Remove Deprecated Arithmetic Conversion on Enumerations From C++26

Document #:	P2864R2
Date:	2023-11-10
Project:	Programming Language C++
Audience:	Core

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

Contents

1 Abstract

C++ has deprecated the implicit conversions of enumeration values in the *usual arithmetic conversions*, as the results are often misleading and easily used accidentally. This paper proposes removing those features from C++26.

2 Revision History

R2: December 2023 (post-Kona mailing)

— Recorded EWG review at the Kona meeting
— Forwarded to Core working group for C++26
— Reordered some sections for a better flow
— Updated analysis of deprecation warnings
— Rebased wording onto latest working draft, [N4964]
— Addressed removing the deprecated stable label
— Fixed the scrambled text for **Difficulty of converting:** from C in Annex C
— Core review
    — objects -> values
    — enumerations -> enumeration valuies
    — added example for ternary operator
    — removed use of `bool` in C compatibility annex

R1: August 2023 (mid-term mailing)

— Provisionally close work on this paper with no consensus to continue
— Corrected last alternative resolution to propose a partial undeprecation
— Revised Annex C wording after review by Jens Maurer
— Clarified some parts of the Analysis
— Recorded release dates of when compilers started warning on deprecation
— Assorted editorial cleanups suggested by Lori Hughes

R0: 2023 May (pre-Varna mailing)

— 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, [P2863R2], 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 Proposal

This paper proposes removing *both* kinds of deprecated conversions from C++26. Note that `enum` conversions and comparisons for `enum class` were never supported, but both changes would be a compatibility concern for code that compiles with both C and C++.

5 Rationale

With the introduction of the three-way comparison (a.k.a. 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.

6 Analysis

6.1 Scope of change

Certain arithmetic conversion on enumerations were deprecated for C++20 by [P1120R0] as part of the effort to make the new three-way comparison operator always type-safe. 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. Otherwise, a type-cast is required.

```
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 where the *usual arithmetic conversions* apply, such as in arithmetic expressions and comparisons; in all other contexts, such conversions are already ill-formed.

6.2 Implementation experience

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

```
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:

Compiler	First deprecated	Release Date
Clang	10	2020/03/24
GCC	11.1	2021/04/27
MSVC	19.22 (`/W4` for differing enums)	2019/09/10
EDG/nvc++	22.7 (first available on Godbolt	2022/07

Most of these compilers do not require any flags to observe the deprecation warning, but MSVC requires `/W4` for the warning on different enum types; it does warn on comparing floating point values with enums by default.

7 Alternative Proposals

7.1 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.

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

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

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

7.3 Remove floating-point comparison and undeprecate `enum` comparisons

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

We note that earlier polling leaned more strongly against undeprecation than 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.

8 C++23 Feedback

For information the review of this proposal for C++23 is included, which recommended a partial removal before the author ran out of time to update the single large paper containing all deprecation reviews.

8.1 EWG review: Telecon 2020/06/09

Concerns were raised about ongoing C compatibility. Any recommendation on removing deprecated support in D.2 [depr.arith.conv.enum] will be forwarded to our WG14 liaison in hopes of having 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 supposedly simple forced promotion through unary `operator+` is too clever or 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 gathering 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.

8.1.1 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.

9 C++26 Feedback

Here we record a summary of all reviews of this proposal for C++26.

9.1 EWG: 2023-06-13, Varna

Concerns raised about backward compatibility, especially in the case of SFINAE behavior that does not see a deprecation warning. Would like to see more experience before removal, but not clear to the author how to achieve that.

Poll

Forward D2864R1 (Remove Deprecated Arithmetic Conversion on Enumerations From C++26) to CWG for inclusion in the Working Draft for C++26.

```
SF  F  N  A SA
 3  4 11  1  0
```

No consensus to move forward.

There are no desire in the eroom to poll undeprecation.

Author notes that the strong majority are neutral, which suggests this paper needs to present a stronger motivation to make the change.

That completes the work on this paper for C++29, unless the author comes back with a stronger presentation to motivate making the change.

9.2 EWG: 2023-11-07, Kona

Re-reviewed with some representatives of SG22 in the room, able to raise concerns from the C interoperation perspective.

Took a straw poll and forwarded the paper to CWG for C++26, with a strong consensus and no votes against.

10 Proposed Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N4964], the latest draft at the time of writing.

10.1 Amend core wording

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:

(1.1) — The lvalue-to-rvalue conversion (7.3.2 [conv.lval]) is applied to each operand and the resulting prvalues are used in place of the original operands for the remainder of this section.

(1.2) — 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.

(1.n) — 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.

(1.3) — Otherwise, if either operand is of floating-point type, the following rules are applied:
(1.3.1) — If both operands have the same type, no further conversion is needed.
(1.3.2) — 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.
(1.3.3) — 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.
(1.3.4) — 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.
(1.3.5) — Otherwise, the expression is ill-formed.

(1.4) — 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`:
(1.4.1) — If `T1` and `T2` are the same type, `C` is that type.
(1.4.2) — Otherwise, if `T1` and `T2` are both signed integer types or are both unsigned integer types, `C` is the type with greater rank.
(1.4.3) — Otherwise, let `U` be the unsigned integer type and `S` be the signed integer type.
(1.4.3.1) — If `U`has rank greater than or equal to the rank of `S`, `C` is `U`.
(1.4.3.2) — Otherwise, if `S` can represent all of the values of `U`, `C` is `S`.
(1.4.3.3) — 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]).

10.2 Add entries to Annex C

[diff.cpp23.expr] Clause 7: Expressions

x **Affected subclause:** 7.4 [expr.arith.conv]

Change: Operations mixing a value of an enumeration type and a value of a different enumeration type or of a floating-point type are no longer valid.

Rationale: Reinforcing type safety.

Effect on original feature: A valid C++ 2023 program that performs operations mixing a value of an enumeration type and a value of a different enumeration type or of a floating-point type is ill-formed. For example:

```
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
auto x = true ? e : f;  // ill-formed; previously well-formed
```

1 **Affected subclause:** 9.4.5 [dcl.init.list]

Change: Pointer comparisons between `initializer_list` objects' backing arrays are unspecified.

Rationale: Permit the implementation to store backing arrays in static read-only memory.

Effect on original feature: Valid C++ 2023 code that relies on the result of pointer comparison between backing arrays may change behavior. For example:

```
  bool ne(std::initializer_list<int> a, std::initializer_list<int> b) {
    return a.begin() != b.begin() + 1;
}
bool b = ne({2,3}, {1,2,3}); // unspecified result; previously false
```

C.7.4 [diff.expr] Clause 7: expressions

1 **Affected subclause:** 7.3.12 [conv.ptr]

Change: Converting `void*` to a pointer-to-object type requires casting.

```
  char a[10];
  void* b=a;
  void foo() {
    char* c=b;
 }
```

ISO C accepts this usage of pointer to `void` being assigned to a pointer to object type. C++ does not.

Rationale: C++ tries harder than C to enforce compile-time type safety.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Can be automated. Violations will be diagnosed by the C++ translator. The fix is to add a cast. For example:

```
  char* c = (char*) b;
```

How widely used: This is fairly widely used but it is good programming practice to add the cast when assigning pointer-to-void to pointer-to-object. Some ISO C translators will give a warning if the cast is not used.

y **Affected subclause:** 7.4 [expr.arith.conv]

Change: Operations mixing a value of an enumeration type and a value of a different enumeration type or of a floating-point type are not valid. For example:

```
enum E1 { e };
enum E2 { f };
int b = e <= 3.7;       // valid in C; ill-formed in C++
int k = f - e;          // valid in C; ill-formed in C++
int x = 1 ? e : f;      // valid in C; ill-formed in C++
```

Rationale: Reinforcing type safety in C++.

Effect on original feature: Well-formed C code will not compile with this International Standard.

Difficulty of converting: Violations will be diagnosed by the C++ translator. The original behavior can be restored with a cast or integral promotion. For example:

```
enum E1 { e };
enum E2 { f };
int b = (int)e <= 3.7;
int k = +f - e;
```

How widely used: Uncommon.

2 **Affected subclauses:** 7.6.1.6 [expr.post.incr] and 7.6.2.3 [expr.pre.incr]

Change: Decrement operator is not allowed with `bool` operand.

Rationale: Feature with surprising semantics.

Effect on original feature: A valid ISO C expression utilizing the decrement operator on a `bool` lvalue (for instance, via the C typedef in `<stdbool.h>` (17.14.5 [stdbool.h.syn])) is ill-formed in C++.

10.3 Strike wording from Annex D

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

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*]

10.4 Update cross-reference for stable labels for C++23

Cross-references from ISO C++ 2023

All clause and subclause labels from ISO C++ 2023 (ISO/IEC 14882:2023, *Programming Languages — C++*) are present in this document, with the exceptions described below.

container.gen.reqmts *see*
container.requirements.general

depr.arith.conv.enum *removed*
depr.res.on.required *removed*

11 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!

Thanks to Jens Maurer for the initial wording review and corrections. Thanks to Lori Hughes for reviewing this paper and providing editorial feedback.

12 References

[N4964] Thomas Köppe. 2023-10-15. Working Draft, Programming Languages — C++.
https://wg21.link/n4964

[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

[P2863R2] Alisdair Meredith. 2023-10-15. Review Annex D for C++26.
https://wg21.link/p2863r2