Allow user overriding of `strong_order` in P0768R1

Gašper Ažman

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 library extension that has been voted into the working draft as part of P0768R1[1].

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

2 Abstract

Rule 1.1 for `strong_order` algorithm (an exception for iec559 (floating point) types) suggests the algorithm is a customization point, but the rest of the rules disallow that.

This paper makes a case for why `strong_order` should be a customization point, and then explores the consequences and a possible fix.

3 Current Status

For reference, the current proposal for the `strong_order` algorithm looks like this:

```
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.

4 Exposition: The Natural and Default Orderings

Obviously, there are many reasons for sorting. However, this paper is chiefly concerned with the division between the *natural ordering* and the *default total ordering* as required for *Regular*[1] types by Stepanov and McJones in their seminal work Elements of Programming ([2], page 62, section 4.4).

The **natural ordering** is the ordering that makes semantic sense for a type. This is the ordering that operator `<=>` and its library extensions are tailor-made for: not every type is ordered (or even equality-comparable), and when a type supports an ordering, it might be strong, partial, or weak.

We use these orderings when we need them to make sense - heaps, scheduling tasks by topological sorts, various displays for users, etc. Not all value types have a natural ordering, because not all types are ordered. The gaussian integers are one such type.

The **default ordering**[2] is the strongest ordering that a type admits. Its equality is defined by value-substitutability, and unequal elements must be ordered; it is always strong and total, and might not make semantic sense.

According to Elements of Programming, every *Regular* type should provide a default ordering.

A type with a default ordering is far more useful than one without; ordering enables the use of tree-based containers (i.e. `map`, `set`), and algorithms based on sorted data (`unique`, the various set algorithms, and the various versions of binary search) – and this is just the tip of the iceberg. The only requirement for the above is having *a* total strong ordering - what the ordering *means* is utterly irrelevant.

The lexicographic ordering of the gaussian integers is a good example of a default ordering.

Another excellent example is `float` – its various `NaN`s and infinities are not ordered, which is why its natural ordering is not suitable as a default ordering. However, iec559 defines a total strong ordering for those values, thus enabling the uses outlined above.

[1] The Elements of Programming concept, not the ISO C++ Regular, which is weaker.
[2] The name comes from Elements of Programming

4.1 On Compatibility Between the Natural and Default Orderings

Elements of Programming specifies that for types where the natural and default orderings differ, the default ordering should be compatible with the natural one: that is, if `a` and `b` are comparable and compare unequal under `<=>`, the default order produces the same result (less or greater).

However, requiring this in the language of the standard library as a mandatory semantic constraint seems like a bad idea.

For instance, if one takes the gaussian integers ordered by the manhattan-distance to zero (sum of absolute values of the two components), the compatible total order (a lexicographic ordering of every equivalence class) is far slower to compute than the simple lexicographic one.

Furthermore, if needed, a compatible total order can always be achieved on the fly by comparing with the natural order first - if the result is `less` or `greater`, keep the result - otherwise, fall back on the default ordering.

5 Problem Description

The current C++ standard does not have an explicitly designated customization point for providing a default ordering. Elements of Programming uses `less<T>::operator()`, as does the global order for pointers; but in the wake of operator `<=>`, `less<T>` is missing features, such as computing equality without calling it twice. It has also failed to get adoption for this purpose throughout the years.

The wording of point 1.1 of the `strong_order` algorithm suggests that `strong_order` is finally this missing customization point for specifying a default ordering for types whose natural ordering is not strong and total, since it does exactly that for the iec559 types.

The issue is that the rest of the points make this function rather unsuitable for use as a customization point, since the language explicitly makes it not SFINAE-friendly. In the event that it cannot be synthesized, it is marked as *deleted*, and not as *"shall not participate in overload resolution"*.

6 Code Example

Let me illustrate on a trivial example. Say we have a template struct representing the gaussian integers, with a "natural order"[3] defined by the manhattan distance from $0 + 0i$. This struct still defines a `strong_order` to be *Regular*.

```
namespace user {
  template <typename T>
  struct gaussian {
    static_assert(std::is_integral_v<T>);
    T re;
    T im;

    constexpr std::strong_equality operator==(gassian const& other) const {
      return re == other.re && im == other.im;
    }
    constexpr std::weak_ordering operator<=>(gaussian const& other) const {
      return (*this == other) ? std::weak_ordering::equal
      : (abs(*this) == abs(other))
      ? std::weak_ordering::equivalent
      : abs(*this) <=> abs(other);
    }
    friend constexpr T abs(gaussian const&) {
      using std::abs;
      return abs(re) + abs(im);
    }
```

[3]There is no natural order on gaussian integers, but humor this example, please.

```
    friend constexpr std::strong_ordering strong_order(gaussian const& x,
    gaussian const& y) {
      // compare lexicographically
      return std::tie(x.re, x.im) <=> std::tie(y.re, y.im);
    }
  };
}
```

Consider a transparent ordering operator for `map`:

```
struct strong {
  struct less {
    template <typename T, typename U>
    bool operator()(T const& x, U const& y) {
      using std::strong_order; // use ADL
      return strong_order(x, y) < 0;
    }
    using is_transparent = std::true_type;
  };
  // also equal, greater, ge_eq, less_eq etc.
};
```

Also say we had a type with an implicit conversion to our `gaussian`:

```
template <typename T>
struct lazy {
  std::function<T()> make;
  operator T() const { return make(); }
};
```

This function now fails to compile, because the chosen `strong_order` is deleted.

```
bool exists(lazy<gaussian<int>> const& x,
std::set<gaussian<int>, strong::less> const& in) {
  /* imagine this being a template in both parameters - it's pretty normal */
  return in.count(x);
}
```

7 Proposal

7.1 Make `strong_order` An Explicit Customization Point

Depending on the final direction of the wording on customization points (either the current one, with an emphasis on *shall not participate in overload resolution*, or the one outlined in P0551R0[**p0551r0**], the wording shall differ here.

I am asking LWG for guidance on this subject.

7.2 Remove the iec559 Exception (point 1.1)

Since this paper adds explicit support for this customization point, the exception can now be implemented using whichever mechanism for customization points is chosen, and this special case moved to that part of the standard. For instance, a "more specialized" template based on a `requires` clause and the `numeric_limts<T>::is_iec559` trait can be added to `namespace std`.

The minimal fix for the current situation would be:

Change point 1.3 to read: *Otherwise, if the expression* `a <=> b` *is well-formed,*[4] *the function does not participate in overload resolution.*

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

After the list, add a Note:

If operator <=> provides an order weaker than strong, this function allows the provision of a default strong order for a user-defined type. In that case, `strong_order` *should define a strict, total ordering.*

7.3 Fix The Rules for Synthesis of Weaker Algorithms

The algorithms section contains a few other algorithms:[5]

- `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 accident[6], with no explicit 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 no longer hold for such types (you might have `strong_order` but not `weak_order`, for instance).

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).

7.4 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.

8 Acknowledgments

I would like to thank

Roger Orr for bringing this to my attention;

Thomas Köppe for his valuable comments, review, and most of all some extremely clear and laconic wording;

Sam Finch for *thoroughly* breaking my examples, some example code, great substantive comments, and pointing out that the current definition actually breaks types that define a partially-ordered set of comparison operators;

Richard Smith for further fixing my example in light of Concepts, and example code.

Thanks!

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

[6] the rules for those algorithms are identical but for the iec559 exception in `strong_order`; since floating-point types possess `operator<` and `operator==`, they enable the synthesis of all those algorithms.

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.