Harmonizing the definitions of total order for pointers

Document #:	P1961R0
Date:	2019-11-05
Project:	Programming Language C++ Library (LWG)

Reply-to:	Gašper Ažman <gasper.azman@gmail.com>

Contents

1 Abstract

This paper adresses NB comments US220 and US176.

The comments point out that we provide a total order for pointers 3 times:

— 17.11.06 [cmp.object]
— 20.14.7 [comparisons]
— 20.14.8 [range.cmp]

We also do not require that they produce the same results. This paper unifies the wording and clarifies that they do produce the same results.

It is the intention of this paper to not introduce any change in behaviour. It merely clarifies that there is only one implementation-defined total order over pointers, and all parts of the library use the same one.

2 Proposed Wording

Diff against n4835.

In section [definitions], insert a subclause:

16.3.XX: **implementation-defined strict total order over pointers** [defns.order.ptr]

implementation-defined strict total ordering over all pointer values such that the ordering is consistent with the partial order imposed by the builtin operators `<`, `>`, `<=`, `>=`, and `<=>`

In section [cmp.object], change the reference:

(4.1) – If the expression `std::forward<T>(t) <=> std::forward<U>(u)` results in a call to a built-in operator`<=>` comparing pointers of type `P`, returns `strong_ordering::less` if (the converted value of) `t` precedes `u` in the implementation-defined strict total order ~~(20.14.8)~~ over pointers ([defns.order.ptr]) ~~of type `P`~~, `strong_ordering::greater` if `u` precedes `t`, and otherwise `strong_ordering::equal`.

In section [comparisons], change the reference, and normalize wording:

2 For templates `less`, `greater`, `less_equal`, and `greater_equal`, the specializations for any pointer type yield a result consistent with the implementation-defined strict total order over pointers ([defns.order.ptr])~~a strict total order that is consistent among those specializations and is also consistent with the partial order imposed by the built-in operators `<`,`>`,`<=`,`>=`.~~ [*Note*: If~~When~~ `a < b` is well-defined for pointers `a` and `b` of type `P`,

then~~this implies~~ `(a < b) == less<P>()(a, b)`,`(a > b) == greater<P>()(a, b)`, and so forth.— *end note*] For template specializations `less<void>`, `greater<void>`, `less_equal<void>`, and `greater_equal<void>`, if the call operator calls a built-in operator comparing pointers, the call operator yields a result consistent with the implementation-defined strict total order over pointers. ~~that is consistent among those specializations and is also consistent with the partial order imposed by those built-in operators.~~

In section [range.cmp], remove the definition, and add reference:

2 ~~There is an implementation-defined strict total ordering over all pointer values of a given type. This total ordering is consitent with the partial order imposed by the builtin operators <, >, <=, >=, and <=>.~~

(4.1) – If the expression `std::forward<T>(t) == std::forward<U>(u)` results in a call to a built-in operator`==` comparing pointers of type `P`: returns `false` if either (the converted value of) `t` precedes `u` or `u` precedes `t` in the implementation-defined strict total order over pointers ([defns.order.ptr]) ~~of type P~~ and otherwise `true`.

(8.1) – If the expression `std::forward<T>(t) < std::forward<U>(u)` results in a call to a built-in operator`<` comparing pointers of type `P`: returns `true` if (the converted value of) `t` precedes `u` in the implementation-defined strict total order over pointers ([defns.order.ptr]) ~~of type P~~ and otherwise `false`.