An Extensible Approach to Obtaining Selected Operators

Document #: WG21 P0436R1
Date: 2016-11-09
Revises: P0436R0 (see §10)
Project: JTC1.22.32 Programming Language C++
Audience: EWG ⇒ CWG
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

In light of WG21's recent rejection of [P0221R2], which proposed default-generated comparison operators, this paper investigates and proposes *operator reinterpretation* as a new but slightly less ambitious approach to the subject. Addressing National Body comments US 5 and RU 5, this proposal is straightforward to specify, offers both *opt-in* and *opt-out* features, has no immediate impact on the Standard Library, is fully backwards-compatible with existing well-formed ordinary user code, eliminates the need for certain boilerplate code, and can in the future be extended to selected other (non-comparison) operators.

If you can write **x < y**, *you also want* **x > y**, **x >= y**, *and* **x <= y**.

— Dave Abrahams *and* Jeremy Siek

The more I thought about this the more I realized our society is obsessed with comparisons.

— Stephanie Hester

1 Introduction

At WG21's Oulu meeting (2016-06), attendees declined to adopt default-generated comparison operators as proposed by [P0221R2]. Many consider this an unfortunate outcome, as considerable committee resources had been expended in developing and refining the proposal to reach that final form; the Bibliography (§9) lists recent (and even some not-so-recent) WG21 papers on the topic.

From informal discussions with a number of the Oulu WG21 participants, it seems clear that several parts of the proposal were considered both desirable and relatively uncontroversial. The present paper proposes to adopt those (and only those) elements via a new approach, herein termed *operator reinterpretation*, that (a) seems to avoid most or all of the controversial issues that accompanied the recent efforts and also (b) provides opportunity for future extension to other operators.

In §2, we will review the underlying principles and prior art on which this proposal is based. The proposal itself is then presented in §3 followed by an analysis (§4) and examples (§5) of its impact on the Standard Library and on existing well-formed user code. We conclude with a discussion (§6) of possible future directions, followed in §7 by our proposed wording.

2 Principles and prior art

While the topic has been under discussion for quite some time,[1] the first recent paper on the subject of default-generated comparison operators appears to be [N3950]. Under the heading of "Correctness," its author argues:

> It is vital that equal/unequal, less/more-or-equals and more/less-or-equal pairs behave as boolean negations of each other. After all, the world would make no sense if both **`operator==()`** and **`operator!=()`** returned *false*! As such, it is common to implement these operators in terms of each other: [code omitted].

This position seems relatively uncontroversial,[2] as it is fully consistent with the generally-accepted concepts *EqualityComparable* and *LessThanComparable* as conceived by Alexander Stepanov[3] and as implemented throughout both today's Standard Library and the draft future **`concept`**-ified version thereof.

It is further a long-accepted *de facto* principle of the Standard Library that two of the six comparison operators, namely *equal-to* and *less-than*, are in some sense special:

- This can perhaps most obviously be seen in the Library's **`std::rel_ops`** namespace, where we find implementations of the other four comparison operators in terms of these special two.
- Moreover, quite a number of Standard Library algorithms are specified in pairs, one specified in terms of a notional **`operator==`** or **`operator<`** and the other specified in terms of a function object having equivalent effect; **`std::equal`** exemplifies the former (**`operator==`**) case, while **`std::sort`** exemplifies the latter (**`operator<`**) case.
- Finally, *Boost.Operators*[4] (which is one of the oldest Boost components) provides templates **`less_than_comparable<>`** and **`equality_comparable<>`** that inject the remaining comparison operators, defining them in terms of these special two. This same design is preserved by Daniel Frey's *The Art of C++/Operators* library,[5] a modernized (e.g., move-aware) rewrite of *Boost.Operators*.

3 Proposal

Unlike the several past attempts to have the compiler generate some or all of the comparison functions, **this paper proposes *operator reinterpretation***[6] as a new, yet backwards compatible,

[1] See, for example, the 1995 (!) paper [N0618].

[2] [N4367] and its successors ([P0100R0] and [P0100R1]) do briefly discuss alternative definitions of **`operator<=`** and **`operator>=`**. These alternatives, while perhaps providing mathematically superior characteristics, seem inconsistent with the long-established precedents enshrined in the Standard Library: a "consistent weak ordering" is, where needed, tacitly assumed throughout. Adoption of these alternatives seems to have introduced some of the concerns that led to [P0221R2]'s rejection.

[3] See the SGI STL web sites http://www.sgi.com/tech/stl/EqualityComparable.html and http://www.sgi.com/tech/stl/LessThanComparable.html.

[4] See http://www.boost.org/doc/libs/1_61_0/libs/utility/operators.htm.

[5] See https://github.com/taocpp/operators.

[6] The term *reinterpretation* was selected because [N4606] already uses the term *interpret* in connection with binary operations **`x @ y`**. As detailed below, we will propose herein that certain failed interpretations be given a second (hence *re-*) interpretation.

approach. While generating no functions, we nonetheless obtain the effect of having the remaining comparison operators by relying on the special ones only. There are therefore two parts to the proposal's details, one for each of the two special comparison operators to be relied on:

1. If no suitable **`operator!=`** is declared[7] for a use of the form **`x != y`**, such an expression is to be (re)interpreted as if (re)written **`!(x == y)`**, but only if that **`operator==`** is sane (i.e., exists and has return type **`bool`**). (Thus, only if no suitable **`operator==`** is declared, as well as no suitable **`operator!=`**, would the original expression yield an ill-formed program.)
2. Similarly, if no suitable operators are declared for uses of the forms **`x <= y`**, **`x > y`**, or **`x >= y`**, such an expression is to be (re)interpreted as if (re)written in terms of another operator, as shown in the following table, but only if that operator is sane as defined above.[8]

Expression	Reinterpretation
`x <= y`	**`!(x > y)`**
`x > y`	**`y < x`**
`x >= y`	**`y <= x`**[9]

It appears that the above approach to obtaining the functionality of (four of) the comparison operators has not received prior WG21 consideration via any of the papers listed in §9. We believe that this approach is viable and reasonably straightforward to specify and implement, removes the need for boilerplate code for these four operators, and completely avoids the known contentious issues that have to date doomed the previous approaches:

- The proposal is *opt-in*, in that an operand type must provide at least some sane (i.e., **`bool`**-returning) operators (typically **`operator==`** and/or **`operator<`**) before this proposal would have any potential effect on the type's interface.
- The proposal is also *opt-out*, in that an operand whose type already provides any or all of **`operator!=`**, **`operator<=`**, **`operator>`**, and/or **`operator>=`**. would use those provided operators in the same way as has been done since at least C++98.

Note that no existing well-formed ordinary[10] program would be affected by the above proposal, as such a program must already provide each operator that is used.[11]

4 Library impact

The proposed new reinterpretations have been carefully designed so as to have no net effect on the behavior of the Standard Library, since the Library already specifies all of each Library type's desired comparison operators.[12] However, this *status quo* may be considered overspecification

[7] More precisely, "no suitable operator is declared" when, during overload resolution, the set of viable functions ([over.match.viable]) is empty.

[8] The reinterpreted expression may itself be subject to additional reinterpretation if no suitable operator has been declared for it.

[9] This Reinterpretation was selected in order to accommodate a possible *opt-out* via a provided **`operator<=`** function. (See §5 for an example of this.) If there is no such function, then this reinterpreted code would be further reinterpreted, this time yielding an expression in terms of **`operator<`** as shown in the table's first row.

[10] We apply the term *ordinary* to describe programs that do not adjust their behavior after probing via the *detection idiom* ([N4502]) or an equivalent technique to determine the validity of a certain expression such as **`x > y`**. Some *extraordinary* programs, i.e., programs that do perform such inspection and self-adjustment, may have a change in behavior under the present proposal. This is because a type that has opted-in to this proposal by supplying **`operator<`** would now report that **`x > y`** is a valid expression even when a corresponding **`operator>`** was not supplied by the user.
Programs making self-adjustments in this manner appear to be exceedingly uncommon. Moreover, in the very few of these we have seen, the programs undertook such inspection in order to compensate for an operator's possible absence. Such compensation would, of course, be no longer needed for most comparison operators once this proposal is adopted.

[11] Stated differently but equivalently, an existing well-formed program must already avoid using any absent operator.

[12] The same analysis holds for all existing well-formed ordinary user code, thus rendering the proposal fully backwards-compatible with such code.

under the proposal.[13] Accordingly, the Library may in future wish to excise some or all of its explicit specifications of operators `!=`, `>`, `>=`, and `<=` and thereby implicitly opt-in to our new core language rule for equivalent behavior.

Independently of such possible simplifications in Library specification, Library implementors could remove their declarations of these functions as soon as the new rule is implemented in their compilers and see no change in the behavior of any existing well-formed program.

Finally, we note that the present proposal completely subsumes the functionality provided by `std::rel_ops`. Clause [operators] (20.2.1) is thus another candidate for future deprecation and excision, should this proposal be adopted.

5 Examples

5.1 `complex<>`

As our first example, consider `std::complex<>`, which has long specified `operator==` and `operator!=`, but no other comparison operators.

- Under the present proposal, the equality operator's specification would remain unchanged. Its presence is necessary to preserve current behavior.
- Under the present proposal, the inequality operator's existing specification becomes redundant, but its presence is not actively harmful. This specification can, at some future date, be removed (or not) at the pleasure of the Library Working Group and/or the Project Editor.
- Under the present proposal, and unlike previous proposals, `std::complex<>` does not suddenly acquire any additional comparison operators.

Thus, this family of types will, in all cases, retain its present behavior under the present proposal.

5.2 Bizarre comparison functions

Let's now consider a hypothetical user-provided type `U` whose comparison operators violate the usual assumptions in some way. Such behavior can arise only if `U` explicitly deletes or otherwise fully defines the corresponding functions. Therefore, `U` would be unaffected by the present proposal, as the presence of these bizarre functions constitutes an explicit opt-out for those operators.

5.3 I want "`x<=y`" to mean "`x<y or x==y`"

If that's the behavior[14] you want for your type, then just provide your type with an `operator<=` function that simply `returns x < y || x == y`. The existence of such a function will constitute an explicit opt-out from our proposal with respect to this `operator<=`. Moreover, our proposed reinterpretation of `operator>=` will use this function. (See our table's last row and corresponding footnote.)

5.4 Comparison defined after use

Suppose we have a user-provided type that provides a comparison operator, but does so only after an expression that uses that operator has already been reinterpreted as defined above. This proposal does not countenance such inconsistency, and provides wording (adapted from [P0221R2]; see §7) to treat any such program as ill-formed.

---

[13] These comparison operators seem to have been originally specified via [N0967R1] for the Library's then-existing types. At the time, it took six single-spaced pages (!) just to identify all these locations, and of course it today takes more than that (in mostly boilerplate specifications) to set forth the desired functionality.

[14] À la [P0100R1], for example.

5.5 `struct tm`

Finally, what about types that declare no comparison operator at all? (**`struct tm`** is a canonical example of such a type.) This proposal does not impact such types in any way. To opt-in, a type must provide, at minimum, **`operator==`**, **`operator<`**, or both. In their absence, this proposal's provisions are inapplicable.

6 Possible future directions

6.1 Generating `operator==`, etc.

Nothing in the present proposal stands in the way of potential future proposals for compiler-generated **`operator==`** and/or **`operator<`**. Even a future proposal for synthesizing **`operator<=`** (from operators **`<`** and **`==`**, as suggested in [P0100R1]) could be gracefully accommodated without impacting our proposed wording. If anything, any such future proposal would become somewhat simpler, as the remaining comparison operators would no longer be at issue.

6.2 Reinterpretations for operator families

If the present proposal were adopted, WG21 could in the future consider expanding the list of operators receiving similar treatment. For example, consider Sutter's formulations[15] of long-accepted coding guidance regarding C++ overloaded operators:

- "If you supply a standalone version of an operator (e.g., **`operator+`**), always supply an assignment version of the same operator (e.g., **`operator+=`**) and prefer implementing the former in terms of the latter."
- "For consistency, always implement postincrement in terms of preincrement, otherwise your users will get surprising (and often unpleasant) results."

These and similar recommendations for C++ programmers have been summarized[16] as "Always provide all out of a set of related operations." Such rules of thumb seem to provide excellent starting points for future WG21 deliberation once we obtain sufficient experience with the present proposal.

6.3 Reinterpretations for some iterator operators

Finally, in a recent posting,[17] Matthew Fioravante reacts (favorably) to a CppCon 2016 talk[18] that reimagines iterator interfaces. After summarizing the talk, Fioravante asks, "how can we make things better?" First among several possible approaches, he proposes to "Add more defaulted operators" as follows[19]:

> * If **`T::operator++()`** is defined and **`T`** is copyable, autogenerate **`T::operator++(int)`**.
> * If **`T::operator--()`** is defined and **`T`** is copyable, autogenerate **`T::operator--(int)`**.
> * If **`T::operator*`** is defined and returns an lvalue reference, autogenerate **`T::operator->()`**.
> * If **`T::operator* const`** is defined and returns an lvalue reference, autogenerate **`T::operator->() const`**.

[15] Herb Sutter: "GotW #4 Solution: Class Mechanics." 2013-05-20. https://herbsutter.com/2013/05/20/gotw-4-class-mechanics/.

[16] sbi [pseudonym]: "The Three Basic Rules of Operator Overloading in C++." 2010-12-12. Revised by Daniel Kamil Kozar, 2013-06-11. http://stackoverflow.com/questions/4421706/operator-overloading/4421708#4421708.

[17] Matthew Fioravante: "[std-proposals] The C++ Iterator API is terrible. How can we fix it?" 2016-10-10. https://groups.google.com/a/isocpp.org/d/msgid/std-proposals/4a6aa8f7-6db6-41af-b128-9848041382ff%40isocpp.org?utm_medium=email&utm_source=footer.

[18] Patrick Niedzielski: "From Zero to Iterators: Building and Extending the Iterator Hierarchy in a Modern, Multicore World." Presented at CppCon 2016, Bellevue, WA, USA. https://www.youtube.com/watch?v=N80hpts1SSk.

[19] For improved clarity, minor typos have been corrected and monospace fonts have been added.

  * If `T::operator+=(U)` is defined, autogenerate `operator+(T, U)`. (or vice-versa)
  * If `T::operator-=(U)` is defined, autogenerate `operator-(T, U)`. (or vice-versa)
  * If `T::operator+(T,U)` is defined and `T::operator*` is defined, autogenerate `T::operator[](U)` (I could see this being problematic)
  * If `T::operator+(T,U)` is defined and `T::operator* const` is defined, autogenerate `T::operator[](U) const` (I could see this being problematic)
  * If `operator==(T,U)` is defined, autogenerate `operator!=(T,U)`. (or vice-versa)
  * If `operator<(T,U)` is defined, autogenerate `operator>=(T,U)`. (or vice-versa)
  * If `operator>(T,U)` is defined, autogenerate `operator<=(T,U)`. (or vice-versa)
  * If `operator<(T,U)` and `operator==(T,U)` are defined, autogenerate `operator>(T,U)`
  * If `operator>(T,U)` and `operator==(T,U)` are defined, autogenerate `operator<(T,U)`

If "autogenerate" were replaced with "reinterpret" in the above, it seems clear that the present proposal could provide a reasonably straightforward means of achieving Fioravante's vision. At mimimum, it would be interesting to contrast this approach with the well-known "iterator facade" approach.[20]

6.4 Reinterpreting `operator==`

It is well known that `!(x < y) and !(y < x)` is typically equivalent to `x == y`. By appealing to this identity, `operator==` could itself become a candidate for reinterpretation based on `operator<`. While such action is not proposed herein, this decision may be reconsidered at a future date.

7 Proposed wording[21]

7.1 To provide appropriate context for our subsequent wording adjustments, we first reproduce the (sole) paragraph currently constituting subclause [over.binary] (13.5.2).[22]

1 A binary operator shall be implemented either by a non-static member function (9.2.1) with one parameter or by a non-member function with two parameters. Thus, for any binary operator `@`, `x @ y` can be interpreted as either `x.operator@(y)` or `operator@(x,y)`. If both forms of the operator function have been declared, the rules in 13.3.1.2 determine which, if any, interpretation is used.

7.2 Append a new paragraph and accompanying table to [over.binary] (13.5.2) as shown below. (The second sentence is adapted from similar wording in [P0221R2].)

2 If no viable functions ([over.match.viable]) are found during overload resolution, then for each binary operator `@` appearing in an Expression in Table *n*, `x @ y` shall, if the corresponding

[20] For example, see "Iterator Facade and Adaptor" in David Abrahams, Jeremy Siek, and Thomas Witt: "The Boost.Iterator Library." 2003. http://www.boost.org/doc/libs/1_62_0/libs/iterator/doc/index.html.

[21] All proposed additions and ~~deletions~~ are relative to the post-Oulu Working Draft [N4606]. Drafting and editorial notes are highlighted like this .

[22] In correspondence dated 2016-10-25, J. Maurer provided the following example, observing that "13.5.2 doesn't redirect you to 13.3.1.2 for this case, because you don't have both member and non-member operator functions declared, but it most certainly should."

```
struct C ;
bool operator==(C&, C&);
bool operator==(const C&, const C&);
bool b = C() == C();
```

The present proposal seems unaffected by this potential wording defect; we mention the issue here only because it was brought to our attention during a review of this paper.

Precondition is satisfied, be reinterpreted (treated as if originally written) as shown in the corresponding Reinterpretation. [*Note:* The Reinterpretation may itself be subject to a (further round of) reinterpretation if no viable functions are found during its overload resolution. — *end* note] If an expression is thus reinterpreted in a context whose nearest enclosing namespace is `N`, and an expression with the same operator and the same operand types in another context whose nearest enclosing namespace is also `N` is not thus reinterpreted, the program is ill-formed; no diagnostic is required if the two expressions appear in different translation units.

Table *n* — Reinterpretations of selected binary expressions [reinterpretations]

Expression	Precondition	Reinterpretation
`x != y`	`decltype(x == y)` is `bool`.	`!(x == y)`
`x <= y`	`decltype(x > y)` is `bool`.	`!(x > y)`
`x > y`	`decltype(y < x)` is `bool`.	`y < x`
`x >= y`	`decltype(y <= x)` is `bool`.	`y <= x`

7.3 For the purposes of SG10, a feature test macro named `__cpp_reinterpreted_operators` is recommended.

8 Acknowledgments

Many thanks to the readers of pre-publication drafts for their careful proofreading and thoughtful general comments. Your contributions in materially improving this paper is greatly and gratefully appreciated.

Special thanks to Jens Maurer, who provided several particularly useful examples in support of his suggestions for improved wording.

9 Bibliography

[N0618] Nathan Myers: "Removing STL Global Operators `!= > <= >=`." ISO/IEC JTC1/SC22/WG21 document N0618 (pre-Austin mailing), 1995-01-26. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1995/N0618.asc.

[N0967R1] Randy Smithey: "Relational Operators for Standard Library Classes." ISO/IEC JTC1/SC22/WG21 document N0967R1 (post-Stockholm mailing), 1996-07-11. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1996/N0967R1.pdf.

[N3950] Oleg Smolsky: "Defaulted comparison operators," ISO/IEC JTC1/SC22/WG21 document N3950 (post-Issaquah mailing), 2014-02-19. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3950.html.

[N4114] Oleg Smolsky: "Defaulted comparison operators," ISO/IEC JTC1/SC22/WG21 document N4114 (post-Rappersville mailing), 2014-07-02. Revises [N3950]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4114.htm.

[N4126] Oleg Smolsky: "Explicitly defaulted comparison operators," ISO/IEC JTC1/SC22/WG21 document N4126 (pre-Urbana mailing), 2014-07-29. Revises [N4114]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4126.htm.

[N4175] Bjarne Stroustrup: "Default comparisons," ISO/IEC JTC1/SC22/WG21 document N4175 (pre-Urbana mailing), 2014-10-11. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4175.pdf.

[N4176] Bjarne Stroustrup: "Thoughts about Comparisons," ISO/IEC JTC1/SC22/WG21 document N4176 (pre-Urbana mailing), 2014-10-11. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4176.pdf.

[N4239] Andrew Tomazos, Michael Spertus: "Defaulted Comparison Using Reflection," ISO/IEC JTC1/SC22/WG21 document N4239 (pre-Urbana mailing), 2014-10-12. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4239.pdf.

[N4367] Lawrence Crowl: "Comparison in C++," ISO/IEC JTC1/SC22/WG21 document N4367 (mid-Urbana-Lenexa mailing), 2015-02-08. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4367.html.

[N4401] Michael Price: "Defaulted comparison operator semantics should be uniform," ISO/IEC JTC1/SC22/WG21 document N4401 (pre-Lenexa mailing), 2015-04-07. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4401.html.

[N4436] Walter E. Brown: "Proposing Standard Library Support for the C++ Detection Idiom." ISO/IEC JTC1/SC22/WG21 document N4436 (pre-Lenexa mailing), 2015-04-09. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4436.pdf.

[N4475] Bjarne Stroustrup: "Default comparisons (R2)," ISO/IEC JTC1/SC22/WG21 document N4475 (pre-Lenexa mailing), 2015-04-09. Revises [N4175]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4475.pdf.

[N4476] Bjarne Stroustrup: "Thoughts about Comparisons (R2)," ISO/IEC JTC1/SC22/WG21 document N4476 (pre-Lenexa mailing), 2015-04-09. Revises [N4176]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4476.pdf.

[N4502] Walter E. Brown: "Proposing Standard Library Support for the C++ Detection Idiom, v2." ISO/IEC JTC1/SC22/WG21 document N4502 (post-Lenexa mailing), 2015-05-03. Revises [N4436]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4502.pdf.

[N4532] Jens Maurer: "Proposed wording for default comparisons," ISO/IEC JTC1/SC22/WG21 document N4532 (post-Lenexa mailing), 2015-05-22. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4532.html.

[N4606] Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4606 (post-Oulu mailing), 2016-07-12. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4606.pdf.
Same content as "C++17 CD Ballot Document," ISO/IEC JTC1/SC22/WG21 document N4604 (post-Oulu mailing), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4604.pdf.

[P0100R0] Lawrence Crowl: "Comparison in C++," ISO/IEC JTC1/SC22/WG21 document P0100R0 (pre-Kona mailing), 2015-09-27. Revises [N4367]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0100r0.html.

[P0100R1] Lawrence Crowl: "Comparison in C++," ISO/IEC JTC1/SC22/WG21 document P0100R1 (post-Kona mailing), 2015-11-07. Revises [P0100R0]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0100r1.html.

[P0221R0] Jens Maurer: "Proposed wording for default comparisons, revision 2," ISO/IEC JTC1/SC22/WG21 document P0221R0 (pre-Jacksonville mailing), 2016-02-10. Revises [N4532]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0221r0.html.

[P0221R1] Jens Maurer: "Proposed wording for default comparisons, revision 3," ISO/IEC JTC1/SC22/WG21 document P0221R1 (post-Jacksonville mailing), 2016-03-17. Revises [P0221R0]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0221r1.html.

[P0221R2] Jens Maurer: "Proposed wording for default comparisons, revision 4," ISO/IEC JTC1/SC22/WG21 document P0221R2 (post-Oulu mailing), 2016-06-23. Revises [P0221R1]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0221r2.html.

10 Document history

Rev	Date	Changes
0	2016-10-10	• Published as P0436R0.
1	2016-11-09	• Mentioned NB comments in the Abstract. • Reworked `operator>=` reinterpretation in §3 (table entry and new footnote), §5 (new subsection), and §7 (table entry). • Added new footnote in §3 and improved wording in §7 to clarify and formalize the intent of "no suitable operator is defined". • Added new footnote in §3 and new Note in §7 re a possible second round of reinterpretation. • Noted in §6 (new subsection) that `operator==` could in future be reinterpreted. • Added footnote in §7 re possible wording issue in [over.binary]/1. • Shortened the recommended SG10 macro name in §7. • Credited J. Maurer in §8. • Corrected some bibliography links in §9. • Tweaked vocabulary, grammar, and punctuation throughout. • Published as P0436R1.