Consistent comparison

Document Number: **P0515 R2**
Date: 2017-09-30
Authors: Herb Sutter, Jens Maurer, Walter E. Brown
Audience: EWG, CWG, LEWG, LWG

Abstract

This paper presents a design for comparisons that unifies the uncontroversial parts of previous proposals and follows EWG+LEWG direction in Issaquah 2016 and Kona 2017. The aim is to present a clean design that is:

complete by addressing all comparisons including three-way comparison, partial orderings, and symmetric heterogeneous comparisons;

correct by ensuring that all proposed defaults are sound;

as efficient as a programmer could write by hand; and

simple and teachable so that a type author just writes one function to opt into all comparisons.

It also includes from the outset notes about standard library use of the feature (see §2.2.4).

Contents

1 Overview

1.1 Background and motivation

Comparisons have been discussed in WG21 broadly since N3950 followed by N4126 (Smolsky) in 2014. We explored various approaches (see Bibliography), culminating in the development and rejection in Oulu 2016 of N4475 and N4476 (Stroustrup, motivation and discussion) and P0221R2 (Maurer, wording).

This proposal unifies and regularizes the noncontroversial parts of previous proposals, and incorporates EWG direction to pursue three-way comparison, letting default copying guide default comparison, and having a simple way to write a memberwise comparison function body. For a detailed side-by-side comparison with previous proposals, see revision R0 of this paper. For additional motivation and discussion, see P0100R2 (Crowl) and N4475 and N4476 (Stroustrup) which remain relevant even though parts of those papers have been superseded by subsequent discussions.

1.2 Design principles

Note These principles apply to all design efforts and aren't specific to this paper. Please steal and reuse.

The primary design goal is conceptual integrity [Brooks 1975], which means that the design is coherent and reliably does what the user expects it to do. Conceptual integrity's major supporting principles are:

- **Be consistent:** Don't make similar things different, including in spelling, behavior, or capability. Don't make different things appear similar when they have different behavior or capability. – For example, this paper follows the principle that by default `a=b` implies `a==b`, so that after copying a value, we can assert equality. Also, all types can get all the comparison operators they want by uniformly writing the same function, the three-way comparison operator `<=>`, and express the kind of comparison they support by the returned comparison category type (e.g., returning `strong_ordering` vs. `weak_ordering`).
- **Be orthogonal:** Avoid arbitrary coupling. Let features be used freely in combination. – For example, in this paper a type's comparison category is expressed orthogonally to the operators, by specifying a different category by just selecting a different return type on the same operator function. Especially, it makes all previously controversial design points into independent options that can later be proposed as pure extensions to this design and that do not affect this core proposal (see revision R0 of this proposal for treatment of the optional parts that did not receive support in EWG+LEWG Kona 2017)).
- **Be general:** Don't restrict what is inherent. Don't arbitrarily restrict a complete set of uses. Avoid special cases and partial features. – For example, this paper supports all seven comparison operators and operations, including adding three-way comparison via `<=>`. It also supports all five major comparison categories, including partial orders.

These also help satisfy the principles of least surprise and of including only what is essential, and result in features that are additive and so directly minimize concept count (and therefore also redundancy and clutter).

1.3 Acknowledgments

Thanks to all of the following recent comparison proposal authors for reviewing drafts of this paper: Walter Brown, Lawrence Crowl, Jens Maurer, Oleg Smolsky, David Stone, Bjarne Stroustrup, Tony Van Eerd.

Thanks also to the following for detailed comments on various drafts of this paper: Casey Carter, Gabriel Dos Reis, Vicente J. Botet Escriba, Hal Finkel, Charles-Henri Gros, Howard Hinnant, Loïc Joly, Nicolai Josuttis, Tomasz Kamiński, Andrzej Krzemieński, Alisdair Meredith, Patrice Roy, Mikhail Semenov, Richard Smith, Jeff Snyder, Peter Sommerlad, Daveed Vandevoorde, and Ville Voutilainen.

1.4 Proposal overview: Guidance and examples

The goal is to be simple enough to be teachable, while still enabling the most powerful and precise comparisons in any major programming language. In this proposal, we teach:

> There's a new three-way comparison operator, `<=>`. The expression `a <=> b` returns an object that compares `<0` if `a < b`, compares `>0` if `a > b`, and compares `==0` if `a` and `b` are equal/equivalent.
>
> **Common case:** To write all comparisons for your type `X` with type `Y`, with memberwise semantics, just write:
>
> ```
> auto X::operator<=>(const Y&) =default;
> ```
>
> **Advanced cases:** To write all comparisons for your type `X` with type `Y`, just write `operator<=>` that takes a `Y`, can use `=default` to get memberwise semantics if desired, and returns the appropriate category type:

- Return an `_ordering` if your type naturally supports `<`, and we'll efficiently generate symmetric `<`, `>`, `<=`, `>=`, `==`, and `!=`; otherwise return an `_equality`, and we'll efficiently generate symmetric `==` and `!=`.
- Return `strong_` if for your type `a == b` implies `f(a) == f(b)` (substitutability, where `f` reads only comparison-salient state that is accessible using the public `const` members), otherwise return `weak_`.

Expressing the same in table form:

Write an `operator<=>` that returns…		*Should `a < b` be supported?*	
		Yes: `_ordering`	*No: `_equality`*
Does `a == b` imply `f(a) == f(b)` (substitutability)?	*Yes: `strong`*	**`std::strong_ordering`**	**`std::strong_equality`**
	No: `weak`	**`std::weak_ordering`**	**`std::weak_equality`**

The design also supports returning `std::partial_ordering` which additionally permits `unordered` results.

> **Note** Normally, `operator<=>` should be just a member function; you will still get conversions on each parameter because of the symmetric generation rules in §2.3. In the rare case that you also want to support conversions on *both* parameters at the same time (to enabling compare two objects neither of which is of this type, but using this type's comparison function), make it a nonmember friend.

1.4.1 Example: Totally ordered comparison, memberwise

> **Note** Herein, "memberwise" is shorthand for "for each base or member subobject."

To get totally ordered memberwise comparison for our type, write `<=>` returning `strong_ordering`, with `=default` as the definition. Here is an example, where the C++17 code uses a non-member function as usual good style to enable conversions on each parameter, which is provided automatically in this proposal:

C++17 style

```
class Point {
  int x;
  int y;
public:
  friend bool operator==(const Point& a, const Point& b) { return a.x == b.x && a.y == b.y; }
  friend bool operator< (const Point& a, const Point& b) { return a.x < b.x
                                                               || (a.x == b.x && a.y < b.y); }
  friend bool operator!=(const Point& a, const Point& b) { return !(a==b); }
  friend bool operator<=(const Point& a, const Point& b) { return !(b<a); }
  friend bool operator> (const Point& a, const Point& b) { return b<a; }
```

This paper (proposed)

```
class Point {
  int x;
  int y;
public:
  auto operator<=>(const Point&) const =default;

  // ... non-comparison functions ...
};
```

```
  friend bool operator>=(const Point& a, const Point& b) { return !(a<b); }

  // ... non-comparison functions ...
};
```

`Point` supports all comparisons, all efficiently implemented as-if a single call to `<=>` and without creating another actual function:

```
Point pt1, pt2;
if (pt1 == pt2) { /*...*/ }     // ok

set<Point> s;                   // ok
s.insert(pt1);                  // ok

if (pt1 <= pt2) { /*...*/ }     // ok, single call to <=>
```

Note I say "as if" because, for example, for `pt1==pt2` it is expected that a quality implementation will invoke `==` on the two `int` members. See also §2.2.3 which describes the built-in `<=>` operators (e.g., `int <=> int`), whose semantics are known to the compiler as usual.

1.4.2 Example: Totally ordered type, custom comparison

To get a non-memberwise ordering, just write your own body instead of `=default`.

Consider this class, which uses a custom comparison because its members need to be compared in a different order than they can be lexically declared (let's say the `tax_id` had to be declared first for some reason, and for comparisons we want similar names bucketed while still falling back to a unique disambiguation by `tax_id`):

```
class TotallyOrdered : Base {
    string tax_id;
    string first_name;
    string last_name;

public:
    std::strong_ordering operator<=>(const TotallyOrdered& that) const {
        if (auto cmp = (Base&)(*this) <=> (Base&)that;     cmp != 0) return cmp;
        if (auto cmp = last_name      <=> that.last_name;  cmp != 0) return cmp;
        if (auto cmp = first_name     <=> that.first_name; cmp != 0) return cmp;
        return tax_id <=> that.tax_id;
    }

    // ... non-comparison functions ...
};
```

Notes If a member does not have a `strong_ordering`, we get a nice compile-time error. A major benefit to this paper's approach is that we can catch such semantic comparison bugs at compile time.

The most effective way to ensure that a user-defined `operator<=>` is a total order is to explicitly compare all data members and bases. Leaving out a data member runs the risk of failing to provide the substitutability property that `a==b` ⇒ `f(a)==f(b)`.

Comparing memberwise `<=>` against `0` is intentional to make the body agnostic to the comparison category of the individual data members (which could vary). This results in code that is both cleaner and more robust under maintenance if the data members' types and comparison categories may

change. Furthermore, in the next examples we will see that the body continues to be the same regardless of this type's own comparison category. This elegance will be explored further in §2.4.

In this proposal, code that uses `TotallyOrdered` can perform all comparisons including totally-ordered three-way comparison, and `<=` and the others are efficiently implemented as-if a single call to `<=>`:

```
TotallyOrdered to1, to2;
if (to1 == to2) { /*...*/ }     // ok

set<TotallyOrdered> s;          // ok
s.insert(to1);                  // ok

if (to1 <= to2) { /*...*/ }     // ok, single call to <=>
```

1.4.3 Example: Weakly ordered type, custom and heterogeneous comparison

To get a weak ordering, just return `weak_ordering`.

Note A type is weakly comparable when its `==` operator does not provide substitutability; that is, there exists a value-inspecting function such that `a==b` but `f(a)!=f(b)` (example shown below).

Consider this class, which uses a custom comparison because it compares one of its members differently from the member's own comparison:

```
class CaseInsensitiveString {
    string s;

public:
    std::weak_ordering operator<=>(const CaseInsensitiveString& b) const {
        return case_insensitive_compare(s.c_str(), b.s.c_str());
    }

    // ... non-comparison functions ...
};
```

In this proposal, code that uses `CaseInsensitiveString` can perform all comparisons including weakly-ordered three-way comparison, and `<=` and the others are efficiently implemented using a single `<=>`:

```
CaseInsensitiveString cis1, cis2;

if (cis1 == cis2) { /*...*/ }   // ok

set<CaseInsensitiveString> s;   // ok
s.insert(/*...*/);              // ok

if (cis1 <= cis2) { /*...*/ }   // ok, performs one comparison operation
```

To additionally provide symmetric heterogeneous comparisons with C-style `char*` strings, also provide `<=>` that takes `CaseInsensitiveString` and `char*`:

```
    std::weak_ordering operator<=>(const char* b) const {
        return case_insensitive_compare(s.c_str(), b);
    }
```

In this proposal, code that uses `CaseInsensitiveString` can additionally perform all `string`/`char*` and `char*`/`string` comparisons, and `<=` and the others are efficiently implemented using a single `<=>`:

```
if (cis1 <= “xyzzy”) { /*...*/ }      // ok, performs one comparison operation
if (“xyzzy” >= cis1) { /*...*/ }      // ok, identical semantics
```

Here is the full code for all 18 comparisons side by side with today’s code.

C++17 style, reusing an existing three-way comparison function which makes things shorter

```
class CaseInsensitiveString {
    string s;

public:
    friend bool operator==(const CaseInsensitiveString& a, const CaseInsensitiveString& b)
        { return case_insensitive_compare(a.s.c_str(), b.s.c_str()) == 0; }
    friend bool operator< (const CaseInsensitiveString& a, const CaseInsensitiveString& b)
        { return case_insensitive_compare(a.s.c_str(), b.s.c_str()) <  0; }
    friend bool operator!=(const CaseInsensitiveString& a, const CaseInsensitiveString& b) { return !(a==b); }
    friend bool operator<=(const CaseInsensitiveString& a, const CaseInsensitiveString& b) { return !(b<a); }
    friend bool operator> (const CaseInsensitiveString& a, const CaseInsensitiveString& b) { return b<a; }
    friend bool operator>=(const CaseInsensitiveString& a, const CaseInsensitiveString& b) { return !(a<b); }

    friend bool operator==(const CaseInsensitiveString& a, const char* b)
        { return case_insensitive_compare(a.s.c_str(), b) == 0; }
    friend bool operator< (const CaseInsensitiveString& a, const char* b)
        { return case_insensitive_compare(a.s.c_str(), b) <  0; }
    friend bool operator!=(const CaseInsensitiveString& a, const char* b) { return !(a==b); }
    friend bool operator<=(const CaseInsensitiveString& a, const char* b) { return !(b<a); }
    friend bool operator> (const CaseInsensitiveString& a, const char* b) { return b<a; }
    friend bool operator>=(const CaseInsensitiveString& a, const char* b) { return !(a<b); }

    friend bool operator==(const char* a, const CaseInsensitiveString& b)
        { return case_insensitive_compare(a, b.s.c_str()) == 0; }
    friend bool operator< (const char* a, const CaseInsensitiveString& b)
        { return case_insensitive_compare(a, b.s.c_str()) <  0; }
    friend bool operator!=(const char* a, const CaseInsensitiveString& b) { return !(a==b); }
    friend bool operator<=(const char* a, const CaseInsensitiveString& b) { return !(b<a); }
    friend bool operator> (const char* a, const CaseInsensitiveString& b) { return b<a; }
    friend bool operator>=(const char* a, const CaseInsensitiveString& b) { return !(a<b); }

    // ... non-comparison functions ...
};
```

This paper (proposed)

```
class CaseInsensitiveString {
    string s;
public:
    std::weak_ordering operator<=>(const CaseInsensitiveString& b) const
        { return case_insensitive_compare(s.c_str(), b.s.c_str()); }

    std::weak_ordering operator<=>(const char* b) const
        { return case_insensitive_compare(s.c_str(), b); }

    // ... non-comparison functions ...
```

```
};
```

1.4.4 Example: Partially ordered type, custom comparison

A class that is partially ordered should define an `operator<=>` that returns `partial_ordering`, and gets all the two-way comparisons as compiler-generated comparisons. The result can express that two objects are `unordered`, in which case all of the two-way comparisons return `false`.

Consider this class, whose ordering is topological:

```
class PersonInFamilyTree { // ...
public:
    std::partial_ordering operator<=>(const PersonInFamilyTree& that) const {
        if (this->is_the_same_person_as ( that)) return partial_ordering::equivalent;
        if (this->is_transitive_child_of( that)) return partial_ordering::less;
        if (that. is_transitive_child_of(*this)) return partial_ordering::greater;
        return partial_ordering::unordered;
    }

    // ... non-comparison functions ...
};
```

In this proposal, code that uses `PersonInFamilyTree` can perform all comparisons:

```
PersonInFamilyTree per1, per2;

if      (per1 == per2) { /*...*/ }     // ok, per1 is per2
else if (per1 <  per2) { /*...*/ }     // ok, per2 is an ancestor of per1
else if (per1 >  per2) { /*...*/ }     // ok, per1 is an ancestor of per2
else                   { /*...*/ }     // per1 and per2 are unrelated

if      (per1 <= per2) { /*...*/ }     // ok, per2 is per1 or an ancestor of per1

if      (per1 >= per2) { /*...*/ }     // ok, per1 is per2 or an ancestor of per2

if      (per1 != per2) { /*...*/ }     // ok, per1 is not per2
```

1.4.5 Example: Equality comparable type, custom comparison

A class that is equality comparable and has a custom ordering should define only an `operator<=>` that returns `strong_equality`, and gets `==` and `!=` as a compiler-generated comparisons. For example:

```
class EqualityComparable {
    string name;
    BigInt number1;
    BigInt number2;

public:
    strong_equality operator<=>(const EqualityComparable& that) const {
        if (auto cmp = number1 <=> that.number1; cmp != 0) return cmp;
        if (auto cmp = number2 <=> that.number2; cmp != 0) return cmp;
        return name <=> that.name;
    }
```

```
    // ... non-comparison functions ...
};
```

In this proposal, code that uses `EqualityComparable` can perform `==` or `!=` comparisons:

```
EqualityComparable ec1, ec2;
if (ec1 != ec2) { /*...*/ }     // ok
```

1.4.6 Example: Equivalence comparable type, custom comparison

A class that is equivalence comparable and has a custom ordering should define only an `operator<=>` that returns `weak_equality`, and gets `==` and `!=` as a compiler-generated comparisons.

> **Note** A comparable class is equivalence comparable when its `==` operator does not provide substitutability; that is, there exists a value-inspecting function such that `a==b` but `f(a)!=f(b)`.

Consider this class, where we write `operator<=>` by hand because we want to compare members in an order that is not the declaration order (let's say) and performs case-insensitive comparisons for `name`:

```
class EquivalenceComparable {
    CaseInsensitiveString name;
    BigInt number1;
    BigInt number2;

public:
    weak_equality operator<=>(const EquivalenceComparable& that) const {
        if (auto cmp = number1 <=> that.number1; cmp != 0) return cmp;
        if (auto cmp = number2 <=> that.number2; cmp != 0) return cmp;
        return name <=> that.name;
    }

    // ... non-comparison functions ...
};
```

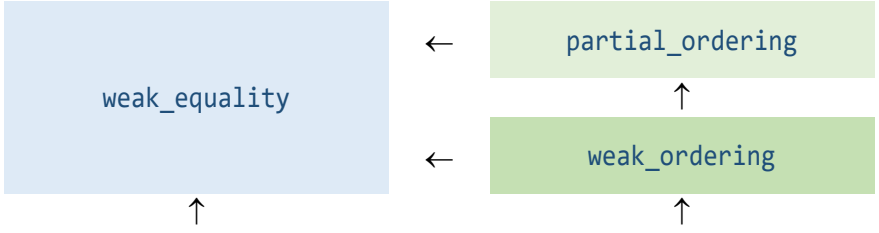
In this proposal, code that uses `EquivalenceComparable` can perform `==` or `!=` comparisons:

```
EquivalenceComparable ec1, ec2;
if (ec1 != ec2) { /*...*/ }     // ok
```

2 Design

2.1 Comparison categories

We define five comparison categories as `std::` types (see also §3). Arrows show "IS-A" implicit conversions.



Notes Making them types enables using the type system to guide generation of the correct appropriate related comparisons, and allows future standard algorithms to perform better checking.

I proposed these names instead of the standard mathematical terms instead, because I have found that these are simpler to teach. In earlier drafts I received questions like "are equivalence-comparison and weak-ordering related?" – the answer is that yes they are, and here both are consistently named `weak_` with exactly the same distinction from `strong_`; since I made this naming change, the question has not arisen again. LEWG agreed and approved the names (Kona 2017).

A `weak_equality` is not just the equality part of `weak_ordering`. There are ordering relationships that satisfy `weak_equality` but not `weak_ordering`. In particular, `weak_ordering` implies that the equality partitions are ordered. The `weak_equality` does not imply that.

Each has predefined values, three numeric values for each `_ordering` and two for each `_equality`. Additionally, `partial_ordering` can represent the value `unordered`, separately from the numeric values.

	Numeric values			
Category	`-1`	`0`	`+1`	Non-numeric values
`strong_ordering`	`less`	`equal`	`greater`	
`weak_ordering`	`less`	`equivalent`	`greater`	
`partial_ordering`	`less`	`equivalent`	`greater`	`unordered`
`strong_equality`		`equal`	`nonequal`	
`weak_equality`		`equivalent`	`nonequivalent`	

Note See §3 for sample implementation details. For example, `strong_*` types also support `*equivalent` for convenience when writing generic code, so that a template that can operate on any `_equality` can write its code to say equivalent regardless of the exact `_equality` type.

We define implicit conversions among these following "IS-A":

- `strong_ordering` with values {`less`, `equal`, `greater`} implicitly converts to:
  - `weak_ordering` with values {`less`, `equivalent`, `greater`} (i.e., keep same values)
  - `partial_ordering` with values {`less`, `equivalent`, `greater`} (i.e., keep same values)
  - `strong_equality` with values {`unequal`, `equal`, `unequal`} (i.e., apply `abs()`)
  - `weak_equality` with values {`nonequivalent`, `equivalent`, `nonequivalent`} (i.e., apply `abs()`)
- `weak_ordering` with values {`less`, `equivalent`, `greater`} implicitly converts to:
  - `partial_ordering` with values {`less`, `equivalent`, `greater`} (i.e., keep same values)
  - `weak_equality` with values {`nonequivalent`, `equivalent`, `nonequivalent`} (i.e., apply `abs()`)
- `partial_ordering` with values {`less`, `equivalent`, `greater`, `unordered`} implicitly converts to:
  - `weak_equality` with values {`nonequivalent`, `equivalent`, `nonequivalent`, `nonequivalent`} (i.e., `unordered` or apply `abs()`)
- `strong_equality` with values {`equal`, `unequal`} implicitly converts to:
  - `weak_equality` with values { `equivalent`, `nonequivalent`} (i.e., keep same values)

Notes Astute readers will have noticed that the examples in §1.4.3 through §1.4.6 rely on these conversions.

This aims to hit a "teachable" sweet spot, that is both mathematically powerful but hides that power except when you really want it; see §1.4.

To compute the strongest common comparison category type, we provide:

```
template <class ...Ts>
struct common_comparison_category {
    using type = /* as specified below */ ;
};

template <class ...Ts>
using common_comparison_category_t
    = typename common_comparison_category<Ts>::type;
```

where `::type` is computed as follows from `Ts`:

- If `Ts` is empty, `::type` is `strong_ordering`.
- Otherwise, if each T_i supports `<=>` returning type Cmp_i, `::type` is the strongest category type that all Cmp_i can be converted to.
- Otherwise, `C` is `void`.

2.2 Three-way comparison `<=>`

2.2.1 `<=>` token

We introduce one new token, `<=>`.

Notes Tokenization follows max munch as usual.

Code that uses the source character sequence `<=>` today tokenizes to `<= >`. The only examples I know of where that sequence can legally occur is when using the address of `operator<=` to instantiate a template (e.g., `X<&Y::operator<=>`) or as the left-hand operand of a `>` comparison (e.g., `x+&operator<=>y`). Under this proposal such existing code would need to add a space to retain its current meaning. This is the only known backward source incompatibility, and it is intentional. (We could adopt a special parsing rule to keep such code working without a space, but I would discourage that as having too little benefit for the trouble.)

2.2.2 `operator<=>`

We introduce one new overloadable operator, `<=>`, often called the "spaceship operator" in other languages.

`operator<=>` is a generalized three-way comparison function and has precedence higher than `<` and lower than `<<`. It normally returns a type that can be compared against literal `0`, but other return types are allowed such as to support expression templates. All `<=>` operators defined in the language and standard library return one of the `std::` comparison category types (see §2.1).

Notes **Homogenous vs. heterogeneous:** There is no restriction on parameter types. They can be the same (homogeneous) or different (heterogeneous) in which case we generate symmetric operations; see §2.3).

`<=>` is for type implementers: User code (including generic code) outside the implementation of an `operator<=>` should almost never invoke an `<=>` directly (as already discovered as a good practice in other languages); for example, code that wants to test `a<b` should just write that, not `a<=>b < 0`. See also related notes in §2.6 regarding library compatibility.

Operator vs named function: I prefer a new operator for `<=>`, instead of a named function such as `compare`, for two main reasons: (1) It is symmetric with `==`, `!=`, `<`, `<=`, `>`, and `>=`. (2) It avoids collision on a common name, because this name will be widely used and so will encounter the usual pressure to make it ugly to minimize conflicts when mixing this with existing code (e.g., using existing classes as base classes). Also, it follows existing practice in other languages.

Precedence: I considered giving `<=>` the same precedence as `<`. However, then we would have a situation where `a<=>b @ 0` would correctly evaluate `<=>` first when `@` is any comparison operator, but writing `0 @ a<=>b` would have the inconsistent behavior that `<=>` would be evaluated first when `@` is in {`==`, `!=`} but evaluated second when `@` is in {`<`, `>`, `<=`, `>=`}. To make `a<=>b @ 0` and `0 @ a<=>b` consistent, `<=>` should have (slightly) higher precedence than `<`.

Return value comparable to 0, but not convertible to int: I prefer allowing the return value to numerically compare to `0` (only), but not convert to an integer, for three main reasons:

(1) ***Follows all existing practice ('the best parts'):*** The majority of existing practice for three-way comparison returns a signed integer:

C	`strcmp`, `memcmp`, `qsort`
C#	`IComparable.CompareTo` (since 1.1), `Comparison<T>` (since 2.0)
Java	`Comparable.compareTo` (since J2SE 1.2)
Groovy	`<=>` (delegates to `compareTo`)
OCaml	`compare`
Perl	`<=>`
PHP	`<=>` (since PHP 7)
Python	`cmp`
Ruby	`<=>`

Of the major languages, only Haskell returns a type (an enumeration). This proposal does both: Like Haskell it leverages the type system by returning a type (and, more than Haskell, uses that type to distinguish the comparison category), and like the vast majority of existing practice it returns a value that can be compared against `0` (as the embodiment of the mapping to the two-way comparisons).

(2) ***Consistency:*** It permits expressing all other comparisons `a @ b` as `a<=>b @ 0` and `0 @ b<=>a`:

source code:	default rewrite generation does:	and also this to support `T2 @ T1`:
`a @ b`	`a<=>b @ 0`	`0 @ b<=>a`
`a == b`	`a<=>b == 0`	`0 == b<=>a`
`a != b`	`a<=>b != 0`	`0 != b<=>a`
`a < b`	`a<=>b < 0`	`0 < b<=>a`
`a <= b`	`a<=>b <= 0`	`0 <= b<=>a`

```
a >  b       a<=>b >  0       0 >  b<=>a
a >= b       a<=>b >= 0       0 >= b<=>a
```

(3) ***Efficiency:*** It avoids closing the door to returning a signed value with magnitude greater than `1` to preserve additional information that would otherwise be thrown away, such as the distance between `a` and `b` if such information is computed while computing `<`. Many of the just-listed examples of existing practice specify a non-equal return value's sign, rather than requiring a non-equal return to be exactly `-1` or `+1`.

Basing everything on `<=>` and its return type: This model has major advantages, some unique to this proposal compared to previous proposals for C++ and the capabilities of other languages:

(1) ***Clean tagging:*** It implicitly "tags" types' ordering, without resorting to separate tag traits; code can query a type's ordering by just querying the return type of `<=>`. Only this proposal and P0100R2 (Crowl) give the standard library the option of enabling its algorithms to check, and even overload based on, the ordering of the type being supplied.

(2) ***Uniform tagging for user-defined and fundamental types:*** In the following section we provide `<=>` for fundamental types, including that for the first time we have a model that tags the fundamental types' ordering (see §2.2.3). Any comparison queries/overloading on `<=>`'s return type that STL and other algorithms might perform work uniformly across all comparable types.

(3) ***Consistency:*** As noted, it consistently gives `a @ b` the *default* meaning `a<=>b @ 0` (i.e., the default for those `@` compiler-generated by rewrite; see §2.3) Then the only difference among the comparison categories is "which" operators get compiler-generated, not what they mean if compiler-generated, which seems correct and is much cleaner.

(4) ***Powerful expressiveness for all comparison categories:*** This is the only proposal other than P0100R2 (Crowl) that is designed to give direct support for partial orders, and the first proposal that actually regularizes partial ordering into the operators, that is, bringing partial ordering support also to the language operators instead of doing something asymmetric such as resorting to a named function.

(5) ***Simplicity:*** It regularizes what we tell type authors to write for comparisons, namely "just write one function, `<=>`, returning the appropriate comparison category type."

(6) ***Efficiency, including finally achieving zero-overhead abstraction for comparisons:*** The vast majority of comparisons are always single-pass. The only exception is generated `<=` and `>=` in the case of types that support both partial ordering and equality. For `<`, single-pass is essential to achieve the zero-overhead principle to avoid repeating equality comparisons, such as for `struct Employee { string name; /*more members*/ };` used in `struct Outer { Employee e; /*more members*/ };` – today's comparisons violates zero-overhead abstraction because `operator<` on `Outer` performs redundant equality comparisons, because it performs `if (e != that.e) return e < that.e;` which traverses the equal prefix of `e.name` twice (and if the name is equal, traverses the equal prefixes of other members of `Employee` twice as well), and this cannot be optimized away in general. As Kamiński notes, zero-overhead abstraction is a pillar of C++, and achieving it for comparisons for the first time is a significant advantage of this design based on `<=>`.

Branches: I deliberately do not propose a three-way branch language extension, because that capability should just fall out of future general pattern matching rather than being baroquely hardwired

into `if`. As a historical example, Fortran had a three-way arithmetic `IF` and then later deprecated it. However, Crowl notes that `switch` works fine with a comparison operator that returns an object whose value is exactly one of {`-1`, `0`, `1`}.

2.2.3 Language types and `operator<=>`

We additionally provide the following built-in `<=>` comparisons. All are `constexpr`. Except where otherwise noted below, these are homogeneous (same-type) comparisons only, and cannot be invoked heterogeneously using scalar promotions/conversions.

- For fundamental `bool`, integral, and pointer types, `<=>` returns `strong_ordering`.
- For pointer types, the different cv-qualifications and derived-to-base conversions are allowed to invoke a homogeneous built-in `<=>`, and there is a built-in heterogeneous `operator<=>(T*, nullptr_t)`. Also, we distinguish that comparisons of pointers into the same object/allocation are constant expressions, and otherwise are not constant expressions.
- For fundamental floating point types, `<=>` returns `partial_ordering`, and can be invoked heterogeneously by widening arguments to a larger floating point type.
- For enumerations, `<=>` returns the same as the enumeration's underlying type's `<=>`. If there is more than one enumerator with the same value (where "same value" is determined using the same rules as `switch` statements and non-type template arguments), which means substitutability does not hold, then if the type is `strong_ordering` adjust it to `weak_ordering`, and if it is `strong_equality` adjust it to `weak_equality`.
- For `nullptr_t`, `<=>` returns `strong_ordering` and always yields `equal`.
- For copyable arrays `T[N]` (i.e., that are nonstatic data members), `T[N] <=> T[N]` returns the same type as `T`'s `<=>` and performs lexicographical elementwise comparison. For other arrays, there is no `<=>` because the arrays are not copyable.
- For `void`, there is no `<=>` because objects of type `void` are not allowed.

Notes For **fundamental types except `int`**, optionally the core language could consider removing the definitions of the existing comparisons to let those comparisons just be compiler-generated.

For integral types, the implementation can make use of as-if but should be sound. For example, transforming `a <=> b` to use `a - b` typically does not work due to undefined behavior in the event of overflow.

For **character types**, which are "integral" types too, I'd love not to add `<=>`, but I think that ship has sailed in the current language. The argument is that, because a character should not be an integer ("`char`" and "`int` of size `1`" should be distinct types if we had a time machine), the character fundamental types shouldn't have been given arithmetic operations, and so it would be nice not to promote that further by adding another. However, they are arithmetic types, so we should just be consistent and provide `<=>` too to avoid creating needless user surprise.

For **raw pointers** we have two choices: (a) give them `<=>` that returns `strong_ordering` (which deliberately ignores segmented architectures) and is not `constexpr` (because even with flat memory it is not possible to give a total ordering over pointers that is the same at compile time and run time, due to compile/link/load time phases), or (b) don't give them `<=>` at all. It would be wrong to give raw pointers a `<=>` that returns something weaker than `strong_ordering`, because the issue with segmented pointers applies to all six comparisons equally. – For now, I'm going with `strong_ordering`, on the basis of maintaining a strict parallel between default copying and default

comparison (we copy raw pointer members, so we should compare them too unless there is a good reason to do otherwise – but the only such "otherwise" reason I know of would be if we want to explicitly keep the door open for segmented architectures). Also, the standard library smart pointers support all comparison operators with a total ordering. – If we decide to provide `<=>` for raw pointers, we can leave the existing two-way comparisons defined the way they are today, or make them a total ordering as well, but that is an independent choice. If we decide not to provide `<=>` for raw pointers, however, then we should provide a `std::strong_order` for raw pointers too (see §2.5), and use it in the default `<=>` comparison for a type's pointer members.

For **floating point types**, we use `partial_ordering` which supports both signed zero and `NaN`s, with the usual semantics that `-0 <=> +0` returns `equivalent` and `NaN <=> anything` returns `unordered`.

For **enumeration types**, the default ordering is never `partial_ordering`.

For **nullptr_t**, since we always return `equal` we could also just return `strong_equality`, but returning `strong_ordering` is usable in more contexts.

For **arrays**, we don't provide comparison if the array is not copyable in the language, to keep copying and comparison consistent. Note that for two arrays, `arr1<=>arr2` is ill-formed because the array-to-pointer conversion is not applied.

Making **scalar comparisons homogeneous** without promotions/conversions avoids bugs like this:

```
unsigned int i = 1;
return -1 < i;            // existing pitfall: returns 'false'
                          // -1 <=> i should not repeat this mistake
```

2.2.4 Standard library types and `operator<=>`

For each standard library type that already supports comparison, provide a nonmember `operator<=>` comparison that returns the appropriate comparison category type and yields consistent results with the operators already specified.

Notes Some `std::` types, such as `pair`, can use the `=default` definition to get memberwise comparison (as shown in §1.4.1).

Some `std::` types, such as containers (including `string`), `string_view`, `optional`, `any`, `unique_ptr`, and `shared_ptr`, would both declare and define a custom `operator<=>` to get custom semantics. (Note that `string` and `string_view` in particular would gain an equivalent to `strcmp`, and one that is superior because unlike `strcmp` it would respect embedded nulls.)

Some `std::` types, such as `complex`, would have `operator<=>` return a weaker comparison category, such as an `_equality`.

The `std::` product types, such as `pair` and `tuple`, should have `operator<=>` adapt its comparison category return type: Because the current types support all six existing comparison operators, with the semantics that `<=` et al. are generated from `<` and `!` (not `<` and `==`), this means the current specification assumes at least a `weak_ordering`. Therefore, if the library wants to preserve backward source compatibility, it should write elementwise `<=>` that returns `strong_ordering` if all element types support that and `weak_ordering` otherwise.

The standard library should review each "unspecified" and "implementation-defined" type, such as `jmp_buf`, to determine whether comparisons needs to be specified explicitly for those types, or follow naturally from the kinds of types permitted.

We should consider providing `<=>` for random-access iterators, or at least contiguous iterators.

We should consider providing heterogeneous `<=>` for `std::chrono` durations.

Optionally, the existing comparison functions for standard library types could be removed to let those comparisons just be compiler-generated. This would be a binary breaking change, however, unless implementations were given latitude to continue providing all the operators; and it would be a source breaking change for programs that take the address of a comparison operator.

This proposal does not currently attempt to list the recommended standard library changes, but could do so.

2.3 Generating two-way comparisons: Rewrite

For a comparison expression `a @ b` where `@` is a two-way comparison operator, perform name lookup for `a @ b`, `a <=> b`, and `b <=> a`. For each potential candidate `<=>` found, include it in overload resolution if any of the following are true:

- `<=>` returns `std::*_ordering` and `@` is one of `== != < > <= >=`
- `<=>` returns `std::*_equality` and `@` is one of `== !=`

Then select the best match using normal overload resolution rules, with the addition that if `a @ b`, `a <=> b`, and/or `b <=> a` are ambiguous, as a final tie-break we prefer `a @ b` over `a <=> b` over `b <=> a`.

Then:

- If `a <=> b` is the best match, then rewrite `a @ b` to `a<=>b @ 0`.
- If `b <=> a` is the best match, then rewrite `a @ b` to `0 @ b<=>a`.

Notes We try both `a <=> b` and `b <=> a` so that a member or heterogeneous comparison (e.g., `operator<=>(const string&, const char*)`) preserves symmetry, so that users don't have to needlessly remember to write both versions and leave another common source of error.

For a user-declared `operator<=>`, we look up both `@` and `<=>` (e.g., `<` and `<=>`) and let overload resolution pick the better match. That favors getting the version that best matches the types even if there is a more specific operator.

2.4 `=default`

To avoid having to write out the memberwise comparison cascade, a user-declared comparison operator can opt into memberwise comparison by default using `=default`, with the following semantics:

- If the function is `<=>`, the parameter types must be the same, the return type must be one of the `std::` comparison types, and the default body performs lexicographical comparison by successively comparing the base (left-to-right depth-first) and then member (in declaration order) subobjects of `T` to compute `<=>`, stopping early when a not-equal result is found, that is:

```
for each base or member subobject o of T
    if (auto cmp = lhs.o <=> rhs.o; cmp != 0) return cmp;
```

```
        return strong_ordering::equal;   // converts to everything
```

- If the function is `<`, `>`, `<=`, `>=`, `==`, or `!=`, the parameter types must be the same, the return type must be `bool`, and the default body is the same as applying the rewrite rules in §2.3 and invoking the corresponding `<=>`.

Note `=default` for two-way operators is useful to conveniently force the creation of functions whose addresses can be taken.

Defaulting `<=>` in particular is useful to opt in to memberwise comparisons easily. For example:

```
class MyClass {
    // ... possibly many members here, taking lots of code to compare ...

    /*...*/ operator<=>(const MyClass&) const = default;
};
```

Notes We could restrict `=default` to respect accessibility; that is, prevent a nonmember nonfriend comparison function from using an `=default` definition on a class with nonpublic data members.

As illustrated by the examples in §1.4, in this design, when opting into comparisons the programmer writes `operator<=>` and the body usually takes the following form, when the members to be compared are directly held data members:

```
class MyClass {  // ...

    /*category*/ operator<=>(/*...*/) {
        if (auto cmp = lhs.member1 <=> rhs.member1; cmp != 0) return cmp;
        if (auto cmp = lhs.member2 <=> rhs.member2; cmp != 0) return cmp;
        // ... etc. ...
        return         lhs.memberN <=> rhs.memberN;
    }

};
```

Given that everything except the comparison category is now regularized, it is tempting to provide a shorthand syntax for the entire function that just lists the members to be compared in order:

```
/*category*/ operator<=>(const MyClass& that) const = default(member1,...,memberN);
```

However, I'm not adding such a novelty to the proposal unless EWG tells me to (and picks a syntax).

Note that this could alternatively be enabled via `std::tie` as follows, at least for homogenous comparison categories:

```
/*category*/ operator<=>(const MyClass& that) const {
    return std::tie(member1, ..., memberN) <=>
           std::tie(that.member1, ..., that.memberN);
}
```

A defaulted `operator<=>` is implicitly deleted and returns `void` if not all base and member subobjects have a compiler-generated or user-declared `operator<=>` declared in their scope (i.e., as a nonstatic member or as a friend) whose result is one of the `std::` comparison category types.

A homogeneous defaulted `operator<=>` may have a return type of `auto`, in which case the return type is `std::common_comparison_category_t<Ms>` where `Ms` is the list (possibly empty) of base and member subobject types. This makes it easier to write cases where the return type non-trivially depends on the members, such as:

```
template<class T1, class T2>
struct P {
    T1 x1;
    T2 x2;
    auto operator<=>(const P&, const P&) = default;
};
```

2.5 Named comparison functions and algorithms

Following the example of the `<` operator and `std::less`, in addition to the operators we provide the following standard comparison function templates in header `<functional>` that default to using `<=>` if available, and can be customized by user-defined types (by specialization or ADL overload).

Notes These are intended to be implemented using reflection if available, compiler support otherwise.

See wording paper P0768 for the fully detailed specification, including details such as the division of functions like `lexicographical_compare_3way` into overloads to handle what is shown here as a default argument, final application of `constexpr`, `noexcept`, etc. as decided by LEWG and LWG, and similar tidying details. P0768 implements the design in this paper without conceptual design changes, but this paper is not attempting to duplicate the details of P0768 as it progresses and the wording is tweaked by LWG.

The names below are as twice seen by LEWG, but since LEWG review in Kona and Toronto some have expressed concern about the names of the algorithms and the comparison category types being too similar, e.g., `weak_ordering` and `weak_order()`. I'm open to changing them, but I would prefer not to do so because this similarity is by design, and I don't expect confusion between the two to arise because: (a) one is a type and one is a function so using the wrong name gives a clear message, and (b) most users will use the comparison category types only (and then only as the return type of `operator<=>`) whereas the algorithms are expected to see only rare use by library writers building algorithms.

```
template<class T, class Comparison>
constexpr bool can_3compare_as() {      // to save typing
    { return std::is_convertible_v<decltype(compare_3way(declval<T>(), declval<T>())), Comparison)>; }

template<class T>
std::strong_ordering strong_order(const T& a, const T&b) {
    if constexpr (can_3compare_as<T, std::strong_ordering>()) return a <=> b;
    else return void();
}

template<class T>
std::weak_ordering weak_order(const T& a, const T&b) {
    if constexpr (can_3compare_as<T, std::weak_ordering>()) return a <=> b;
    else if constexpr (/* can invoke a<b and a==b */) return a==b ? weak_ordering::equivalent :
                                                             a<b ? weak_ordering::less : weak_ordering::greater;
```

```
    else if constexpr (/* can invoke weak_order(a.M, b.M) for each member M of T */) /* do that */;
}

template<class T>
std::partial_ordering partial_order(const T& a, const T&b) {
    if constexpr (can_3compare_as<T, std::partial_ordering>()) return a <=> b;
    else if constexpr (/* can invoke a<b and a==b */) return a==b ? partial_ordering::equivalent :
                                                             a<b ? partial_ordering::less : partial_ordering::greater;
    // --- no fallback here to only a < b, existing operator< usually tries to express a weak order
    else if constexpr (/* can invoke partial_order(a.M, b.M) for each member M of T */) /* do that */;
}

template<class T>
std::strong_equality strong_equal(const T& a, const T&b) {
    if constexpr (can_3compare_as<T, std::strong_equality>()) return a <=> b;
    else if constexpr (/* can invoke strong_equal(a.M, b.M) for each member M of T */) /* do that */;
}

template<class T>
std::weak_equality weak_equal(const T& a, const T&b) {
    if constexpr (can_3compare_as<T, std::weak_equality>()) return a <=> b;
    else if constexpr (/* can invoke a==b */) return a == b ? weak_equality::equivalent : weak_equality::nonequivalent;
    else if constexpr (/* can invoke weak_equal(a.M, b.M) for each member M of T */) /* do that */;
}
```

Note that the user never has to explicitly specialize or overload these, except in exactly the cases where they want to provide something that is explicitly different and inconsistent with the operators. For example, a type that has a non-total order (e.g., `struct F { float f; };`) would not get `std::strong_order` by default, but could customize `std::strong_order` if wants to opt into an imposed total ordering, thereby simultaneously providing the total ordering and also documenting that it is "something different" from the operators.

We also provide a `std::strong_order<F>` specialization for (only) each fundamental floating point type `F` for which `std::numeric_limits<F>::is_iec559` is `true`, that implements the IEEE `totalOrder` operation:

```
template<class T, std::enable_if_t<std::numeric_limits<T>::is_iec559, int> = 0>
std::strong_ordering strong_order(const T& a, const T&b) { return /* IEEE totalOrder */; }
```

Note It is deliberate that for floating point types `<=>` is a partial ordering to be consistent with the existing two-way comparison operators, but total ordering is available as a named function. A total ordering should not be provided through the comparison operators because the operators (including now `<=>`) should be fully compatible with the existing comparison operator semantics for floating point types, and because imposing the ordering incurs at least minimal overhead. However, having a total ordering available (outside the operators) is desirable because it makes floating point types work as intended with STL containers and algorithms.

For cases like implementing `optional<T>` on existing types that do not have `<=>`, we need a function that will give the strongest ordering available for a given type `T`:

```
template<class T, class U>
auto compare_3way(const T& a, const U& b) {
    if constexpr (/* can invoke a <=> b */)
        return a <=> b;
```

```
    else if constexpr (/* can invoke a<b and a==b */)
        return a==b ? strong_ordering::equal : a<b ? strong_ordering::less : strong_ordering::greater;
    else if constexpr (/* can invoke a==b */)
        return a == b ? strong_equality::equal : strong_equality::unequal;
    else if constexpr (is_same_v<T,U> && /* can invoke a.M <=> b.M for each member M of T */) /* do that */;
}
```

We also provide a comparison algorithm, where the default argument shown can be provided as-if by overload:

```
auto lexicographical_compare_3way(
    InputIterator b1, InputIterator e1,
    InputIterator b2, InputIterator e2,
    Comparison comp /* = [](auto l, auto r) { return l <=> r; } */
) {
    for ( ; b1!=e1 && b2 != e2; ++b1, ++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;
}
```

Additionally, although most user code will not use `<=>` directly (see note in §2.2.2), for code that does some have expressed a preference for having an operation to write a named function rather than `a<=>b @ 0`:

```
bool is_eq   (std::weak_equality    cmp) { return cmp == 0; };
bool is_neq  (std::weak_equality    cmp) { return cmp != 0; };
bool is_lt   (std::partial_ordering cmp) { return cmp <  0; };
bool is_lteq (std::partial_ordering cmp) { return cmp <= 0; };
bool is_gt   (std::partial_ordering cmp) { return cmp >  0; };
bool is_gteq (std::partial_ordering cmp) { return cmp >= 0; };
```

2.6 Library compatibility

This is seamlessly compatible with C++17 code including all of STL, because the new comparison is merely a unified way of generating efficient and consistent versions of all the existing comparisons. Nearly all new calling code, and all existing C++17 calling code, uses the existing two-way comparison operators. Existing code that uses distinct-style types transparently gains the performance and semantic benefits without any change; the caller is not aware of `<=>` and so never calls it directly, and the ordinary comparisons generated by default have consistent and efficient implementations. Additionally, new calling code that wants to use the new three-way comparison can call it directly on the types that support it.

This subsumes namespace `std::rel_ops`, so we propose also removing (or deprecating) `std::rel_ops`.

Note For total orders, which are common, this is important for both clarity and efficiency, because it avoids losing information and repeating computation. It also has long-standing precedent in the C standard library `strcmp`/`qsort`, and all other major languages. See Crowl's analysis in P0100R2.

The C language comparison operators and the STL library focus on `<` and `==` as the primitive or fundamental comparison operations. This works but has some drawbacks:

(1) Clarity: This design leads to simpler defaults. In STL `relops`, to get all six comparison functions, the type author must write two functions, `<` and `==`, from which the rest are generated (and with the foregoing caveats). In this proposal, the type author must write only one, `<=>`, and the other six are generated (and with optimal efficiency).

(2) Correctness: This design avoids semantic pitfalls. In STL, the major pitfall is taught as "equality vs. equivalence": For many types, the type author must remember to implement consistent comparisons where `a==b` gives the same result as `!(a<b) && !(b<a)`. Type users must be aware whether a given type is consistent, and can encounter pitfalls when using containers or algorithms where some use equality and others use equivalence (e.g., switching between `unordered_map` and `map`, or between `find` and `lower_bound`, respectively). Another example is Lawrence Crowl's suggested optimization for special-casing the comparison of a single-element struct (to preserve the same behavior and efficiency as if the element were not in a struct), which here falls out for the single-element struct case.

(3) Performance (algorithmic): This design avoids inefficiencies by not throwing away common work. For example, in STL `relops`, expressing `<=` in terms of `==` and `<` requires making two function calls, and the functions typically repeat work because each throws away work the other could know about; this leaves a performance incentive to write `<=` and the others manually. The usual problem arises from compound operations like `<=` and common prefixes, such as comparing `name1 <= name2` that share a common prefix that must be traversed twice individually by the calls to `<` and `==`, which does not arise when calling a single three-way comparison function.

(4) Performance (hardware): This design avoids potential inefficiencies from less optimal mapping to hardware. Some processors support three-way comparison instructions for machine types, and code generation can naturally take advantage of this capability where present. Conversely, having three-way comparison in the language does not disadvantage hardware that does not support it (such as x86 SIMD vector instructions); code generation can fall back to `<` and `==`, and is no worse than without three-way comparison in the source semantics.

This proposal makes programmers generally immune to all of these problems, because it lets the programmer follow the "don't repeat yourself" principle and write just one function.

3 Sample category types implementation

Notes I would prefer using `enum`s, but for now the language doesn't allow expressing what we need using `enum`s. In particular, `enum`s don't currently support a way to express value conversion relationships, for example that a `strong_ordering` value converts correctly to a `weak_ordering` or `partial_ordering` value (which preserve the integral enumerator value) or an `strong_equality` or `weak_equality` value (which adjust the integral value because of mapping it to fewer options).

We want to allow comparing against literal `0`, but not comparing against just any integer. In this sample implementation, I'm comparing with `nullptr_t` as a "hacky but useful" way to permit comparison against literal `0` (abusing its wonky dual nature) but no other integer value, not even an `int` lvalue holding value `0`. In the wording, we can say an "unspecified type."

This is a sample only. See wording paper P0768 for the actual detailed specification including the removal of C-style casts, and the application of `constexpr`, `noexcept`, etc. as decided by LEWG and LWG.

```
#include <stdexcept>

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

//===========================================================================
//   _equality:
//        - use int as underlying type + default copying semantics
//        - can only be constructed from specific values
//        - can be compared against literal 0 (== and != only)
//===========================================================================

//---------------------------------------------------------------------------
class weak_equality {
     int value;

     // constructors
     explicit weak_equality(eq v) : value{ (int)v } { }

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

     // comparisons
     friend bool operator==(weak_equality v, nullptr_t) { return v.value == 0; }
     friend bool operator!=(weak_equality v, nullptr_t) { return v.value != 0; }
     friend bool operator==(nullptr_t i, weak_equality v) { return 0 == v.value; }
     friend bool operator!=(nullptr_t i, weak_equality v) { return 0 != v.value; }
};

//---------------------------------------------------------------------------
class strong_equality {
     int value;

     // constructors
     explicit strong_equality(eq v) : 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};

    // implicit conversions to weaker types
    operator weak_equality() const { return *this == equal ? weak_equality::equivalent : weak_equality::nonequivalent; }

    // comparisons
    friend bool operator==(strong_equality v, nullptr_t) { return v.value == 0; }
    friend bool operator!=(strong_equality v, nullptr_t) { return v.value != 0; }
    friend bool operator==(nullptr_t, strong_equality v) { return 0 == v.value; }
    friend bool operator!=(nullptr_t, strong_equality v) { return 0 != v.value; }
};


//===========================================================================
//   _ordering:
//        - use int as underlying type + default copying semantics
//        - can only be constructed from specific values
//        - can be compared against literal 0
//        - can be explicitly converted to int
//        - do not convert to bool, to prevent the "if( strcmp(...) )" mistake
//===========================================================================

//---------------------------------------------------------------------------
class partial_ordering {
    struct {
        int cmp : 7;
        bool is_ordered : 1;
    } value;

    // constructors
    explicit partial_ordering(eq   v) : value{ (int)v, true  } { }
    explicit partial_ordering(ord  v) : value{ (int)v, true  } { }
    explicit partial_ordering(ncmp v) : 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 };

    // implicit conversions to weaker types
    operator weak_equality() const { return *this == equivalent ? weak_equality::equivalent : weak_equality::nonequivalent; }

    // comparisons
    friend bool operator==(partial_ordering v, nullptr_t) { return  v.value.is_ordered && v.value.cmp == 0; }
    friend bool operator!=(partial_ordering v, nullptr_t) { return !v.value.is_ordered || v.value.cmp != 0; }
    friend bool operator< (partial_ordering v, nullptr_t) { return  v.value.is_ordered && v.value.cmp <  0; }
    friend bool operator<=(partial_ordering v, nullptr_t) { return  v.value.is_ordered && v.value.cmp <= 0; }
    friend bool operator> (partial_ordering v, nullptr_t) { return  v.value.is_ordered && v.value.cmp >  0; }
    friend bool operator>=(partial_ordering v, nullptr_t) { return  v.value.is_ordered && v.value.cmp >= 0; }
    friend bool operator==(nullptr_t, partial_ordering v) { return  v.value.is_ordered && 0 == v.value.cmp; }
    friend bool operator!=(nullptr_t, partial_ordering v) { return !v.value.is_ordered || 0 != v.value.cmp; }
    friend bool operator< (nullptr_t, partial_ordering v) { return  v.value.is_ordered && 0 <  v.value.cmp; }
    friend bool operator<=(nullptr_t, partial_ordering v) { return  v.value.is_ordered && 0 <= v.value.cmp; }
    friend bool operator> (nullptr_t, partial_ordering v) { return  v.value.is_ordered && 0 >  v.value.cmp; }
```

```
    friend bool operator>=(nullptr_t, partial_ordering v) { return  v.value.is_ordered && 0 >= v.value.cmp; }
};

//------------------------------------------------------------------------
class weak_ordering {
    int value;

    // constructors
    explicit weak_ordering(eq  v) : value{ (int)v } { }
    explicit weak_ordering(ord v) : 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 };

    // implicit conversions to weaker types
    operator weak_equality() const   { return *this == equivalent ? weak_equality::equivalent : weak_equality::nonequivalent;   }
    operator partial_ordering() const { return *this == equivalent ? partial_ordering::equivalent
                                            : *this == less ? partial_ordering::less : partial_ordering::greater; }

    // comparisons
    friend bool operator==(weak_ordering v, nullptr_t) { return v.value == 0; }
    friend bool operator!=(weak_ordering v, nullptr_t) { return v.value != 0; }
    friend bool operator< (weak_ordering v, nullptr_t) { return v.value <  0; }
    friend bool operator<=(weak_ordering v, nullptr_t) { return v.value <= 0; }
    friend bool operator> (weak_ordering v, nullptr_t) { return v.value >  0; }
    friend bool operator>=(weak_ordering v, nullptr_t) { return v.value >= 0; }
    friend bool operator==(nullptr_t, weak_ordering v) { return 0 == v.value; }
    friend bool operator!=(nullptr_t, weak_ordering v) { return 0 != v.value; }
    friend bool operator< (nullptr_t, weak_ordering v) { return 0 <  v.value; }
    friend bool operator<=(nullptr_t, weak_ordering v) { return 0 <= v.value; }
    friend bool operator> (nullptr_t, weak_ordering v) { return 0 >  v.value; }
    friend bool operator>=(nullptr_t, weak_ordering v) { return 0 >= v.value; }
};

//------------------------------------------------------------------------
class strong_ordering {
    int value;

    // constructors
    explicit strong_ordering(eq  v) : value{ (int)v } { }
    explicit strong_ordering(ord v) : 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 };

    // implicit conversions to weaker types
    operator weak_equality() const    { return *this == equivalent ? weak_equality::equivalent : weak_equality::nonequivalent;   }
    operator strong_equality() const  { return *this == equal      ? strong_equality::equal    : strong_equality::nonequal; }
    operator partial_ordering() const { return *this == equivalent ? partial_ordering::equivalent
                                           : *this == less       ? partial_ordering::less    : partial_ordering::greater; }
    operator weak_ordering() const    { return *this == equivalent ? weak_ordering::equivalent
                                           : *this == less       ? weak_ordering::less       : weak_ordering::greater; }
```

```
    // comparisons
    friend bool operator!=(strong_ordering v, nullptr_t) { return v.value != 0; }
    friend bool operator< (strong_ordering v, nullptr_t) { return v.value <  0; }
    friend bool operator==(strong_ordering v, nullptr_t) { return v.value == 0; }
    friend bool operator<=(strong_ordering v, nullptr_t) { return v.value <= 0; }
    friend bool operator> (strong_ordering v, nullptr_t) { return v.value >  0; }
    friend bool operator>=(strong_ordering v, nullptr_t) { return v.value >= 0; }
    friend bool operator==(nullptr_t, strong_ordering v) { return 0 == v.value; }
    friend bool operator!=(nullptr_t, strong_ordering v) { return 0 != v.value; }
    friend bool operator< (nullptr_t, strong_ordering v) { return 0 <  v.value; }
    friend bool operator<=(nullptr_t, strong_ordering v) { return 0 <= v.value; }
    friend bool operator> (nullptr_t, strong_ordering v) { return 0 >  v.value; }
    friend bool operator>=(nullptr_t, strong_ordering v) { return 0 >= v.value; }
};
```

4 Proposed wording

4.1 Language wording

Note An initial version of this wording appeared in paper P0564R0.

This section presents the detailed wording changes to implement the foregoing design. Any differences in semantics are unintentional.

In 5.12 [lex.operators], add `<=>` as an option for the grammar non-terminal *preprocessing-op-or-punc*.

Add a new section 8.9 [expr.spaceship] before the existing 5.9 [expr.rel]:

8.9 Three-way comparison operator [expr.spaceship]

The three-way comparison operator groups left-to-right.

compare-expression:

shift-expression

compare-expression `<=>` *shift-expression*

If both operands have (possibly different) floating-point types, the usual arithmetic conversions are applied to the operands. The operator yields a prvalue of type `std::partial_ordering`. The expression `a <=> b` yields `std::partial_ordering::less` if `a` is less than `b`, `std::partial_ordering::greater` if `a` is greater than `b`, `std::partial_ordering::equivalent` if `a` is equivalent to `b`, and `std::partial_ordering::unordered` otherwise.

If both operands have the same enumeration type E: If E has more than one enumerator with a given value, the operator yields a prvalue of type `std::weak_ordering`, otherwise it yields a prvalue of type `std::strong_ordering`. In either case, the operator yields the result of converting the operands to the underlying type of E and applying `<=>` to the converted operands.

If at least one of the operands is a pointer, pointer conversions (7.11 [conv.ptr]), function pointer conversions (7.13 [conv.fctptr]), and qualification conversions (7.5 [conv.qual]) are performed on both operands to bring them to their composite pointer type (Clause 8 [expr]). If at least one of the operands is a pointer to member, pointer to member conversions (7.12) and qualification conversions (7.5) are performed on both operands to bring them to their composite pointer type (Clause 8). If both operands are null pointer constants, but not both of integer type, pointer conversions (7.11 [conv.ptr]) are performed on both operands to bring them to their composite pointer type (Clause 8 [expr]). In all cases, after the conversions, the operands shall have the same type. [Note: Array-to-pointer conversions (7.2 [conv.array]) are not applied. -- end note]

If the composite pointer type is a function pointer type, a pointer-to-member type, or `std::nullptr_t`, the operator yields a prvalue of type `std::strong_equality`; the operator yields `std::strong_equality::equal` if the (possibly converted) operands compare equal (8.10 [expr.eq]) and `std::strong_equality::unequal` if they compare unequal, otherwise the result of the operator is unspecified.

If the composite pointer type is an object pointer type, the operator yields the result of converting both operands to `std::uintptr_t` and comparing the converted operands using `<=>`, where the result is consistent with the result of equality and relational comparisons. [Note: That means, if two pointer operands `p` and `q` compare equal (8.10 [expr.eq]), `p <=> q` yields `std::strong_ordering::equal`; if `p` and `q` compare unequal, `p <=> q` yields `std::strong_ordering::less` if `q` compares greater than `p` and `std::strong_ordering::greater` if `p` compares greater than `q` (8.9 [expr.rel]). -- end note]

If both operands have the same integral type, the operator yields a prvalue of type `std::strong_ordering`. The result is `std::strong_ordering::equal` if both operands are arithmetically equal, `std::strong_ordering::less` if the first operand is arithmetically less than the second operand, and `std::strong_ordering::greater` otherwise. [Note: Integral promotions (7.6 [conv.prom]) or integral conversions (7.8 [conv.integral]) are not applied. -- end note]

Otherwise, the program is ill-formed.

Change the grammar in 8.9(old) [expr.rel]:

relational-expression:

~~shift-expression~~ compare-expression

relational-expression `<` *~~shift-expression~~ compare-expression*

relational-expression `>` *~~shift-expression~~ compare-expression*

relational-expression `<=` *~~shift-expression~~ compare-expression*

relational-expression `>=` *~~shift-expression~~ compare-expression*

Insert a bullet in 8.20 [expr.const] paragraph 2:

- ...
- a three-way comparison (8.9(new) [expr.spaceship]) comparing pointers that do not point to subobjects of the same complete object;
- a relational (8.9) or equality (8.10) operator where the result is unspecified; or
- ...

Add a new section to clause 15 [special]:

15.9 Comparisons [class.compare]

A defaulted comparison operator function (8.9(new) [expr.spaceship], 8.9 [expr.rel], 8.10 [expr.eq]) for some class C shall be a non-template function declared in the *member-specification* of C that

- is a non-static member of C having one parameter of type `const C&` or
- a static member or friend of C having two parameters of type `const C&`,

in all cases naming the injected-class-name.

15.9.1 Three-way comparison [class.spaceship]

The *common comparison type* R of a possibly empty list of types T_0, T_1, … T_{n-1} is defined as follows:

- If n is 0, R is `std::strong_ordering`.

- Otherwise, if any T_i is not a comparison category type, the type is `void`.
- Otherwise, if
  - at least one T_i is `std::weak_equality` or
  - at least one T_i is `std::strong_equality` and at least one T_j is `std::partial_ordering` or `std::weak_ordering`,

  R is `std::weak_equality`.
- Otherwise, if at least one T_i is `std::strong_equality`, R is `std::strong_equality`.
- Otherwise, if at least one T_i is `std::partial_ordering`, R is `std::partial_ordering`.
- Otherwise, if at least one T_i is `std::weak_ordering`, R is `std::weak_ordering`.
- Otherwise, R is `std::strong_ordering`.

The direct base class subobjects of C, in the order of their declaration in the *base-specifier-list* of C, followed by the non-static data members of C, in the order of their declaration in the *member-specification* of C, form a list of subobjects. In that list, any subobject of array type is recursively expanded to the sequence of its elements, in the order of increasing subscript. Let x_i denote the i^{th} element in the expanded list of subobjects for an object x, where x_i is an lvalue if it is has reference type, and a const xvalue otherwise. [Note: This yields the same result as a class member access (8.2.5 [class.mem]) on `const C`. -- end note] The type of the expression x_i `<=>` x_i is denoted by R_i.

For a defaulted three-way comparison operator function, the declared return type shall be either `auto`, in which case the return type is deduced as the common comparison type of R_0, R_1, … R_{n-1}, or one of the comparison category types, in which case a value of the deduced return type shall be implicitly convertible to the declared return type. If the return type of a defaulted three-way comparison operator function is `void`, the operator function is defined as deleted.

The return value V of the three-way comparison operator function invoked with arguments x and y of the same type is determined by comparing corresponding elements x_i and y_i in the expanded lists of subobjects for x and y and converting each of the resulting values to type R. Let i denote the first index where x_i <=> y_i yields a result value different from R_i`::equivalent`; V is that result value converted to R. If no such index exists, V is `std::strong_ordering::equal` converted to R.

15.9.2 Other comparison operators [class.rel.eq]

A defaulted relational (8.9 [expr.rel]) or equality (8.10 [expr.eq]) operator function for some operator @ shall have a declared return type `bool`.

The operator function with parameters x and y is defined as deleted if

- overload resolution (16.3), as applied to `x <=> y` (also considering synthesized candidates with reversed order of parameters), results in an ambiguity or a function that is deleted or inaccessible from the operator function, or
- the operator @ cannot be applied to the return type of `x <=> y` or `y <=> x`.

Otherwise, the operator function yields `x <=> y @ 0` if an `operator<=>` with the original order of parameters was selected, or `0 @ y <=> x` otherwise.

[Example:

```
struct C {
  friend std::strong_equality operator<=>(const C&, const C&);
  bool operator==(const C& x, const C& y) = default;  // ok, returns x <=> y == 0
  bool operator<(const C&, const C&) = default;       // ok, function is deleted
};
```

-- end example]

Change in 16.3.1.2 [over.match.oper] paragraph 6 and add a new paragraph after that:

The set of candidate functions for overload resolution is the union of the member candidates, the non-member candidates, and the built-in candidates , all for operator@. If the operator is a relational (5.9 [exp.rel]) or equality (5.10 [expr.eq]) operator, a member or non-member candidate operator<=> is added to the set of candidate functions for overload resolution if

- the candidate has return type `std::strong_ordering`, `std::weak_ordering`, or `std::partial_ordering` or
- the candidate has return type `std::strong_equality` or `std::weak_equality` and @ is `==` or `!=`.

For each such added candidate whose parameter types differ, a synthesized candidate is added to the candidate set where the order of the two parameters is reversed.

The argument list contains all of the operands of the operator. The best function from the set of candidate functions is selected according to 16.3.2 and 16.3.3. [Footnote: ...] [Example: ... -- end example]

If a candidate for `operator<=>` is selected by overload resolution, but @ is not `<=>`, the call to `operator@` with arguments x and y yields the value of `0 @ operator<=>(y,x)` if the selected candidate is a synthesized candidate with reversed order of parameters, or `operator<=>(x,y) @ 0` otherwise.

If a built-in candidate is selected by overload resolution, the operands of class type are converted to the types of the corresponding parameters of the selected operation function, except that ...

Add new bullets before the deduction guide bullet in 16.3.3 [over.match.best] paragraph 1:

- ...
- F1 is an operator function for a relational (8.9 [expr.rel]) or equality (8.10 [expr.eq]) operator and F2 is not [Example:

```
struct S {
    auto operator<=>(const S&, const S&) = default;  // #1
    bool operator<(const S&, const S&);              // #2
};
bool b = S() < S();   // calls #2
```

-- end example] or, if not that,

- F1 and F2 are operator functions for `operator<=>` and F2 is a synthesized candidate with reversed order of parameters and F1 is not [Example:

```
struct S {
  std::weak_ordering operator<=>(const S&, int);  // #1
  std::weak_ordering operator<=>(int, const S&);  // #2
};
bool b = 1 < S();   // calls #2
```

-- end example] or, if not that,

- F1 is generated from a deduction-guide (16.3.1.8) and F2 is not …

In 16.5 [over.oper] paragraph 1, add `<=>` as an option for the grammar non-terminal operator.

Add two new paragraphs after 16.6 [over.built] paragraph 13:

> For every integral type T there exist candidate operator functions of the form
>
> ```
> std::strong_ordering operator<=>(T , T );
> ```
>
> For every pair of floating-point types L and R, there exist candidate operator functions of the form
>
> ```
> std::partial_ordering operator<=>(L , R );
> ```

Change in 16.6 [over.built] paragraphs 16 and 17:

> For every T, where T is an enumeration type or a pointer type, there exist candidate operator functions of the form
>
> ```
> bool    operator<(T , T );
> bool    operator>(T , T );
> bool    operator<=(T , T );
> bool    operator>=(T , T );
> bool    operator==(T , T );
> bool    operator!=(T , T );
> R       operator<=>(T , T );
> ```
>
> where R is the result type specified in 8.9 [expr.spaceship].
>
> For every pointer to member type T or type `std::nullptr_t` there exist candidate operator functions of the form
>
> ```
> bool     operator==(T , T );
> bool     operator!=(T , T );
> std::strong_equality operator<=>(T , T );
> ```

Add a new paragraph after 16.6 [over.built] paragraph 17:

> For every array type T there exist candidate operator functions of the form
>
> ```
> R     operator<=>(T& , T& );
> ```

```
R     operator<=>(T&& , T&& );
```

where R is the result type specified in 8.9 [expr.spaceship].

Add a new paragraph after 18.4 [except.spec] paragraph 10:

A deallocation function (6.7.4.2) with no explicit noexcept-specifier has a non-throwing exception specification.

The exception specification for a three-way comparison without a *noexcept-specifier* that is defaulted on its first declaration, is potentially-throwing if and only if the invocation of any comparison operator in the implicit definition is potentially-throwing.

4.2 Library wording

See [P0768].

5 Bibliography

[N3950] O. Smolsky. “Defaulted comparison operators” (WG21 paper, 2014-02-19). Initial proposal of the current series of papers attempting to add comparisons to C++. Successively revised by N4114 and N4126 to implement EWG direction during 2014.

[N4475] B. Stroustrup. “Default Comparisons (R2)” (WG21 paper, 2015-04-09). Motivational and design paper. Update of the original N4175 which was a counterproposal focused especially on default generation.

[N4476] B. Stroustrup. “Thoughts about Comparisons (R2)” (WG21 paper, 2015-04-09). Discussion paper. Update of the original N4176 which was a response arguing against particular design points in the previous EWG directions, including declaration verbosity.

[Smolsky 2015] O. Smolsky. “On generating default comparisons” (unpublished, Kona 2015 wiki, Oct 2015). Discussion paper on comparison generation.

[P0100R2] L. Crowl. “Comparison in C++” (WG21 paper, 2016-11-27). Update of N4367 initially presented in Lenexa.

[P0221R2] J. Maurer. “Proposed wording for default comparisons, revision 4” (WG21 paper, 2016-06-23). Wording for N4475.

[P0474R0] L. Crowl. “Comparison in C++: Basic Facilities” (WG21 paper, 2016-10-15). The first step of P0100R2.

[P0436R1] W. E. Brown. “An Extensible Approach to Obtaining Selected Operators” (WG21 paper, 2016-10-10).

[P0481R0] T. Van Eerd. “Bravely Default” (WG21 paper, 2016-10-15). Argues that default comparison follow default copying.

[P0432R0] D. Stone. “Implicit and Explicit Default Comparison Operators” (WG21 paper, 2016-09-18).

[P0515R0] H. Sutter. “Consistent comparison” (WG21 paper, 2017-02-05).

[P0515R1] H. Sutter. “Consistent comparison” (WG21 paper, 2017-06-16).

[P0768R0] W. E. Brown. “Library Support for the Spaceship (Comparison) Operator” (WG21 paper, 2017-09-30).

6 Revision history

R2 (pre-Albuquerque, 2017-09):

- Implemented Toronto LEWG guidance.
- Added library wording in companion paper P0768R0 (Walter E. Brown).

R1 (pre-Toronto, 2017-06):

- Implemented Kona EWG guidance: Removed the optional parts that did not get consensus.
- Implemented Kona LEWG guidance: Kept `strong_` naming. Removed conversion to integer. Added note mapping to mathematical terminology. Added `common_comparison_category<Ts>::type` version. Distinguished which pointer comparisons are constant expressions. Enumerator equality is determined using the same rules as switch statements and non-type template arguments. Moved proposed wording from Maurer's separate paper P0564 into this paper.

R0 (pre-Kona, 2017-02): Initial revision.