXT macro need not even be expanded for further error reporting.

5.5.1 gcc 14.2

Gcc issues a diagnostic (warning) that it is ignoring the final \, so produces a well-formed empty string and accepts the program.

```
<source>:8:35: warning: invalid string literal, ignoring final '\'
    8 |     const char *x = TEXT(BACKSLASH);
      |                                   ^
```

The preprocessed code:

```
int main() {
    const char *x = "";
}
```

5.5.2 clang 19.1.0

Clang issues a diagnostic (warning) that it is ignoring the final \, so produces a well-formed empty string and accepts the program.

Compiler warning:

```
<source>:8:26: warning: invalid string literal, ignoring final '\'
    8 |     const char *x = TEXT(BACKSLASH);
      |                          ^
<source>:5:19: note: expanded from macro 'BACKSLASH'
    5 | #define BACKSLASH \\
      |                   ^
1 warning generated.
```

Preprocessed code:

```
int main() {
    const char *x = "";
}
```

5.5.3 MSVC 19.40 VS17.10

No warning or error during preprocessing. However, this compiler gives an excellent example of undefined behavior when examining the preprocessed output, with the line of the `main` function declaration somehow incorporated into the string literal that follows it within the source file.

Preprocessed code:

```
    const char *x = "\\nint main() {";
}
```

Subsequent experimentation showed that in the `#define BACKSLASH`, the `\\` token is interpreted as a line continuation, and the MSVC compiler seeks the next non-empty line to complete the `#define` preprocessor directive.

5.5.4 EDG 6.6

No error or warning during macro expansion. However, a diagnostic for an ill-formed string literal is produced as a result of that expansion.

Preprocessed code:

```
int main() {
    const char *x = "\";
}
```

15.6.3 [cpp.stringize] The # operator

2 A *character string literal* is a *string-literal* with no prefix. If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument (excluding placemarker tokens). Let the *stringizing argument* be the preprocessing token sequence for the corresponding argument with placemarker tokens removed. Each occurrence of whitespace between the stringizing argument's preprocessing tokens becomes a single space character in the character string literal. Whitespace before the first preprocessing token and after the last preprocessing token comprising the stringizing argument is deleted. Otherwise, the original spelling of each preprocessing token in the stringizing argument is retained in the character string literal, except for special handling for producing the spelling of *string-literal*s and *character-literal*s: a \ character is inserted before each " and \ character of a *character-literal* or *string-literal* (including the delimiting " characters). If the replacement that results is not a valid character string literal, the program is ill-formed~~, no diagnostic required~~. The character string literal corresponding to an empty stringizing argument is "". The order of evaluation of # and ## operators is unspecified.

5.5.5 Compatibility

The proposed changes to not make any currently valid programs ill-formed nor change their meaning. Hence, we suggest no Annex C wording is necessary, although there may be an impact on some implementations rejecting programs with specific undefined behaviors in C++23 as invalid in C++26.

5.5.6 Implementation experience

EDG already diagnoses the production of ill-formed string literals.

Clang and gcc should promote their warning diagnostic to an error in strictly conforming compiler modes, and can continue with their warning as an extension.

MSVC should produce a specific diagnostic in this case, rather than relying on the garbled output that might or might not become a valid program.

5.5.7 Resolve issues

Resolve [CWG2578] Undefined behavior when token pasting does not create a preprocessing token

5.6 The ## operator

This is core issue [CWG2579]. Much of the original concern for this wording was resolved by paper [P2621R2] adopted in Varna. However, one use of undefined behavior -> "ill-formed, no diagnostic required" remains, when the final result of token pasting does not produce a valid preprocessing token.

Given that subsequent rescanning would expose an invalid preprocessing token, it is not clear what benefit is achieved by declaring such a replacement ill-formed (or UB) before a subsequent rescanning could splice appropriate content to produce a valid token. It would be particularly confusing for the program to survive rescanning, yet still be ill-formed without a diagnostic informing the user of an intermediate state during macro expansion that rendered the program ill-formed.

While we would like to strike the ill-formed, no diagnostic required condition entirely, deferring to subsequent rescanning to report the error, we take the more conservative approach in the initial wording to merely diagnose the error at the point the program becomes ill-formed.

```
#define DO_CONCAT(a,b) a##b
#define CONCAT(a,b) DO_CONCAT(a,b)

#define MINUS -

int main() {
    int word = 0;
    auto x = CONCAT(MINUS, word);
}
```

5.6.1 gcc 14.2

Gcc diagnoses a compilation error for the exact undefined behavior.

```
<source>:4:15: error: pasting "-" and "word" does not give a valid preprocessing token
    4 | #define MINUS -
      |               ^
<source>:1:24: note: in definition of macro 'DO_CONCAT'
    1 | #define DO_CONCAT(a,b) a##b
      |                        ^
<source>:8:13: note: in expansion of macro 'CONCAT'
    8 |     int x = CONCAT(MINUS, word);
      |             ^~~~~~
<source>:8:20: note: in expansion of macro 'MINUS'
    8 |     int x = CONCAT(MINUS, word);
      |                    ^~~~~
```

5.6.2 clang 19.1.0

Clang diagnoses a compilation error for the exact undefined behavior.

```
<source>:8:13: error: pasting formed '-word', an invalid preprocessing token
    8 |     int x = CONCAT(MINUS, word);
      |             ^
<source>:2:21: note: expanded from macro 'CONCAT'
    2 | #define CONCAT(a,b) DO_CONCAT(a,b)
      |                     ^
<source>:1:25: note: expanded from macro 'DO_CONCAT'
    1 | #define DO_CONCAT(a,b) a##b
      |                         ^
1 error generated.
```

5.6.3 MSVC 19.40 VS17.10

MSVC issues no warnings or errors during preprocessing, and accepts the program since the invalid preprocessing token is never used in a preprocessing directive where a valid preprocessing token is required, thus allowing that invalid preprocessing token to decompose into two regular tokens in phase 7.

Preprocessed code:

```
int main() {
    int word = 0;
    int x = -word;
    return x;
}
```

5.6.4 EDG 6.6

EDG issues no warnings or errors during preprocessing, and accepts the program since the invalid preprocessing token is never used in a preprocessing directive where a valid preprocessing token is required, thus allowing that invalid preprocessing token to decompose into two regular tokens in phase 7.

Preprocessed code:

```
int main() {
    int word = 0;
    int x = -word;
    return x;
}
```

5.6.5 Wording

15.6.4 [cpp.concat] The ## operator

3 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a `##` preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token.

[*Note 1:* Concatenation can form a *universal-character-name*. *—end note*]

If the result is not a valid preprocessing token, the program is ill-formed~~, no diagnostic required~~. The resulting token is available for further macro replacement. The order of evaluation of `##` operators is unspecified.

5.6.6 Compatibility

The proposed changes to not make any currently valid programs ill-formed nor change their meaning. Hence, we suggest no Annex C wording is necessary, although there may be an impact on some implementations rejecting programs with specific undefined behaviors in C++23 as invalid in C++26.

5.6.7 Implementation experience

Clang and gcc already diagnose the production of ill-formed preprocessing tokens.

EDG and MSVC would have diagnose the ill-formed tokens, even if they continued to allow the program to be well-formed as a conforming compiler extension.

5.6.8 Resolve issues

Resolve [CWG2579] Undefined behavior when token pasting does not create a preprocessing token

5.7 Predefined macros

This is Core issue [CWG2581].

```
#define __cplusplus 12345678L

#define define gibberish
#define SOMETHING 42

int main() {
    int i = SOMETHING;
    long c = __cplusplus;
}
```

5.7.1 gcc 14.2

Redefining the word `define` has no effect on the preprocessor `#define`. Compiler warning for predefined macro redefined, but no warning for undefining the predefined macro:

```
<source>:1:9: warning: "__cplusplus" redefined
    1 | #define __cplusplus 12345678L
      |         ^~~~~~~~~~
<built-in>: note: this is the location of the previous definition
```

The preprocessed code:

```
int main() {
    int i = 42;
    long c = 12345678L;

    long x = __cplusplus;
}
```

5.7.2 clang 19.1.0

Redefining the word `define` has no effect on the preprocessor `#define`. Essentially the same behavior as gcc:

```
<source>:1:9: warning: redefining builtin macro [-Wbuiltin-macro-redefined]
    1 | #define __cplusplus 12345678L
      |         ^
1 warning generated.
```

Preprocessed code:

```
int main() {
    int i = 42;
    long c = 12345678L;

    long x = __cplusplus;
}
```

5.7.3 MSVC 19.40 VS17.10

Microsoft seems to have the sanest behavior.

Redefining the word `define` has no effect on the preprocessor `#define`. Attempts of changing predefined macros result in warnings and the attempts are ignored:

```
<source>(1): warning C4117: macro name '__cplusplus' is reserved, '#define' ignored
<source>(9): warning C4117: macro name '__cplusplus' is reserved, '#undef' ignored
```

Preprocessed code:

```
int main() {
    int i = 42;
    long c = 199711L;

    long x = 199711L;
}
```

5.7.4 EDG 6.5

Redefining the word `define` has no effect on the preprocessor `#define`. Essentially the same as gcc and clang:

```
"<source>", line 1: warning: incompatible redefinition of macro "__cplusplus"
  #define __cplusplus 12345678L
          ^
```

Preprocessed code:

```
int main() {
    int i = 42;
    long c = 12345678L;

    long x = __cplusplus;
}
```

5.7.5 Implementation experience

5.7.5.1 `#define` macro

	`defined`	`__cplusplus`	`__DATE__`	`__FILE__`	`__LINE__`	`__TIME__`
clang	Error	Warning	Warning	Warning	Warning	Warning
gcc	Error	Warning	Warning	Warning	Warning	Warning
EDG	Error	Warning	Error	Error	Error	Error
MSVC	Warning	Warning	Warning	Warning	Warning	Warning
—–	———	————-	———-	———-	———-	———-
C++23	UB	UB	UB	UB	UB	UB
P2843	Error	Error	Error	Error	Error	Error

5.7.5.2 `#undef` macro

	`defined`	`__cplusplus`	`__DATE__`	`__FILE__`	`__LINE__`	`__TIME__`
clang	Error	OK	Warning	Warning	Warning	Warning
gcc	Error	OK	Warning	Warning	Warning	Warning
EDG	Error	OK	Error	Error	Error	Error
MSVC	Warning	Warning	Warning	Warning	Warning	Warning
—–	———	————-	———-	———-	———-	———-
C++23	UB	UB	UB	UB	UB	UB
P2843	Error	Error	Error	Error	Error	Error

5.7.6 Implementation Guidance

MSVC diagnoses misuse so is already a conforming extension. This paper proposes making each case ill-formed in a strictly conforming mode.

Clang renders attempts to manipulate the `defined` token and the macros `__DATE__`, `__FILE__`, `__LINE__`, and `__TIME__`, as ill-formed, exactly as this paper proposes. While the compiler will warn on attempts to redefine any other predefined macro, it will accept attempts to undefine such macros, which according to this paper should be diagnosed as ill-formed, or at least issue a warning as a compiler extension.

Clang renders attempts to manipulate the `defined` token as ill-formed, exactly as this paper proposes. For attempts to manipulate`the two macros controlled by the`#LINE`directive this compiler issues a warning diagnost` macros. While the compiler will warn on attempts to redefine any other predefined macro, it will accept attempts to undefine such macros, which according to this paper should be diagnosed as ill-formed, or at least issue a warning as a compiler extension.

Gcc behaves the same as Clang and renders attempts to manipulate the `defined` token as ill-formed, exactly as this paper proposes. For attempts to manipulate`the two macros controlled by the`#LINE`directive this compiler iss` macros. While the compiler will warn on attempts to redefine any other predefined macro, it will accept attempts to undefine such macros, which according to this paper should be diagnosed as ill-formed, or at least issue a warning as a compiler extension.

5.7.7 Proposed wording

15.11 [cpp.predefined] Predefined macro names

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 program is ill-formed~~, no diagnostic required~~. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

5.7.8 Resolve issues

Resolve [CWG2580] Undefined behavior when redefining a macro

5.8 The #line directive

This is Core issue [CWG2580].

The first part of this issue is overflow behavior when a compiler tries to pass a number larger than the maximum value for a 32 bit signed integer. Rather than try to analyse this behavior in detail, we will observe that in addition to passing such large numbers to `#line`, we should also consider the overflow behavior that results from setting a number just below this limit, and allowing the file to naturally grow beyond that number of lines. It is observed that the existing compilers behave very differently in this regard today so the proposed change is to simply make values greater than the current maximum conditionally supported with implementation defined behavior. This will place a small documentation burden on all implementations, as the nature of conditionally supported behavior means that they must also document if they reject such programs. This overflow behavior should still be better specified in the case of natural file growth past that number, and such specification is left to another paper.

To determine the undefined behavior for invalid invalid forms of the `#line` directive we tested the following program.

```
#line
#line sdf
#line "xyz"
#line -32
#line 0x123
#line (123)

#line 09    // valid decimal integer
#line 10 2 3
#line 11 "2" 3
#line 12 "4" "5"
#line 13 "6" "7" "8"
```

5.8.1 Implementation experience

All compilers reject the first 6 lines, correctly accept `09` as a decimal number and not an ill-formed octal number, and reject the following line with three integers. However, only EDG rejects the last three lines, which all the other compilers accept with a warning, rejecting any trailing tokens after the filename. They do not perform string concatenation, as observed in the reported error messages for the redundant final tests.

5.8.2 Proposed wording

15.7 [cpp.line] Line control

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 specified by the digit sequence (interpreted as a decimal integer). ~~If the digit sequence specifies zero or a number greater than 2147483647, the program is ill-formed, no diagnostic required~~ Digit sequences representing zero or a number greater than 2,147,483,647 are conditionally supported with implementation defined semantics.

5 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 program is ill-formed~~, no diagnostic required~~; otherwise, the result is processed as appropriate.

5.8.3 Resolve issues

Resolve [CWG2580] regarding undefined behavior for bad usage of the `#line` directive.

6 Further Concerns

6.1 Defining keywords as macros

The core language specifying the preprocessor appears to have no restriction against users defining macros with names that match keywords, alternate tokens, contextual keywords, or standard attribute names. However, there is such a restriction in the standard library requirements on conforming programs, and it is not predicated on including (or importing) a standard header (or module).

Note that while such programs should be diagnosed as ill-formed today, our analysis will show a variety of non-conforming behaviors in current implementations. For implementers concerned that they would break code for existing customers by strictly enforcing the precise specification of the Standard, it is perfectly valid to accept such programs as a conforming extension as long as such usage is diagnosed. Hence, where compilers issue a warning today while accepting the program, they are conforming — although it might be helpful to update those diagnostics to note that they are relying on a compiler extension. Compilers that do not diagnose such macros with bad names should issue an extension diagnostic, but are otherwise unaffected.

6.1.1 Implementation experience

Although this paragraph is normative today, none of our current compilers appear to implement it in full — likely because the paragraph is hidden away in the Standard Library introductory clause, where we would rarely go looking for normative text.

Example program for testing:

```
#define defined 1               // 1
#define defined(...)            // 2
#undef defined                  // 3
#undef _Pragma                  // 4
#define and                     // 5
#undef and                      // 6
#undef __LINE__                 // 7
#undef __cplusplus              // 8
#define public private          // 9
#undef public                   // 10
#undef private                  // 11
#define final                   // 12
#undef override                 // 13
#define noreturn                // 14
#undef noreturn                 // 15
#undef alignas                  // 16
```

Test Line	Clang	EDG	GCC	MSVC
1	Error	Error	Error	Warning
2	Error	Error	Error	Warning
3	Error	Error	Error	Warning
4	Warning	Error	Warning	
5	Error		Error	
6	Error		Error	
7	Warning	Error	Warning	Warning
8	Warning			Warning
9	Warning			
10				
11				
12				
13				

14
15
16

The results are not encouraging. A conforming compiler should report an error for every row. A compiler with conforming extensions could report a warning, but as it is not allowed to change the meaning of a conforming program it is questionable whether that warning can do more than silently ignore the directive.

Note that lines 4 and 5, dealing with `and`, are not subject to the rule in the normative paragraph we propose moving because `and` is *never* an *identifier*, but is always an *operator-or-punctuator* in the grammar as a consequence of the specification for alternate tokens.

No compiler even warns for violating lines 9 onwards, and those are entirely the subject of the normative paragraph being moved. Given that theory and reality do not match we might consider removing this paragraph entirely, but that does not feel very satisfactory.

Another approach would be to allow these redefinitions and undefinitions as conditionally supported behavior with implementation defined semantics. That path would place a new documentation burden on vendors to say what they do with such redefinitions and undefinitions, but all them to either accept the same code that they do today, or to reject such code with a diagnosable error. By making the feature conditionally supported, we remove any requirements to issue a diagnostic when supported.

A further tweak, if we want to move in the direction of bringing vendors into conformance with the standard of today, would be to make this feature both conditionally supported, and deprecated. We might also specify that the conditionally supported forms have no effect.

A further observation on the current specification is that, despite its inclusion in the table above, `_Pragma` is not subject to the same rules against that `defined` is subject to. However, it is still an identifier that is reserved to the implementation by 5.11 [lex.name] so is still not available to users to define and undefine for their own purposes.

The following names are also not called out in 15.11 [cpp.predefined] but have their usage reserved in the subclauses that specify them.

- `__has_include`
- `__has_cpp_attribute`
- `__VA_ARGS__`
- `__VA_OPT__`

6.1.2 Proposed wording

Move the normative paragraph from the library and place it between paragraph 8 and 9 of the core specification for defining macros.

15.6.1 [cpp.replace.general] General

8 The identifier immediately following the `define` is called the *macro name*. There is one name space for macro names. Any whitespace characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

X A translation unit shall not `#define` or `#undef` macro names lexically identical to keywords, to the identifiers listed in Table 4, or to the *attribute-tokens* described in 9.12 [dcl.attr], except that the names `likely` and `unlikely` may be defined as function-like macros (15.6 [cpp.replace]).

9 If a `#` preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive can begin, the identifier is not subject to macro replacement.

16.4.5.3.3 [macro.names] Macro names

1 A translation unit that includes a standard library header shall not #define or #undef names declared in any standard library header.

2 A translation unit shall not #define or #undef names lexically identical to keywords, to the identifiers listed in Table 4, or to the *attribute-tokens* described in 9.12 [dcl.attr], except that the names likely and unlikely may be defined as function-like macros (15.6 [cpp.replace]).

6.2 Ill-formed comments

The control character form-feed and vertical-tab are not allowed in a comment that is introduced by //, unless all the following characters are whitespace. Any other character will render the program ill-formed, no diagnostic required, meaning that the whole program is undefined. It is not clear why no diagnostic would be required here, nor why such characters might not be conditionally supported if the IFNDR wording was to allow programs to successfully compile on existing tool-chains that do not diagnose this error.

We believe that the preferred resolution would be remove the whole concern about characters following the specific cases of form-feed and vertical-tab, but as a compromise to avoid forcing changes on existing implementations that diagnose errors today, we choose conditionally supported behavior. Note that the onus is on implementations that to *not* support conditionally supported behavior to document that they diagnose such programs as ill-formed, where implementations that accept such comments have no impact at all.

If WG21 prefers to fully accept such comments, simply drop the additional text in the wording below that follows the strike-out of existing text.

6.2.1 Proposed wording

5.4 [lex.comment] Comments

1 The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters // start a comment, which terminates immediately before the next new-line character. ~~If there is a form-feed or a vertical-tab character in such a comment, only whitespace characters shall appear between it and the new-line that terminates the comment; no diagnostic is required~~. Non-whitespace characters that appear between a form-feed or a vertical-tab character and the new-line that terminates the comment are conditionally supported.

6.3 Use of reserved identifiers

There is a small set of patters for identifiers that are reserved for use by the implementation and a program trying to make its own use of these identifiers, as macros, variables, function names, type names, enumerators, etc., are ill-formed, no diagnostic required.

If we were to enforce the rule that such programs are plain ill-formed and so requiring a diagnostic, an implementation remains free to continue accepting such programs as a conforming extension as long as such use of reserved identifiers is diagnosed as such.

Hence, in the interests of advancing a well-specified standard, we propose that use of identifiers reserved to the C++ Standard make a program plain ill-formed, which just adds a diagnostic requirement to the current Standard. Note that we are making no such recommendation for names reserved by other standards, even those incorporated by reference. In particular, POSIX reserves a larger set of names, some of which are actively used by identifiers in the C++ Standard itself.

6.3.1 Proposed wording

5.11 [lex.name] Identifiers

3 In addition, some identifiers appearing as a token or preprocessing-token are reserved for use by C++ implementations and shall not be used otherwise~~; no diagnostic is required~~.

(3.1) — Each identifier that contains a double underscore `__` or begins with an underscore followed by an uppercase letter, other than those specified in this document (for example, `__cplusplus` (15.11)), is reserved to the implementation for any use.

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

7 Impact on Implementations

We collect the following observations to summarize the known impact on current implementations to come into conformance with this paper.

7.1 Nonfunctional changes

The change from undefined behavior to ill-formed, no diagnostic required makes no demands on implementations, and is purely a tightening of our specification. There is no risk.

For other changes, we can lean heavily into conforming extension modes, where accepting an ill-formed program where diagnostics are required is permitted, as long as a diagnostic is emitted — and in many of these cases compilers do emit warnings. In some cases, those warnings might want to be recategorized as extensions warnings, and compilers with strict conformance modes would promote those warnings to errors.

For the subset of changes where we want to endorse specific behavior, we can introduce that behavior as conditionally supported, so that compilers diagnosing errors today remain conforming, but strictly speaking should update their documentation to say that they do not implement the conditionally supported behavior.

7.2 Functional changes

In a variety of cases implementations already diagnose and reject programs, so no further work is needed.

In some cases we make a program ill-formed, even where implementations do not emit a diagnostic today. As noted above, in order to preserve customer code and migrate at a pace that implementers are happy no negotiate with their clients, each of these cases can be turned into a conforming extension by issuing a warning while accepting the program. The meaning of such programs would naturally be for said implementations to define, but the Standard places no documentation requirements on implementations to document the behavior of their extensions, but merely to diagnose their use.

7.3 Analysis by compiler

Listing the main front ends alphabetically, we will list the changes that are not diagnosed today, and those cases that are warned today but should be diagnosed as extensions, and rejected in a strictly conforming mode.

7.3.1 All

First, we will examine behaviors common to all compilers.

7.3.1.1 No change of behavior

Some parts of this proposal are cleaning up the existing specification and imply no change of behavior. Any compilers that fail to meet these changes already have those same issues against the current standard.

— Turn undefined behavior into ill-formed, no diagnostic required
— Reject use of keywords as macro names

7.3.1.2 Conforming

Some parts of this paper are formalizing existing practice.

— Conditional support for comments with non-whitespace after form-feed or vertical-tab
— Accept macros defined as `defined`
— Reject all misuse of the `defined` operator

7.3.1.3 Must diagnose

— Use of keywords as macros

Note that these diagnostics are already required for a conforming implementation, unlike most of the rest of this paper, and also listed under no change of behavior. However, this requirement is found in the library specification which can be a surprise for compiler implementers, even though there are expected requirements on compilers throughout Library clause 17 [support], and all implementers require some extra diagnostics for conforming extensions here, so this feature is listed twice.

7.3.2 Clang

The Clang compiler should audit its behavior for preprocessor directive within macro argument lists. It has a mix of cases that are ill-formed, warnings, or accepted (but ill-considered?) that should be updated so that all are either ill-formed (conforming to this proposal) or continue to warn as conforming compiler extensions.

7.3.2.1 Must diagnose

— `#undef` predefined macro other than `__DATE__`, `__FILE__`, `__LINE__`, and `__TIME__`

7.3.2.2 Extensions

— Creating an invalid string literal via the `#` operator
— `#define` any predefined macro
— `#undef __DATE__`, `__FILE__`, `__LINE__`
— Diagnose trailing preprocessor tokens follow the filename of a `#line` directive

7.3.3 EDG

The EDG front end should audit its behavior for preprocessor directive within macro argument lists. It has a mix of cases that are ill-formed, warnings, or accepted (but ill-considered?) that should be updated so that all are either ill-formed (conforming to this proposal) or continue to warn as conforming compiler extensions.

7.3.3.1 Must diagnose

— Creating an invalid preprocessing token via the `##` operator
— `#undef` predefined macro other than `__DATE__`, `__FILE__`, `__LINE__`, and `__TIME__`

7.3.3.2 Extensions

— `#define` predefined macro other than `__DATE__`, `__FILE__`, `__LINE__`, and `__TIME__`

7.3.4 Gcc

The gcc compiler should audit its behavior for preprocessor directive within macro argument lists. It has a mix of cases that are ill-formed, warnings, or accepted (but ill-considered?) that should be updated so that all are either ill-formed (conforming to this proposal) or continue to warn as conforming compiler extensions.

7.3.4.1 Must diagnose

— `#undef` predefined macro other than `__DATE__`, `__FILE__`, `__LINE__`, and `__TIME__`

7.3.4.2 Extensions

— Creating an invalid string literal via the `#` operator
— `#define` any predefined macro
— `#undef __DATE__`, `__FILE__`, `__LINE__`, and `__TIME__`
— Diagnose trailing preprocessor tokens follow the filename of a `#line` directive

7.3.5 Microsoft Visual C++

7.3.5.1 Must diagnose

— Creating an invalid string literal via the `#` operator
— Creating an invalid preprocessing token via the `##` operator

7.3.5.2 Extensions

— Allowing a trailing string literal on `#include "header"`
— `#define defined`
— `#undef defined`
— `#define` any predefined macro
— `#undef` any predefined macro
— Diagnose trailing preprocessor tokens follow the filename of a `#line` directive

8 Proposed wording

Make the following changes to the C++ Working Draft. All wording is relative to [N5001], the latest draft at the time of writing.

5.4 [lex.comment] Comments

1 The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters // start a comment, which terminates immediately before the next new-line character. ~~If there is a form-feed or a vertical-tab character in such a comment, only whitespace characters shall appear between it and the new-line that terminates the comment; no diagnostic is required~~. Non-whitespace characters that appear between a form-feed or a vertical-tab character and the new-line that terminates the comment are conditionally supported.

5.11 [lex.name] Identifiers

3 In addition, some identifiers appearing as a token or preprocessing-token are reserved for use by C++ implementations and shall not be used otherwise~~; no diagnostic is required~~.

(3.1) — Each identifier that contains a double underscore __ or begins with an underscore followed by an uppercase letter, other than those specified in this document (for example, `__cplusplus` (15.11)), is reserved to the implementation for any use.

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

15.2 [cpp.cond] Conditional inclusion

10 Prior to evaluation, 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. If ~~the preprocessing token `defined` is generated as a result of this replacement process or~~ use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the ~~behavior is undefined~~program is ill-formed.

15.3 [cpp.include] Source file inclusion

4 A preprocessing directive of the form

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

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text (i.e., 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~~program is ill-formed, no diagnostic required. The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined into a single header name preprocessing token is implementation-defined.

15.6.1 [cpp.replace.general] General

8 The identifier immediately following the `define` is called the *macro name*. There is one name space for macro names. Any whitespace characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

X A translation unit shall not `#define` or `#undef` macro names lexically identical to keywords, to the identifiers listed in Table 4, or to the *attribute-tokens* described in 9.12 [dcl.attr], except that the names `likely` and `unlikely` may be defined as function-like macros (15.6 [cpp.replace]).

9 If a # preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive can begin, the identifier is not subject to macro replacement.

13 The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the ~~behavior is undefined~~program is ill-formed.

15.6.3 [cpp.stringize] The # operator

2 A *character string literal* is a *string-literal* with no prefix. If, in the replacement list, a parameter is immediately preceded by a `#` preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument (excluding placemarker tokens). Let the *stringizing argument* be the preprocessing token sequence for the corresponding argument with placemarker tokens removed. Each occurrence of whitespace between the stringizing argument's preprocessing tokens becomes a single space character in the character string literal. Whitespace before the first preprocessing token and after the last preprocessing token comprising the stringizing argument is deleted. Otherwise, the original spelling of each preprocessing token in the stringizing argument is retained in the character string literal, except for special handling for producing the spelling of *string-literal*s and *character-literal*s: a `\` character is inserted before each `"` and `\` character of a *character-literal* or *string-literal* (including the delimiting `"` characters). If the replacement that results is not a valid character string literal, the ~~behavior is undefined~~program is ill-formed, no diagnostic required. The character string literal corresponding to an empty stringizing argument is `""`. The order of evaluation of `#` and `##` operators is unspecified.

15.6.4 [cpp.concat] The ## operator

3 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a `##` preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token.

[*Note 1:* Concatenation can form a *universal-character-name* (5.3.1 [lex.charset]). *—end note*]

If the result is not a valid preprocessing token, the ~~behavior is undefined~~program is ill-formed, no diagnostic required. The resulting preprocessing token is available for further macro replacement. The order of evaluation of `##` operators is unspecified.

15.7 [cpp.line] Line control

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 specified by the digit sequence (interpreted as a decimal integer). ~~If the digit sequence specifies zero or a number greater than 2147483647, the program is ill-formed, no diagnostic required~~ Digit sequences representing zero or a number greater than 2,147,483,647 are conditionally supported with implementation defined semantics.

5 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~~program is ill-formed, no diagnostic required; otherwise, the result is processed as appropriate.

15.11 [cpp.predefined] Predefined macro names

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~~program is ill-formed. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

16.4.5.3.3 [macro.names] Macro names

1 A translation unit that includes a standard library header shall not `#define` or `#undef` names declared in any standard library header.

2 A translation unit shall not `#define` or `#undef` names lexically identical to keywords, to the identifiers listed in Table 4, or to the *attribute-tokens* described in 9.12 [dcl.attr], except that the names `likely` and `unlikely` may be defined as function-like macros (15.6 [cpp.replace]).

8.1 Update Annex C

No changes expected, as no valid program is changed.

8.2 Close issues as resolved

Close the following Core issues resolved as ill-formed by this paper:

- [CWG2575] Undefined behavior when macro-replacing `defined` operator
- [CWG2576] Undefined behavior with macro-expanded `#include` directives
- [CWG2577] Undefined behavior for preprocessing directives in macro arguments
- [CWG2578] Undefined behavior when creating an invalid string literal via stringizing
- [CWG2579] Undefined behavior when token pasting does not create a preprocessing token
- [CWG2581] Undefined behavior for predefined macros

Close the following Core issue resolved as conditionally supported by this paper, and further specification would require a new paper:

- [CWG2580] Undefined behavior with `#line`

9 Acknowledgements

Thanks to Michael Park for the pandoc-based framework used to transform this document's source from Markdown.

Thanks to Joshua Berne for making the painstaking effort to be sure I understand the subtle differences between undefined behavior, unspecified behavior, implementation-defined behavior, and programs that are ill-formed, no diagnostic required. I am not the easiest of students!

Thanks to Attila Fehér for providing many of the examples used to analyse the existing preprocessor behavior for troublesome code.

10 References

[CWG2575] US. 2019-10-23. Undefined behavior when macro-replacing "defined" operator.
https://wg21.link/cwg2575

[CWG2576] US. 2019-10-23. Undefined behavior with macro-expanded #include directives.
https://wg21.link/cwg2576

[CWG2577] US. 2019-10-23. Undefined behavior for preprocessing directives in macro arguments.
https://wg21.link/cwg2577

[CWG2578] US. 2019-10-23. Undefined behavior when creating an invalid string literal via stringizing.
https://wg21.link/cwg2578

[CWG2579] US. 2019-10-23. Undefined behavior when token pasting does not create a preprocessing token.
https://wg21.link/cwg2579

[CWG2580] US. 2019-10-23. Undefined behavior with #line.
https://wg21.link/cwg2580

[CWG2581] US. 2019-10-23. Undefined behavior for predefined macros.
https://wg21.link/cwg2581

[N3801] Gabriel Dos Reis. 2013-10-14. Removing Undefined Behavior from the Preprocessor.
https://wg21.link/n3801

[N4220] David Krauss. 2014-10-10. An update to the preprocessor specification (rev. 2).
https://wg21.link/n4220

[N4858] Barry Hedquist. 2020-02-15. Disposition of Comments: SC22 5415, ISO/IEC CD 14882.
https://wg21.link/n4858

[N4986] Thomas Köppe. 2024-07-16. Working Draft, Programming Languages — C++.
https://wg21.link/n4986

[N5001] Thomas Köppe. 2024-12-17. Working Draft, Programming Languages — C++.
https://wg21.link/n5001

[P1705R1] Shafik Yaghmour. 2019-10-07. Enumerating Core Undefined Behavior.
https://wg21.link/p1705r1

[P2234R1] Scott Schurr. 2021-02-13. Consider a UB and IF-NDR Audit.
https://wg21.link/p2234r1

[P2621R2] Corentin Jabot. 2023-02-08. UB? In my Lexer?
https://wg21.link/p2621r2