Chained comparisons: Safe, correct, efficient

Document Number: **P3439 R2**
Date: 2025-05-03
Reply-to: Herb Sutter (herb.sutter@gmail.com)
Audience: EWG

Contents

Abstract

This paper proposes that we adopt Barry Revzin's [P0893R1] (based on my [P0515R0] section 3.3) with refined proposed rules that may address previous concerns, but for now only make the problematic chains deprecated or ill-formed to allow for a transition period.

Updates in this revision

R2: Per the last poll in EWG Hagenberg, this revision changes the proposal to **for now only deprecate or make ill-formed the chains that are already problematic and whose meaning would change** in the future if the rest of this proposal were someday adopted. Thanks to Corentin Jabot and other Clang implementers for implementing part of this R2 proposal as an error by default (Compiler Explorer example), and to Jan Schultke for giving feedback on wording.

R1: Added **proposed wording**. Added **fold-expressions** support, following EWG Wrocław direction. Simplified to '**the entire rewritten expression** is valid and contextually convertible to `bool`' as the trigger rule, rather than 'each individual binary comparison.' Thanks to Gašper Ažman for this suggestion over the break, and the observation in Q&A 4.4. Added **section 4 "Q&A"** covering: that previous implementability concerns have been addressed; that user-defined mathematical types Just Work; whether to have a deprecation period; and additional benefits discovered since R0.

1 Background and motivation: Why consider this again now?

1.1 Overview

Today, comparison chains like `min <= index_expression < max` are valid code that do the wrong thing; for example, `0 <= 100 < 10` means `true < 10` which means `true`, certainly a bug. Yet that is exactly a natural bounds check; for indexes, such subscript chains' current meaning is **always** a potentially exploitable out-of-bounds violation.

[P0893R1] reported that code searches performed by Barry Revzin with Nicolas Lesser and Titus Winters found:

- Lots of instances of such bugs in the wild: in real-world code "of the `assert(0 <= ratio <= 1.0);` variety," and "in questions on StackOverflow [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]."
- "**A few thousand** instances over just a few code bases" (**emphasis** original) where programmers today write more-brittle and less-efficient long forms such as `min <= index_expression && index_expression < max` because they must, but with multiple evaluation of `index_expression` and with bugs because of having to write the boilerplate and sometimes getting it wrong.

Today, fold expressions currently always generate parenthesized chains such as `((a <= b) <= max)`; as [P0893R1] observes, "this makes today's fold expressions for comparisons not useful and actually buggy."

1.2 History

In [P0515R0] which introduced `<=>`, section 3.3 proposing chained comparisons was the only part not adopted.

In [P0893R1], Barry Revzin re-proposed chained comparisons, including researching real-world code (with Nicolas Lesser and Titus Winters and finding shipping code bugs that expected comparisons to chain. The paper was rejected in EWG San Diego 2018, partly because the bugs seemed rare and lack of implementation experience.

1.3 What's new: Why consider this again now?

New motivation (safety): Since 2022, WG21 has been more receptive to proposals that improve safety and correctness, especially when those proposals are of the form "recompile your existing code and it gets safer and/or faster" as we did with erroneous behavior for uninitialized locals. Comparison chains' correctness has safety implications, because one of the most common examples of such chains is the natural bounds checks `min <= idx < max`, which [P0893R1] showed are actually written by accident in the wild and are currently wrong.

New C++ implementation and usage experience: Chained comparisons are implemented and used in [cppfront], and the resulting C+ code works in all recent versions of MSVC, GCC, and Clang.

New information: Since 2024, Clang 19 warns for boolean chains like `0 <= i < j` for integer variables `i` and `j`, so now all recent compilers already warn on such chains, but are required to accept them. (All compilers have long warned on literal cases like `0 <= i < 20` (GCC with `-Wall`, MSVC and Clang by default) but again are required to accept them.)

New tool (erroneous behavior): Since Tokyo 2024, we have the new tool of "erroneous behavior" in draft C++26 that we could apply to make invalid chains like `a <= b > c` either ill-formed or erroneous.

New simplified proposed rules: This paper refines the proposed rules to avoid changing the meaning of constructs like `a<b == c<d` (unchanged in this proposal) and of DSLs that use heavy operator overloading.

1.4 Acknowledgments

Thanks very much to Barry Revzin for [P0893R1] carrying the torch further for this feature, and T.C., Nicolas Lesser, and Titus Winters who helped him gather data.

Thanks to the following people for their feedback on this paper, and on my paper [P0515R0] where section 3.3 originally proposed this feature: Gašper Ažman, Walter Brown, Casey Carter, Lawrence Crowl, Gabriel Dos Reis, Vicente J. Botet Escriba, Hal Finkel, Charles-Henri Gros, Howard Hinnant, Corentin Jabot, Loïc Joly, Nicolai Josuttis, Tomasz Kamiński, Andrzej Krzemieński, Jens Maurer, Alisdair Meredith, Patrice Roy, Jan Schultke, Mikhail Semenov, Richard Smith, Oleg Smolsky, Jeff Snyder, Peter Sommerlad, David Stone, Bjarne Stroustrup, Daveed Vandevoorde, Tony Van Eerd, and Ville Voutilainen.

2 Rationale and design alternatives

For detailed rationale and discussion, see [P0893R1].

For convenience, here is a copy of the key results reported in that paper, in the section “Existing Code in C++” (**emphasis** is original, highlights are added to draw attention to some key parts):

> ***Existing Code in C++***
>
> *The first question we sought to answer is the last question implied above: How much code exists today that uses chained comparison whose meaning would change in this proposal, and of those cases, how many were intentional (wanted the current semantics and so would be broken by this proposal) or unintentional (compile today, but are bugs and would be silently fixed by this proposal)? Many instances of the latter can be found in questions on StackOverflow* [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] … *.*
>
> *To that end, we created a clang-tidy check for all uses of chained comparison operators, ran it on many open source code bases, and solicited help from the C++ community to run it on their own. The check itself casts an intentionally wide net, matching any instance of a @ b @ c for any of the six comparison operators, regardless of the types of these underlying expressions.*
>
> *Overall, what we found was:*
>
> - ***Zero** instances of chained arithmetic comparisons that are correct today. That is, intentionally using the current standard behavior.*
> - *Four instances of currently-erroneous arithmetic chaining, of the `assert(0 <= ratio <= 1.0);` variety. These are bugs that compile today but don't do what the programmer intended, but with this proposal would change in meaning to become correct.*
> - *Many instances of using successive comparison operators in DSLs that overloaded these operators to give meaning unrelated to comparisons.*
>
> *Finding zero instances in many large code bases where the current behavior is intended means this proposal has low negative danger (not a significant breaking change). However, a converse search shows this proposal has existing demand and high positive value: we searched for expressions that would benefit from chaining if it were available (such as `idx >= 0 && idx < max`) and found **a few thousand** instances over just a few code bases. That means that this proposal would allow broad improvements across existing code bases, where linter/tidying tools would be able to suggest rewriting a large number of cases of existing code to be clearer, less brittle, and potentially more efficient (such as suggesting rewriting `idx >= 0 && idx < max` to `0 <= idx < max`, where the former is easy to write incorrectly now or under maintenance, and the latter is both clearer and potentially more efficient because it avoids multiple evaluation of `idx`). It also adds strong justification to pursuing this proposal, because the data show the feature is already needed and its lack is frequently being worked around today by forcing programmers to write more brittle code that is easier to write incorrectly.*

3 Proposal: Start a transition period for boolean chains and folds, so we can later make them mathematically transitive

Based on EWG Hagenberg feedback, the proposal is as before but for now to always reject the chains in question, so that someday later we can give them new meanings (where mathematically transitive) or continue rejecting them (where not mathematically transitive):

- For an expression E that is an unparenthesized chain of $x_1\ @_1\ x_2\ @_2\ \ldots\ @_{n-1}\ x_n$, where each $@_i$ is a relational operator ([expr.rel]) or equality operator ([expr.eq]), let the expression E' be $((x_1\ @_1\ x_2)$ `&&` $(x_2\ @_2\ x_3)$ `&&` … `&&` $(x_{n-1}\ @_{n-1}\ x_n))$ where every x_i for $0 < i < n$ is materialized as an lvalue, and if E' is valid and of a type contextually convertible to `bool`:
    - If in E every $@_i$ is one of `<` and `<=`, or is one of `>` and `>=`, or is `==`, or is `!=`, and `n>2`, then E is **ill-formed (alternative: deprecated)**.

Note For example, comparing integers using `min <= index_expression < max` will be ill-formed. My understanding is that every major compiler already warns on this case.

Future In the future, we can then make these chains have mathematically valid transitive meaning.

-
    - Otherwise, E every $@_i$ is a relational operator or is an equality operator, then E is **ill-formed (alternative: deprecated)**.

Note For example, `a <= b > c` would be ill-formed, just as it is anti-recommended in the languages that currently allow it (e.g., Python).

Future In the future, we would continue to reject these, because they are not mathematically transitive.

- For an expression E that is a fold-expression ([expr.prim.fold]) whose fold-operator is a relational operator or equality operator, E is **ill-formed (alternative: deprecated)**.

Note For example, `(... <= max)` will be ill-formed. It has no sensible meaning today.

Future In the future, we can then make these chains have mathematically valid transitive meaning by just expanding them without injecting parentheses (in combination with the first Future note above).

- No change to other chains. Expressions like `a<b == c<d` and `((a < b) < c)` retain their current reasonable meaning. Existing DSLs that overload comparison operators retain their current meaning.

4 Q&A

4.1 What about the implementability concerns previously raised in EWG, that this would be very hard to implement on compilers that mix parsing and sema phases?

I've discussed the concern with that implementer, and my understanding is that we now agree it's no longer a red flag. Because the proposed chains are always unparenthesized, and the compiler already has to look ahead to the next operator because it might have higher precedence, the parsing can key off the handling of a chained relational operator.

4.2 Will my user-defined mathematical type work with chained comparisons?

Yes, because any mathematical type for which `if(a<b)` works already provides relational comparisons whose results are contextually convertible to `bool`, and so will work with chained comparisons.

4.3 Does this allow any other strong typing benefits?

Gašper Ažman points out that this feature also makes partial bounds easy to use correctly using the type system.

Consider this comparison, where we only want to use `LowerBound` and `UpperBound` as part of a chain, not in isolation because we don't want those partial bounds to be used in isolation:

```
LowerBound{1} < y < UpperBound{5}
```

In this proposal, this performs

```
lower_ok_t __a = LowerBound{1} < y;

upper_ok_t __b = y < UpperBound{5};
```

and then

```
__a && __b
```

This enables us to make `lower_ok_t` and `upper_ok_t` individually not convert to bool because they're part of a cohesive check, to prevent them from being used in isolation, and provide `bool operator&&(lower_ok_t, upper_ok_t)` which enables the chain.

This approach allows modeling split "in-range" comparisons with strong types, and `LB < y < UB` reads correctly and verifies that the programmer tested both bounds.

5 Proposed wording

After [expr.rel] and [expr.eq], add a new subclause:

x.x.x Chained comparisons [expr.chain]

For an expression E of the form x1 op1 x2 op2 … opN-1 xN where N > 2 and each opi is a relational operator ([expr.rel]) or each opi is an equality operator ([expr.eq]), let the expression E’ be `( (x1 op1 x2) && (x2 op2 x3) && … && (xN-1 opN-1 xN) )` where each `xi` for 0 < i < N is treated as an lvalue. E’ is unevaluated. E is ill-formed if E’ is valid and the type of E’ is contextually convertible to `bool`.

Change [temp.variadic] paragraph 14 as follows:

A *fold-expression* whose *fold-operator* is a relational operator or equality operator is ill-formed. Otherwise, t~~T~~he instantiation of a *fold-expression* ([expr.prim.fold]) produces:

- (((E_1 op E_2) op ⋯) op E_N) for a unary left fold,
- (E_1 op (⋯ op (E_{N-1} op E_N))) for a unary right fold,
- ((((E op E_1) op E_2) op ⋯) op E_N) for a binary left fold, and
- (E_1 op (⋯ op E_{N-1} (op E_N (op E)))) for a binary right fold.

In each case *[… etc. as currently …]*

6 References

[cppfront] H. Sutter. Cppfront compiler (GitHub, 2022-2025).

[P0515R0] H. Sutter. “Consistent comparison” (WG21 paper, February 2017).

[P0893R1] B. Revzin. “Chaining comparisons” (WG21 paper, April 2018).