Making Optional Greater Equal Again

Document number: P0307R0
Date: 2016-03-15
Audience: LEWG/LWG
Reply-to: Tony Van Eerd. optional at forecode.com

tl;dr:

```
optional<X> opt = ....;

assert(( opt >=  opt) == true);
assert((*opt >=  opt) == true);
assert(( opt >= *opt) == true);
assert((*opt >= *opt) == false);
```

One of those is not like the others.

Similarly, for some classes Y:

```
optional<Y> opt = ....;

... ( opt >=  opt) ... // OK
... (*opt >=  opt) ... // OK
... ( opt >= *opt) ... // OK
... (*opt >= *opt) ... // compile error!!!
```

This inconsistency in optional is a very simple and small problem with a very simple and small fix: `optional>=` needs to call T's `operator>=`. (optional currently instead calls `!operator<()`, which is typically, *but not always*, the same result as `X::operator>=`, ie consider `X` == `float`)

If you agree with the small fix (or already assumed it worked that way) you don't really need to read any further. It is that simple.

3 categories of classes

Aggregates: pair, tuple; struct { int m, n; }; struct { float m; }; class MyFoo {...};
Wrappers: optional, variant, any, expected, ...
Containers: int x[17], std::array, vector, vector, vector, ...

Notes:

- you can argue that `int x[17]` is an Aggregate. But it is not a Wrapper.
- I hear there are other containers past vector, but you should just use vector :-)
- `struct { float m; }` is an aggregate, not a wrapper

Wrappers

So what "defines" or at least hints at a Wrapper ('Proxy'? Some better name?)

- *implicit* construction from the 'wrappee'
- conversion to the wrappee
- relational operators between wrapper and wrappee ie `optional<X>{} < X{}`
- in general trying to 'act like' the wrappee (proxy etc)

The above is why `struct { float m; }` is not a Wrapper/Proxy, but an Aggregate.

Consistency

We want everything to be consistent. Sometimes this is not possible. What should we do?

The following is probably obvious when stated, but still needs to be stated sometimes:
Not all consistency is valued equally. There is a scale (from greatest to least value):

- Self consistency
- Similar consistency
- ...
- ...
- Global Consistency

So how does consistency apply to the current situation with optional? ***Wait.*** First, optional needs to be correct. Being consistently wrong is not nearly as good as consistently right. Of these lines:

```
optional<float> opt = NaN;

assert(( opt >=  opt) == true);
assert((*opt >=  opt) == true);
assert(( opt >= *opt) == true);
assert((*opt >= *opt) == false);
```

only the bottom one is *correct*. (or you can argue that only the bottom one can't change, as we are not changing how float works)

So we need to change the other three. And it is not that "we have to", it is what makes sense. If `optional>=` calls T's `>=` we get (of course)

```
assert(( opt >=  opt) == false);
assert((*opt >=  opt) == false);
assert(( opt >= *opt) == false);
assert((*opt >= *opt) == false);
```

This makes optional *consistent with itself* and T.

Consistency with neighbours

- This small fix makes optional *closer* to being consistent with aggregates - `optional<float>` now gives the same results as `struct { float m; }` (particularly if/when we get default generation of relational operators from EWG). But optional is still slightly inconsistent vs Aggregates when dealing with exotic types. For Aggregates, `operator>=` calls `operator>` and `operator=` instead of calling memberwise `>=`. (We can't change how Aggregates work here. For aggregates with more than one member, you cannot build a sensible lexicographical `>=` from only memberwise `>=`. For *single* member, you could use `>=` directly, but then single-member aggregates would not be consistent with multi-member aggregates.)
- This small fix makes optional *slightly* inconsistent with Containers. Fine. Optional is not a Container, it is a Wrapper. And for "normal" types, they are all still consistent. More importantly, they are each consistent within their own category.
- (Also, regardless of this fix, Containers are not consistent with Aggregates (of floats, for example). We don't suggest changing Containers or breaking code. Also, Containers (often) are used to find() things via an equivalence (not equality) relation, so maybe Containers aren't broken. Maybe.)
- (Currently `pair` and `tuple` act like Containers (with respect to `>=`). They should probably act like Aggregates. This paper does not suggest changing them at this time.)

A Slight Alternative

Ville suggested that `optional<T>` use `T`'s `>=` - *if it exists* - but that if `T` does *not* have `>=`, then `optional<T>` could generate `>=` from `<`. If P0221R1 (default comparison operators) is implemented, then it doesn't matter - `T` will already have a generated `>=` (unless, of course, the developer intentionally deleted it).

As shown above (near the beginning) it may be more consistent to only define `optional<T>::operator>=` when `T` defines `>=`, but falling back to a `<` based definition may be seen as "convenience".

Variant

Ditto for variant, particularly if accepted into C++17. And for any other potential wrapper classes (std::expected, etc).

Acknowledgements

Thanks to Chandler and Nico and many others for encouraging me, and for Ville and Nevin for putting up with me :-)

Wording

(For optional. Based on current wording. (I'm not sure why the wording of `optional < optional` *is so different from* `optional < T`*))*

`template<class T> constexpr bool operator>(const optional<T>&x, const optional<T>&y);`
Requires: Expression `*x > *y` shall be well-formed.
Returns: If `!x`, `false`; otherwise, if `!y`, `true`; otherwise `*x > *y`.
Remarks: Instantiations of this function template for which `*x > *y` is a core constant expression, shall be constexpr functions.

`template<class T> constexpr bool operator<=(const optional<T>&x, const optional<T>&y);`
Requires: Expression `*x <= *y` shall be well-formed.
Returns: If `!x`, `true`; otherwise, if `!y`, `false`; otherwise `*x <= *y`.
Remarks: Instantiations of this function template for which `*x <= *y` is a core constant expression, shall be constexpr functions.

`template<class T> constexpr bool operator>=(const optional<T>&x, const optional<T>&y);`
Requires: Expression `*x >= *y` shall be well-formed.
Returns: If `!y`, `true`; otherwise, if `!x`, `false`; otherwise `*x >= *y`.
Remarks: Instantiations of this function template for which `*x >= *y` is a core constant expression, shall be constexpr functions.

`template<class T> constexpr bool operator!=(const optional<T>&x, const optional<T>&y);`
Requires: Expression `*x != *y` shall be well-formed.
Returns: If `bool(x) != bool(y)`, `true`; otherwise, if `bool(x) == false`, `false`; otherwise `*x != *y`.
Remarks: Instantiations of this function template for which `*x != *y` is a core constant expression, shall be constexpr functions.

Comparisons with T

`template <class T> constexpr bool operator!=(const optional<T>& x, const T& v);`
Returns: `bool(x) ? *x != v : true`.
`template <class T> constexpr bool operator!=(const T& v, const optional<T>& x);`
Returns: `bool(x) ? v != *x : true`.
`template <class T> constexpr bool operator<=(const optional<T>& x, const T& v);`
Returns: `bool(x) ? *x <= v : true`.
`template <class T> constexpr bool operator<=(const T& v, const optional<T>& x);`
Returns: `bool(x) ? v <= *x : false`.
`template <class T> constexpr bool operator>(const optional<T>& x, const T& v);`
Returns: `bool(x) ? *x > v : false`.
`template <class T> constexpr bool operator>(const T& v, const optional<T>& x);`
Returns: `bool(x) ? v > *x : true`.
`template <class T> constexpr bool operator>=(const optional<T>& x, const T& v);`
Returns: `bool(x) ? *x >= v : false`.
`template <class T> constexpr bool operator>=(const T& v, const optional<T>& x);`
Returns: `bool(x) ? v >= *x : true`.