Remove Deprecated Array Comparisons from C++26

Document #:	P2865R4
Date:	2023-11-11
Project:	Programming Language C++
Audience:	SG22 C interoperability

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

Contents

1 Abstract

Comparison operators applied to arrays perform that comparison on the results of array-to-pointer decay, i.e., on the address, not the contents, of each array. That behavior was deprecated by C++20, and this paper proposes making such comparison ill-formed in C++26.

2 Revision History

R4: December 2023 (post-Kona mailing)

- Confirmed wording against latest working draft, [N4964]
- Track committee progress: CWG -> EWG -> SG22
- Added preliminary analysis for SG22 (C compatibility)
- Cleaned up formatting and pagination
- Updated deprecation warning archaeology
- Wording updates
  - Removed [*Example N:* —*end example*] wrappers around Annex C examples

R3: September 2023 (midterm mailing)

- Track committee progress: EWG -> CWG
- Removed revision history's redundant subsection numbering
- Added analysis for equality comparison with a null pointer constant
- Fixed assorted typos and grammar issues
- Updated wording following CWG review
  - Allow nondeprecated equality comparison with null pointer constants
  - Correct use of present tense in Annex C when comparing with the C language
  - Move code example to "Difficulty of converting" in Annex C
- Additional wording changes
  - Rebased wording onto the latest working draft, N4958
  - Changed `bool` to `int` for the C language comparison in Annex C
  - Updated stable label cross-reference to C++23
- Addressed a multitude of editorial issues to improve the clarity of the paper. Thanks, Lori!

R2: August 2023 (midterm mailing)

- Track committee progress: EWG -> CWG -> EWG
- Noted initial CWG review
- Wording fixes
  - "array-to-pointer conversions" -> "the array-to-pointer conversion" in two places
  - Use present tense consistently in Annex C wording
  - Remove superfluous and inexact sentence in the C.7.4 [diff.expr], **How widely used**

R1: June 2023 (Varna meeting)

- Added SG22 to the target audience
- Updated analysis to show GCC warns since v12.1, with `-Wall`
- Confirmed wording is valid with latest working draft, N4950
- Fixed a wording detail describing operands and array-to-pointer decay
- Addressed issues with Annex C following wordsmith preview by Jens Maurer

R0: May 2023 (pre-Varna)

This is the initial draft of the paper, inspired by [P2139R2] and based on the C++ Standard working draft, N4944. Several notable changes have been made since the original C++23 proposal.

- Added the concern for relational comparison, rather than just equality tests
- Overhauled Core wording, initially based on working draft N4928
- Retested compilers for deprecation warnings

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, [P2863], 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 comparison operators for array types, D.4 [depr.array.comp].

4 Analysis

4.1 History

Comparison of array types was deprecated by C++20 as part of the effort to make the new spaceship operator do the right thing, adopted by paper [P1120R0]. It potentially impacts code written against C++98 and later Standards.

The deprecated comparison operator for arrays was less a deliberately designed feature than an accidental oversight that array-to-pointer decay would kick in. Hence, for equality comparison, we test whether two arrays are literally the same array at the same address, rather than whether two distinct arrays have the same contents. Identity tests are more typically performed by explicitly taking the address of the objects we wish to validate; it would be highly unusual to rely on an implicit pointer decay to intentionally perform that task. Implicit decay offers no efficiency gain over explicitly taking the address of the array, and relying on this idiom to compare two array addresses would fool a large number of subsequent code readers and reviewers who are unfamiliar with this idiom. We do note that function comparison performs exactly the same decay, and users are not surprised that comparing functions is an identity test.

The situation is worse for greater or less than comparisons, where the result is often unspecified. The only cases where the result *is* specified are

— both arrays are the same object (identity test)
— both arrays are data members of the same object
— both arrays are elements of the same array of arrays

In all other cases, the result is unspecified, as per pointer comparisons, although the result may be defined for specific ABIs. Given the likelihood that any usage of these comparison operators is a bug waiting to be detected, this behavior could be a real concern for software reliability. Thus, we recommend removing this feature from C++26.

Take note when drafting and reviewing the wording that the comparison of an array with a pointer value was never deprecated and is certainly not deprecated in C++23. For practitioners of C++ to compare arrays to pointers, anticipating the implicit array-to-pointer decay, is reasonably idiomatic. Hence, that comparison behavior should be preserved and indeed is the same behavior as the spaceship operator given the same arguments.

4.2 Equality comparison with a null pointer constant

When the CWG performed its initial review in Varna (2023-06-16), it noted that the proposed wording also excludes equality comparison between an array and a null pointer constant (`nullptr` or a literal `0`). Null pointer constants are *not* pointer types and are clearly not arrays either, so such comparisons have not yet been deprecated and should be preserved. Comparing an array and a null pointer constant with a relational operator (`<`,`>`,`<=`,`>=`) has been ill-formed since C++98, so the drafting issue relates to only the equality comparison operators, `==` and `!=`. Jens Maurer pointed out that the initial lvalue-to-rvalue conversion should address any latent concerns about lvalues of type `std::nullptr_t` becoming a third category to worry about.

The author and EWG chair confirm that the issues reported by the CWG are genuinely wording issues, as the design approved by the EWG did not include such a change of semantics; a separate follow-up paper will look into the potential for deprecation and removal of the null pointer constant comparisons with arrays.

4.3 Overlap with ISO C

The proposal has been forwarded to SG22 to consider the impact on interoperability between ISO C and ISO C++ and whether the C community, through WG14, would be interested in adopting a similar proposal.

The author believes that the overlap for language interoperability is limited to the impact on shared header files, where the affected operators can occur only inside the definition of inline functions.

There has been no exploration of widespread C practice as the November 2023 revision of this paper is being submitted.

4.4 Deployment experience

To test the current status of this deprecated feature, the following test program was used across a range of compilers and their versions available through Godbolt Compiler Explorer, invoked with the necessary switch to access C++20 features:

```
int main() {
   int a[5] = {};
   int b[5] = {};

   if (a == b) {
      return 1;
   }

   if (a > b) {
      return 2;
   }
}
```

Compiler	First deprecated	Release date
Clang	2.8, deprecated 9.0	2019/09/19
GCC	12.1 (with `-Wall`)	2022/05/06
MSVC	19.22	2019/09/10
EDG/nvc++	(no warnings)	

Note that GCC requires the `-Wall` command line switch, while none of the other compilers need any special warning flags to warn about the deprecation, as long as the language dialect supports C++20.

The Clang compiler has been diagnosing warnings in these cases as a regular QoI warning since Clang 2.8 and specifically with the deprecated notice since Clang 9.

The Clang and GCC compilers produced a program that returned 2, whereas the Microsoft and Intel compilers produced a program that returned 0, demonstrating real-world differences for the unspecified return value of the comparison.

5 Historical Feedback From C++23

5.1 LEWG: 2020-06-09 telecon

The review noted that comparison of pointer values is often unspecified, rather than well defined, unless both arrays happen to be members of the same class or are both elements of the same multi-dimensional array. The group generally agreed that this proposal offers a good opportunity to remove from the language a potential danger that offers little benefit, even when correctly used as intended.

A concern arose about ongoing C compatibility, which should be forwarded to our WG14 liaison (now SG22), to ideally have a coordinated process for removing this feature.

Poll:

Q: In C++23, remove the deprecated use of array comparisons?

SF	F	N	A	SA
9	9	3	0	0

Consensus: Follow up with this direction.

6 C++26 Feedback

6.1 EWG: 2023-06-13 Varna

No concerns were raised during the review.

Poll:

Q: Forward D2865R1 (Remove Deprecated Array Comparisons from C++26) to CWG for inclusion in the working draft for C++26.

SF	F	N	A	SA
8	16	1	0	0

6.2 CWG: 2023-06-16 Varna

The CWG raised concern that the wording in P2865R1 makes array comparison with a null pointer ill-formed, even though that was not previously deprecated. The proposal was sent back to EWG to confirm its intent.

6.3 EWG/SG22: 2023-11-06 Kona

EWG is content to pass this paper to Core, as amended to retain the comparison with null pointer literals, but would like SG22 (C compatibility) to review and (hopefully) sign-off first.

Dispatched to SG22 for further processing.

7 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.

7.1 Wording intent

The key change is to perform array-to-pointer conversion only if the other operand is either a pointer or a null pointer constant (a literal `0` or `nullptr`). This change makes the distinction between operand and converted operand more important to call out, particularly for the case of pointer types since none of the conversions will change the type category of an operand unless they change it to a pointer type (array-to-pointer conversion and function-to-pointer conversion).

In both subclauses below, the second paragraph has a list of types that the *converted* operand *shall* be, and neither subclause supports array types at that point. Thus, the comparison of arrays with anything but pointers is an error, just as any comparison with `void` is an error.

7.2 Core wording changes

7.6.9 [expr.rel] Relational operators

1 The relational operators group left-to-right.

[*Example 1:* `a<b<c` means `(a<b)<c` and *not* `(a<b)&&(b<c)`. —*end example*]

relational-expression :
compare-expression
relational-expression `<` *compare-expression*
relational-expression `>` *compare-expression*
relational-expression `<=` *compare-expression*
relational-expression `>=` *compare-expression*

The lvalue-to-rvalue (7.3.2 [conv.lval])~~, array-to-pointer (7.3.3 [conv.array]),~~ and function-to-pointer (7.3.4 [conv.func]) standard conversions are performed on the operands. If one of the operands is a pointer, the array-to-pointer conversion (7.3.3 [conv.array]) is performed on the other operand. ~~The comparison is deprecated if both operands were of array type prior to these conversions (D.4 [depr.array.comp]).~~

2 The converted operands shall have arithmetic, enumeration, or pointer type. The operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) all yield `false` or `true`. The type of the result is `bool`.

3 The usual arithmetic conversions (7.4 [expr.arith.conv]) are performed on operands of arithmetic or enumeration type. If both converted operands are pointers, pointer conversions (7.3.12 [conv.ptr]) and qualification conversions (7.3.6 [conv.qual]) are performed to bring them to their composite pointer type (7.2.2 [expr.type]). After conversions, the operands shall have the same type.

4 The result of comparing unequal pointers to objects …

7.6.10 [expr.eq] Equality operators

equality-expression :
relational-expression
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

1 The `==` (equal to) and the `!=` (not equal to) operators group left-to-right. The lvalue-to-rvalue (7.3.2 [conv.lval])~~,~~ ~~array-to-pointer (7.3.3 [conv.array]),~~ and function-to-pointer (7.3.4 [conv.func]) standard conversions are performed on the operands. If one of the operands is a pointer or a null pointer constant (7.3.12 [conv.ptr]), the

array-to-pointer conversion (7.3.3 [conv.array]) is performed on the other operand. ~~The comparison is deprecated if both operands were of array type prior to these conversions (D.4 [depr.array.comp]).~~

2 The converted operands shall have arithmetic, enumeration, pointer, or pointer-to-member type, or type `std::nullptr_t`. The operators `==` and `!=` both yield `true` or `false`, i.e., a result of type `bool`. In each case below, the operands shall have the same type after the specified conversions have been applied.

3 If at least one of the converted operands is a pointer, pointer conversions (7.3.12 [conv.ptr]), function pointer conversions (7.3.14 [conv.fctptr]), and qualification conversions (7.3.6 [conv.qual]) are performed on both operands to bring them to their composite pointer type (7.2.2 [expr.type]). Comparing pointers is defined as follows:

- If one pointer represents the address of a complete object, and another pointer represents the address one past the last element of a different complete object, the result of the comparison is unspecified.
- Otherwise, if the pointers are both null, both point to the same function, or both represent the same address (6.8.4 [basic.compound]), they compare equal.
- Otherwise, the pointers compare unequal.

4 If at least one of the operands is a pointer to member, …

7.3 Add new sections to Annex C

C.1.2 [diff.cpp23.expr] Clause 7: Expressions

x **Affected subclause:** 7.6.9 [expr.rel] and 7.6.10 [expr.eq]

Change: Comparing two objects of array type is no longer valid.

Rationale: The old behavior was confusing since it compared not the contents of the two arrays, but their addresses. Depending on context, this comparison 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 array objects is rejected as ill-formed in this International Standard. For example:

```
int arr1[5];
int arr2[5];
bool same = arr1 ==  arr2;       // ill-formed; previously well-formed
bool idem = arr1 == +arr2;       // compare addresses
bool less = arr1 <  +arr2;       // compare addresses, unspecified result
```

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

…

C.7 [diff.iso] C++ and ISO C

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

3 **Affected subclauses:** 7.6.2.5 [expr.sizeof] and 7.6.3 [expr.cast]

…

x **Affected subclauses:** 7.6.9 [expr.rel] and 7.6.10 [expr.eq]

Change: C allows directly comparing two objects of array type; C++ does not.

Rationale: The behavior is confusing because it compares not the contents of the two arrays, but their addresses. Depending on context, this comparison would either report whether the two arrays were the same object or have an unspecified result.

Effect on original feature: Deletion of semantically well-defined feature that had unspecified behavior in common use cases.

Difficulty of converting: Violations will be diagnosed by the C++ translator. The original behavior can be replicated by explicitly casting either array to a pointer, possibly with unary `+` to force a promotion. For example:

```
int arr1[5];
int arr2[5];
int same = arr1 ==  arr2;       // valid C, ill-formed C++
int idem = arr1 == +arr2;       // valid in both C and C++
```

How widely used: Rare.

4 **Affected subclauses:** 7.6.16 [expr.cond], 7.6.19 [expr.ass], and 7.6.20 [expr.comma]

…

7.4 Strike wording from Annex D

D.4 [depr.array.comp] Array comparisons

1 Equality and relational comparisons (7.6.10 [expr.eq], 7.6.9 [expr.rel]) between two operands of array type are deprecated.

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

[*Example 1:*

```
int arr1[5];
int arr2[5];
bool same = arr1 == arr2;    // deprecated, same as &arr1[0] == &arr2[0],
                             // does not compare array contents
auto cmp = arr1 <=> arr2;    // error
```

—*end example*]

7.5 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.array.comp *removed*
depr.res.on.required *removed*

8 Acknowledgements

Thanks to Michael Park 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.

9 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

[P2863] Alisdair Meredith. 2023-12-15. Review Annex D for C++26.
https://wg21.link/p2863