Allow user overriding of `strong_order` in p0768

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 represents 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`:
   (a) 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.
   (b) Otherwise, returns `a <=> b` if that expression is well-formed and convertible to `strong_ordering`.
   (c) Otherwise, if the expression `a <=> b` is well-formed, then the function shall be defined as deleted.

(d) 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`.

(e) Otherwise, the function shall be defined as deleted.

Point 1.1 hints at the potential for `strong_order` to be the elusive *default ordering* that Stepanov and McJones require for *Regular* types to enable logarithmic searching and algorithms such as unique([2], page 62, section 4.4).

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), the *default order* should agree with it wherever defined.

As an example, the lexicographic ordering of the gaussian integers would be one such default order: any two comparable numbers ($1 + 0j < 2 + 0j$, $0 + 1j < 0 + 2j$) still compare correctly, and any two incomparable numbers have a consistent default order.

Unfortunately, the hope of finally having a canonical way of naming the default ordering (`std::less` was never really that, except for pointers) is destroyed by Point 1.3.

This is because the spaceship operator is explicitly designed[1] 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 achieving our fabled *default ordering*. However, as per point 1.3, the user is not allowed to specify this extension themselves, because the function is specified as deleted, which means 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,*[2] *the function does not participate in overload resolution.*

After the list, add remark:

This function is the idiomatic way to provide a default strong order for your types. This strong order should be consistent with the natural order provided by operator <=> ($a <=> b = strong\_order(a, b)$ *for all* a, b *where* $a <=> b \neq 0$*).*

4 Fixups

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, 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 no longer holds.

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).

[1] This is clear because of the various orderings that it supports.

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

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.

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.