Consistent comparison

Document Number: **P0515 R0**
Date: 2017-02-05
Author: Herb Sutter (hsutter@microsoft.com)
Audience: EWG, LEWG

Abstract

This paper presents a design for comparisons organized as:

- a core design that follows the design points that were generally uncontroversial and new ones that have current EWG support from Issaquah, and
- a separate "optional" part to decouple the previously controversial question of whether/how to implicitly declare comparisons when no comparisons are user-declared (a pure extension, nothing in the core proposal depends on whether this is adopted).

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 EWG and WG21 broadly since N3950 followed by N4126 (Smolsky) in 2014. We explored various approaches, culminating in the development and rejection in Oulu of the most-developed proposal yet, N4475 and N4476 (Stroustrup, motivation and discussion) and P0221R2 (Maurer, wording).

In Issaquah (Nov 2016), EWG discussed four new comparison proposals – P0474R0 as a first step of P0100R2 (Crowl), P0436R1 (Brown), P0481R0 (Van Eerd), P0432R0 (Stone) – and expressed moderate to strong interest in: (a) including three-way comparison; (b) letting implicit memberwise copy imply implicit memberwise comparison; and (c) have a simple way to write a memberwise comparison function body.

> **Note** For a detailed side-by-side comparison with the previous proposals, see §5.

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see P0100R2 (Crowl) and N4475 and N4476 (Stroustrup) which remain relevant even though parts of those papers have been superseded by subsequent discussions.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 do not affect the core proposal.
- **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, and Ville Voutilainen.

1.4 Proposal overview: Guidance and examples

1.4.1 Guidance: What we teach

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.
>
> To write all comparisons for your type, just write `operator<=>` that returns the appropriate category type:

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

Expressing the same in table form:

Write `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.

1.4.2 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, just write `<=>` returning `strong_ordering`, with `=default` as the definition. Here is an example, and we'll use a non-member function as usual good style to enable conversions on both arguments:

```
class Point {
    int x;
    int y;

public:
    friend std::strong_ordering operator<=>(const Point&, const Point&) = default;

    // ... other functions, but no other user-declared comparisons ...
};
```

`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.

1.4.3 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;
    }

    // ... other functions, but no other comparisons ...
};
```

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 <=>
```

1.4.4 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:
    friend std::weak_ordering operator<=>(const CaseInsensitiveString& a,
                                          const CaseInsensitiveString& b) {
        return case_insensitive_compare(a.s.c_str(), b.s.c_str());
    }

    // ... other functions, but no other comparisons ...
};
```

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*`:

```
    // add this function in CaseInsensitiveString

    friend std::weak_ordering operator<=>(const CaseInsensitiveString& a,
                                          const char* b) {
        return case_insensitive_compare(a.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
```

1.4.5 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;
    }

    // ... other functions, but no other comparisons ...
};
```

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

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

1.4.6 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;
    }

    // ... other functions, but no other comparisons ...
};
```

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

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

1.4.7 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;
    }

    // ... other functions, but no other comparisons ...
};
```

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

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

1.4.8 (optional) Compiler-generated <=> comparison

Consider this class, which does not declare any comparisons:

```
class Point {
    int x;
    int y;

public:
    // ... other functions, but no user-defined copying or comparisons ...
};
```

§3.1 describes three options for this type:

- (§3.1.1) Option 1: Generate nothing. Then this class has no comparisons, as today.
- (§3.1.2) Option 2: Generate all comparisons, by generating `<=>` to return `_ordering`.
- (§3.1.3) Option 3: Generate `==` and != only, by generating `<=>` to return `_equality`.

If Option 1 is accepted, then this class has no comparisons, as today.

If either Option 2 or Option 3 is accepted, then `==` and `!=` work, and are efficiently implemented in terms of a single `<=>`, and none create actual functions:

```
Point pt1, pt2;

pt1 = pt2;
assert(pt1 == pt2);             // ok, if §3.1 Option 2 or 3 accepted
```

If Option 2 is accepted, then additionally ordered comparisons work, and are efficiently implemented in terms of a single `<=>`, and none create actual functions:

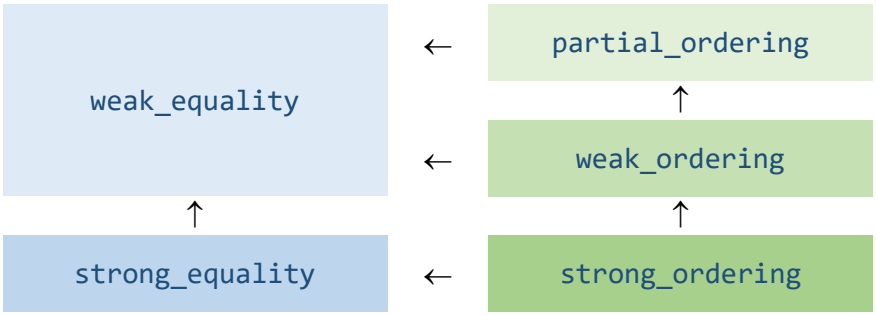
```
set<Point> s;                   // ok, if §3.1 Option 2 or §3.2 accepted
s.insert(pt1);                  // ok, if §3.1 Option 2 or §3.2 accepted

if (pt1 <= pt2) { /*...*/ }     // ok, if §3.1 Option 2 accepted
```

2 Core design

2.1 Comparison categories

We define five comparison categories as `std::` types (see also §4). 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.

Naming bikeshed: We can easily instead follow the standard mathematical terms instead, but I think these are probably simpler to teach; see note in §5.1.2 regarding the choice of names.

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 §4 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 §1.4.4 through §1.4.7 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 §1.4.1.

`common_comparison_category_t<class ...Ts>` is the type `C` computed as follows from `Ts`:

- If `Ts` is empty, `C` is `strong_ordering`.
- Otherwise, if each T_i supports `<=>` returning type Cmp_i, `C` 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 convertible to int: I prefer allowing the return value to convert to a signed 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 used as a signed integer (and adds safety by making the conversion explicit).

(2) ***Consistency:*** It permits expressing all other comparisons `a @ b` as `a<=>b @ 0` (and, when `operator<=>` is heterogeneous, taking parameters of different types, `0 @ b<=>a`):

source code:	default rewrite generation does:	+ when `<=>(T1,T2)` is heterogeneous also this to support `T2 @ T1` (see §2.3)
`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
```

This definition is not recursive for `int`, because `int` defines all operators without rewrite.

(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 homogeneous (same-type) comparisons only, and cannot be invoked heterogeneously using scalar promotions/conversions unless mentioned otherwise below.

- 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)`.
- 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 (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 If we adopt §3.1 option 2 or 3, some `std::` types, such as `array` and some `_tag` types, could omit explicit comparison entirely and rely on defaults. In particular, empty types in the library would be comparable by default and always-equal semantics. Although that default could be overridden, it seems like it shouldn't need to be: It's a pure extension that current code doesn't use, and I suspect it aids generic code for the same reasons as `void_t`, by turning a "doesn't compile" into "compiles as no-op" for regularity instead of disabling a special case that generic code has to work around.

Some `std::` types, such as `pair`, would declare `operator<=>` to opt back in because they have custom copying functions, but can use the `=default` definition to get memberwise comparison (as shown in §1.4.2).

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 `strong_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, and custom memberwise two-way comparison operators that delegate to elementwise two-way comparison operators as today (this makes `<=>` a little inconsistent with the other comparisons, but remains source compatible). Alternatively, if the library is willing to take a small source breaking change, it could be consistent and apply the same rules as in §3.1.2 for deducing the comparison category: Write only `<=>` and return the weakest common comparison category (with the exclusion that the set cannot contain both `*_equality` and `partial_ordering`, because neither has a value domain that is is a proper subset of the other's); this would mean that a `pair` or `tuple` that contains a floating point fundamental type would now support only `<=>`, `<`, and `>` comparison, but not `<=`, `>=`, `==`, or `!=` comparison.

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.

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.

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 comparison operator), perform name lookup for `a @ b`, `a <=> b`, and `b <=> a`, excluding any compiler-generated `operator<=>` (this restriction matters only if §3.1 Option 2 or 3 is adopted). 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`.

If §3.1 Option 2 or 3 is adopted, add this fallback step: If no viable candidate is found, then perform the lookup for `a <=> b` only, this time considering also compiler-generated `operator<=>` functions.

Notes We try both `a <=> b` and `b <=> a` so that a 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.

For a compiler-generated `operator<=>` (which exists only if §3.1 Option 2 or 3 is adopted), we treat it as a fallback for backward source compatibility, even though a worse-match `operator@` might be selected at some call sites. For example, some call site may already have a local `operator@` function in scope that is being called, and we don't want to change the meaning of that call site by injecting a compiler-generated `operator<=>` that would be an equal match (which would make the call ambiguous) or a better match (which would silently change the meaning of the call). This is the same compatibility motivation and similar approach as in N4475 (Stroustrup) and P0221R2 (Maurer).

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 if we do not provide compiler-generated `<=>` (see §3.1) or if we do provide that but the generation is suppressed (such as for types that write a custom copying function but want memberwise comparison). For example:

```
class MyClass {
    // ... possibly many members here ...

public:                             // even if §3.1 Option 2 or 3 is accepted, this
    MyClass(const MyClass&);        // user-declared copy still suppresses comparison

    // ...

    friend std::strong_ordering operator<=>(const MyClass&, const MyClass&) =default;
};                                  // but we can opt back in like this
```

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).

TODO All of these should additionally be made conditionally `noexcept` and `constexpr`, and the comparisons should be specialized for `<>`. Excluded for initial presentation clarity.

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

LEWG question: Should these SFINAE away ("don't participate in overload resolution [etc.]") if none of the underlying operations are valid, like `greater<>` (with `<>`) et al. do?

```
template<class T, class Comparison>
constexpr bool can_3compare_as() {   // to save typing
    { return std::is_convertible_v<std::result_of_t< std::declval<T>() <=> std::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 if constexpr (/* can invoke strong_order(a.M, b.M) for each member M of T */) /* do that */;
}

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::equal :
                                                             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 ? weak_ordering::equal :
                                                             a<b ? weak_ordering::less : weak_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;
    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;
    // note: heterogeneous case has no fallback to memberwise (the homogeneous case below adds this)
}

template<class T>
auto compare_3way(const T& a, const T& 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 (/* can invoke a.M <=> b.M for each member M of T */) /* do that */;
}
```

We also provide a comparison algorithm:

```
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 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 Optional design points/alternatives

These design points can be included or excluded separately. Nothing in the core design depends on them.

3.1 Generating `<=>`: nothing, or `_ordering` (`<`, `>`, `<=`, `>=`, `==`, `!=`), or `_equality` (`==`, `!=`)

Generating comparison operators when the type does not declare any at all was controversial, but in this proposal can be elegantly supported or disabled independently from the rest of this proposal.

For this subsection only, because we have learned that the consensus choice in EWG and in plenary can differ, I have a procedural suggestion: I suggest that EWG as usual discuss and poll all three options, but in this case additionally forward to Core a wording paper that contains wording for *all* options that achieve support in EWG (e.g., EWG should choose what wording they want for each such option if that option ends up being the one chosen) structured so that the wording alternatives are clearly marked and separable; because of the structure of this proposal, this should be easy to do as shown in the rest this subsection. That way, plenary can consider EWG's recommended option as well as the full-group preference and easily select the strongest option that has a consensus in plenary.

3.1.1 Option 1: Generate nothing

This is the default if we add nothing below to the core proposal.

3.1.2 Option 2: Generate all comparisons, by generating `<=>` to return `_ordering`

To have a type that does not declare any comparisons get all comparisons, if and only if it is default-copyable, add the following.

If a type `T` meets the following requirements:

- `T`'s copy construction and copy assignment operator are both defaulted (either explicitly or implicitly) and not deleted, and therefore are known to perform memberwise copying;
- `T` has no user-declared destructor (this bullet exists to duplicate the rules in D.2 to exclude the already-deprecated cases of implicitly-declared copy);
- `T`'s copy construction and copy assignment operator have parameters that are either `T` or `const T&` and therefore are known not to modify the source object;
- `T`'s class definition has no user-declared comparison functions (including in public base classes); and
- an `operator<=>` that is `=default` would not be implicitly deleted (see §2.4);

then there is a compiler-generated non-member friend `operator<=>` that performs memberwise comparison:

- takes two parameters `a` and `b` of type `const T&`;
- has `constexpr` deduced following the same rules as implicitly-declared copy functions;
- is unconditionally `noexcept`;
- returns `auto`; and
- is defined as `=default` (see §2.4).

Notes "Compiler-generated" is placeholder wording – we should decide whether this is an actual function. The main argument for making it a real function is consistency with compiler-generated copy/move

operations. The main argument for making it a semantic rewrite is consistency with the other compiler-generated comparisons in §2.3.

Because it is a friend it is declared in the scope of the class, which makes it visible for convenient scoped name lookup as in §2.4).

For "T has no user-declared comparison functions," any rule should only care about those provided by the author of T, not just any operator that happens to be available in a given calling scope (which can vary). These are virtually always (a) members or (b) friends, which are covered by the rule above. The only case not covered by this rule is if there are nonmember nonfriends in the namespace in which T is declared, which are found by ADL. However, the rule in §2.3 covers this by using fallback in the case of a compiler-generated `operator<=>`, so we can't change the meaning of existing code.

We follow "non-deprecated" default copying. We have already deprecated the implicitly-declared copy cases that have been long known to be problematic. For example, classes that have a raw pointer data member for which copying is not the safe default will virtually always write a destructor, and that will disable compiler-generated comparisons as it already disables implicitly-declared moves (and, when we remove the deprecated copy cases from the standard, it will also fix copying too and all will be fully consistent).

We do not generate a `<=>` that would be implicitly deleted because that would participate in overloading. We want to follow the same approach as SFINAE and have it not exist.

These operational semantics rely on the implicit conversions in §2.1.

Accessibility is ignored, which does not violate encapsulation any more than for any other implicitly-defined function.

We deduce `constexpr` as we do for implicitly-declared copying, but we do not deduce `noexcept` similarly because whereas copying can fail by throwing, comparison is a `const` operation and should not throw.

Supporting empty types avoids special cases in the language and in generic code. For example:

> In generic code, a tuple may contain an empty type. We don't want that to suppress comparison for (say) `tuple<int, tag, string>`.
>
> In regular code, a class may contain an empty base. We don't want that to suppress comparison for (say) `class Point : tag { int x, y; };` just because it has an empty base.

Like all other types, lambdas should be default comparable if they are default copyable, but the order of their members is not specified. If we make lambdas default comparable if they are default copyable, we should also specify the order of their members.

3.1.3 Option 3: Generate `==`/`!=` only, by generating `<=>` to return `_equality`

If we do not want to generate the ordered comparisons, there are two simple ways to achieve the result:

Option 3(a): Take Option 2 (§3.1.2), adding the restriction that a compiler-generated `<=>` returns an `_equality` (for which only `==` and `!=` are in turn generated), not an `_ordering` (for which all six others are in turn generated) by adding the following rule in §3.1.2:

- If the compiler-generated `<=>`'s return type $\texttt{Cmp}_{weakest}$ would be calculated as `strong_ordering` then adjust it to `strong_equality`, else if it would be calculated as `weak_ordering` or `partial_ordering` then adjust it to `weak_equality`.

Notes In previous discussions, including in Oulu plenary discussion and Issaquah EWG discussion, a number of experts said that when there are no user-declared comparisons, they were happy to have compiler-generated `==` and `!=` comparisons, but expressed sustained concerns about having compiler-generated ordered comparisons `<`, `>`, `<=`, and `<=` for a variety of reasons, such as that the result was not semantically meaningful to the class, and/or that it exposed the implementation detail of the data member order. Note that this paper's key design principle that ***by default, `a = b` implies `a == b`*** is a direct and strong argument for compiler-generated `==` and `!=`, but is silent about whether to generate the others. (I personally don't share the concern and prefer Option 2; as others have also noted, private members are not exposed, and the observable side effect of their type and order on compiler-generated comparison doesn't seem much different from other observable side effects of data members' type and ordering, such as whether their copying throws. – The key point is that this paper's design makes the result easy to implement and *teachable* either way, as Option 2 or Option 3.)

Because this design has strong support for the comparison categories, it's easy to make this choice as a fully separable design point that does not affect the basic design of this paper, and in a more teachable way than in other proposals to generate `==` and `!=` but not the others: Under the other proposals, the best way I knew to teach Option 3 was as a rote rule, "remember the compiler generates these two operators, but you have to opt into those four operators." Under this design, we could teach "the compiler generates an `_equality`; you have to opt into an `_ordering`."

Option 3(b): Take Option 2 (§3.1.2), but add a special case in §2.3 that a compiler-generated `<=>` never in turn generates `<`, `>`, `<=`, or `>=`.

Note The main rationale for this is that if we do get Option 3 (but not Option 2), type authors might want to define `<` etc. explicitly based on defaulted `<=>` of members and base classes, or of their own class. If any `<=>` in the hierarchy is downgraded as prescribed by 3(a), that fails. By keeping the return type being the "best possible," that succeeds.

However, this rationale worries me. First, type authors should not be having to write such two-way comparison definitions by hand all the time anyway – if they still have to do that, we've failed. Second, if we return (say) `strong_ordering` but don't actually support `<` etc., that would be deeply confusing; for example, it would break the ability of generic code to inspect the return type of `operator<=>` to reliably know what two-way comparisons can be performed. It would not be good to break the guarantee that a given comparison category reliably ensures a known set of orderings – it would probably eliminate the advantage 3(b) is trying to achieve of retaining the "best possible" category.

This note is probably the best argument against Option 3 overall: The above rationale points out a fundamental problem created by Option 3 (disabling natural composability by adding an artificial limitation) that may not be well solved by either 3(a) or 3(b).

3.2 `std::less` fallback

If we choose to accept §3.1.3 to generate `==` and `!=` but not ordered comparisons (`<`, `>`, `<=`, `>=`) when there are no user-declared comparisons, some experts would like the `set<Point>` in the previous section to work anyway by default even though `Point` has no `<` comparison.

Two ways to achieve that are: (a) Change `std::set` et al. to use the `std::weak_order()` function in §2.5; this would be a binary breaking change for the ordered associative containers. Alternatively: (b) Extend `std::less` to use the same fallback strategy as the function templates in §2.5, to add additional fallbacks not legal today:

```
template<class T>
struct less {
    constexpr bool operator()(const T& a, const T&b) {
        if constexpr (/* can invoke a < b */) return a < b; // preserves the cases that are legal today
        else if constexpr (/* can invoke weak_order(a,b) */) return weak_order(a,b) < 0;
    }
};
```

3.3 Chaining comparisons

C++17 has added fold expressions, which are very useful. However, as Voutilainen and others have reported, fold expressions do not currently work with comparisons. For example:

```
if (args <...)      // not possible with correct semantics in C++17
```

We can permit two-way comparisons to be chained with the usual pairwise mathematical meaning when the mathematical meaning preserves transitivity (which also always means they have equal precedence). The valid chains are:

- all `==`, such as `a == b == c == d`;
- all {`<`, `<=`}, such as `a < b <= c < d`; and
- all {`>`, `>=`} (e.g., `a >= b > c > d`).

For example, this:

```
if (a < b <= c < d)
```

would be rewritten by the compiler as-if as follows except with single evaluation of `b` and `c`:

```
if ((a < b) && (b <= c) && (c < d))   // but no multiple eval of b and c
```

To illustrate how the compiler would implement this, here is one valid implementation that would satisfy the requirements including single evaluation, by just defining and invoking a lambda:

```
if ([](const auto& a, const auto& b, const auto& c, const auto& d)
    { return a<b && b<=c && c<d; } (a,b,c,d))
```

Notes Chaining support was one alternative suggested by Ville Voutilainen to permit natural use of comparisons in C++17 fold-expressions, such as `if (args <...)`. However, chaining is also broadly useful throughout people's code, so instead of baking the feature into fold-expressions only, it's better to provide general-purpose support that can also express concepts like `first <= iter < last`. Providing general chaining also enables fold-expressions as a special case (and with the "transitive" restriction above avoids the design pitfall of just providing chaining "for all comparison

fold-expressions," when they should correctly be supported "for all comparison fold-expressions *except* `!=`" because `!=` is not transitive).

Without chaining, today we either perform double evaluation or introduce a temporary variable. I've many times wanted to write code like `0 <= expr < max` without either evaluating `expr` twice or else having to invent a temporary variable (and usually a new scope) to store the evaluated value. A number of times, I've actually written the code without thinking, forgetting it wasn't supported, and of course it either didn't compile or did the wrong thing. As an example of "did the wrong thing," this proposal does change the meaning of some code like the following that is legal today, but that is dubious because it probably doesn't do what the programmer intended:

```
int x = 1, y = 3, z = 2;
assert (x < y < z);     // today, means "if (true < 2)" – succeeds
```

In this proposal, the meaning of the condition would be `if ((1 < 3) && (3 < 2))` and the assertion will fire. To use Stroustrup's term, I consider this "code that deserves to be broken;" the change in meaning is probably fixing a bug. (Unless of course we do a code search and find examples that are actually intended.)

Non-chained uses such as `(a<b == c<d)` keep their existing meaning.

4 Sample category types implementation

Notes These types themselves would be much shorter if this paper were adopted, because the bulk of the boilerplate is writing the comparison functions.

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 `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`. The only wacky bug that allows is comparing against `nullptr` itself, and that's a much less likely mistake than accidentally comparing against a nonzero `int` and getting nonsense results.

TODO All of these should additionally be made to be `noexcept constexpr` literal types.

```
#include <stdexcept>

enum class eq   { equal = 0, equivalent = 0, nonequal = 1, nonequivalent = 1 };
enum class ord  { less = -1, greater = 1 };
enum class ncmp { unordered = 255 };

//=============================================================================
//   _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;

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

     // valid values
     static const weak_equality equivalent, nonequivalent;

     // raw value access
     explicit operator int() const { return value; }

     // comparisons (this boilerplate would be mostly eliminated if we could use P0515 itself)
     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; }
     friend bool operator==(weak_equality v, weak_equality v2) { return v.value == v2.value; }
     friend bool operator!=(weak_equality v, weak_equality v2) { return v.value == v2.value; }
};
```

```
const weak_equality weak_equality::equivalent{ eq::equivalent };
const weak_equality weak_equality::nonequivalent{ eq::nonequivalent };

//-----------------------------------------------------------------------
class strong_equality {
     int value;
public:
     // constructors
     explicit strong_equality(eq v) : value{ (int)v } { }

     // valid values
     static const strong_equality equal, equivalent, nonequal, nonequivalent;

     // raw value access
     explicit operator int() const { return value; }

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

     // comparisons (this boilerplate would be mostly eliminated if we could use P0515 itself)
     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; }
     friend bool operator==(strong_equality v, strong_equality v2) { return v.value == v2.value; }
     friend bool operator!=(strong_equality v, strong_equality v2) { return v.value == v2.value; }
};
const strong_equality strong_equality::equal{ eq::equal };
const strong_equality strong_equality::equivalent{ eq::equivalent };        // for convenient substitutability in generic code
const strong_equality strong_equality::nonequal{ eq::nonequal };
const strong_equality strong_equality::nonequivalent{ eq::nonequivalent };  // for convenient substitutability in generic code


//=======================================================================
//    _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 {
public:
     struct result {
          int cmp : 7;
          bool unordered : 1;
     };
private:
     result value;

public:
     // constructors
     explicit partial_ordering(eq  v) : value{ (int)v, false } { }
     explicit partial_ordering(ord v) : value{ (int)v, false } { }
     explicit partial_ordering(ncmp ) : value{   -255, true  } { }

     // valid values
     static const partial_ordering less, equivalent, greater, unordered;
```

```
    // raw value access
    operator result() const { return value; }

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

    // comparisons (this boilerplate would be mostly eliminated if we could use P0515 itself)
    friend bool operator==(partial_ordering v, nullptr_t) { return !v.value.unordered && v.value.cmp == 0; }
    friend bool operator!=(partial_ordering v, nullptr_t) { return  v.value.unordered || v.value.cmp != 0; }
    friend bool operator< (partial_ordering v, nullptr_t) { return !v.value.unordered && v.value.cmp <  0; }
    friend bool operator<=(partial_ordering v, nullptr_t) { return !v.value.unordered && v.value.cmp <= 0; }
    friend bool operator> (partial_ordering v, nullptr_t) { return !v.value.unordered && v.value.cmp >  0; }
    friend bool operator>=(partial_ordering v, nullptr_t) { return !v.value.unordered && v.value.cmp >= 0; }
    friend bool operator==(nullptr_t, partial_ordering v) { return !v.value.unordered && 0 == v.value.cmp; }
    friend bool operator!=(nullptr_t, partial_ordering v) { return  v.value.unordered || 0 != v.value.cmp; }
    friend bool operator< (nullptr_t, partial_ordering v) { return !v.value.unordered && 0 <  v.value.cmp; }
    friend bool operator<=(nullptr_t, partial_ordering v) { return !v.value.unordered && 0 <= v.value.cmp; }
    friend bool operator> (nullptr_t, partial_ordering v) { return !v.value.unordered && 0 >  v.value.cmp; }
    friend bool operator>=(nullptr_t, partial_ordering v) { return !v.value.unordered && 0 >= v.value.cmp; }
    friend bool operator==(partial_ordering v, partial_ordering v2) { return v.value.unordered == v2.value.unordered && v.value.cmp == v2.value.cmp; }
    friend bool operator!=(partial_ordering v, partial_ordering v2) { return v.value.unordered != v2.value.unordered || v.value.cmp != v2.value.cmp; }
};
const partial_ordering partial_ordering::less{ ord::less };
const partial_ordering partial_ordering::equivalent{ eq::equivalent };
const partial_ordering partial_ordering::greater{ ord::greater };
const partial_ordering partial_ordering::unordered{ ncmp::unordered };

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

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

    // valid values
    static const weak_ordering less, equivalent, greater;

    // raw value access
    explicit operator int() const { return value; }

    // 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 (this boilerplate would be mostly eliminated if we could use P0515 itself)
    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; }
    friend bool operator==(weak_ordering v, weak_ordering v2) { return v.value == v2.value; }
    friend bool operator!=(weak_ordering v, weak_ordering v2) { return v.value != v2.value; }
};
const weak_ordering weak_ordering::less{ ord::less };
const weak_ordering weak_ordering::equivalent{ eq::equivalent };
const weak_ordering weak_ordering::greater{ ord::greater };

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

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

    // valid values
    static const strong_ordering less, equal, equivalent, greater;

    // raw value access
    xplicit operator int() const { return value; }

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

    // comparisons (this boilerplate would be mostly eliminated if we could use P0515 itself)
    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; }
    friend bool operator==(strong_ordering v, strong_ordering v2) { return v.value == v2.value; }
    friend bool operator!=(strong_ordering v, strong_ordering v2) { return v.value != v2.value; }
};
const strong_ordering strong_ordering::less{ ord::less };
const strong_ordering strong_ordering::equal{ eq::equal };
const strong_ordering strong_ordering::equivalent{ eq::equivalent };        // for convenient substitutability in generic code
const strong_ordering strong_ordering::greater{ ord::greater };
```

5 Comparison with recent proposals

This paper presents a design for comparisons that separates the core proposal and previously controversial design points so that the proposal can proceed with or without those optional parts. Nothing in the core proposal depends on any of the optional parts.

5.1 Core proposal Q&A

5.1.1 Should we support three-way comparison?

Yes, per EWG sentiment in Issaquah (see §1.1), and because otherwise in common cases we cannot reliably generate `<=` that is both correct and efficient. This paper proposes adding a new `operator<=>` (see §2.2).

Similarities to previous proposals: This paper follows the arguments for three-way comparisons in P0100R2 (Crowl) and that paper's comparison category enumerations which are expressed herein as distinct types. It also follows existing practice in essentially all mainstream modern languages (for a partial list, see §2.2.2).

Differences from previous proposals: Other recent proposals did not include three-way comparison, but often just to simplify the proposal. Notably, N4475 (Stroustrup) did not propose three-way comparison, but as a compromise motivated by time pressure: It was felt that EWG would not be able to reach agreement on adding three-way comparison in time to get default comparisons into C++17. That time pressure no longer applies.

5.1.2 How should programmers write comparison for their type?

They nearly always write a single function, `operator<=>`:

To model this	Write `operator<=>` that returns a	And we'll generate a@b as **`a<=>b @ 0`** for
Total order	`std::strong_ordering`	`< > <= >= == !=`
Weak order	`std::weak_ordering`	`< > <= >= == !=`
Partial order	`std::partial_ordering`	`< > <= >= == !=`
Equality comparable	`std::strong_equality`	`== !=`
Equivalence comparable	`std::weak_equality`	`== !=`

Notes Bikeshed: We can easily instead follow the standard mathematical terms from the first column (e.g., `equivalence_comparison` instead of `weak_equality`), but this orthogonal naming is easier to teach and avoids FAQs like "are `weak_ordering` and `equivalence_comparison` related?" The answer is easier to teach (and more obvious) with the names `weak_ordering` and `weak_equality`: "yes, they differ from `strong_ordering` and `strong_equality`, respectively, in exactly the same way."

I like and prefer the consistent naming of `strong` and `weak`, but changing `strong` to `total` would even more closely match the mathematical terms.

Similarities to previous proposals: Same as in §5.1.1.

Differences from previous proposals: The other recent proposals that did not have three-way comparisons required writing all operators, or specific operators that would be combined in various ways to generate others. This caused design disagreements (see also §5.1.3), and moved design asymmetries around to different places.

5.1.3 How should a <= b be generated?

As `a<=>b <= 0`, which is both efficient (single-pass) and correct for totally ordered and weakly ordered types.

Similarities to previous proposals: P0100R2 (Crowl) mentions in passing that `<=` can be generated from three-way comparison, but relies on "as-if" whereas this paper makes it a requirement.

Differences from previous proposals: Other proposals proposed generation from `<` and `!` which doesn't work for partially ordered types, or from `<` and `=` which is two-pass and so can be less efficient and violates the zero-overhead principle; neither is ideal.

5.1.4 What should default `<=>` comparison semantics be?

Default comparison should follow default copying: It should have memberwise semantics.

Similarities to previous proposals: This paper follows P0100R2 (Crowl), P0481R0 (Van Eerd), and P0432R0 (Stone).

Differences from previous proposals: The major difference from N4475 (Stroustrup) is that this paper avoids special cases. It proposes a simple rule, without resorting to skipping `mutable` members. Default comparison should do exactly the same thing as default copying.

5.1.5 Should compiler-generated comparisons create real functions?

No. They should be built-in (for `<=>`) or synonyms/rewrites (for the others), not proliferate functions. (User-declared defaulted comparisons are functions as usual.)

Similarities to previous proposals: This paper follows N4475 (Stroustrup), P0436R1 (Brown), P0481R0 (Van Eerd), and P0432R0 (Stone) to generate rewrite rules (e.g., "reinterpretation" in P0436) instead of additional functions.

Differences from previous proposals: P0100R2 (Crowl) is agnostic about whether defaulted comparisons are functions or some other mechanism such as rewrites, but this paper is definite that they should be rewrites.

5.1.6 What about floating point types?

Floating point types will gain a built-in `<=>` that returns `partial_ordering` (see §2.2.3).

IEEE 754 floating point types also get a specialization of the new `std::strong_order` that use IEEE 754 `totalOrder` (see §2.5).

5.1.7 Should there be a terse syntax to declare a comparison operator?

No, but nothing in this proposal precludes that as a separate later proposal. In this proposal, the type author always writes exactly zero or one comparison function, so there is little motivation to add novel syntax to the language to make writing the one function declaration any terser.

Similarities to previous proposals: This paper follows N4475 (Stroustrup), P0100R2 (Crowl), P0436R1 (Brown), and P0432R0 (Stone) in not proposing a novel comparison function declaration syntax. In addition to nearly all the other proposals, a number of people in EWG discussions agreed with not adding a novel declaration syntax.

Differences from previous proposals: P0481R0 (Van Eerd) proposes a novel comparison function declaration syntax motivated by terseness, but this paper's ability to write only a single function already removes most or all of the verbosity that the terse declaration syntax would mitigate (and if §3.1 Option 2 or 3 is accepted that removes even more).

5.1.8 Should there be a terse syntax to define a memberwise comparison body?

Yes. When you need to write a comparison function, it is useful to be able to ask for default memberwise semantics, which is most of the work of writing the function. Note that this argument directly parallels special member functions: When we want to opt into the implicit semantics, that's exactly what `=default` was

invented for. Therefore, this paper proposes simply using `=default` for the identical motivation, purpose, and semantics also for comparison functions.

Similarities to previous proposals: This paper follows P0481R0 (Van Eerd) and P0432R0 (Stone) with `=default`.

Differences from previous proposals: This paper draws a distinction between declaration and definition, and so it agrees with N4475 (Stroustrup), P0100R2 (Crowl), and P0436R1 (Brown) to avoid a terse declaration syntax (see §5.1.7), but disagrees with them by providing a terse definition syntax.

5.2 Optional design question: Generating `<=>` (see §3)

Whether `<=>` comparison should be implicitly generated when not user-defined, and if so which comparisons, and under what conditions, is an optional design point; see §3.1. Nothing in the core proposal depends on this.

If §3.1 Option 2 or 3 is accepted, it would be a simple rule that parallels copying, per EWG sentiment in Issaquah (see §1.1). This paper's optional suggestion follows the arguments favoring generation in N4475 (Stroustrup), P0481R0 (Van Eerd), and P0432R0 (Stone). Default comparison can be reliably generated when the class has non-deprecated default copying (construction and assignment, which are therefore known to be memberwise) that do not modify their source parameter (and so are known to be nonmodifying so that the source and target can be reliably compared) and no user-defined comparisons.

Similarities to previous proposals: All of the proposals that include generated comparisons have seen the need to align comparison with copying in some way. This proposal is closest to P0481R0 (Van Eerd) to generate default comparison based on non-deprecated default copying without additional special rules.

Differences from previous proposals: The major difference from N4475 (Stroustrup) and P0432R0 (Stone) is that this paper avoids special cases and relying on heuristics to guess the programmer's intent. It proposes a simple rule, without resorting to special reasoning about members that are pointers, references, `mutable`, `volatile`, etc. Memberwise comparison should be generated by default iff we know its semantics are the same as memberwise copying, which means in exactly the cases when copying is defaulted (and doesn't modify its source object). Adding special rules to suppress default comparisons for special cases such as virtual functions/bases or pointer/`mutable`/`volatile` members seems unnecessary, because types that have those should not be using default copying anyway. If a type with a pointer or `mutable` or `volatile` member, or a virtual function or base, does support default copying, that's either because it's appropriate (in which case default memberwise comparison is also appropriate), or it's a bug where the type will have much bigger problems than merely comparisons and it should either suppress copying or provide non-default copying (which would also suppress generating comparisons). As long as we simply keep default comparison consistent with non-deprecated default copying, the right things will happen for comparison.

5.3 Summary: Side by side comparison with previous proposals

Here is a brief summary of the foregoing. Shaded cells highlight the differences from this paper.

	N4126 (Smolsky)	N4475 (Stroustrup) + P0221R2 (Maurer)	P0100R2 (Crowl)	P0436R1 (Brown)	P0481R0 (Van Eerd)	P0432R0 (Stone)	This paper
Core proposal (§2)							
Three-way comparison	No	No	Yes	No	No	No	Yes
==, != comparison	Yes	Yes	Yes	Yes	Yes	Yes	Yes
<, <=, >, >= comparison	Yes	Yes	Yes	Yes	No	Yes	Yes
Generated <= is single-pass	Yes	No	As-if	Yes	n/a	No	Yes
Generated <= is correct for partially ordered types	Yes	Yes	Yes	No	n/a	Yes	Yes
Supports partial ordering	Implicitly	No	Yes	No	No	No	Yes
Rewrites vs. real functions	Functions	Rewrites	Functions	Rewrites	Rewrites	Rewrites	Rewrites
Opt-in using shorter declaration	Yes	No	No	No	Yes	No	No
Opt-in to memberwise body	Yes, `=default`	No	No for operators; yes for functions, `=default`	No	Yes, `=default`	Yes, `=default`	Yes, `=default`
Default semantics	Memberwise	Memberwise	Memberwise	n/a	Memberwise	Memberwise	Memberwise
Optional feature: Generation (§3)							
Implicit generation, when no user-defined comparisons	No	Yes, for all	No	No	Yes, for `==` and `!=`	Yes, for `==` and `!=`	No, `==`/`!=`, or all (see §3.1)
Implicit generation conditions	n/a	Default copy constructor and copy assignment, no pointer members, no virtual functions, no mutable members	n/a	n/a	Default copy constructor	Default copy constructor, no mutable members, no virtual functions, no virtual bases	Non-deprecated default copy constructor and copy assignment

6 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. 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).