Default == is >, default < is < so

Document number: P0354R0
Date: 2016-05-15
Audience: EWG
Reply-to: Tony Van Eerd. order at forecode.com

tl;dr:

Contrary to P0221R1, we should not generate `operator<()` by default.

Motivation/Explanation

```
class chair { ... };
```

We can all easily imagine what is in class `chair`. It probably tells you the colour, size, shape, material, number of legs, etc,... of the chair.

```
chair1 == chair2
```

What does `chair1 == chair2` *mean*?

It is unsurprising to want to compare two chairs and determine that they are, for salient properties, equal. Default memberwise equality works fine. In fact, when I learned C circa 1987, I tried to compare two structs for equality, and was saddened that it didn't work. I think that default generation of == (and !=), a la P0221R1, is *great*, and *for most classes* both obvious and useful.

```
chair1 == chair2
```

Now, what does `chair1 < chair2` *mean*?

What does it mean to ask "is this chair *less* than that chair?" ? Is the chair smaller? Shorter? Lighter? Less legs? I think this question has little to no meaning. Less *red*? (imagine that the first member of `chair` is colour in RGB format.)

`operator<()` on `chair` is *meaningless*.

I understand an ordering *might* be useful, in particular when used with `std::map` (but maybe you should use `unordered_map`?), etc. But I don't appreciate *meaningless* API being added to **all** my classes. (Why not add a `calculate_volume` function that doesn't calculate the volume of the chair, or a `calculate_pi()` function, which doesn't calculate pi?) Why not memberwise operator+ and divide by scalar? At least then I could maybe calculate the *average chair*, which makes more sense than the *least chair*.

Ordering can be useful, but it shouldn't be tied to less. "Representative ordering" and "less" are different concepts, and each has their uses. They should not be conflated - at least not by default.

How bad is it?

If `operator<()` is generated by default, I will recommend, as a coding guideline, that the average class opt-out of this default generation. My default will be to disable the default. I'll go as far as allowing, maybe even recommending, a MACRO for this purpose. *It's that bad.*

Ways Out

1. **Just *don't* it**. Don't generate `operator<()`.
2. Make default generation of `operator<()` **opt in**. This has been discussed in the past. I'm not against it. I'd still like `==` to be default-in, opt-out. Because `==` almost always makes sense, `<` almost always doesn't.
3. **Generate a specialization of `std::order` instead** (which would then be used by `std::map` et al). `std::map` should never have defaulted to `std::less` but rather it should have defaulted to some `std::order` (which could defer to `std::less` if/when `std::order` wasn't specialized). See Alisdair's P0181R0 for further work (on the library side) in a similar direction.
4. **A new operator - the *ordering operator***. I know no one likes new syntax except the one proposing it, but... For now, to avoid bikeshedding, imagine it is `operator<@` (see footnotes). The new operator could be generated by default - *without ambiguity of meaning*, and used by `std::map` et al (for now, it could be called by `std::less` if/when `<` is invalid, and/or called by `std::order` and have `map` use that, etc).

The difference between 3 and 4 is just whether the *language* should generate *library* specializations, or whether it should stick to language-level syntax.

I recommend 1 followed by 3 or 4. ie for C++17, just don't generate `operator<()` and then introduce a new operator post C++17. These are better than option 2 (opt-in) because order *is* worthwhile, even when "less" doesn't make sense - it is a separate concept, and should be kept separate.

Conclusion(s)

1. Most importantly, please don't generate `operate<()` *by default*. It is just wrong.
2. Please take some other path towards default ordering - one of the paths suggested above, or some other path, just not default generated `operator<()`.

The rest of this paper discusses why separation of "less" and "representative order" is important, and why generating representative order some other way than `operator<()` would be worthwhile, but the main point of the paper has already been made: *we should not generate* `operator<()` *by default*.

The rest of this paper is probably post-C++17 discussion.

Other uses

I think "less" and "representative order" are fundamentally different, and if we had both as independent concepts, we would find many natural uses. The first use I found, years ago, was an `immutable_string` class. (Adobe, for example, had at least 2 classes like this.) For `immutable_string`, all instances that are equal (by string equality) can share the same storage for the string. (Like copy-on-write, but you never write!) The storage address becomes the implementation of `==`. Address is also useful for implementing `<` when used in `std::map` (if/when lookup is more important than order). But you still want `<` to be string-based less, for other uses, ie for display in a UI. Separating "order" from "less", and `std::order` from `std::less` and `operator<>` from `operator<` solves these issues.

I think there are many other uses, waiting to be found. The problem is common enough that many well-respected C++ leaders (eg Sean Parent, Alex Stepanov,...) have a stock recommendation: implement `std::less` but not `operator<` in cases where you want order, but `<` is meaningless. It is a common/real issue.

Take back std::less

Implementing `std::less` but not `operator<` is a viable work-around, but it is a hack. The point of `std::less` was for it to be the function-object form of `operator<`; exploiting the use of `std::less` as an extension point for `std::map` et al perverts the meaning of `std::less`. If `std::less` was meant to be an extension point, it probably should have been named differently, and have been specific to containers - ie `std::order`, for example. (Note also that these specializations of `std::less` may be prohibited by the standard - 17.6.4.2.1 "only if the declaration depends on a user-defined type and the specialization meets the standard library requirements for the original template" - what are the requirements of `std::less`? - it is defined to return "x < y", so if returning x < y is a requirement....)

By separating "less" from "representation order", we can keep `std::less` as having the single meaning of "calls `x < y`". I would in fact go further, and deprecate allowing users to specialization `std::less`. It should only have one meaning.

Conclusion(s) again

1. Most importantly, please don't generate `operate<()` *by default*. It is just wrong.
2. Please take some other path towards default ordering - one of the paths suggested above, or some other path, just not default generated `operator<()`.

Footnotes

- It is tempting to suggest that the new ordering operator should be "less-dot" ie `<.` because adding a dot seem to be in vogue, but in this case `<.` would just lead to ambiguities like `.1 < .0`, as would `.<` and `.<.` :-)
- `*<` works (but `<*` doesn't - ie `p < *q` vs `p<*q`).
- `<>` works (even though that means `!=` in some languages). It can be read to mean "some order, not necessarily greater or less, but some order"
- I would NOT recommend any of the addition operators that could be made from `<@` such as `<@=` etc.