Allow user overriding of `strong_order` in P0768R1

Gašper Ažman

2018-01-06

Document #: DxxxxR0
Date: 2018-01-06
Audience: Library Working Group
Reply-to: Gašper Ažman, gasper.azman@gmail.com

Contents

1 Status of this paper

This paper is a defect-report to a paper that has been voted into the working draft. It seeks to highlight an issue with the currently proposed `strong_order` algorithm.

The wording for the entire fix is not provided in this paper, and shall be written if this paper receives support.

2 Problem description

The paper P0768R1[1] proposes the library extensions for operator <=>. Among them is the function `strong_order(const T& a, const T& b)`, specified in section [cmp.alg].

This is the specification in that paper:

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

1. Effects: Compares two values and produces a result of type `strong_ordering`:
   1.1. 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.
   1.2. Otherwise, returns `a <=> b` if that expression is well-formed and convertible to `strong_ordering`.

1.3. Otherwise, if the expression `a <=> b` is well-formed, then the function shall be defined as deleted.

1.4. 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`.

1.5. Otherwise, the function shall be defined as deleted.

Point 1.1 hints at the potential for `strong_order` to be the elusive *default ordering*, required by Stepanov and McJones for *Regular* types, to enable sorting for the purposes of speeding up further processing such as logarithmic searching ([2], page 62, section 4.4). For this purpose, *any* strong ordering would do.

Elements of Programming stresses that many types do not have a *natural order*; even then, a *default order* (a total order that respects at least representational equality) should be provided for all *Regular* types, because the efficiency gains enabled by sorting are enormous. For types that do have a natural total order (possibly only in some of the domain), they specify the *default order* should agree with it wherever defined.

As an example, consider the lexicographic ordering of the gaussian integes. This forms a total order, and its restriction to the integers (gaussian integers of the form $n + 0j$ agrees with the natural order on the integers.

Unfortunately, the hope of finally having a canonical way of naming the default ordering[1] is destroyed by Point 1.3.

Operator <=> seems to be explicitly designed[2] to represent the *natural ordering* over the values of `T`. In the case of floating point, iec559 extends this natural order to a total order, thus providing our fabled *default ordering*. However, as per point 1.3, the user is not allowed to specify this extension to `strong_order` themselves, because the function is specified as deleted – it still participates in overload resolution.

3 Proposal

This paper proposes changing point 1.3 to read:

Otherwise, if the expression `a <=> b` is well-formed,[3] *the function does not participate in overload resolution.*

After the list, add a Note:

This function is the way to provide a default strong order for user-defined types. This strong order should be consistent with the natural order provided by operator <=>. If operator <=> already orders a and b, the default order should agree. Otherwise (if operator <=> deems a and b unordered or equivalent), the default order should order them if they are unequal.[4] [5]

[1] `std::less` was never really the canonical way of referencing the default ordering, except for pointers.

[2] Because of the various orderings that it supports; they map out the semantic gamut of natural orderings of value types.

[3] Note: point 1.2 already takes care of the case where <=> provides a strong (and thus valid default) order.

[4] An ordering weaker than *strong* implies the existence of elements that are unequal, but are not distinguished (deemed either equivalent or unordered) by <=>.

[5] *unequal* does not refer to `operator==`, but to the notion of equality exemplified by iec559 for floating point. It's the answer to the question "what should `unique` do (by default)?".

4 Fixups

The algorithms section contains a few other algorithms:[6]

- `weak_order(const T& a, const T& b)`
- `partial_order(const T& a, const T& b)`
- `strong_equal(const T& a, const T& b)`
- `weak_equal(const T& a, const T& b)`
- `partial_equal(const T& a, const T& b)`

Intuitively, one would expect that if `strong_order` is available, then so are `strong_equal`, `weak_order` and `partial_order` (with `weak_equal` and `partial_equal` being consequences of those). The current situation seems to provide for that by pure accident[7], with no reference to this fact.

However, if `strong_order` is the customization point for a default order that *may* be stronger than the order on operator <=>, then the above expectation may longer hold for such types.

The fix-up for each of the sections describing the above primitives would be to insert, after point x.1 (which describes the algorithm in terms of <=>) the automatic fallback to a call to `strong_order`, if it is resolvable through an unqualified call (thus enabling argument-dependent lookup).

5 Alternative

If the purpose of `strong_order` is not enabling a default-ordering for types, the iec559 exception should be removed from the wording, and a different customization point (perhaps called `total_order`) added for the express purpose of providing an arbitrary total order on the entire domain of a type.

6 Acknowledgments

I would like to thank Thomas Köppe for his valuable comments and review, and Roger Orr for bringing this to my attention.

References

[1] Walter E. Brown. "Library Support for the Spaceship (Comparison) Operator". In: *Post-Albuquerque Mailing* (2017). URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0768r1.pdf.

[2] Alexander Stepanov and Paul McJones. *Elements of Programming*. 1st. Addison-Wesley Professional, 2009. ISBN: 032163537X, 9780321635372.

[6] Not to be confused with the types of their results; those end in `-ing`: `strong_ordering`, `weak_ordering` etc.

[7] the rules are identical, except for the iec559 exception in `strong_order`, while floating-point types possess `operator<` and `operator==`, thus enabling the presence of all those primitives.