Preprocessing is never undefined

Replacing UB with IFNDR

Document #:	P2843R0
Date:	2023-05-18
Project:	Programming Language C++
Audience:	SG12 Undefined Behavior
Revises:	N/A
Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net>

Contents

1 Abstract

This paper revises all specification of the C++ preprocessor using the term *undefined behavior* with the more appropriate term, *ill-formed, no diagnostic required*.

2 Revision history

2.1 R0: Varna 2023

Initial draft of the 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.

4 Basic Proposal

A better formulation for all cases where the preprocessor specifies undefined behavior in the preprocessor is that the program is ill-formed, no diagnostic required. Such a change would have no effect on any implementations today, and serves as the basis for the follow up work below, making most of those cases diagnosable, or well-defined.

All wording is relative to [N4944], the latest working draft at the time of writing.

4.1 Original 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 ~~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-literals* and character-literals: 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. If the result begins with a sequence matching the syntax of *universal-character-name*, the ~~behavior is undefined~~ program is ill-formed, no diagnostic required.

[*Note 1:* This determination does not consider the replacement of *universal-character-name*s in translation phase 3 (5.2 [lex.phases]). —*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.

4.2 Conflict resolution

Corentin Jabot has paper [P2621R2] that resolves the outstanding concerns for undefined behavior in the lexer, which is tentatively ready to land at the Varna meeting. Thus, we do not cover the lexer in this paper, but do note that both papers make changes to the same paragraph in 15.6.4 [cpp.concat]. If both papers are applied at the same meeting, P2621 takes precedence.

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. ~~If the result begins with a sequence matching the syntax of *universal-character-name*, the behavior is undefined~~.

[*Note 1:* This determination does not consider the replacement of *universal-character-names* in translation phase 3 (5.2 [lex.phases]). *—end note*]

[*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.

5 Towards a better specification

While the above wording is the minimum progress we should aim for, replacing an inappropriate term with a more correct one, there is a long history of trying to address the undefined nature of the preprocessor. Note that this section of the paper is the start of an ongoing review that is incomplete in this revision of paper.

5.1 Past efforts

— [N3801] Removing Undefined Behavior from the Preprocessor, Gabriel Dos Reis
— [N4219] Fixing the specification of universal-character-names, David Krauss
— [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

6 Reviewing each ill-formed constructions

For the benefit of the following examples, we are assuming the UB -> IFNDR change at the start of this paper has been applied.

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

};
```

7 Acknowledgements

Thanks to Michael Parks 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!

8 References

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

[N4219] David Krauss. 2014-10-10. Fixing the specification of universal-character-names (rev. 2).
https://wg21.link/n4219

[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

[N4944] Thomas Köppe. 2023-03-22. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4944

[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