Library Support for the Spaceship (Comparison) Operator

Document #:	WG21 P0768R0
Date:	2017-09-30
Project:	JTC1.22.32 Programming Language C++
Audience:	LEWG ⇒ LWG
Reply to:	Walter E. Brown <webrown.cpp@gmail.com>

Contents

1 **Introduction** 1
2 **Comparison category types** 2
3 **Discussion** 2
4 **Proposed wording** 2
5 **Acknowledgments** 13
6 **Bibliography** 13
7 **Document history** 13

Abstract

This paper proposes standard library wording to accompany the core language wording in Sutter's proposal [P0515R2], "Consistent comparison."

Isn't it strange how a lamb can feel like a lion when comparing itself to a mouse, whereas a lion feels like a lamb when measuring itself against dragons?

— Richelle E. Goodrich

What makes the Universe so hard to comprehend is that there's nothing to compare it with.

— Ashleigh Brilliant

Contrast is what makes photography interesting.

— Conrad Hall

1 Introduction

The major contribution of Sutter's paper [P0515R2], "Consistent comparison," is the design and specification of a new C++ operator. Spelled **`<=>`**, it is formally termed the *three-way comparison* operator and colloquially known as the *spaceship* operator.

Although it is a core language feature, this new operator's behavior relies on new standard library components known as *comparison category types*. This paper provides standard library wording to specify those components and their (notional) underlying **`enum`**s,[1] together with some related objects, functions, and algorithms.

[1]Ideally, **`enum`**s alone would suffice. Alas, as Sutter's paper notes at the top of §3, "**`enum`**s don't currently support a way to express value conversion relationships [that are desired]."

Application of this new language feature in the context of the standard library is beyond the scope of the present paper. Only those facilities proposed by Sutter's paper are specified herein.

2 Comparison category types

In section 2.1, [P0515R2] proposes five *comparison category types*, each of which is a standard library type. Here are some of their salient features:

- **`weak_equality`** and **`strong_equality`** categorize/characterize the spaceship operator's result when a type permits only equality (**`==`**, **`!=`**) comparisons.
- **`strong_ordering`** and **`weak_ordering`** categorize/characterize the spaceship operator's result when a type permits all six comparison operators, among which exactly one of **`x < y`**, **`x == y`**, and **`x > y`** will be true.[2]
- **`partial_ordering`** categorizes/characterizes the spaceship operator's result when a type permits all six comparison operators, but none of **`x < y`**, **`x == y`**, and **`x > y`** need be true.
- The **`strong_`** and **`weak_`** comparison category types are distinguished by the *substitutability* property, namely, whether **`a == b`** implies **`f(a) == f(b)`**.[3]
- "Each [comparison category type] has predefined values, three numeric values for each **`_ordering`** and two for each **`_equality`**." Each call to a spaceship operator returns one of these values.
- Finally, there are selected implicit conversions among these comparison category types, as well as six named comparison functions taking an argument of comparison category type.[4]

Please see §4 below for the proposed detailed specifications of these and related components. For further design details, tutorial information, proposed core language wording, and a bibliography of recent WG21 papers that explored other approaches, please consult Sutter's paper.

3 Discussion

Following its review of Sutter's paper, LEWG in Toronto approved all the library components specified below. However, Sutter's paper does not recommend a name for the header in which the standard library will provide these components. Since all are in support of the comparison operator, we herein propose the header name **`<cmp>`**, a commonly-used short form that we find much easier to type than **`<comparison>`**, **`<compare>`**, **`<comparing>`**, **`<3way>`**, or **`<spaceship>`**.[5]

4 Proposed wording[6]

4.1 Insert, in alphabetical order, the following new entry into the *C++ library headers* table in subclause [headers]:

`<cmp>`

---

[2]In mathematics, this is known as the *trichotomy* property of an order relation. See, for example, the explanation at https://en.wikipedia.org/wiki/Trichotomy_(mathematics).

[3]This assumes that "**`f`** reads only comparison-salient state that is accessible using the public **`const`** members."

[4]These functions are intended for users who prefer to avoid writing **`a<=>b @ 0`**, where **`@`** denotes any of the six traditional comparison operators.

[5]We could, of course, also consider **`#include <=>`**. ☺

[6]Throughout this paper, all proposed additions are relative to [N4687], the post-Toronto Working Draft. Editorial notes are displayed against a gray background.

4.2 Insert the following new row into the *Language support library summary* table in subclause [support.general]:

21.9	Initializer lists	`<initializer_list>`
21.x	Comparisons	`<cmp>`
21.10	Other runtime support	`<csignal> <csetjmp> <cstdarg> <cstdlib>`

4.3 Insert the following new subclause after subclause [support.initlist] and before subclause [support.runtime]:

21.x Comparisons [cmp]

21.x.1 Header `<cmp>` synopsis [cmp.syn]

1 The header `<cmp>` specifies types, objects, and functions for use primarily in connection with the three-way comparison operator ([expr.spaceship]).

```
namespace std {
  // comparison category types
  class weak_equality;
  class strong_equality;
  class partial_ordering;
  class weak_ordering;
  class strong_ordering;

  // named comparison functions
  constexpr bool is_eq  (weak_equality    cmp) noexcept { return cmp == 0; }
  constexpr bool is_neq (weak_equality    cmp) noexcept { return cmp != 0; }
  constexpr bool is_lt  (partial_ordering cmp) noexcept { return cmp <  0; }
  constexpr bool is_lteq(partial_ordering cmp) noexcept { return cmp <= 0; }
  constexpr bool is_gt  (partial_ordering cmp) noexcept { return cmp >  0; }
  constexpr bool is_gteq(partial_ordering cmp) noexcept { return cmp >= 0; }

  // [cmp.common], common comparison category type
  template<class... Ts>
    struct common_comparison_category { using type = see below; };
  template<class... Ts>
    using common_comparison_category_t
      = typename common_comparison_category<Ts...>::type;

  // [cmp.alg], comparison algorithms
  template<class T, class U> auto compare_3way(const T& a, const U& b);

  template<InputIterator I1, InputIterator I2, class Cmp>
    auto lexicographical_compare_3way(I1 b1, I1 e1, I2 b2, I2 e2, Cmp comp)
    -> common_comparison_category_t<decltype(comp(*b1,*b2)), strong_ordering>;
  template<InputIterator I1, InputIterator I2>
    auto lexicographical_compare_3way(I1 b1, I1 e1, I2 b2, I2 e2);

  template<class T> strong_ordering  strong_order (const T& a, const T& b);
  template<class T> weak_ordering    weak_order   (const T& a, const T& b);
```

```
  template<class T> partial_ordering partial_order(const T& a, const T& b);
  template<class T> strong_equality  strong_equal (const T& a, const T& b);
  template<class T> weak_equality    weak_equal   (const T& a, const T& b);
}
```

21.x.2 Comparison category types [cmp.categories]

1 The **`_equality`** and **`_ordering`** types are collectively termed the *comparison category types*. Each is specified in terms of an exposition-only data member named **`value`** whose value typically corresponds to that of an enumerator from one of the following exposition-only enumerations:

```
enum class eq { equal = 0, equivalent = equal,
                nonequal = 1, nonequivalent = nonequal };
enum class ord { less = -1, greater = 1 };
enum class ncmp { unordered = -127 };
```

2 [*Note:* The types **`strong_ordering`** and **`weak_equality`** correspond, respectively, to the terms *total ordering* and *equivalence* in mathematics. — *end note*]

3 The comparison category types' relational and equality friend functions are specified with an anonymous parameter of *unspecified* type. This type shall be selected by the implementation such that these parameters can accept literal **`0`** as a corresponding argument. [*Example:* **`nullptr_t`** satisfies this requirement. — *end example*] In this context, the behavior of a program that supplies an argument other than a literal **`0`** is undefined.

4 For the purposes of this subclause, *substitutability* is the property that **`f(a) == f(b)`** is **`true`** whenever **`a == b`** is **`true`**, where **`f`** denotes a function that reads only comparison-salient state that is accessible via the argument's public **`const`** members.

21.x.2.1 Class `weak_equality` [cmp.weakeq]

1 The **`weak_equality`** type is typically used as the result type of a three-way comparison operator that (a) admits only equality and inequality comparisons, and (b) does not imply substitutability.

```
namespace std {
  class weak_equality {
    int value;  // exposition only

    // exposition-only constructor
    explicit constexpr weak_equality(eq v) noexcept : value(int(v)) {}

  public:
    // valid values
    static constexpr weak_equality equivalent   {eq::equivalent};
    static constexpr weak_equality nonequivalent{eq::nonequivalent};

    // comparisons
    friend constexpr bool operator==(weak_equality v, unspecified) noexcept;
    friend constexpr bool operator!=(weak_equality v, unspecified) noexcept;
    friend constexpr bool operator==(unspecified, weak_equality v) noexcept;
    friend constexpr bool operator!=(unspecified, weak_equality v) noexcept;
  };
}
```

```
constexpr bool operator==(weak_equality v, unspecified) noexcept;
constexpr bool operator==(unspecified, weak_equality v) noexcept;
```

2 *Returns:* `v.value == 0`.

```
constexpr bool operator!=(weak_equality v, unspecified) noexcept;
constexpr bool operator!=(unspecified, weak_equality v) noexcept;
```

3 *Returns:* `v.value != 0`.

21.x.2.2 `strong_equality` **[cmp.strongeq]**

1 The **`strong_equality`** type is typically used as the result type of a three-way comparison operator that (a) admits only equality and inequality comparisons, and (b) does imply substitutability.

```
namespace std {
  class strong_equality {
    int value;  // exposition only

    // exposition only constructor
    explicit constexpr strong_equality(eq v) noexcept : value(int(v)) {}

  public:
    // valid values
    static constexpr strong_equality equal        {eq::equal};
    static constexpr strong_equality nonequal     {eq::nonequal};
    static constexpr strong_equality equivalent   {eq::equivalent};
    static constexpr strong_equality nonequivalent{eq::nonequivalent};

    // conversion
    constexpr operator weak_equality() const noexcept;

    // comparisons
    friend constexpr bool operator==(strong_equality v, unspecified) noexcept;
    friend constexpr bool operator!=(strong_equality v, unspecified) noexcept;
    friend constexpr bool operator==(unspecified, strong_equality v) noexcept;
    friend constexpr bool operator!=(unspecified, strong_equality v) noexcept;
  };
}
```

```
constexpr operator weak_equality() const noexcept;
```

2 *Returns:* `*this == equal ? weak_equality::equivalent : weak_equality::nonequivalent`.

```
constexpr bool operator==(strong_equality v, unspecified) noexcept;
constexpr bool operator==(unspecified, strong_equality v) noexcept;
```

3 *Returns:* `v.value == 0`.

```
constexpr bool operator!=(strong_equality v, unspecified) noexcept;
constexpr bool operator!=(unspecified, strong_equality v) noexcept;
```

4 *Returns:* `v.value != 0`.

21.x.2.3 Class `partial_ordering` [cmp.partialord]

1 The **`partial_ordering`** type is typically used as the result type of a three-way comparison operator that (a) admits all of the six comparison operators, (b) does not imply substitutability, and (c) permits two values to be incomparable (i.e., **`a < b`**, **`a == b`**, and **`a > b`** might all be **`false`**).

```
namespace std {
  class partial_ordering {
    struct {
      int cmp
      bool is_ordered;
   } value;  // exposition only

    // exposition-only constructors
    explicit constexpr
      partial_ordering(eq   v) noexcept : value{int(v), true }  {}
    explicit constexpr
      partial_ordering(ord  v) noexcept : value{int(v), true }  {}
    explicit constexpr
      partial_ordering(ncmp v) noexcept : value{int(v), false}  {}

  public:
    // valid values
    static constexpr partial_ordering less      {ord::less};
    static constexpr partial_ordering equivalent{eq::equivalent};
    static constexpr partial_ordering greater   {ord::greater};
    static constexpr partial_ordering unordered {ncmp::unordered};

    // conversion
    constexpr operator weak_equality() const noexcept;

    // comparisons
    friend constexpr bool operator==(partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator!=(partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator< (partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator<=(partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator> (partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator>=(partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator==(unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator!=(unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator< (unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator<=(unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator> (unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator>=(unspecified, partial_ordering v) noexcept;
  };
}
```

`constexpr operator weak_equality() const noexcept;`

2 *Returns:* **`*this == equivalent ? weak_equality::equivalent : weak_equality::nonequivalent`**.

```
constexpr bool operator==(partial_ordering v, unspecified) noexcept;
constexpr bool operator< (partial_ordering v, unspecified) noexcept;
constexpr bool operator<=(partial_ordering v, unspecified) noexcept;
constexpr bool operator> (partial_ordering v, unspecified) noexcept;
constexpr bool operator>=(partial_ordering v, unspecified) noexcept;
```

3 *Returns:* `false` if `v.is_ordered` is `false`; otherwise, `operator@` returns `v.value.cmp @ 0`.

```
constexpr bool operator==(unspecified, partial_ordering v) noexcept;
constexpr bool operator< (unspecified, partial_ordering v) noexcept;
constexpr bool operator<=(unspecified, partial_ordering v) noexcept;
constexpr bool operator> (unspecified, partial_ordering v) noexcept;
constexpr bool operator>=(unspecified, partial_ordering v) noexcept;
```

4 *Returns:* `false` if `v.is_ordered` is `false`; otherwise, `operator@` returns `0 @ v.value.cmp`.

```
constexpr bool operator!=(partial_ordering v, unspecified) noexcept;
constexpr bool operator!=(unspecified, partial_ordering v) noexcept;
```

5 *Returns:* `true` if `v.is_ordered` is `false`; otherwise, returns `v.value.cmp != 0`.

21.x.2.4 Class `weak_ordering` [cmp.weakord]

1 The `weak_ordering` type is typically used as the result type of a three-way comparison operator that (a) admits all of the six comparison operators, and (b) does not imply substitutability.

```
namespace std {
  class weak_ordering {
    int value;  // exposition only

    // exposition-only constructors
    explicit constexpr weak_ordering(eq  v) noexcept : value(int(v)) {}
    explicit constexpr weak_ordering(ord v) noexcept : value(int(v)) {}

  public:
    // valid values
    static constexpr weak_ordering less      {ord::less};
    static constexpr weak_ordering equivalent{eq::equivalent};
    static constexpr weak_ordering greater   {ord::greater};

    // conversions
    constexpr operator weak_equality() const noexcept;
    constexpr operator partial_ordering() const noexcept;

    // comparisons
    friend constexpr bool operator==(weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator!=(weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator< (weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator<=(weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator> (weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator>=(weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator==(unspecified, weak_ordering v) noexcept;
    friend constexpr bool operator!=(unspecified, weak_ordering v) noexcept;
    friend constexpr bool operator< (unspecified, weak_ordering v) noexcept;
    friend constexpr bool operator<=(unspecified, weak_ordering v) noexcept;
```

```
    friend constexpr bool operator> (unspecified, weak_ordering v) noexcept;
    friend constexpr bool operator>=(unspecified, weak_ordering v) noexcept;
  };
}
```

```
constexpr operator weak_equality() const noexcept;
```

2 *Returns:* `*this == equivalent ? weak_equality::equivalent : weak_equality::nonequivalent`.

```
constexpr operator partial_ordering() const noexcept;
```

3 *Returns:* `*this == equivalent ? partial_ordering::equivalent : *this == less ? partial_ordering::less : partial_ordering::greater`.

```
constexpr bool operator==(weak_ordering v, unspecified) noexcept;
constexpr bool operator!=(weak_ordering v, unspecified) noexcept;
constexpr bool operator< (weak_ordering v, unspecified) noexcept;
constexpr bool operator<=(weak_ordering v, unspecified) noexcept;
constexpr bool operator> (weak_ordering v, unspecified) noexcept;
constexpr bool operator>=(weak_ordering v, unspecified) noexcept;
```

4 *Returns:* `v.value @ 0` for `operator@`.

```
constexpr bool operator==(unspecified, weak_ordering v) noexcept;
constexpr bool operator!=(unspecified, weak_ordering v) noexcept;
constexpr bool operator< (unspecified, weak_ordering v) noexcept;
constexpr bool operator<=(unspecified, weak_ordering v) noexcept;
constexpr bool operator> (unspecified, weak_ordering v) noexcept;
constexpr bool operator>=(unspecified, weak_ordering v) noexcept;
```

5 *Returns:* `0 @ v.value` for `operator@`.

21.x.2.5 Class `strong_ordering` [cmp.strongord]

1 The `strong_ordering` type is typically used as the result type of a three-way comparison operator that (a) admits all of the six comparison operators, and (b) does imply substitutability.

```
namespace std {
  class strong_ordering {
    int value;  // exposition only

    // exposition-only constructors
    explicit constexpr strong_ordering(eq  v) noexcept : value(int(v)) {}
    explicit constexpr strong_ordering(ord v) noexcept : value(int(v)) {}

    public:
    // valid values
    static constexpr strong_ordering less      {ord::less};
    static constexpr strong_ordering equal     {eq::equal};
    static constexpr strong_ordering equivalent{eq::equivalent};
    static constexpr strong_ordering greater   {ord::greater};

    // conversions
    constexpr operator weak_equality() const noexcept;
```

```
    constexpr operator strong_equality() const noexcept;
    constexpr operator partial_ordering() const noexcept;
    constexpr operator weak_ordering() const noexcept;

    // comparisons
    friend constexpr bool operator==(strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator!=(strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator< (strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator<=(strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator> (strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator>=(strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator==(unspecified, strong_ordering v) noexcept;
    friend constexpr bool operator!=(unspecified, strong_ordering v) noexcept;
    friend constexpr bool operator< (unspecified, strong_ordering v) noexcept;
    friend constexpr bool operator<=(unspecified, strong_ordering v) noexcept;
    friend constexpr bool operator> (unspecified, strong_ordering v) noexcept;
    friend constexpr bool operator>=(unspecified, strong_ordering v) noexcept;
  };
}
```

`constexpr operator weak_equality() const noexcept;`

2 *Returns:* `*this == equivalent ? weak_equality::equivalent : weak_equality::nonequivalent`.

`constexpr operator strong_equality() const noexcept;`

3 *Returns:* `*this == equal ? strong_equality::equal : strong_equality::nonequal`.

`constexpr operator partial_ordering() const noexcept;`

4 *Returns:* `*this == equivalent ? partial_ordering::equivalent : *this == less ? partial_ordering::less : partial_ordering::greater`.

`constexpr operator weak_ordering() const noexcept;`

5 *Returns:* `*this == equivalent ? weak_ordering::equivalent : *this == less ? weak_ordering::less : weak_ordering::greater`.

```
constexpr bool operator==(strong_ordering v, unspecified) noexcept;
constexpr bool operator!=(strong_ordering v, unspecified) noexcept;
constexpr bool operator< (strong_ordering v, unspecified) noexcept;
constexpr bool operator<=(strong_ordering v, unspecified) noexcept;
constexpr bool operator> (strong_ordering v, unspecified) noexcept;
constexpr bool operator>=(strong_ordering v, unspecified) noexcept;
```

6 *Returns:* `v.value @ 0` for `operator@`.

```
constexpr bool operator==(unspecified, strong_ordering v) noexcept;
constexpr bool operator!=(unspecified, strong_ordering v) noexcept;
constexpr bool operator< (unspecified, strong_ordering v) noexcept;
constexpr bool operator<=(unspecified, strong_ordering v) noexcept;
constexpr bool operator> (unspecified, strong_ordering v) noexcept;
constexpr bool operator>=(unspecified, strong_ordering v) noexcept;
```

7 *Returns:* `0 @ v.value` for `operator@`.

21.x.3 Class template `common_comparison_category` [cmp.common]

1 The type `common_comparison_category` provides an alias for the strongest comparison category that all of the template arguments can be converted to. [*Note:* A comparison category type is stronger than another if they are distinct types and an instance of the former can be converted to an instance of the latter. — *end note*]

```
template<class... Ts>
struct common_comparison_category { using type = see below; };
```

2 *Remarks:* The member *typedef-name* `type` shall denote the common comparison type ([class.spaceship]) of `Ts...`, the expanded parameter pack. [*Note:* This is well-defined even if the expansion is empty or includes a type that is not a comparison category type. — *end note*]

21.x.4 Comparison algorithms [cmp.alg]

1 For the purposes of this subclause, to carry out an action in a *memberwise* fashion means that the action is to be carried out, in the following order, on corresponding members of the given objects:

(1.1) — First, the direct base class subobjects, if any, in order of their declaration in the *base-specifier-list*.

(1.2) — Then, the non-static data members, if any, in the order of their declaration in the *member-specification*. Any subobject of array type is recursively expanded to the sequence of its elements, in the order of increasing subscript.

```
template<class T, class U> auto compare_3way(const T& a, const U& b);
```

2 *Effects:* Compares two values and produces a result of the strongest applicable comparison category type:

(2.1) — Returns `a <=> b` if that expression is well-formed.

(2.2) — Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to `bool`, returns:
(a) `strong_ordering::equal` when `a == b` is `true`,
(b) `strong_ordering::less` when `a < b` is `true`, or
(c) `strong_ordering::greater` when neither is `true`.

(2.3) — Otherwise, if the expression `a == b` is well-formed and convertible to `bool`, returns:
(a) `strong_equality::equal` when `a == b` is `true`, or
(b) `strong_equality::nonequal` when `a == b` is `false`.

(2.4) — Otherwise, if `is_same_v<T, U>` is `true`, let r_i denote the result, of type R_i, of the i^{th} call in a sequence of memberwise calls `compare_3way(a.m, b.m)` for each subobject `m` of `T`. Then let R denote the common comparison type ([class.spaceship]) of all R_i. Further, let r denote the first r_i whose result is not convertible to R_i`::equivalent` or, if there is no such r, let r instead denote `strong_ordering::equivalent`. Returns r converted to R.

(2.5) — Otherwise, the function shall be defined as deleted.

```
template<InputIterator I1, InputIterator I2, class Cmp>
  auto lexicographical_compare_3way(I1 b1, I1 e1, I2 b2, I2 e2, Cmp comp)
  -> common_comparison_category_t<decltype(comp(*b1,*b2)), strong_ordering>;
```

3 *Requires:* `Cmp` shall be a function object type whose return type is a comparison category type.

4 *Effects:* Lexicographically compares two ranges and produces a result of the strongest applicable comparison category type. Equivalent to:

```
for ( ; b1 != e1 && b2 != e2; ++b1, void(++b2) )
  if (auto cmp = comp(*b1,*b2); cmp != 0)
    return cmp;
return b1 != e1 ? strong_ordering::greater
     : b2 != e2 ? strong_ordering::less : strong_ordering::equal;
```

```
template<InputIterator I1, InputIterator I2>
  auto lexicographical_compare_3way(I1 b1, I1 e1, I2 b2, I2 e2)
```

5 *Returns:*
```
lexicographical_compare_3way(b1, e1, b2, e2,
                             compare_3way<decltype(*b1),decltype(*b2)> ).
```

```
template<class T> strong_ordering strong_order(const T& a, const T& b);
```

6 *Effects:* Compares two values and produces a result of type `strong_ordering`:

- (6.1) — 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.
- (6.2) — Otherwise, returns `a <=> b` if that expression is well-formed and convertible to `strong_ordering`.
- (6.3) — Otherwise, the function shall be defined as deleted.

```
template<class T> weak_ordering weak_order(const T& a, const T& b);
```

7 *Effects:* Compares two values and produces a result of type `weak_ordering`:

- (7.1) — Returns `a <=> b` if that expression is well-formed and convertible to `weak_ordering`.
- (7.2) — Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to `bool`, returns
  (a) `weak_ordering::equivalent` when `a == b` is `true`,
  (b) `weak_ordering::less` when `a < b` is `true`, or
  (c) `weak_ordering::greater` when neither expression is `true`.
- (7.3) — Otherwise, if it is well-formed to do so, calls `weak_order(a.m, b.m)` in a memberwise fashion for each subobject `m` of `T`. Let `r` denote the result of the first call whose result is not `weak_ordering::equivalent`. If there is such an `r`, returns it; otherwise, returns `weak_ordering::equivalent`.
- (7.4) — Otherwise, the function shall be defined as deleted.

```
template<class T> partial_ordering partial_order(const T& a, const T& b);
```

8 *Effects:* Compares two values and produces a result of type `partial_ordering`:

- (8.1) — If the expression `a <=> b` is well-formed and produces a result of a type convertible to `partial_ordering`, returns the result of evaluating that expression.
- (8.2) — Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to `bool`, returns

(a) `partial_ordering::equivalent` when `a == b` is `true`,
(b) `partial_ordering::less` when `a < b` is `true`, or
(c) `partial_ordering::greater` when neither expression is `true`.

(8.3) — Otherwise, if it is well-formed to do so, calls `partial_order(a.m, b.m)` in a memberwise fashion for each subobject `m` of `T`. Let `r` denote the result of the first call whose result is not `partial_ordering::equivalent`. If there is such an `r`, returns it; otherwise, returns `partial_ordering::equivalent`.

(8.4) — Otherwise, the function shall be defined as deleted.

```
template<class T> strong_equality strong_equal(const T& a, const T& b);
```

9 *Effects:* Compares two values and produces a result of type `strong_equality`:

(9.1) — Returns `a <=> b` if that expression is well-formed and convertible to `strong_equality`.

(9.2) — Otherwise, if it is well-formed to do so, calls `strong_equal(a.m, b.m)` in a memberwise fashion for each subobject `m` of `T`. Let `r` denote the result of the first call whose result is not `strong_equality::equal`. If there is such an `r`, returns it; otherwise, returns `strong_equality::equal`.

(9.3) — Otherwise, the function shall be defined as deleted.

```
template<class T> weak_equality weak_equal(const T& a, const T& b);
```

10 *Effects:* Compares two values and produces a result of type `weak_equality`:

(10.1) — If the expression `a <=> b` is well-formed and produces a result of a type convertible to `weak_equality`, returns the result of evaluating that expression.

(10.2) — Otherwise, if the expression `a == b` is well-formed and convertible to `bool`, returns
(a) `weak_equality::equivalent` when `a == b` is `true`, or
(b) `weak_equality::nonequivalent` when `a == b` is not `true`.

(10.3) — Otherwise, if it is well-formed to do so, calls `weak_equal(a.m, b.m)` in a memberwise fashion for each subobject `m` of `T`. Let `r` denote the result of the first call whose result is not `weak_equivalent::equal`. If there is such an `r`, returns it; otherwise, returns `weak_equivalent::equivalent`.

(10.4) — Otherwise, the function shall be defined as deleted.

4.4 Deprecate `rel_ops` as follows:

- Create a new subclause [rel_ops] in Annex D.
- Populate that new subclause with the following text as its initial paragraph, followed by the namespace `std::rel_ops` synopsis from [utility.syn], followed in turn by (suitably renumbered) paragraphs 1 through 9 from subclause [operators].
- Remove subclause [operators] and also remove the namespace `rel_ops` synopsis from [utility.syn].

1 The header `<utility>` has the following additions:

4.5 Preserve the normative intent of the original [operators]/10 as follows:

- Create a new subclause within [requirements].
- Populate that new subclause with the following text as its initial paragraph, followed by paragraphs 2 through 9 from the original subclause [operators]:

1 In this library, whenever a declaration is provided for an **`operator!=`**, **`operator>`**, **`operator>=`**, or **`operator<=`**, its requirements and semantics are as follows, unless explicitly specified otherwise.

5 Acknowledgments

Many thanks to the following gentlemen for their respective comments re this paper's technical content: Alisdair Meredith, Casey Carter, Herb Sutter, Jens Maurer, Stephan T. Lavavej, and Titus Winters.

6 Bibliography

[N4687] Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4687 (post-Toronto mailing), 2017–07–31. http://wg21.link/n4687.

[P0515R2] Herb Sutter, et al.: "Consistent comparison." ISO/IEC JTC1/SC22/WG21 document P0515R2 (pre-Albuquerque mailing), 2017–09–30. http://wg21.link/p0515r2.

7 Document history

Rev.	Date	Changes
0	2017-09-30	• Published as P0768R0, pre-Albuquerque.