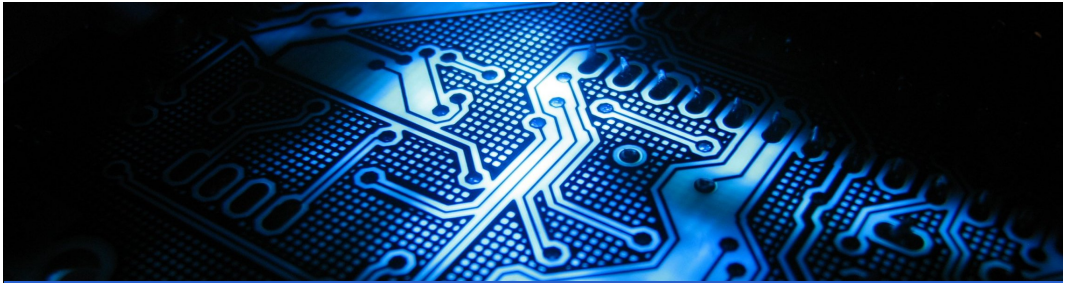


Using C++20's Three-way Comparison <=>

Jonathan Müller — @foonathan — CC BY 4.0

Type Classification

Guideline 0

Every type should either

1. overload ==, !=, <, <=, >=, *and* > (**ordering**);
2. overload only == and != (**equality**);
3. overload none of those.

Type Classification

Guideline 0

Every type should either

1. overload ==, !=, <, <=, >=, *and* > (**ordering**);
2. overload only == and != (**equality**);
3. overload none of those.

In particular: don't overload only <, <=, >=, >.

Copying and Equality

Copying and Equality

```
struct A
{
    int i;

    A(const A&) = default;
};

bool operator==(const A& lhs, const A& rhs)
{
    return lhs.i == rhs.i;
}
```

What we copy	What we don't copy	What we compare	Good Type?
integer `i`	nothing	integer `i`	yes

Copying and Equality

```
struct B
{
    std::mutex m;
    int i;

    B(const B& other) : i(other.i) {}
};

bool operator==(const B& lhs, const B& rhs)
{
    return lhs.i == rhs.i;
}
```

What we copy	What we don't copy	What we compare	Good Type?
integer `i`	mutex `m`	integer `i`	yes

Copying and Equality

```cpp
struct C
{
    std::vector<int> v;

    C(const C&) = default;
};

bool operator==(const C& lhs, const C& rhs)
{
    return lhs.v == rhs.v;
}
```

What we copy	What we don't copy	What we compare	Good Type?
elements of v	capacity of v	elements of v	yes

Copying and Equality

```
struct D
{
    int a, b;

    D(const D&) = default;
};

bool operator==(const D& lhs, const D& rhs)
{
    return lhs.a == rhs.a;
}
```

What we copy	What we don't copy	What we compare	Good Type?
integer a, b	nothing	integer a	maybe

Copying and Equality

```
struct E
{
    int a, b;

    E(const E& other) : a(other.a), b(42) {}
};

bool operator==(const E& lhs, const E& rhs)
{
    return lhs.a == rhs.a && lhs.b == rhs.b;
}
```

What we copy	What we don't copy	What we compare	Good Type?
integer a	integer b	integer a, b	no

Copying and Equality

Guideline 1

What is compared should be a subset of what is copied.

- **Strong Equality**: everything that is copied is compared
- **Weak Equality**: a proper subset of what is copied is compared

Equality and Ordering

Equality and Ordering

Two objects `a`, `b` of an ordered type:

- `a` and `b` are equal
- `a` is less than `b`
- `a` is greater than `b`

That's a relationship!

Sean Parent

Equality and Ordering

Relationships between `a`, `b` in an ordered type:

- a and b are equal
- a is less than b
- a is greater than b

Equality and Ordering

Relationships between `a`, `b` in an ordered type:

- `a` and `b` are equal
- `a` is less than `b`
- `a` is greater than `b`

But comparison operators give us only a `bool`!

Equality and Ordering

< is all you need:

- a is less than b: `a < b`

Equality and Ordering

< is all you need:

- a is less than b: `a < b`
- a is greater than b: `b < a`

Equality and Ordering

< is all you need:

- a is less than b: `a < b`
- a is greater than b: `b < a`
- a and b are “equal”: `!(a < b) && !(b < a)`

Equality and Ordering

Equivalence of a and b: `!(a < b) && !(b < a)`

Equality of a and b: `a == b`

Equality and Ordering

Equivalence of a and b: `!(a < b) && !(b < a)`

Equality of a and b: `a == b`

Guideline 2

Equivalence and equality of a type should be identical.

`a <= b` should mean `a < b || a == b`.

Equality and Ordering

Equivalence of a and b: `!(a < b) && !(b < a)`

Equality of a and b: `a == b`

Guideline 2

Equivalence and equality of a type should be identical.

`a <= b` should mean `a < b || a == b`.

Corollary

The same things should be compared in == and <.

Taxonomy of Comparable Types

Taxonomy	`==`, `!=`	`==`, `!=`, `<`, `<=`, `>=`, `>`
compares all copied members	strong equality	strong ordering
compares some copied members	weak equality	weak ordering

Three-Way Comparison

Relationships between `a`, `b` in an ordered type:

- a and b are equal
- a is less than b
- a is greater than b

Three-Way Comparison

Relationships between `a`, `b` in an ordered type:

- `a` and `b` are equal
- `a` is less than `b`
- `a` is greater than `b`

Return relationship directly!

Three-Way Comparison

`std::strcmp(a, b)`:

- a and b are equal: `== 0`
- a is less than b: `< 0`
- a is greater than b: `> 0`

C++20 Three-Way Comparison

C++20 Three-Way Comparison

Language:

- new operator `<=>` for a three-way comparison
- new operator rewrite rules for comparisons
- ability to `= default` comparison operators

Library (mostly in header `<compare>`):

- comparison categories: `std::strong_ordering`, `std::weak_ordering`
- concepts, utility functions
- completely transparent removal of many comparison operators

C++20 Three-Way Comparison

Language:

- new operator `<=>` for a three-way comparison
- new operator rewrite rules for comparisons
- ability to `= default` comparison operators

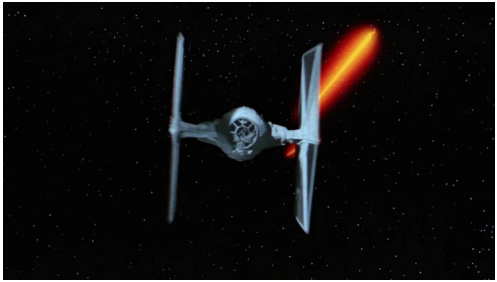
Library (mostly in header `<compare>`):

- comparison categories: `std::strong_ordering`, `std::weak_ordering`
- concepts, utility functions
- completely transparent removal of many comparison operators

Resources: jonathanmueller.dev/talk/cppcon2019

Spaceship?

C++20 Three-Way Comparison



| = |

TIE fighter.

https://www.starwars.com/databank/tie-fighter

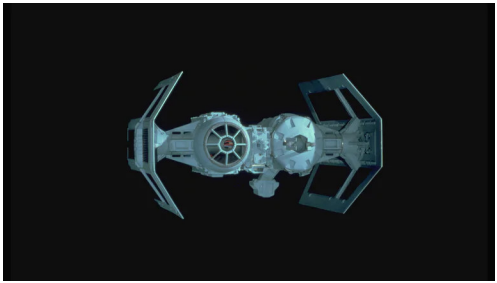
C++20 Three-Way Comparison

<=>

Darth Vader's TIE fighter.

https://www.starwars.com/databank/darth-vader-s-tie-fighter

C++20 Three-Way Comparison

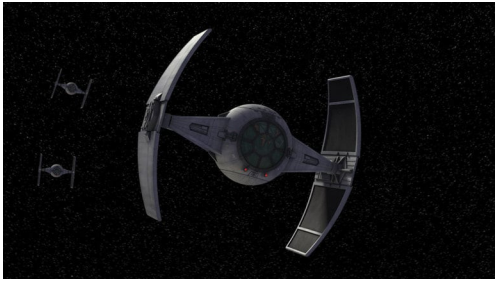


<=>

TIE Bomber.

https://www.starwars.com/databank/tie-bomber

C++20 Three-Way Comparison

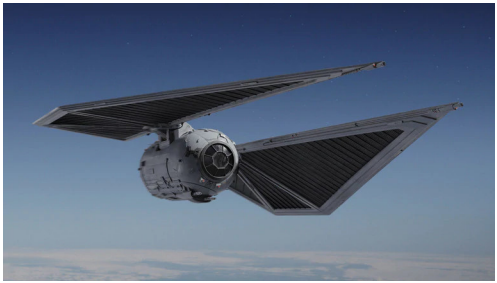


(=)

Inquisitor's TIE Advanced Prototype.

https://www.starwars.com/databank/inquisitor-s-tie-advanced-prototype

C++20 Three-Way Comparison



/o\

TIE Striker.

https://www.starwars.com/databank/tie-striker

C++20 Three-Way Comparison

`a <=> b`:

- a and b are equal: `== 0`
- a is less than b: `< 0`
- a is greater than b: `> 0`

C++20 Three-Way Comparison

`a <=> b`:

- a and b are equal: `== 0`
- a is less than b: `< 0`
- a is greater than b: `> 0`

But `<=>` does not return `int`.

C++20 Three-Way Comparison

Wrong, but useful:

```
enum class weak_ordering
{
    less = -1,
    equivalent = 0,
    greater = 1
};

// + comparison with 0
```

```
enum class strong_ordering
{
    less = -1,
    equal = 0,
    equivalent = 0,
    greater = 1
};

// + comparison with 0
```

Writing Comparison: Application Classes

Writing Comparison: Application Classes

```
struct Point
{
    int x, y;
};
```

Goal: member-wise equality and inequality comparison.

Writing Comparison: Application Classes

Pre C++20

```cpp
struct Point
{
    int x, y;
};

bool operator==(const Point& lhs, const Point& rhs)
{
    return lhs.x == rhs.x && lhs.y == rhs.y;
}

bool operator!=(const Point& lhs, const Point& rhs)
{
    return !(lhs == rhs);
}
```

Writing Comparison: Application Classes

C++20

```
struct Point
{
    int x, y;

    bool operator==(const Point&) const = default;
};
```

Writing Comparison: Application Classes

```
Point a = {0, 0};
Point b = {0, 1};
```

```
a == b          →
```

Writing Comparison: Application Classes

```
Point a = {0, 0};
Point b = {0, 1};
```

```
a == b        →        a.operator==(b)
```

Writing Comparison: Application Classes

```
Point a = {0, 0};
Point b = {0, 1};
```

```
a == b    →    a.operator==(b)
a != b    →
```

Writing Comparison: Application Classes

```
Point a = {0, 0};
Point b = {0, 1};
```

```
a == b        →        a.operator==(b)
a != b        →        !(a.operator==(b))
```

Writing Comparison: Application Classes

```
struct PointLike { operator Point(); };

Point p = {0, 0};
PointLike pl;
```

```
pl == p          →
```

Writing Comparison: Application Classes

```
struct PointLike { operator Point(); };

Point p = {0, 0};
PointLike pl;
```

```
pl == p  →  p.operator==(pl)
```

Interlude: Why Member Function?

Pre C++20

Guideline

Make comparison operator overloads free functions.

- required for conversions on the left-hand side

Interlude: Why Member Function?

C++20

Guideline

Make comparison operator overloads member functions:

- it just works now
- no `friend` needed
- fewer overload resolution candidates:
  - potential for faster compilation
  - better error messages (!)

Writing Comparison: Application Classes

C++20

Technique

To get member-wise equality and inequality comparison:

```
class C
{
public:
    bool operator==(const C&) const = default;
};
```

Writing Comparison: Application Classes

```
struct Person
{
    PersonID id;
    std::string name;
    …
};
```

Goal: equality and inequality comparison of `Person` forwarding to `id` member.

Writing Comparison: Application Classes

Pre C++20

```
struct Person
{
    PersonID id;
    std::string name;
    …
};

bool operator==(const Person& lhs, const Person& rhs)
{
    return lhs.id == rhs.id;
}
bool operator!=(const Person& lhs, const Person& rhs)
{
    return lhs.id != rhs.id;
}
```

Writing Comparison: Application Classes

C++20

```
struct Person
{
    PersonID id;
    std::string name;
    …

    bool operator==(const Person& other) const
    {
        return other.id == id;
    }
};
```

Writing Comparison: Application Classes

C++20

```
struct Person
{
    PersonID id;
    std::string name;
    …

    bool operator==(const Person& other) const
    {
        return other.id == id;
    }
};
```

Goal: equality and inequality comparison of `Person` with `PersonID`.

Writing Comparison: Application Classes

Pre C++20

```
struct Person
{
    PersonID id;
    std::string name;
    …
};

bool operator==(const Person& lhs, const Person& rhs);
bool operator!=(const Person& lhs, const Person& rhs);

bool operator==(const Person& lhs, PersonID rhs) { return lhs.id == rhs;
bool operator==(PersonID lhs, const Person& rhs) { return lhs == rhs.id;
bool operator!=(const Person& lhs, PersonID rhs) { return lhs.id != rhs;
bool operator!=(PersonID lhs, const Person& rhs) { return lhs != rhs.id;
```

Writing Comparison: Application Classes

C++20

```
struct Person
{
    PersonID id;
    std::string name;
    …

    bool operator==(const Person& other) const
    {
        return id == other.id;
    }
    bool operator==(PersonID other) const
    {
        return id == other;
    }
};
```

Writing Comparison: Application Classes

```
Person p;
PersonID id;
```

```
p == id          →
```

Writing Comparison: Application Classes

```
Person p;
PersonID id;
```

```
p == id        →        p.operator==(id)
```

Writing Comparison: Application Classes

```
Person p;
PersonID id;
```

```
p == id        →        p.operator==(id)
id == p        →
```

Writing Comparison: Application Classes

```
Person p;
PersonID id;
```

```
p == id        →        p.operator==(id)
id == p        →        p.operator==(id)
```

Writing Comparison: Application Classes

```
Person p;
PersonID id;
```

```
p == id     →     p.operator==(id)
id == p     →     p.operator==(id)
id != p     →
```

Writing Comparison: Application Classes

```
Person p;
PersonID id;
```

```
p == id     →     p.operator==(id)
id == p     →     p.operator==(id)
id != p     →     !(p.operator==(id))
```

Writing Comparison: Application Classes

C++20

Technique

To get custom or mixed equality and inequality comparison:

```
class C
{
public:
    bool operator==(const T& other) const
    {
        // compare *this and other by calling == of members
    }
};
```

Writing Comparison: Application Classes

```
struct PersonID
{
    int impl;
};
```

Goal: full member-wise ordering.

Writing Comparison: Application Classes

Pre C++20

```
struct PersonID
{
    int impl;
};

bool operator==(PersonID lhs, PersonID rhs) { return lhs.impl == rhs.impl
bool operator<(PersonID lhs, PersonID rhs)  { return lhs.impl < rhs.impl

bool operator!=(PersonID lhs, PersonID rhs) { return !(lhs == rhs); }
bool operator>(PersonID lhs, PersonID rhs)  { return rhs < lhs; }
bool operator<=(PersonID lhs, PersonID rhs) { return !(rhs < lhs); }
bool operator>=(PersonID lhs, PersonID rhs) { return !(lhs < rhs); }
```

Writing Comparison: Application Classes

C++20

```
struct PersonID
{
    int impl;

    std::strong_ordering operator<=>(const PersonID&) const = default;
};
```

Writing Comparison: Application Classes

```
struct PersonIDLike { operator PersonID(); };

PersonID a, b;
PersonIDLike c;
```

a < b →

Writing Comparison: Application Classes

```
struct PersonIDLike { operator PersonID(); };

PersonID a, b;
PersonIDLike c;
```

```
a < b        →        a.operator<=>(b) < 0
```

Writing Comparison: Application Classes

```
struct PersonIDLike { operator PersonID(); };

PersonID a, b;
PersonIDLike c;
```

```
a < b     →     a.operator<=>(b) < 0
a >= b    →
```

Writing Comparison: Application Classes

```
struct PersonIDLike { operator PersonID(); };

PersonID a, b;
PersonIDLike c;
```

```
a < b     →     a.operator<=>(b) < 0
a >= b    →     a.operator<=>(b) >= 0
```

Writing Comparison: Application Classes

```
struct PersonIDLike { operator PersonID(); };

PersonID a, b;
PersonIDLike c;
```

```
 a < b   →   a.operator<=>(b) < 0
a >= b   →   a.operator<=>(b) >= 0
 c > b   →
```

Writing Comparison: Application Classes

```
struct PersonIDLike { operator PersonID(); };

PersonID a, b;
PersonIDLike c;
```

`a < b`	→	`a.operator<=>(b) < 0`
`a >= b`	→	`a.operator<=>(b) >= 0`
`c > b`	→	`0 > b.operator<=>(c)`

Writing Comparison: Application Classes

```
struct PersonIDLike { operator PersonID(); };

PersonID a, b;
PersonIDLike c;
```

`a < b`	→	`a.operator<=>(b) < 0`
`a >= b`	→	`a.operator<=>(b) >= 0`
`c > b`	→	`0 > b.operator<=>(c)`
`a == b`	→	

Writing Comparison: Application Classes

```
struct PersonIDLike { operator PersonID(); };

PersonID a, b;
PersonIDLike c;
```

```
 a < b    →    a.operator<=>(b) < 0
a >= b    →    a.operator<=>(b) >= 0
 c > b    →    0 > b.operator<=>(c)
a == b    →    a.operator==(b)
```

Writing Comparison: Application Classes

C++20

```cpp
struct PersonID
{
    int impl;

    std::strong_ordering operator<=>(const PersonID&) const = default;

    // bool operator==(const PersonID&) const = default;
};
```

Writing Comparison: Application Classes

C++20

Technique

To get full member-wise ordering:

```
class C
{
public:
    std::strong_ordering operator<=>(const C&) const = default;
};
```

Writing Comparison: Application Classes

```
struct ID
{
    ByteString value;
    int domain;
};
```

Goal: member-wise comparison, but `domain` first then `value` (and member order can't be changed).

Writing Comparison: Application Classes

Pre C++20

```
bool operator==(const ID& lhs, const ID& rhs)
{
    return lhs.domain == rhs.domain && lhs.value == rhs.value;
}
bool operator<(const ID& lhs, const ID& rhs)
{
    if (lhs.domain != rhs.domain)
        return lhs.domain < rhs.domain;
    return lhs.value < rhs.value;
}

bool operator!=(ID lhs, ID rhs) { return !(lhs == rhs); }
bool operator>(ID lhs, ID rhs)  { return rhs < lhs; }
bool operator<=(ID lhs, ID rhs) { return !(rhs < lhs); }
bool operator>=(ID lhs, ID rhs) { return !(lhs < rhs); }
```

Writing Comparison: Application Classes

C++20

```
std::strong_ordering ID::operator<=>(const ID& other) const
{
    if (std::strong_ordering cmp = domain <=> other.domain;
        cmp != 0)
        return cmp;

    return value <=> other.value;
}
```

Writing Comparison: Application Classes

C++20

```
std::strong_ordering ID::operator<=>(const ID& other) const
{
    if (std::strong_ordering cmp = domain <=> other.domain;
        cmp != 0)
        return cmp;

    if (value == other.value)
        return std::strong_ordering::equal;
    else if (value < other.value)
        return std::strong_ordering::less;
    else
        return std::strong_ordering::greater;
}
```

Writing Comparison: Application Classes

C++20

```
template <typename T, std::totally_ordered_with<T> U>
auto synth_three_way(const T& lhs, const U& rhs)
{
    …
}
```

Writing Comparison: Application Classes

C++20

```
template <typename T, std::totally_ordered_with<T> U>
auto synth_three_way(const T& lhs, const U& rhs)
{
    if constexpr (std::three_way_comparable_with<T, U>)
        return lhs <=> rhs;
    else
        …
}
```

Writing Comparison: Application Classes

C++20

```cpp
template <typename T, std::totally_ordered_with<T> U>
auto synth_three_way(const T& lhs, const U& rhs)
{
    if constexpr (std::three_way_comparable_with<T, U>)
        return lhs <=> rhs;
    else
    {
        if (lhs == rhs)
            return std::strong_ordering::equal;
        else if (lhs < rhs)
            return std::strong_ordering::less;
        else
            return std::strong_ordering::greater;
    }
}
```

Writing Comparison: Application Classes

C++20

```
std::strong_ordering ID::operator<=>(const ID& other) const
{
    if (auto cmp = domain <=> other.domain;
        cmp != 0)
        return cmp;

    return synth_three_way(value, other.value);
}
```

Writing Comparison: Application Classes

C++20

```
std::strong_ordering ID::operator<=>(const ID& other) const
{
    if (auto cmp = domain <=> other.domain;
        cmp != 0)
        return cmp;

    return synth_three_way(value, other.value);
}

bool ID::operator==(const ID& other) const
{
    return lhs.domain == rhs.domain && lhs.value == rhs.value;
}
```

Writing Comparison: Application Classes

C++20

Technique

To get custom or mixed ordering:

```
class C
{
public:
    std::strong_ordering operator<=>(const T& other) const
    {
        // compare with <=> or a synth_three_way helper
    }
};
```

And implement equality by providing `operator==`.

Writing Comparison: Application Classes

C++20

```cpp
struct ID
{
    // swapped members!
    int domain;
    ByteString value;

    std::strong_ordering operator<=>(const ID&) const = default;
};
```

Using Comparison: Algorithms

Using Comparison: Algorithms

```
template <typename Iter>
Iter min_element(Iter begin, Iter end)
{
    // return minimum according to `<`
}
```

Using Comparison: Algorithms

Pre C++20

```
template <typename Iter>
Iter min_element(Iter begin, Iter end)
{
    Iter min = begin;
    for (auto cur = begin; cur != end; ++cur)
    {
        if (*cur < *min)
            min = cur;
    }
    return min;
}
```

Using Comparison: Algorithms

C++20

```
template <typename Iter>
Iter min_element(Iter begin, Iter end)
{
    Iter min = begin;
    for (auto cur = begin; cur != end; ++cur)
    {
        if (*cur < *min)
            min = cur;
    }
    return min;
}
```

Using Comparison: Algorithms

C++20

Guideline

If you only need ==, <, >= etc., just call it; don't call <=>.

Using Comparison: Algorithms

```cpp
template <typename Iter, typename T>
bool binary_search(Iter begin, Iter end, const T& value)
{
    // return true if value is in [begin, end)
    // range is sorted according to `<`
}
```

Assumption: we can use all comparison operators.

Using Comparison: Algorithms

Pre C++20

```
template <typename Iter, typename T>
bool binary_search(Iter begin, Iter end, const T& value)
{
    if (begin == end)
        return false; // empty range, value not there

    auto mid = begin + std::distance(begin, end) / 2;
    if (value == *mid)
        return true; // found it
    else if (value < *mid)
        return binary_search(begin, mid, value); // first half
    else
        return binary_search(mid + 1, end, value); // second half
}
```

(**Note:** not a great binary search implementation)

Using Comparison: Algorithms

C++20

```
auto mid = begin + std::distance(begin, end) / 2;
if (auto cmp = value <=> *mid;
    cmp == 0)
    return true; // found it
else if (cmp < 0)
    return binary_search(begin, mid, value); // first half
else
    return binary_search(mid + 1, end, value); // second half
```

Using Comparison: Algorithms

C++20

```
auto mid = begin + std::distance(begin, end) / 2;
if (auto cmp = synth_three_way(value, *mid);
    cmp == 0)
    return true; // found it
else if (cmp < 0)
    return binary_search(begin, mid, value); // first half
else
    return binary_search(mid + 1, end, value); // second half
```

Using Comparison: Algorithms

C++20

Guideline

Don't call <=> in generic code; use something like `synth_three_way`.

Using Comparison: Algorithms

Pre C++20

```
template <typename Iter, typename T, class Predicate>
bool binary_search(Iter begin, Iter end, const T& value, Predicate p)
{
    // return true if value is in [begin, end)
    // range is partioned according to p
}
```

Using Comparison: Algorithms

Pre C++20

```
template <typename Iter, typename T, class Predicate>
bool binary_search(Iter begin, Iter end, const T& value, Predicate p)
{
    // return true if value is in [begin, end)
    // range is partioned according to p
}
```

- `p(a, b)` means a is “less than” b
- `!p(a, b) && !p(b, a)` means a is “equivalent to” b

Using Comparison: Algorithms

Pre C++20

```
auto mid = begin + std::distance(begin, end) / 2;
if (p(value, *mid))
    return binary_search(begin, mid, value); // first half
else if (p(*mid, value))
    return binary_search(mid + 1, end, value); // second half
else
    return true; // found it
```

Using Comparison: Algorithms

C++20

```
template <typename Iter, typename T, class Ordering>
bool binary_search(Iter begin, Iter end, const T& value, Ordering o)
{
    // return true if value is in [begin, end)
    // range is partitioned according to o
}
```

Using Comparison: Algorithms

C++20

```
template <typename Iter, typename T, class Ordering>
bool binary_search(Iter begin, Iter end, const T& value, Ordering o)
{
    // return true if value is in [begin, end)
    // range is partitioned according to o
}
```

- `o(a, b) == 0` if a and b are “equivalent”
- `o(a, b) < 0` if a is “less than” b
- `o(a, b) > 0` if a is “greater than” b

Using Comparison: Algorithms

C++20

```
auto mid = begin + std::distance(begin, end) / 2;
if (auto cmp = o(value, *mid);
    cmp == 0)
    return true; // found it
else if (cmp < 0)
    return binary_search(begin, mid, value); // first half
else
    return binary_search(mid + 1, end, value); // second half
```

Using Comparison: Algorithms

C++20

```
template <typename T, typename U>
std::strong_ordering my_order(const T& a, const U& b);
```

- `my_order(a, b) == 0` means `a == b`

e.g. <=>

Using Comparison: Algorithms

C++20

```
template <typename T, typename U>
std::strong_ordering my_order(const T& a, const U& b);
```

- `my_order(a, b) == 0` means `a == b`

e.g. <=>

```
template <typename T, typename U>
std::weak_ordering my_weaker_order(const T& a, const U& b);
```

- `my_weaker_order(a, b) == 0` does not necessarily mean `a == b`
- but: `a == b` means `my_weaker_order(a, b) == 0`!

e.g. `case_insensitive_compare`

Using Comparison: Algorithms

C++20

Guideline

Consider a three-way ordering instead of a “less than” predicate if you need three-way comparison.

- `order(a, b) == 0`
- `order(a, b) < 0`
- `order(a, b) > 0`

Using Comparison: Algorithms

C++20

Guideline

Use `std::weak_ordering` (only) as the return type of a three-way ordering with a weaker equality.

Writing Comparison: Library Classes

Writing Comparison: Library Classes

Assumptions:

1. Types provide equality or an ordering or none (Guideline 0)

Standard library assumptions:

Writing Comparison: Library Classes

Assumptions:

1. Types provide equality or an ordering or none (Guideline 0)
2. < and == define the same equality (Guideline 2)

Standard library assumptions:

Writing Comparison: Library Classes

Assumptions:

1. Types provide equality or an ordering or none (Guideline 0)
2. < and == define the same equality (Guideline 2)

Standard library assumptions:

1. Types provide == or < or both. If a type provides < it does not necessarily have ==.

Writing Comparison: Library Classes

Assumptions:

1. Types provide equality or an ordering or none (Guideline 0)
2. < and == define the same equality (Guideline 2)

Standard library assumptions:

1. Types provide == or < or both. If a type provides < it does not necessarily have ==.
2. < and == do not necessarily define the same equality.

Writing Comparison: Library Classes

```
template <typename T>
struct wrapper
{
    T value;
};
```

Goal: full member-wise ordering.

Writing Comparison: Library Classes

C++20

```cpp
template <typename T>
bool wrapper<T>::operator==(const wrapper& other) const
{
    return value == other.value;
}
```

Writing Comparison: Library Classes

C++20

```
template <typename T>
bool wrapper<T>::operator==(const wrapper&) const = default;
```

Writing Comparison: Library Classes

C++20

```
template <typename T>
auto wrapper<T>::operator<=>(const wrapper& other) const
{
    return value <=> other.value;
}
```

Writing Comparison: Library Classes

C++20

```
template <typename T>
auto wrapper<T>::operator<=>(const wrapper& other) const
{
    return synth_three_way(value, other.value);
}
```

Writing Comparison: Library Classes

C++20

```
template <typename T>
auto wrapper<T>::operator<=>(const wrapper& other) const
{
    return synth_three_way(value, other.value);
}
```

Note: implementing <=> with < and == requires assuming a comparison category.

Writing Comparison: Library Classes

C++20

```
template <typename T>
auto wrapper<T>::operator<=>(const wrapper& other) const
{
    return synth_three_way(value, other.value);
}
```

Note: implementing <=> with < and == requires assuming a comparison category.

- my recommendation: assume `std::strong_ordering`
- standard library recommendation: assume `std::weak_ordering`

Writing Comparison: Library Classes

C++20

```
template <typename T>
auto wrapper<T>::operator<=>(const wrapper&) const = default;
```

Writing Comparison: Library Classes

C++20

```
template <typename T, std::totally_ordered_with<T> U>
auto synth_three_way(const T& lhs, const U& rhs)
{
    …
}

template <typename T, typename U = T>
using synth_three_way_category
    = decltype(synth_three_way(std::declval<const T&>(),
                               std::declval<const U&>()));
```

Writing Comparison: Library Classes

C++20

```
template <typename T>
struct wrapper
{
    T value;

    synth_three_way_category<T> operator<=>(const wrapper&) const
        = default;
};
```

Writing Comparison: Library Classes

Technique

To get full member-wise ordering in generic code:

```
template <typename T>
class C
{
public:
    compute-category<T> operator<=>(const C&) const = default;
};
```

Note: keep in mind that this assumes the comparison category the type provides if it doesn't have a spaceship operator.

See also: `std::common_comparison_category`.

Writing Comparison: Library Classes

```
template <typename T>
class container
{
    T* ptr;
    std::size_t size;
};
```

Goal: full member-wise ordering of elements.

Writing Comparison: Library Classes

C++20

```cpp
template <typename T>
bool container<T>::operator==(const container& other) const
{
    return std::equal(ptr, ptr + size,
                      other.ptr, other.ptr + other.size);
}
```

Writing Comparison: Library Classes

C++20

```
template <typename T>
auto container<T>::operator<=>(const container& other) const
{
    return std::lexicographical_compare_three_way(ptr, ptr + size,
                other.ptr, other.ptr + other.size);
}
```

Writing Comparison: Library Classes

C++20

```
template <typename T>
auto container<T>::operator<=>(const container& other) const
{
    return std::lexicographical_compare_three_way(ptr, ptr + size,
                other.ptr, other.ptr + other.size,
                synth_three_way);
}
```

Writing Comparison: Library Classes

C++20

```
template <typename T, std::totally_ordered_with<T> U>
auto synth_three_way(const T& lhs, const U& rhs)
{
    if constexpr (std::three_way_comparable_with<T, U>)
        return lhs <=> rhs;
    else
    {
        if (lhs == rhs)
            return std::strong_ordering::equal;
        else if (lhs < rhs)
            return std::strong_ordering::less;
        else
            return std::strong_ordering::greater;
    }
}
```

Writing Comparison: Library Classes

C++20

```
struct synth_three_way_t
{
    template <typename T, std::totally_ordered_with<T> U>
    auto operator()(const T& lhs, const U& rhs) const
    {
        …
    }
};

inline constexpr synth_three_way_t synth_three_way;
```

Writing Comparison: Library Classes

C++20

Technique

- To get equality of a container: implement `operator==` using the `std::equal` algorithm.

Writing Comparison: Library Classes

C++20

Technique

- To get equality of a container: implement `operator==` using the `std::equal` algorithm.
- To get ordering of a container: implement `operator<=>` the `std::lexicographical_compare_three_way` algorithm with a `synth_three_way` comparison function.

Writing Comparison: Library Classes

C++20

Technique

- To get equality of a container: implement `operator==` using the `std::equal` algorithm.
- To get ordering of a container: implement `operator<=>` the `std::lexicographical_compare_three_way` algorithm with a `synth_three_way` comparison function.
- Implement the `synth_three_way` comparison function as a function object.

Interlude: Why is == separate from <=>?

<=> of two ranges:

1. Iterate and compare elements until one range ended.
2. Return less/greater depending on the range that ended.

Interlude: Why is == separate from <=>?

<=> of two ranges:

1. Iterate and compare elements until one range ended.
2. Return less/greater depending on the range that ended.

== of two ranges:

1. Compare size (if possible). If not equal, range can't be equal.
2. Iterate and compare elements.

Writing Comparison: Library Classes

```cpp
template <typename T>
class optional
{
public:
    bool has_value() const;
    const T& value() const;
};
```

Goal: full member-wise ordering, mixed comparisons with `T` and `std::nullopt`. Empty optional is less than all other values.

Writing Comparison: Library Classes

Pre C++20

- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `optional<U>`

Writing Comparison: Library Classes

Pre C++20

- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `optional<U>`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `U`

Writing Comparison: Library Classes

Pre C++20

- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `optional<U>`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `U`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `U` and `optional<T>`

Writing Comparison: Library Classes

Pre C++20

- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `optional<U>`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `U`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `U` and `optional<T>`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `std::nullopt`

Writing Comparison: Library Classes

Pre C++20

- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `optional<U>`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `U`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `U` and `optional<T>`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `std::nullopt`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `std::nullopt` and `optional<T>`

Writing Comparison: Library Classes

Pre C++20

- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `optional<U>`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `U`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `U` and `optional<T>`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `std::nullopt`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `std::nullopt` and `optional<T>`

Writing Comparison: Library Classes

Pre C++20

- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `optional<U>`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `U`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `U` and `optional<T>`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `optional<T>` and `std::nullopt`
- `==`, `!=`, `<`, `<=`, `>=`, `>` for `std::nullopt` and `optional<T>`

30 comparison operator overloads!

Writing Comparison: Library Classes

`optional<T>` and `std::nullopt`:

C++20

```
template <typename T>
bool optional<T>::operator==(std::nullopt_t) const
{
    return !has_value();
}

template <typename T>
std::strong_ordering optional<T>::operator<=>(std::nullopt_t) const
{
    if (has_value())
        return std::strong_ordering::greater;
    else
        return std::strong_ordering::equal;
}
```

Writing Comparison: Library Classes

C++20

`optional<T>` and `U`:

```
template <typename T, typename U>
bool optional<T>::operator==(const U& other) const
{
    if (has_value())
        return value() == other;
    else
        return false;
}
```

Writing Comparison: Library Classes

C++20

`optional<T>` and `U`:

```
template <typename T, typename U>
auto optional<T>::operator<=>(const U& other) const
{
    if (has_value())
        return synth_three_way(value(), other);
    else
        return std::strong_ordering::less;
}
```

Writing Comparison: Library Classes

C++20

`optional<T>` and `optional<U>`:

```
template <typename T, typename U>
bool optional<T>::operator==(const optional<U>& other) const
{
    if (has_value())
        return value() == other; // forward
    else
        return std::nullopt == other; // forward
}
```

Writing Comparison: Library Classes

C++20

`optional<T>` and `optional<U>`:

```
template <typename T, typename U>
auto optional<T>::operator<=>(const optional<U>& other) const
{
    if (has_value())
        return value() <=> other; // forward
    else
        return std::nullopt <=> other; // forward
}
```

Writing Comparison: Library Classes

C++20

Technique

- Formulating complex orderings using three-way comparison is easier than using <.
- Implement complex operators in terms of simpler ones.

Writing Comparison: Corner Cases

Writing Comparison: Corner Cases

```
struct Temperature
{
    double value;
};
```

Goal: full member-wise ordering.

Writing Comparison: Corner Cases

C++20

```
struct Temperature
{
    double value;

    std::strong_ordering operator<=>(const Temperature&) = default;
};
```

Writing Comparison: Corner Cases

I lied to you.

Writing Comparison: Corner Cases

Relationships between two floats `a`, `b`:

- `a` and `b` are equal
- `a` is less than `b`
- `a` is greater than `b`

Writing Comparison: Corner Cases

Relationships between two floats a, b:

- a and b are equal
- a is less than b
- a is greater than b
- a and b are **unordered** (NaN!)

Writing Comparison: Corner Cases

Relationships between two floats `a`, `b`:

- `a` and `b` are equal
- `a` is less than `b`
- `a` is greater than `b`
- `a` and `b` are **unordered** (NaN!)

This is called a **partial order**.

Writing Comparison: Corner Cases

Wrong, but useful:

```
enum class partial_ordering
{
    less = -1,
    equivalent = 0,
    greater = 1,

    unordered = NaN
};

// + comparison with 0
```

Writing Comparison: Corner Cases

C++20

Guideline

`<=>` should never return `std::partial_ordering`!

- it is surprising if `<`, `==` and `>` are all `false`
- no standard library algorithm works with partial ordering `<`
- `std::set`/`std::map` don't work with partial ordering `<`

Writing Comparison: Corner Cases

C++20

```
struct Temperature
{
    double value;

    std::strong_ordering operator<=>(const Temperature&) = default;
};
```

```
std::partial_ordering operator<=>(double lhs, double rhs);
```

Writing Comparison: Corner Cases

C++20

```cpp
struct Temperature
{
    double value;

    std::strong_ordering operator<=>(const Temperature& other)
    {
        // just ignore NaNs...
        if (value == other.value)
            return std::strong_ordering::equal;
        else if (value < other.value)
            return std::strong_ordering::less;
        else
            return std::strong_ordering::greater;
    }
};
```

Writing Comparison: Corner Cases

Floating Point Ordering:

- `a <=> b` gives a `std::partial_ordering`
  - normal numbers totally ordered
  - `+0`/`-0` is treated as equal
  - NaNs are unordered

Writing Comparison: Corner Cases

Floating Point Ordering:

- `a <=> b` gives a `std::partial_ordering`
  - normal numbers totally ordered
  - `+0`/`-0` is treated as equal
  - NaNs are unordered
- `std::strong_order(a, b)` gives a `std::strong_ordering`
  - normal numbers totally ordered
  - `+0`/`-0` are treated as non-equal
  - NaNs are all totally ordered, different from each other

Writing Comparison: Corner Cases

Floating Point Ordering:

- `a <=> b` gives a `std::partial_ordering`
  - normal numbers totally ordered
  - `+0`/`-0` is treated as equal
  - NaNs are unordered
- `std::strong_order(a, b)` gives a `std::strong_ordering`
  - normal numbers totally ordered
  - `+0`/`-0` are treated as non-equal
  - NaNs are all totally ordered, different from each other
- `std::weak_order(a, b)` gives a `std::weak_ordering`
  - normal numbers totally ordered
  - `+0`/`-0` are treated as equal
  - NaNs all treated equal, totally ordered with others

Writing Comparison: Corner Cases

Guideline

Comparing floating points is hard.

Writing Comparison: Corner Cases

C++20

```
struct Point
{
    int x, y;

    bool operator==(const Point&) const = default;
};

std::map<Point, T> spatial_grid;
```

Writing Comparison: Corner Cases

C++20

```
struct Point
{
    int x, y;

    std::strong_ordering operator<=>(const Point&) const = default;
};

std::map<Point, T> spatial_grid;
```

Writing Comparison: Corner Cases

Guideline

Don't implement <=> (or <) if you need to use it as key in map, only when it makes sense to provide it.

Writing Comparison: Corner Cases

C++20

```cpp
struct Point
{
    int x, y;

    bool operator==(const Point&) const = default;
};

std::strong_ordering strong_order(Point lhs, Point rhs)
{
    if (auto cmp = lhs.x <=> rhs.x; cmp != 0)
        return cmp;

    return lhs.y <=> rhs.y;
}
```

Writing Comparison: Corner Cases

C++20

```
struct strong_order_cmp
{
    using is_transparent = void;

    template <typename T, typename U>
    bool operator()(const T& t, const U& u) const
    {
        return std::strong_order(t, u) < 0;
    }
};
```

```
std::map<point, T, strong_order_cmp> spatial_grid;
```

Writing Comparison: Corner Cases

`std::strong_order`:

1. `strong_order(a, b)` if one available via ADL
2. special case for floats
3. `<=>`

(also `std::weak_order`, `std::partial_order`)

Writing Comparison: Corner Cases

`std::strong_order`:

1. `strong_order(a, b)` if one available via ADL
2. special case for floats
3. `<=>`

(also `std::weak_order`, `std::partial_order`)

`std::compare_strong_order_fallback`:

1. `strong_order(a, b)` if one available via ADL
2. special case for floats
3. `<=>`
4. `==` and `<`

(also `std::compare_weak_order_fallback`, `std::compare_partial_order_fallback`)

Writing Comparison: Corner Cases

C++20

Technique

Provide an ordering under the name `strong_order` (or `weak_order`, or `partial_order`) if:

- there is *some* ordering for the type, just not one that makes sense, or

Writing Comparison: Corner Cases

C++20

Technique

Provide an ordering under the name `strong_order` (or `weak_order`, or `partial_order`) if:

- there is *some* ordering for the type, just not one that makes sense, or
- there is an alternative ordering that is faster than `<=>`.

Writing Comparison: Corner Cases

C++20

Technique

Provide an ordering under the name `strong_order` (or `weak_order`, or `partial_order`) if:

- there is *some* ordering for the type, just not one that makes sense, or
- there is an alternative ordering that is faster than `<=>`.

Call `std::strong_order` (or `std::weak_order`, or `std::partial_order`) if you only need *some* ordering, not one that makes sense.

Conclusion

```
class C
{
public:
    bool operator==(const C&) const/* = default*/;
    std::strong_ordering operator<=>(const C&) const/* = default*/;
};
```

Resources: jonathanmueller.dev/talk/cppcon2019

Twitter: @foonathan

Patreon: patreon.com/foonathan