Project: Programming Language C++, Library Working Group
Document Number: 3765
Date: 2013-9-01
Reply to: Tony Van Eerd
Email: tvaneerd@blackberry.com

N3765: On Optional

1. Use of std::less<>

optional<T>::operator<() is currently defined in terms of std::less(). The reasoning for this was to enable things like optional<T*> to work correctly in standard containers like std::map<>.

However, it would be better to instead specialize std::less for optional<T>.

```
template <class T> struct less<optional<T>> {
   bool operator()(const optional<T>& x, const optional<T>& y) const;
   bool operator()(const optional<T>& x, const T& y) const;
   bool operator()(const T& x, const optional<T>& y) const;
   typedef optional<T> first_argument_type;
   typedef optional<T> second_argument_type;
   typedef bool result_type;
   typedef unspecified is_transparent;
};
   operator()(const optional<T>& x, const optional<T>& y)
      returns x ? y ? std::less<T>{}(*x, *y) : false : y ? true : false.
   operator()(const optional<T>& x, const T& y)
      returns x ? std::less<T>{}(*x, y) : true.
   operator()(const T& x, const optional<T>& y)
      returns y ? std::less<T>{}(x, *y) : false.
```

(constexpr should be added to the operators if/when a contemporaneous proposal, which makes all <funtional> function object types constexpr, is accepted.)

And in **20.6.8 Relational operators**,
change the use of “less<T>{}(*x, *y)” in operator<() to “*x < *y”. ie

```
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), false; otherwise, if (!x), true; otherwise *x < *y.
 Remarks: Instantiations of this function template for which *x < *y is a core
constant expression, shall be constexpr functions.
```

Reasoning:
- std::less<Foo> is not always the same as Foo < Foo. For example, pointers. Also, complex<> may get a specialization of std::less, while not defining operator<(). (As an intuitive ordering of complex doesn't exist, but we would still like to order them for use in containers.)

So optional<T>::operator<() should call T::operator<(), not std::less.

2. Other relational operators

currently there is no operator>() for optional<T>. Nor !=, <=, >=. Assuming we would like these operators (general consensus seems to be yes), our choices (to add to **20.6.8**) are:

A. implement optional::operator>() as the opposite of optional::operator<().

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

B. implement optional::operator>() using T's operator>().

```
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.
```

What's the (observable) difference?
- for T's that have “non-normal” relational operators, ie where (t1 < t2) != (t2 > t1), results using A and B will be different.
- for T's that do not implement >, but do implement <,

```
optional<T>() > optional<T>()
```

compiles for A, but not for B.

Reasoning for A (use the opposite of <)
A.1 - this is what std::tuple et al do. (but not what std::greater et al do)
A.2 - this is fine for “normal” classes.
A.3 - developer does not need to implement > (as optional does it for them).

Reasoning for B (use T > T)
B.1 - this is what std::greater et al do. (but not what std::tuple et al do)
B.2 - As mentioned, there is a difference when using “non-normal” T's. optional<T> is, in many ways, a proxy for T, and many motivating examples in the original paper include examples of easily updating code from using T to optional<T>. When using non-normal T's, the developer would expect optional<T> to behave similar to T. ie:

```
T t1 = ...;
T t2 = ...;
optional<T> ot1 = t1;
optional<T> ot2 = t2;

assert( (ot1 > ot2) == (t1 > t2) );

// and when we include mixed operators:
assert( (ot1 > t2) == (t1 > t2) );
assert( (t1 > ot2) == (t1 > t2) );
```

B.3 - When T *does not have* an operator>(), then neither should optional<T>, as it is not optional's “job” to extend T's interface, beyond extending it for the concept of “optionality”.

Recommendation: **B.** Use T's operator>(). In many ways, optional<T> is a proxy of T. Thus it should work like T when possible.

Similarly for the other relational operators, we get:

```
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.
```

And similarly for Comparison with T **(20.6.10)**:

```
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.
```

Note that “ T < optional<T> “ was missing in the original proposal, see http://cplusplus.github.io/LWG/lwg-active.html#2283. Included here for completeness:

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

3. Specialization of std::greater et al

Assuming we decide (in 1.) to specialize std::less, since std::greater<T> is defined in terms of T > T (and not std::less), if we wish to correctly support std::greater<optional<T*>>, then we need to specialize std::greater, and similarly greater_equal, and less_equal as well:

```
template <class T> struct greater<optional<T>> {
   bool operator()(const optional<T>& x, const optional<T>& y) const;
   bool operator()(const optional<T>& x, const T& y) const;
   bool operator()(const T& x, const optional<T>& y) const;
   typedef optional<T> first_argument_type;
   typedef optional<T> second_argument_type;
   typedef bool result_type;
   typedef unspecified is_transparent;
};
   operator()(const optional<T>& x, const optional<T>& y)
      returns x ? y ? std::greater<T>{}(*x, *y) : true : y ? false : false.
   operator()(const optional<T>& x, const T& y)
      returns x ? std::greater<T>{}(*x, y) : false.
   operator()(const T& x, const optional<T>& y)
      returns y ? std::greater<T>{}(x, *y) : true.

template <class T> struct greater_equal<optional<T>> {
   bool operator()(const optional<T>& x, const optional<T>& y) const;
   bool operator()(const optional<T>& x, const T& y) const;
   bool operator()(const T& x, const optional<T>& y) const;
   typedef optional<T> first_argument_type;
   typedef optional<T> second_argument_type;
   typedef bool result_type;
   typedef unspecified is_transparent;
};
   operator()(const optional<T>& x, const optional<T>& y)
      returns x ? y ? std::greater_equal<T>{}(*x, *y) : true : y ? false : true.
   operator()(const optional<T>& x, const T& y)
      returns x ? std::greater_equal<T>{}(*x, y) : false.
   operator()(const T& x, const optional<T>& y)
      returns y ? std::greater_equal<T>{}(x, *y) : true.

template <class T> struct less_equal<optional<T>> {
   bool operator()(const optional<T>& x, const optional<T>& y) const;
   bool operator()(const optional<T>& x, const T& y) const;
   bool operator()(const T& x, const optional<T>& y) const;
   typedef optional<T> first_argument_type;
   typedef optional<T> second_argument_type;
   typedef bool result_type;
   typedef unspecified is_transparent;
};
   operator()(const optional<T>& x, const optional<T>& y)
      returns x ? y ? std::less_equal<T>{}(*x, *y) : false : y ? true : true.
   operator()(const optional<T>& x, const T& y)
      returns x ? std::less_equal<T>{}(*x, y) : true.
   operator()(const T& x, const optional<T>& y)
      returns y ? std::less_equal<T>{}(x, *y) : false.
```

4. Quirk of optional<bool>

Note that, for the special case of optional<bool> we have this seeming contradiction:

```
optional<bool> ob = false;

assert( (!ob) != (ob == false) );
```

ie:

```
if (!ob)
```

is not the same as

```
if (ob == false)
```

(and similarly for == true). Note that for (hopefully) every other type in the standard,

```
if (!foo)
```

and

```
if (foo == false)
```

are either the same, or one or the other do not compile.

Choices:
- do nothing. Live with it.

- remove explicit operator bool() for optional<bool>
  when checking for (dis)engagement,
  optional<bool> code, and template code, would use comparison to nullopt

```
if (opt == nullopt)
```

- remove operator==() and operator!=() for optional<bool>
  when checking for “is it this value”,
  optional<bool> code, and template code, would use dereference:

```
if (opt && *opt == value)
```

- remove both for operator bool
- remove one, the other, or both, for optional<T> in general

Recommendation: *very weakly* recommending removal of both for optional bool.
Reasoning
- explicitness is clarity
– average usage of optional<> is unaffected
- burden on template writers, but template writers always need to be more careful than average
- less mistakes when using optional<bool>. (Note that Boost.Optional always recommended use of Boost.Tribool to handle this issue.)

(Note also, that if/when we allow looser mixed relations, such as optional<string>() == “foo”, the same will happen for optional<int>, etc.)