`std::ranges::less<>` Should Be More!

Document #: WG21 D1291R0
Date: 2018–10–07
Project: JTC1.22.32 Programming Language C++
Audience: LEWG
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper recapitulates the evolution of `std::less`, and its sibling operator types. The paper then proposes to reformulate these types in light of the improved understanding we have achieved in recent years.

Some objects seem to disappear immediately while others never want to leave.
— Louis Jenkins

1 Background

The C++ standard library has long provided "basic function object classes" corresponding to all the arithmetic, comparison, logical, and bitwise operators in the language.[1] Over the years, there have been a few significant changes to these "classes" (actually, class templates):

- Adopting [N3421] augmented each of these with a `void` specialization[2] "to make `<functional>`'s operator functors easier to use and more generic."
- Adopting [N3657] then injected, into each of these `void` specializations, "a nested type called `is_transparent`" as part of a "protocol" whose purpose was "to allow heterogeneous lookup when using user-defined comparator functions."
- Adopting [N3789] then inserted `constexpr` to provide "the ability to work at compile time."
- Finally, adopting [P0090R0] removed "every mention of `result_type`, `argument_type`, `first_argument_type`, and `second_argument_type`" from these class templates and their specializations.[3]

[1] These are specified in [N4762] subclauses [arithmetic.operations], [comparisons], [logical.operations], and [bitwise.operations], respectively.

[2] Collectively, these `void` specializations have since been sometimes informally termed the *diamond operators* and variables of such types have been sometimes termed *diamond objects*.

[3] A period of deprecation preceded the actual removal. [P0090R0] was merged into [P0005R4], whose adoption deprecated the listed aliases for C++17. Subsequently, the aliases were removed from the nascent C++20 via adoption of [P0619R4].

Using **std::less** as representative of these function object classes, the cumulative effect of those adopted proposals has led to implementations such as the following. First we have the primary template (namespace elided):

```
template< class T = void >
struct less {

  constexpr bool
    operator() ( const T& x, const T& y ) const
  {
    return x < y;
  }
}
```

In addition, we have the **void** specialization:

```
template<>
struct less<void> {

  template<class T, class U>
  constexpr auto
    operator() ( T&& t, U&& u ) const
  -> decltype(std::forward<T>(t) < std::forward<U>(u))
  {
    return std::forward<T>(t) < std::forward<U>(u);
  }

  using is_transparent = unspecified;
};
```

Most recently, specifications leading to such implementations, augmented by appropriate constraints,[4] have been adopted (for comparison function objects only) by the ranges proposal [P0896R2]. In that proposal, these constrained templates are of course declared in the **std::ranges** subnamespace so as to avoid conflicting with the existing unconstrained versions in namespace **std**.

2 Proposal

With the likely near-term adoption of that ranges proposal [P0896R2], we seem to have a once-in-a-generation opportunity to change the specifications of these function templates to obtain the far simpler and more useful design we should have had all along, a design that has now been proven by over six years of experience. Since the homogeneous comparison case is completely subsumed by the heterogeneous case, we can achieve implementations illustrated by the following:

[4] For example, these constraits may take the form of **requires** clauses involving such concepts as **StrictTotallyOrdered<T>** (for the primary template) and **StrictTotallyOrderedWith<T,U>** (for the **void** specialization).

```
struct less {

  template< class T, class U >
    requires StrictTotallyOrderedWith<T, U>
          or BUILTIN_PTR_CMP(T, <, U)  // see footnote 5
  constexpr auto
    operator() ( T&& t, U&& u ) const
  -> decltype(std::forward<T>(t) < std::forward<U>(u))
  {
    return std::forward<T>(t) < std::forward<U>(u);
  }

  using is_transparent = unspecified;
};
```

and analogously for the sibling arithmetic, comparison, logical, and bitwise function objects. Note that we are no longer dealing with class templates; rather, we have ordinary classes, each with a member function template `operator()` as today found in the diamond operator specialization.

As pointed out by Stephan T. Lavavej in personal correspondence, "CTAD allows diamond objects to be constructed with `less{}` which is optimal, but the type still needs to be spelled as `less<>` (e.g., in a `map`) which is an avoidable bit of complexity, and this [proposal] avoids it."

If this proposal is countenanced by LEWG, we would proceed to refine our prototype constrained code and then draft suitable wording for presentation in a subsequent revision of this paper.[5]

3 Acknowledgments

We are grateful to Casey Carter and Stephan T. Lavavej for their respective thoughtful comments on an early draft of this paper.

4 Bibliography

[N3421] Stephan T. Lavavej: "Making Operator Functors greater<>." ISO/IEC JTC1/SC22/WG21 document N3421 (pre-Portland mailing), 2012–09–20. https://wg21.link/n3421.

[N3657] Jonathan Wakely, Stephan T. Lavavej, and Joaquín M[a] López Muñoz: "Adding heterogeneous comparison lookup to associative containers (rev 4)." ISO/IEC JTC1/SC22/WG21 document N3657 (post-Bristol mailing), 2013–03–19. https://wg21.link/n3657.

[N3789] Marshall Clow: "Constexpr Library Additions: functional." ISO/IEC JTC1/SC22/WG21 document N3789 (post-Chicago mailing), 2013–09–27. https://wg21.link/n3789.

[N4762] Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4762 (corrected post-Rappersville mailing), 2018–07–07. https://wg21.link/n4762.

[P0005R4] Alisdair Meredith, Stephan T. Lavavej,, and Tomasz Kamiński: "Adopt `not_fn` from Library Fundamentals 2 for C++17." ISO/IEC JTC1/SC22/WG21 document P0005R4 (post-Jacksonville mailing), 2016–03–01. https://wg21.link/p0005r4.

[P0090R0] Stephan T. Lavavej: "Removing `result_type`, etc." ISO/IEC JTC1/SC22/WG21 document P0090R0 (pre-Kona mailing), 2015–09–24. https://wg21.link/p0090r0.

[5] [P0896R2] defines `BUILTIN_PTR_CMP(T, op, U)` as "a boolean constant expression [that] is `true` if and only if *op* in the expression `declval<T>()` *op* `declval<U>()` resolves to a built-in operator comparing pointers."

[5] While it might have been better to integrate these wording adjustments into the ranges proposal [P0896R2], we prefer to avoid any risk of delayed acceptance of that paper.

[P0619R4] Alisdair Meredith, Stephan T. Lavavej,, and Tomasz Kamiński: "Reviewing Deprecated Facilities of C++17 for C++20." ISO/IEC JTC1/SC22/WG21 document P0619R4 (corrected post-Rappersville mailing), 2018–06–08. https://wg21.link/p0619r4.

[P0896R2] Eric Niebler, Casey Carter, and Christopher Di Bella: "The One Ranges Proposal." ISO/IEC JTC1/SC22/WG21 document P0896R2 (corrected post-Rappersville mailing), 2018–06–25. https://wg21.link/p0896r2.

5 Document history

Rev.	Date	Changes
0	2018–10–07	• Published as D1291R0, pre-San Diego mailing.