Remove Deprecated Arithmetic Conversion on Enumerations From C++26

Document #:	P2864R0
Date:	2023-05-15
Project:	Programming Language C++
Audience:	Evolution SG22 C interoperability
Revises:	N/A

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

Contents

1 Abstract

C++ has deprecated a number of features related to implicit conversions of enumeration values in comparison operators that are often misleading and easily used accidentally. This paper proposes removing those features from C++26.

2 Revision History

2.1 R0: Varna 2023

Initial draft of this paper, based on [P2139R2].

3 Introduction

At the start of the C++23 cycle, [P2139R2] tried to review each deprecated feature of C++ to see which we would benefit from actively removing and which might now be better undeprecated. Consolidating all this analysis into one place was intended to ease the (L)EWG review process but in return gave the author so much feedback that the next revision of the paper was not completed.

For the C++26 cycle, a much shorter paper, [P2863R0], will track the overall analysis, but for features that the author wants to actively progress, a distinct paper will decouple progress from the larger paper so that the delays on a single feature do not hold up progress on all.

This paper takes up the deprecated arithmetic conversion on enumerations, D.2 [depr.arith.conv.enum].

4 Rationale

With the introduction of the three-way comparison "spaceship" operator, committee members were concerned with avoiding some implicit comparisons that might be lossy (due to rounding of floating point values) or giving up intended type safety (by using enumeration rather than integer types to indicate more than just a value). While the three-way comparison operators are specified to reject such comparisons, the pre-existing comparison operators were granted ongoing compatibility in these cases but were deprecated. Most but not all such usage relying on implicit conversion is likely a latent bug, and reading such code would always be clearer if the implicit conversions were made explicit. Note that avoiding explicit casts is still possible by using unary `operator+`, forcing integral promotion.

5 Analysis

Certain arithmetic conversion on enumerations were deprecated for C++20 by [P1120R0] as part of the effort to make the new spaceship operator Do The Right Thing. These deprecations potentially impact code written according to C++98 and later Standards.

Specifically, `enum` objects comparing against floating point values were deprecated by C++20 as were comparisons between different `enum` types and arithmetic operations between different `enum` types. For clarity, operations between integer types and enumerations are not impacted, so promoting an `enum` value to an integer with unary `operator+` is often a quick fix:

```
int main() {
   enum E1 { e };
   enum E2 { f };
   int k =  f - e;      // deprecated
   int x = +f - e;      // OK
}
```

Note that these conversions are deprecated only when evaluating comparison operators; in all other contexts, such conversions are already ill-formed.

5.1 Implementation experience

The following program was tested on Godbolt compiler explorer, pulled from Annex D of the Standard.

```
int main() {
   enum E1 { e };
   enum E2 { f };
   bool b = e <= 3.7;  // deprecated
   int  k = f - e;     // deprecated
}
```

Built in (sometimes experimental) C++20 mode, we find deprecation warnings from the following compiler versions:

```
clang      10
GCC        11.1
MSVC       19.22 (floating point only)
EDG/Intel  Does not warn with EDG front end
```

6 Past Feedback

6.1 Initial review: Telecon 2020/06/09

Concerns were raised about ongoing C compatibility. Any recommendation on removing deprecated support in D.1 will be forwarded to our WG14 liaison to hopefully have a coordinated process for removing such features.

Concerns were raised over several issues of interoperating with different enumeration types, including expressions used to initialize global variables, and for expressions used to define array bounds. Similarly, metaprogramming idioms from C++98 often used enumerations to denote result values (saving on storage for a static data member), and comparing such named values from different instantiations of the same template is the expected usage.

Another concern was raised regarding idioms that externally extend another enumeration without intruding on the original, e.g.,

```
enum original { e_FIRST, ... , e_LAST };
enum extended { e_NEXT = e_LAST + 1, ... , e_MORE };
```

with the intent that the extended enumeration can interoperate with the original.

Fewer concerns arose for removing the interoperation with floating point types, although some discussion focused on the workarounds. Concerns were raised that the "simple" forced promotion through unary operator `+` is too clever/cute but that `static_cast`, or C-style casts, are verbose and risk converting to a different type than the previous implicit integer promotion.

An ongoing concern focuses on gaining more specific feedback on implementation experience with the deprecation warning before making any change and on gaining experience with any removal that might silently change behavior in SFINAE contexts.

6.1.1 Polls:

Q1: In C++23, un-deprecate enum vs. enum?

```
| SF  F  N  A SA
|  1  2  7  6  4
```

No consensus.

Q2: In C++23, un-deprecate enum vs. floating-point?

```
| SF  F  N  A SA
|  8  1  4  8  6
```

No consensus.

Q3: In C++23, remove the deprecated enum vs. enum facilities?

SF	F	N	A	SA
1	2	10	7	1

No consensus.

Q4: In C++23, remove the deprecated enum vs. floating-point facilities?

SF	F	N	A	SA
7	10	4	0	1

Consensus.

7 Proposal

This paper proposes removing *both* kinds of deprecated conversions. Note that `enum` conversions and comparisons for `enum class` were never supported, but both changes would be a backward compatibility concern for code that compiles with both C and C++.

8 Alternative Proposals

We might also consider a few options.

8.1 Undeprecate both forms

If we believe the proposed changes are actively harmful to good code and best practices, we should undeprecate both behaviors. Anecdotal data from resolving deprecation warnings in our own code shows that the warnings raised were good to consider and relatively simple to resolve, so we reject this option.

8.2 Remove floating-point comparison but retain deprecated `enum` comparisons

Noting the feedback from the C++23 cycle, acting on only removing the floating-point comparisons while retaining, as deprecated, the support for different `enum` types might be preferred.

The question with this approach is what we expect to gain by retaining the deprecated conversions. Compilers have been warning for a few years, and two Standards have shipped. What more would it take to persuade us to change the status of those conversions?

8.3 Remove floating-point comparison but retain deprecated `enum` comparisons

Noting the feedback from the C++23 cycle, acting on only removing the floating-point comparisons and restoring the support for different `enum` types might be preferred.

We note that the earlier polling leaned more strongly against undeprecation than it did toward removal, hence the recommendation for outright removal as the primary proposal. However, if we do not expect to learn anything new in the next one or two Standard cycles, we should consider undeprecating a feature if we lack the confidence to ever remove it.

9 Proposed Wording

All changes are relative to [N4944].

7.4 Usual arithmetic conversions [expr.arith.conv]

[1] Many binary operators that expect operands of arithmetic or enumeration type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*, which are defined as follows:

- If either operand is of scoped enumeration type (9.7.1 [dcl.enum]), no conversions are performed; if the other operand does not have the same type, the expression is ill-formed.
- Otherwise, if one operand is of enumeration type and the other operand is of a different enumeration type or a floating-point type, the expression is ill-formed.
- Otherwise, if either operand is of floating-point type, the following rules are applied:
  - If both operands have the same type, no further conversion is needed.
  - Otherwise, if one of the operands is of a non-floating-point type, that operand is converted to the type of the operand with the floating-point type.
  - Otherwise, if the floating-point conversion ranks (6.8.6 [conv.rank]) of the types of the operands are ordered but not equal, then the operand of the type with the lesser floating-point conversion rank is converted to the type of the other operand.
  - Otherwise, if the floating-point conversion ranks of the types of the operands are equal, then the operand with the lesser floating-point conversion subrank (6.8.6 [conv.rank]) is converted to the type of the other operand.
  - Otherwise, the expression is ill-formed.
- Otherwise, each operand is converted to a common type `C`. The integral promotion rules (7.3.7 [conv.prom]) are used to determine a type `T1` and type `T2` for each operand. Then the following rules are applied to determine `C`:
  - If `T1` and `T2` are the same type, `C` is that type.
  - Otherwise, if `T1` and `T2` are both signed integer types or are both unsigned integer types, `C` is the type with greater rank.
  - Otherwise, let `U` be the unsigned integer type and `S` be the signed integer type.
    - If `U`has rank greater than or equal to the rank of `S`, `C` is `U`.
    - Otherwise, if `S` can represent all of the values of `U`, `C` is `S`.
    - Otherwise, `C` is the unsigned integer type corresponding to `S`.

[2] If one operand is of enumeration type and the other operand is of a different enumeration type or a floating-point type, this behavior is deprecated (D.2 [depr.arith.conv.enum]).

C.1.X Clause 7: Expressions [diff.cpp23.expr]

Affected subclause: 7.4 [expr.arith.conv]

Change: Cannot compare nor perform arithmetic on enumerations with enumerations of a different type nor with a floating-point type.

Rationale: The old behavior was confusing, as it did not compare the contents of the two arrays but their addresses. Depending on context, this would either report whether the two arrays were the same object or have an unspecified result.

Effect on original feature: A valid C++ 2023 program directly comparing two enumeration objects of different type or an enumeration object with a floating-point object will be rejected as ill-formed in this International Standard. [*Example 1:*

```
enum E1 { e };
enum E2 { f };
bool b = e <= 3.7;      // ill-formed; previously well-formed
int k = f - e;          // ill-formed; previously well-formed
```

—*end example*]

C.5.3 Clause 7: expressions [diff.expr]

Affected subclause: 7.4 [expr.arith.conv]

Change: Cannot compare nor perform arithmetic on enumerations with enumerations of a different type nor with a floating-point type.

Rationale: Reinforcing type safety in C++, consistent with the three-way comparison operator.

Effect on original feature: Well-formed C code will not compile with C++26.

Difficulty of converting: Violations will be diagnosed by the C++ translator.

How widely used: Uncommon.

D.2 Arithmetic conversion on enumerations [depr.arith.conv.enum]

1 The ability to apply the usual arithmetic conversions (7.4 [expr.arith.conv]) on operands where one is of one enumeration type and the other is of a different enumeration type or a floating-point type is deprecated.

[*Note 1:* Three-way comparisons (7.6.8 [expr.spaceship]) between such operands are ill-formed. —*end note*]

[*Example 1:*

```
enum E1 { e };
enum E2 { f };
bool b = e <= 3.7;      // deprecated
int k = f - e;          // deprecated
auto cmp = e <=> f;     // error
```

—*end example*]

10 Acknowledgements

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

Thanks again to Matt Godbolt for maintaining Compiler Explorer, the best public resource for C++ compiler and library archaeology, especially when researching the history of deprecation warnings!

11 References

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

[P1120R0] Richard Smith. 2018-06-08. Consistency improvements for <=> and other comparison operators.
https://wg21.link/p1120r0

[P2139R2] Alisdair Meredith. 2020-07-15. Reviewing Deprecated Facilities of C++20 for C++23.
https://wg21.link/p2139r2