Fixing the `partial_order` comparison algorithm

Document #: P0863R1
Date: 2018-10-08
Project: Programming Language C++
Audience: Library Evolution
Reply-to: Jeff Snyder <jeff-isocpp@caffeinated.me.uk>

1 Introduction

At the Albuquerque 2017 meeting, the committee reviewed the *Consistent Comparison* proposal[P0515R3], and had consensus to merge it and its companion library wording paper[P0768R1] into the C++ working draft.

During the meeting, it was noticed that something does not seem quite right about the fallback to `operator==` and `operator<` in the specification of `partial_order`: it checks `a == b` and `a < b`, but does not check `b < a`. Moreover, there is no circumstance in which it will return `partial_order::unordered`, and so it must be returning something else when the inputs are unordered and this fallback is used.

It was suggested that this should be handled as an LWG issue, but discussions with those involved in designing these fallbacks revealed that this was in fact the intended design. As I understand it—and I do not claim to understand it well—the rationale for the current design is based on assuming that `operator<` provides at least a weak order, and so it follows that `!(a == b) && !(a < b)` implies `a > b`, regardless of whether the user called `weak_order` or `partial_order`. This design has some unfortunate consequences.

2 Discussion

If the comparison algorithms are to have fallbacks written in terms of `operator==` and `operator<`, what are the desirable properties of these fallbacks? These might include:

1. If `strong_order(a, b)` is called with objects of a type for which `operator==` and `operator<` implement a strong ordering, then the `strong_ordering` values returned provide a genuine strong order.
2. If `weak_order(a, b)` is called with objects of a type for which `operator==` and `operator<` implement a weak ordering, then the `weak_ordering` values returned provide a genuine weak order.

3. If `partial_order(a, b)` is called with objects of a type for which `operator==` and `operator<` implement a partial ordering, then the `partial_ordering` values returned provide a genuine partial order.
4. If `strong_order(a, b)` is called with objects of a type for which `operator==` and `operator<` implement a strong ordering and returns `strong_order::less`, `strong_order::equal` or `strong_order::greater`, then `weak_order(a, b)` returns `weak_order::less`, `weak_order::equivalent` or `weak_order::greater`, respectively.
5. If `strong_order(a, b)` is called with objects of a type for which `operator==` and `operator<` implement a strong ordering and returns `strong_order::less`, `strong_order::equal` or `strong_order::greater`, then `partial_order(a, b)` returns `partial_order::less`, `partial_order::equivalent` or `partial_order::greater`, respectively.
6. If `weak_order(a, b)` is called with objects of a type for which `operator==` and `operator<` implement a weak ordering and returns `weak_order::less`, `weak_order::equivalent` or `weak_order::greater`, then `partial_order(a, b)` returns `partial_order::less`, `partial_order::equivalent` or `partial_order::greater`, respectively.
7. If `strong_order(a, b)` returns `strong_order::less`, `strong_order::equal` or `strong_order::greater`, then `weak_order(a, b)` returns `weak_order::less`, `weak_order::equivalent` or `weak_order::greater`, respectively.
8. If `strong_order(a, b)` returns `strong_order::less`, `strong_order::equal` or `strong_order::greater`, then `partial_order(a, b)` returns `partial_order::less`, `partial_order::equivalent` or `partial_order::greater`, respectively.
9. If `weak_order(a, b)` returns `weak_order::less`, `weak_order::equivalent` or `weak_order::greater`, then `partial_order(a, b)` returns `partial_order::less`, `partial_order::equivalent` or `partial_order::greater`, respectively.

2.1 The value of the fallback clauses

Properties 1, 2 and 3 allow users to compare two values of a type that does not implement `operator<=>`, but get the result as one of the x`_ordering` types, if the user knows what type of ordering the type's `operator==` and `operator<` implement. This seems like valuable functionality for bridging between pre- and post-C++20 code.

Properties 4, 5 and 6 also seem very desirable—they make the substitution of a correct use[1] of `strong_order` or `weak_order` with `weak_order` or `partial_order` (respectively) behave the same as calling the former and converting the returned value to the result type of the latter.

[1]By a "correct use" here, I mean one where the underlying type's `==` and `<` operators implement an

Properties 7, 8 and 9 are harder to justify—they concern the behaviour of these functions when they are used on types whose `==` and `<` operators implement an ordering that does not have the properties of the ordering being requested, e.g. calling `weak_order` on values of a partially ordered type.

The current wording supports all of these properties except for property 3. Let us examine the consequences of this in more detail. Suppose we have a simple wrapper type for a `double`, which exists for the purpose of improving type safety:

```
struct price
{
    constexpr explicit price(double v) : m_value{v} {}
    constexpr explicit operator double() const { return m_value; }
    constexpr friend bool operator==(price a, price b)
    { return a.m_value == b.m_value; }
    constexpr friend bool operator<(price a, price b)
    { return a.m_value < b.m_value; }
private:
    double m_value;
}
```

If we consider that objects of type `price` could contain `NaN` values, we can achieve some surprising results:

```
static_assert(std::partial_order(price{std::nan("")}, price{1})
                  == std::partial_ordering::greater);           // OK
static_assert(std::partial_order(price{1}, price{std::nan("")})
                  == std::partial_ordering::greater);           // OK
```

We can redefine `partial_order` to avoid this pitfall, by having it test `b < a` as well as `a < b`, and return `partial_ordering::unordered` if both return `false`. This, however, comes at a cost[2]: properties 8 and 9 will no longer hold after this modification.

How valuable are those properties? They pertain to what happens when `strong_order` or `weak_order` is called on a partially ordered[3]type. This is already a nonsensical operation—these functions will yield values of type `strong_ordering` and `weak_ordering` respectively, but these results are unsound. Treating these ordering values as a strong order or a weak order, and relying on the properties of strong or weak ordering respectively, will yield an incorrect program.

I would therefore argue that there is no value in properties 7, 8 and 9 that is not provided by properties 4, 5 and 6, and that we should modify `partial_order` as described above.

ordering at least as strong as the one being requested.

[2]There is also a performance cost, but I am going to ignore this since there is little use in returning an incorrect result quickly.

[3]"partially ordered" here specifically means that the type is *only* partially ordered, i.e. that the partial ordering is not also a weak ordering or a strong ordering.

2.2 The cost of the fallback clauses

If there is no value in being able to call `strong_order` or `weak_order` on a type for which the `==` and `<` operators provide a partial order[4], then we should also consider the cost of allowing `strong_order(a, b)` and `weak_order(a, b)` to be valid expressions when `a` and `b` are values of such a type. If these expressions are valid regardless of the semantics of the operators that they are implemented in terms of, then SFINAE tests on these expressions cannot tell us anything useful about the comparison semantics of the type. `strong_order` will happily yield values of type `strong_ordering` when applied to a type that isn't even *partially* ordered.

This is of great concern to anyone hoping to use these new comparison algorithms in generic code. These comparison algorithms are useful as customisation points, e.g. they let a type that has a "natural" weak order expose a strong order to algorithms that require a strong ordering, but this usefullness is greatly compromised if there are many types for which `strong_order(a, b)` compiles but provides unsound results.

The set of such types is large, as it includes all types which have a floating-point value amongst their salient attributes. We should therefore consider removing the fallbacks to `operator==` and `operator<` entirely, such that (e.g.) `std::strong_order(a, b)` is defined as deleted (or, better yet, does not participate in overload resolution) if the type of `a` and `b` does not have an `operator<=>` returning `std::strong_ordering`.

We could then consider introducing a parallel set of comparison algorithms for the purpose of integration with types that do not support `operator<=>`.

3 Proposals

3.1 Option A

Alter the wording of `partial_order` such that it behaves correctly when used on a partially ordered pre-C++20 type.

3.2 Option B

Remove the fallbacks to `operator==` and `operator<` from all comparison algorithms, so that the comparison algorithms are closer to being useful as customisation points.

Furthermore, remove the `compare_3way` algorithm entirely since it is identical to calling `operator<=>` after the fallbacks are removed.

[4]See footnote 3

3.3 Option C

Remove the comparison algorithms entirely, so that they can be more thoroughly considered in the C++23 timeframe.

4 Wording

4.1 Wording for Option A

Change paragraph **[cmp.alg] (16.11.4)p3** as follows:

```
template<class T>
   constexpr partial_ordering partial_order(const T& a, const T& b);
```

Effects: Compares two values and produces a result of type `partial_ordering`:

- Returns `a <=> b` if that expression is well-formed and convertible to `partial_ordering`.
- Otherwise, if the expression `a <=> b` is well-formed, then the function shall be defined as deleted.
- Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to bool, returns `partial_ordering::equivalent` when `a == b` is true, otherwise returns `partial_ordering::less` when `a < b` is true, otherwise returns `partial_ordering::greater` when `b < a` is true, and otherwise returns `partial_ordering::`~~`greater`~~`unordered`
- Otherwise, the function shall be defined as deleted.

4.2 Wording for Option B

Change paragraph **[cmp.alg] (16.11.4)p1** as follows:

```
template<class T>
   constexpr strong_ordering strong_order(const T& a, const T& b);
```

Effects: Compares two values and produces a result of type `strong_ordering`:

- If `numeric_limits<T>::is_iec559` is true, returns a result of type `strong_ordering` that is consistent with the totalOrder operation as specified in ISO/IEC/IEEE 60559.

- Otherwise, returns `a <=> b` if that expression is well-formed and convertible to `strong_ordering`.
- ~~Otherwise, if the expression `a <=> b` is well-formed, then the function shall be defined as deleted.~~
- ~~Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to bool, returns `strong_ordering::equal` when `a == b` is true, otherwise returns `strong_ordering::less` when `a < b` is true, and otherwise returns `strong_ordering::greater`.~~
- Otherwise, the function shall be defined as deleted.

Change paragraph [**cmp.alg**] **(16.11.4)p2** as follows:

```
template<class T>
   constexpr weak_ordering weak_order(const T& a, const T& b);
```

Effects: Compares two values and produces a result of type `weak_ordering`:

- Returns `a <=> b` if that expression is well-formed and convertible to `weak_ordering`.
- ~~Otherwise, if the expression `a <=> b` is well-formed, then the function shall be defined as deleted.~~
- ~~Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to bool, returns `weak_ordering::equivalent` when `a == b` is true, otherwise returns `weak_ordering::less` when `a < b` is true, and otherwise returns `weak_ordering::greater`.~~
- Otherwise, the function shall be defined as deleted.

Change paragraph [**cmp.alg**] **(16.11.4)p3** as follows:

```
template<class T>
   constexpr partial_ordering partial_order(const T& a, const T& b);
```

Effects: Compares two values and produces a result of type `partial_ordering`:

- Returns `a <=> b` if that expression is well-formed and convertible to `partial_ordering`.
- ~~Otherwise, if the expression `a <=> b` is well-formed, then the function shall be defined as deleted.~~
- ~~Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to bool, returns `partial_ordering::equivalent` when `a == b` is true, otherwise returns `partial_ordering::less` when `a < b` is true, and otherwise returns `partial_ordering::greater`.~~

- Otherwise, the function shall be defined as deleted.

Change paragraph [**cmp.alg**] **(16.11.4)p4** as follows:

```
template<class T>
  constexpr strong_equality strong_equal(const T& a, const T& b);
```

Effects: Compares two values and produces a result of type `strong_equality`:

- Returns `a <=> b` if that expression is well-formed and convertible to `strong_equality`.
- ~~Otherwise, if the expression `a <=> b` is well-formed, then the function shall be defined as deleted.~~
- ~~Otherwise, if the expression `a == b` is well-formed and convertible to bool, returns `strong_equality::equal` when `a == b` is true, and otherwise returns `strong_equality::nonequal`.~~
- Otherwise, the function shall be defined as deleted.

Change paragraph [**cmp.alg**] **(16.11.4)p5** as follows:

```
template<class T>
  constexpr weak_equality weak_equal(const T& a, const T& b);
```

Effects: Compares two values and produces a result of type `weak_equality`:

- Returns `a <=> b` if that expression is well-formed and convertible to `weak_equality`.
- ~~Otherwise, if the expression `a <=> b` is well-formed, then the function shall be defined as deleted.~~
- ~~Otherwise, if the expression `a == b` is well-formed and convertible to bool, returns `weak_equality::equivalent` when `a == b` is true, and otherwise returns `weak_equality::nonequivalent`.~~
- Otherwise, the function shall be defined as deleted.

Change paragraph [**alg.3way**] **(23.7.11p1)** as follows:

~~`template<class T, class U> constexpr auto compare_3way(const T& a, const U& b);`~~

~~*Effects:* Compares two values and produces a result of the strongest applicable comparison category type:~~

- ~~Returns `a <=> b` if that expression is well-formed.~~
- ~~Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to `bool`, returns `strong_ordering`::equal when `a == b`~~

~~is `true`, otherwise returns `strong_ordering::less` when `a < b` is `true`, and otherwise returns `strong_ordering::greater`.~~

- ~~Otherwise, if the expression `a == b` is well-formed and convertible to `bool`, returns `strong_equality::equal` when `a == b` is `true`, and otherwise returns `strong_equality::nonequal`.~~
- ~~Otherwise, the function is defined as deleted.~~

Change paragraph [**alg.3way**] **(23.7.11p3)** as follows:

```
template<class InputIterator1, class InputIterator2>
  constexpr auto
    lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                 InputIterator2 b2, InputIterator2 e2);
```

Effects: Equivalent to:

```
return lexicographical_compare_3way(b1, e1, b2, e2,
                                    [](const auto& t, const auto& u) {
                                        return ~~compare_3way(t, u)~~
                                            t <=> u;
                                    });
```

4.3 Wording for Option C

Remove section [**cmp.alg**] **(16.11.4)**.

Change paragraph [**alg.3way**] **(23.7.11p1)** as follows:

~~`template<class T, class U> constexpr auto compare_3way(const T& a, const U& b);`~~

~~*Effects:* Compares two values and produces a result of the strongest applicable comparison category type:~~

- ~~Returns `a <=> b` if that expression is well-formed.~~
- ~~Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to `bool`, returns `strong_ordering::equal` when `a == b` is `true`, otherwise returns `strong_ordering::less` when `a < b` is `true`, and otherwise returns `strong_ordering::greater`.~~
- ~~Otherwise, if the expression `a == b` is well-formed and convertible to `bool`, returns `strong_equality::equal` when `a == b` is `true`, and otherwise returns `strong_equality::nonequal`.~~
- ~~Otherwise, the function is defined as deleted.~~

Change paragraph [**alg.3way**] **(23.7.11p3)** as follows:

```
template<class InputIterator1, class InputIterator2>
  constexpr auto
    lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                 InputIterator2 b2, InputIterator2 e2);
```

Effects: Equivalent to:

```
return lexicographical_compare_3way(b1, e1, b2, e2,
                                    [](const auto& t, const auto& u) {
                                      return ~~compare_3way(t, u)~~
                                             t <=> u;
                                    });
```

5 Changelog

5.1 Revision 1

- Updated section numbers to be based on N4762
- Changed the wording of option B to also remove bullet points that were made redundant by the proposed changes in R0
- Updated option B to also remove `compare_3way`
- Added option C

6 Acknowledgements

Many thanks to Graham Snyder and Gene Novark for proofreading and commenting on drafts of this paper.

References

[P0515R3] Herb Sutter. Consistent comparison. Proposal P0515R3, ISO/IEC JTC1/SC22/WG21, November 2017.

[P0768R1] Walter E. Brown. Library Support for the Spaceship (Comparison) Operator. Proposal P0768R1, ISO/IEC JTC1/SC22/WG21, November 2017.