Remove Deprecated Array Comparisons from C++26

Document #:	D2865R1
Date:	2023-06-12
Project:	Programming Language C++
Audience:	Evolution, 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

2.1 R0: June 2023 (Varna meeting)

— Added SC22 to the target audience
— Updated analysis to show gcc warns since v12.1, with `-Wall`
— Confirmed wording is valid with latest working draft, [N4950]
— Fix Wording nit describing operands and array-to-pointer decay

— Addressed issues with Annex C following wordsmith preview by Jens Maurer

2.2 R0: May 2023 (pre-Varna)

Initial draft of this paper, based on [N4944].

Notable changes since [P2139R2]

— Added the concern for relational comparison, rather than just equality tests
— Overhauled Core wording, based on [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, [P2863R1], 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

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 on code written against C++98 and later standards.

The deprecated comparison operator for arrays was not so much a deliberately designed feature, but accidental oversight that array-to-pointer decay would kick in. Hence, or 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. Relying on implicit decay offers no efficiency gain over explicitly taking the address of the array, but would fool a large number of subsequent code readers and reviewers who are not familiar 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 may be defined for specific ABIs.

Given the likelihood that any usage of these comparison operators is a bug waiting to be detected, this could be a real concern for software reliability. Thus, we recommend removing this feature from C++26.

However, note for wording, that the comparison of an array with a pointer value was not deprecated in C++20, is reasonably idiomatic for practitioners of C++, and is intended to continue to be supported. This mimics the behavior of the spaceship operator.

4.1 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	10	2020/03/24
GCC	12.1 (with -Wall)	2022/05/06
MSVC	19.22 (floating point only)	2019/09/10
EDG/Intel	(No C++20 support on Godbolt)	

Note that GCC requires the `-Wall` command line switch, while the other compilers did not need any special warnings flags.

The Clang compiler has been diagnosing warnings in these cases since Clang 3.0 as a regular QoI warning, and with the deprecated notice since Clang 10.

Using the classic Intel compiler as a proxy for the EDG front ends, there was no C++20 support as of the last release of that compiler, icc 2021.7.1.

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 C++23 Feedback

There was only the initial LEWG review at the 2020/06/09 telecon.

It was 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.

General agreement that this is a good opportunity to remove a landmine from the language that offers little benefit, even when correctly used as intended.

A concern was raised about ongoing C compatibility, which should be forwarded to our WG14 liaison (now sg22), to hopefully have a co-ordinated process for removing this feature.

Polls:

Q1: 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

Pending

7 Proposed Wording

All changes are relative to [N4950].

7.1 Wording Intent

The key change is to perform array-to-pointer conversion only if the other operand is a pointer. This makes the distinction between operand and converted operand more important to call out, but mostly for the case of pointer types as none of the conversions will change the type category of an operand unless it is 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, 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, array-to-pointer conversions (7.3.3 [conv.array]) are 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, array-to-pointer conversions (7.3.3 [conv.array]) are 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.X Clause 7: Expressions [diff.cpp23.expr]

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, as it did not compare the contents of the two arrays, but compared 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 array objects is rejected as ill-formed in this International Standard. [*Example 1:*

```
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
```

—*end example*]

[diff.expr] Clause 7: expressions

Affected subclause: 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 was confusing, as it did not compare the contents of the two arrays, but compared 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: Deletion of feature that had unspecified behavior in common use cases.

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

How widely used: Rare. In the cases where the result is well defined, it is reporting whether both arguments are the same object, using the same name. The original behavior can be replicated by explicitly casting either array to a pointer, possibly with unary + to force a promotion.

[*Example 1:*

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

—*end example*]

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*]

8 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.

9 References

[N4928] Thomas Köppe. 2022-12-18. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4928

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

[N4950] Thomas Köppe. 2023-05-10. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4950

[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

[P2863R1] Alisdair Meredith. 2023-05-15. Review Annex D for C++26.
https://wg21.link/d2863r1