Avoid UB When Compiling Code That Violates Library Specification

Replacing compile-time undefined behavior with IFNDR

Document #:	P3534R0
Date:	2024-12-11
Project:	Programming Language C++
Audience:	Library Working Group
Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net>

Contents

1 Abstract

This paper revises the specification of the C++ Standard Library so that misuses of the library that are observed during program translation no longer use the term *undefined behavior* and instead use the more appropriate term *ill-formed, no diagnostic required*.

2 Revision History

R0 December 2024 (post-Wrocław mailing)

Initial draft of this paper

3 Introduction

Several parts of the Standard Library specification call for undefined behavior (UB) when code is written in a way that would violate Library assumptions. Some of these rules forbid users to write certain declarations and template specializations, and in such cases, any undefined behavior must necessarily occur when translating the program — i.e., in the compiler — rather than when the program is executing. In some cases, we say that "the program has undefined behavior," but our specification for the terms *program* and *undefined behavior* do not give a meaning to the phrase that includes them; rather, when no requirements are placed on the behavior of a whole program, we specify that the program is *ill-formed, no diagnostic required*.

This paper is the library analog of [P2843R1], but unlike that paper, it merely tries to fix all inappropriate uses of *undefined behavior* without requiring any changes of semantic, such as supporting more behavior or requiring misuse to be diagnosed.

4 Principles

The guiding principle is to use the correct terminology in the appropriate specifications. *Undefined behavior* is most appropriate for library functionality that is not supportable due to encountering a bad program state at run time. We already make good use of *Mandates:* and *Constraints:* to describe misuse that can be recognized at compile time, and in several cases, we already use *ill-formed, no diagnostic required* for situations that cannot be supported but also cannot easily be diagnosed in all cases.

The main outstanding concern is the use of *undefined behavior* to describe situations when *ill-formed, no diagnostic required* is more appropriate, such as when providing a declaration that would interfere with overload resolution in a way that would observably change library behavior such that the specified behavior could no longer be guaranteed.

5 Frequently Asked Questions

5.1 Isn't IFNDR strictly worse than UB?

When considering runtime behavior, then yes, ill-formed, no diagnostic required (IFNDR) is strictly worse than UB because IFNDR places no requirements on the whole program, rather than just the runtime execution that tripped over UB and that might never occur in a program fed with correct data. However, UB in the act of translating a program is strictly worse than IFNDR because UB makes the act of compilation a source of unmitigated risk. UB under such circumstances is both surprising and generally unnecessary.

5.2 Why is skipping review by the Library Evolution Working Group appropriate?

This paper is entirely editorial in nature and has no impact on a well-formed, well-defined C++ program. If we were to make further proposals to diagnose some of the IFNDR constraints, then a process passing through the LEWG would be appropriate.

6 Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N4993], the latest draft at the time of writing.

16.4.5 [constraints] Constraints on programs

16.4.5.1 [constraints.overview] Overview

1 Subclause 16.4.5 [constraints] describes restrictions on C++ programs that use the facilities of the C++ standard library. The following subclauses specify constraints on the program's use of namespaces (16.4.5.2 [namespace.constraints]), its use of various reserved names (16.4.5.3 [reserved.names]), its use of headers (16.4.5.4 [alt.headers]), its use of standard library classes as base classes (16.4.5.5 [derived.classes]), its definitions of replacement functions (16.4.5.6 [replacement.functions]), and its installation of handler functions during execution (16.4.5.7 [handler.functions]).

16.4.5.2 [namespace.constraints] Namespace use

16.4.5.2.1 [namespace.std] Namespace `std`

1 Unless otherwise specified, ~~the behavior of a C++ program is undefined if it~~ a C++ program that adds declarations or definitions to namespace `std` or to a namespace within namespace `std` is ill-formed, no diagnostic required.

2 Unless explicitly prohibited, a program may add a template specialization for any standard library class template to namespace `std` provided that

- — the added declaration depends on at least one program-defined type, and
- — the specialization meets the standard library requirements for the original template.

3 ~~The behavior of a C++ program is undefined if it~~ A C++ program that declares an explicit or partial specialization of any standard library variable template, except where explicitly permitted by the specification of that variable template, is ill-formed, no diagnostic required.

[*Note 1:* The requirements on an explicit or partial specialization are stated by each variable template that grants such permission. —*end note*]

4 ~~The behavior of a C++ program is undefined if it declares~~ A C++ program that declares

- — an explicit specialization of any member function of a standard library class template, or
- — an explicit specialization of any member function template of a standard library class or class template, or
- — an explicit or partial specialization of any member class template of a standard library class or class template, or
- — a deduction guide for any standard library class template~~.~~

is ill-formed, no diagnostic required.

5 A program may explicitly instantiate a class template defined in the standard library only if the declaration

- — depends on the name of at least one program-defined type, and
- — the instantiation meets the standard library requirements for the original template.[1]

No diagnostic is required.

6 Let *F* denote a standard library function (16.4.6.4 [global.functions]), a standard library static member function, or an instantiation of a standard library function template. Unless *F* is designated an *addressable function*, the behavior of a C++ program is unspecified (possibly ill-formed) if it explicitly or implicitly attempts to form a pointer to F.

[1]Any library code that instantiates other library templates must be prepared to work adequately with any user-supplied specialization that meets the minimum requirements of this document.

[*Note 2:* Possible means of forming such pointers include application of the unary `&` operator (7.6.2.2 [expr.unary.op]), `addressof` (20.2.11 [specialized.addressof]), or a function-to-pointer standard conversion (7.3.4 [conv.func]). —*end note*]

Moreover, the behavior of a C++ program is unspecified (possibly ill-formed) if it attempts to form a reference to *F* or if it attempts to form a pointer-to-member designating either a standard library non-static member function (16.4.6.5 [member.functions]) or an instantiation of a standard library member function template.

7 A translation unit shall not declare namespace `std` to be an inline namespace (9.8.2 [namespace.def]).

16.4.5.2.2 [namespace.posix] Namespace `posix`

1 ~~The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `posix` or to a namespace within namespace `posix` unless otherwise specified.~~ The namespace `posix` is reserved for use by ISO/IEC/IEEE 9945 and other POSIX standards. Unless otherwise specified, a program that adds declarations or definitions to namespace `posix` or to a namespace within namespace `posix` is ill-formed, no diagnostic required.

16.4.5.2.3 [namespace.future] Namespaces for future standardization

1 Top-level namespaces whose *namespace-name* consists of `std` followed by one or more *digits* (5.11 [lex.name]) are reserved for future standardization. ~~The behavior of a C++ program is undefined if it~~ A C++ program that adds declarations or definitions to such a namespace is ill-formed, no diagnostic required.

[*Example 1:* The top-level namespace `std2` is reserved for use by future revisions of this International Standard. —*end example*]

16.4.5.3 [reserved.names] Reserved names

16.4.5.3.1 [reserved.names.general] General

1 The C++ standard library reserves the following kinds of names:

— macros
— global names
— names with external linkage

2 ~~If a program~~ A program that declares or defines a name in a context where it is reserved, other than as explicitly allowed by 16 [library], ~~its behavior is undefined~~ is ill-formed, no diagnostic required.

16.4.5.3.2 [zombie.names] Zombie names

1 In namespace `std`, the names shown in Table 38 are reserved for previous standardization:

Table 38 — Zombie names in namespace std [tab:zombie.names.std]

2 The names shown in Table 39 are reserved as members for previous standardization~~, and~~:

[*Note 1:* These names may not be used as a name for object-like macros in portable code~~:~~. —*end note*]

Table 39 — Zombie object-like macros [tab:zombie.names.objmacro]

3 The names shown in Table 40 are reserved as member functions for previous standardization~~, and~~:

[*Note 2:* These names may not be used as a name for function-like macros in portable code~~:~~. —*end note*]

Table 40 — Zombie function-like macros [tab:zombie.names.fnmacro]

4 The header names shown in Table 41 are reserved for previous standardization:

Table 41 — Zombie headers [tab:zombie.names.header]

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. No diagnostic is required.

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]). No diagnostic is required.

16.4.5.4 [alt.headers] Headers

1 If a file with a name equivalent to the derived file name for one of the C++ standard library headers is not provided as part of the implementation, and a file with that name is placed in any of the standard places for a source file to be included (15.3 [cpp.include]), the ~~behavior is undefined~~ program is ill-formed, no diagnostic required.

17.11.5 [cmp.result] Result of three-way comparison

1 ~~The behavior of a~~ A program that adds specializations for the `compare_three_way_result` template defined in this subclause is ~~undefined~~ ill-formed, no diagnostic required.

17.12.4 [coroutine.handle] Class template `coroutine_handle`

[coroutine.handle.general] General

2 ~~If a~~ A program that declares an explicit or partial specialization of `coroutine_handle`~~, the behavior is undefined~~ is ill-formed, no diagnostic required.

21.3.2 [meta.rqmts] Requirements

4 Unless otherwise specified, ~~the behavior of~~ a program that adds specializations for any of the templates specified in 21.3 [type.traits] is ~~undefined~~ ill-formed, no diagnostic required.

5 Unless otherwise specified, an incomplete type may be used to instantiate a template specified in 21.3 [type.traits]. The ~~behavior of a program is undefined~~ program is ill-formed, no diagnostic required, if

— an instantiation of a template specified in 21.3 [type.traits] directly or indirectly depends on an incompletely-defined object type `T`, and

— that instantiation could yield a different result were `T` hypothetically completed.

25.8.3 [coro.generator.class] Class template `generator`

4 ~~The behavior of a~~ A program that adds a specialization for `generator` is ~~undefined~~ ill-formed, no diagnostic required.

26.3.6.2 [execpol.type] Execution policy type trait

3 ~~The behavior of a~~ A program that adds specializations for `is_execution_policy` is ~~undefined~~ ill-formed, no diagnostic required.

28.5.8.1 [format.arg] Class template `basic_format_arg`

2 ~~The behavior of a~~ A program that adds specializations for `basic_format_arg` is ~~undefined~~ ill-formed, no diagnostic required.

29.5.3 [rand.req] Requirements

29.5.3.1 [rand.req.genl] General requirements

1 Throughout 29.5 [rand], ~~the effect of instantiating a template:~~ a program that instantiates a template:

- — that has a template type parameter named `Sseq` is ~~undefined~~ ill-formed, no diagnostic required unless the corresponding template argument is cv-unqualified and meets the requirements of seed sequence (29.5.3.2 [rand.req.seedseq]).
- — that has a template type parameter named `URBG` is ~~undefined~~ ill-formed, no diagnostic required unless the corresponding template argument is cv-unqualified and meets the requirements of uniform random bit generator (29.5.3.3 [rand.req.urng]).
- — that has a template type parameter named `Engine` is ~~undefined~~ ill-formed, no diagnostic required unless the corresponding template argument is cv-unqualified and meets the requirements of random number engine (29.5.3.4 [rand.req.eng]).
- — that has a template type parameter named `RealType` is ~~undefined~~ ill-formed, no diagnostic required unless the corresponding template argument is cv-unqualified and is one of `float`, `double`, or `long double`.
- — that has a template type parameter named `IntType` is ~~undefined~~ ill-formed, no diagnostic required unless the corresponding template argument is cv-unqualified and is one of `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.
- — that has a template type parameter named `UIntType` is ~~undefined~~ ill-formed, no diagnostic required unless the corresponding template argument is cv-unqualified and is one of `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.

30.4.4 [time.traits.is.clock] Class template `is_clock`

2 ~~The behavior of a~~ A program that adds specializations for `is_clock` is ~~undefined~~ ill-formed, no diagnostic required.

32.11.2.3 [saferecl.rcu.base] Class template `rcu_obj_base`

2 ~~The behavior of a~~ A program that adds specializations for `rcu_obj_base` is ~~undefined~~ ill-formed, no diagnostic required.

32.11.3.3 [saferecl.hp.base] Class template `hazard_pointer_obj_base`

2 ~~The behavior of a~~ A program that adds specializations for `hazard_pointer_obj_base` is ~~undefined~~ ill-formed, no diagnostic required.

7 Acknowledgements

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

Thanks to Lori Hughes for reviewing this paper and providing editorial feedback.

8 References

[N4993] Thomas Köppe. 2024-10-16. Working Draft, Programming Languages — C++.
https://wg21.link/n4993

[P2843R1] Alisdair Meredith. Preprocessing is never undefined.
https://wg21.link/d2843r1