Chained comparisons: Safe, correct, efficient

Document Number: **P3439 R0**
Date: 2024-10-13
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. I am raising this now because of the new focus on the importance of safety and correctness in C++, and that this proposal automatically fixes known actual bugs in C++ code to do the right thing, including for safety-related bugs like bounds checks. Also, more implementation and use experience in C++ code is now available via [cppfront]; starting in 2024 all major compilers already warn on (but accept) the current bugs; and since Tokyo 2024 we have the new tool of "erroneous behavior" in draft C++26.

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, and so all such 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.

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 with the same semantics, including researching real-world code (with Nicolas Lesser and Titus Winters) and finding cases in production code that were certainly bugs that expected comparisons to chain. However, the paper was rejected by EWG in San Diego 2018; one of the objections was that the current cases that are bugs seemed rare, and another was that there was no 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. This has safety implications because perhaps the most common example of such chains is bounds checking, `min <= idx < max`, which [P0893R1] showed are actually written by accident in the wild and are currently wrong.

> Note: Just because the bugs in the field today may be rare doesn't mean we should not fix them, especially now when safety is a priority when not in tension with efficiency or correctness, and this is neither: it is more efficient because of single evaluation, and more correct because it has the expected meaning.

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

New information: In 2024, Clang 19 added a warning for boolean chains like `0 <= 100 < 10`, so now all recent compilers already warn on (but accept) such chains.

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.

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: Unparenthesized chains of all `<`/`<=`, all `==`, or all `>`/`>=` that are boolean have their correct transitive meaning

This paper proposes a refinement of [P0893R1] that may address some of the earlier concerns:

- **Making unparenthesized chains of all `<`/`<=`, all `>`/`>=`, or all `==`, when the expression is of a copyable type contextually convertible to `bool`, be correct (transitive pairwise boolean meaning) and efficient (single evaluation).** For example, comparing integers using `min <= index_expression < max` will mean the expected `min <= index_expression && index_expression < max` but with single evaluation of `index_expression`.

- **Making unparenthesized chains of all `<`/`<=`/`>`/`>=` that contain at least one `<` or `<=` and at least one `>` or `>=`, when the expression is of a type contextually convertible to `bool`, be ill-formed (or erroneous behavior).** For example, `a <= b > c` would be ill-formed or erroneous, just as it is anti-recommended in the languages that currently allow it (e.g., Python).

- **Making fold-expressions of `<`, `<=`, `>`, or `>=` be ill-formed (or erroneous behavior).** For example, `(... <= vals)` would be ill-formed or erroneous. These fold-expressions always generate parenthesized chains; as [P0893R1] observes, "this makes today's fold expressions for comparisons not useful and actually buggy."

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

4 References

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

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

[cppfront] H. Sutter. Cppfront compiler (GitHub, 2022-2024).