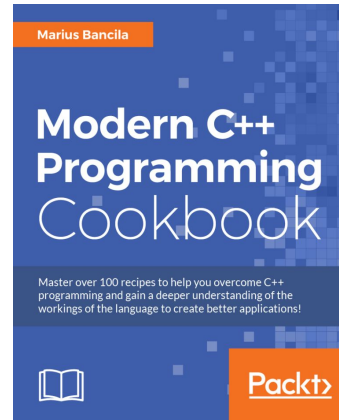# Consistent comparison in C++20

Marius Bancila

https://mariusbancila.ro

https://github.com/mariusbancila

mariusbancila

marius.bancila

The Modern C++ Challenge

Become an expert programmer by solving real-world problems

Packt>
www.packt.com

By Marius Bancila

Marius Bancila

Modern C++ Programming Cookbook

Master over 100 recipes to help you overcome C++ programming and gain a deeper understanding of the workings of the language to create better applications!

Packt>

MVP Microsoft® Most Valuable Professional
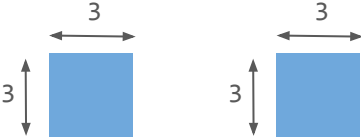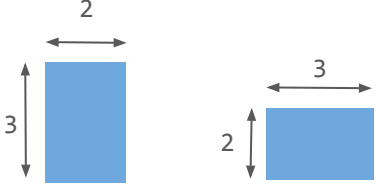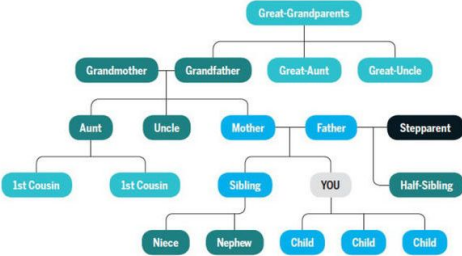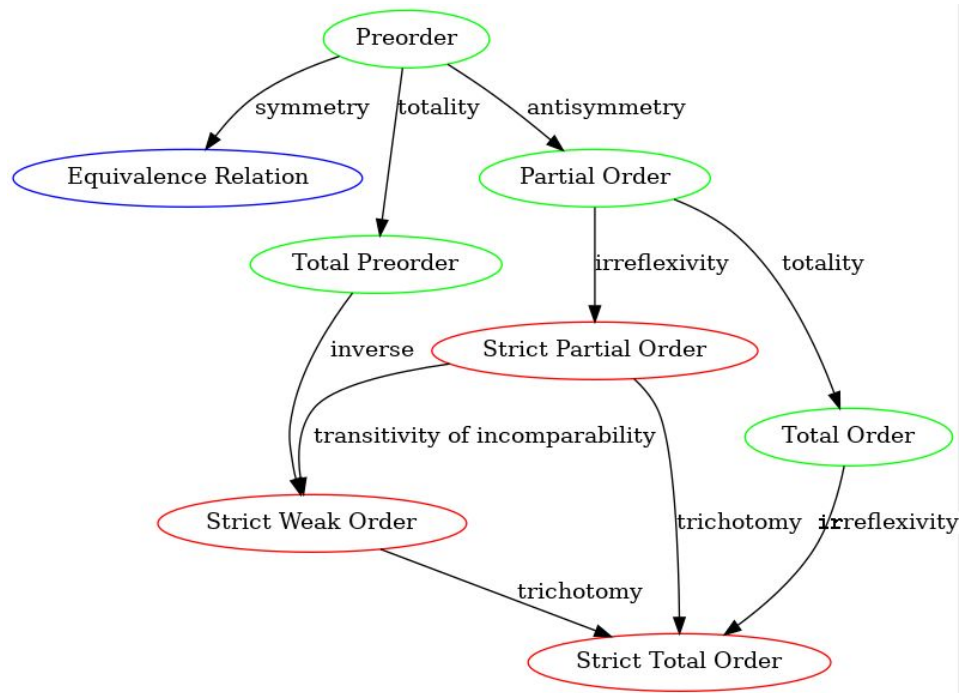
Agenda

- Ordering theory
- How to implement as of C++17
- Rules / best practices
- 3-way comparison operator in C++20
- Comparison categories
- Conversion
- Examples
- Rules / best practices
- And more...

Integers	…, -3, -2, -1, 0, 1, 2, 3, …	equality, total ordering
Floating point	…, -1.5, 0, 3.14, 19.99, 42, …	equality, partial ordering
Complex numbers	…, 1 + 2i, 3 - i, -2 + 2i, 3 + i, …	equality
IPs	192.168.0.1, 10.11.1.1, …	equality, total ordering
Squares		equality, total ordering
Rectangles		equivalence
Family tree		equality, partial ordering

Order theory

Jonathan Müller
https://foonathan.net/blog/2018/07/19/ordering-relations-math.html

- **Equivalence**: Binary relation that is
  - Reflexive
    `f(x,x) = true`
  - Symmetric
    `If f(x, y) then f(y, x)`
  - Transitive
    `If f(x, y) and f(y, z) then f(x, z)`

    Where `f(x,y)` is a comparison function

- **Equality**: Special equivalence relation where every element is equal to and only to itself

- **Preorder**: binary relation that is reflexive and transitive

- **Partial order**: binary relation that is reflexive, transitive, antisymmetric

- **Total order**: a binary relation that is reflexive, transitive, antisymmetric and total (i.e. no incomparable elements)

How to implement comparison

Comparison functions (the C style)

	less	equal	greater
memcmp	<0	0	>0
strcmp	<0	0	>0
strcoll	<0	0	>0

Operators

- Operator == and <
- Operators !=, >, <=, >= can be implemented in terms of the other two

Functors

ipv4 class

```cpp
struct ipv4
{
    explicit ipv4(unsigned char const a = 0,
                  unsigned char const b = 0,
                  unsigned char const c = 0,
                  unsigned char const d = 0) noexcept :
        data{a,b,c,d}
    {}

    unsigned long to_ulong() const noexcept
    {
        return
            (static_cast<unsigned long>(data[0]) << 24) |
            (static_cast<unsigned long>(data[1]) << 16) |
            (static_cast<unsigned long>(data[2]) << 8) |
             static_cast<unsigned long>(data[3]);
    }

private:
    std::array<unsigned char, 4> data;
};
```

```cpp
ipv4 a1 { 192, 169, 0, 1 };
ipv4 a2 = a1;
ipv4 a3 = std::move(a2);
```

assert(a1 == a3); assert(a1 > a4);

```cpp
friend bool operator==(ipv4 const & a1, ipv4 const & a2) noexcept
{ return a1.data == a2.data; }

friend bool operator!=(ipv4 const & a1, ipv4 const & a2) noexcept
{ return !(a1 == a2); }

friend bool operator<(ipv4 const & a1, ipv4 const & a2) noexcept
{ return a1.to_ulong() < a2.to_ulong(); }

friend bool operator>(ipv4 const & a1, ipv4 const & a2) noexcept
{ return a2 < a1; }

friend bool operator<=(ipv4 const & a1, ipv4 const & a2) noexcept
{ return !(a2 < a1); }

friend bool operator>=(ipv4 const & a1, ipv4 const & a2) noexcept
{ return !(a1 < a2); }
```

```
assert(a1 == 0xC0A90001);
```

```cpp
friend bool operator==(ipv4 const & a1, unsigned long const a2) noexcept
{ return a1.to_ulong() == a2; }

friend bool operator!=(ipv4 const & a1, unsigned long const a2) noexcept
{ return a1 != a2; }

friend bool operator<(ipv4 const & a1, unsigned long const a2) noexcept
{ return a1.to_ulong() < a2; }

friend bool operator>(ipv4 const & a1, unsigned long const a2) noexcept
{ return a1.to_ulong() > a2; }

friend bool operator<=(ipv4 const & a1, unsigned long const a2) noexcept
{ return a1.to_ulong() <= a2; }

friend bool operator>=(ipv4 const & a1, unsigned long const a2) noexcept
{ return !(a1 < a2); }
```

```cpp
assert(0xC0A90001 == a1);
```

```cpp
friend bool operator==(unsigned long const a1, ipv4 const & a2) noexcept
{ return a1 == a2.to_ulong(); }

friend bool operator!=(unsigned long const a1, ipv4 const & a2) noexcept
{ return a1 != a2; }

friend bool operator<(unsigned long const a1, ipv4 const & a2) noexcept
{ return a1 < a2.to_ulong(); }

friend bool operator>(unsigned long const a1, ipv4 const & a2) noexcept
{ return a1 > a2.to_ulong(); }

friend bool operator<=(unsigned long const a1, ipv4 const & a2) noexcept
{ return a1 <= a2.to_ulong(); }

friend bool operator>=(unsigned long const a1, ipv4 const & a2) noexcept
{ return !(a1 < a2); }
```

and more...

```cpp
ipv4 from_string(std::string_view text)
{
   unsigned char a = 0, b = 0, c = 0, d = 0;
   int result = sscanf(text.data(), "%hhu.%hhu.%hhu.%hhu", &a, &b, &c, &d);
   if (result != 4) throw std::runtime_error("invalid IPv4 address");
   return ipv4{ a,b,c,d };
}


ipv4 a5 = from_string("192.169.0.1");
assert(a1 == a5);
```

```
assert(a1 == "192.169.0.1");

inline bool operator==(ipv4 const & a1, char const * const a2)
{ return a1 == from_string(a2); }

inline bool operator!=(ipv4 const & a1, char const * const a2)
{ return a1 != a2; }

inline bool operator<(ipv4 const & a1, char const * const a2)
{ return a1 < from_string(a2); }

inline bool operator>(ipv4 const & a1, char const * const a2)
{ return a1 > from_string(a2); }

inline bool operator<=(ipv4 const & a1, char const * const a2)
{ return a1 <= from_string(a2); }

inline bool operator>=(ipv4 const & a1, char const * const a2)
{ return !(a1 < a2); }
```

```
assert("192.169.0.1" == a1);
```

```cpp
inline bool operator==(char const * const a1, ipv4 const & a2)
{ return from_string(a1) == a2; }

inline bool operator!=(char const * const a1, ipv4 const & a2)
{ return a1 != a2; }

inline bool operator<(char const * const a1, ipv4 const & a2)
{ return from_string(a1) < a2; }

inline bool operator>(char const * const a1, ipv4 const & a2)
{ return from_string(a1) > a2; }

inline bool operator<=(char const * const a1, ipv4 const & a2)
{ return from_string(a1) <= a2; }

inline bool operator>=(char const * const a1, ipv4 const & a2)
{ return !(a1 < a2); }
```

```cpp
struct ipv4
{
    explicit ipv4(unsigned char const a = 0,
                  unsigned char const b = 0,
                  unsigned char const c = 0,
                  unsigned char const d = 0) noexcept;
    unsigned long to_ulong() const noexcept;

    friend bool operator==(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator!=(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator<(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator>(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator<=(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator>=(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator==(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator!=(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator<(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator>(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator<=(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator>=(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator==(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator!=(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator<(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator>(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator<=(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator>=(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator==(ipv4 const & a1, char const * const a2);
    friend bool operator!=(ipv4 const & a1, char const * const a2);
    friend bool operator<(ipv4 const & a1, char const * const a2);
    friend bool operator>(ipv4 const & a1, char const * const a2);
    friend bool operator<=(ipv4 const & a1, char const * const a2);
    friend bool operator>=(ipv4 const & a1, char const * const a2);
    friend bool operator==(char const * const a1, ipv4 const & a2);
    friend bool operator!=(char const * const a1, ipv4 const & a2);
    friend bool operator<(char const * const a1, ipv4 const & a2);
    friend bool operator>(char const * const a1, ipv4 const & a2);
    friend bool operator<=(char const * const a1, ipv4 const & a2);
    friend bool operator>=(char const * const a1, ipv4 const & a2);
private:
    std::array<unsigned char, 4> data;
};
```

Overload the equality and comparison operators as non-member functions...

If you want the first operand to be of a type that is not this class, or
If you want implicit type conversion for any of the two operands

Implement an equality relation only if you know the value of the type.

Implement equality by comparing the properties that form the value.

Copies must be equal.

Implement deep equality when you have deep copy.
Implement shallow equality when you have shallow copy.

In you overload any of <, <=, >, >=
implement all of them.

In you overload the comparison operators also overload the equality operators.

The comparison operators should implement a total ordering.

For any other ordering implement a named predicate.

```cpp
struct ipv4
{
    explicit ipv4(unsigned char const a = 0,
                  unsigned char const b = 0,
                  unsigned char const c = 0,
                  unsigned char const d = 0) noexcept;
    unsigned long to_ulong() const noexcept;

    friend bool operator==(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator!=(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator<(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator>(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator<=(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator>=(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator==(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator!=(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator<(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator>(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator<=(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator>=(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator==(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator!=(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator<(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator>(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator<=(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator>=(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator==(ipv4 const & a1, char const * const a2);
    friend bool operator!=(ipv4 const & a1, char const * const a2);
    friend bool operator<(ipv4 const & a1, char const * const a2);
    friend bool operator>(ipv4 const & a1, char const * const a2);
    friend bool operator<=(ipv4 const & a1, char const * const a2);
    friend bool operator>=(ipv4 const & a1, char const * const a2);
    friend bool operator==(char const * const a1, ipv4 const & a2);
    friend bool operator!=(char const * const a1, ipv4 const & a2);
    friend bool operator<(char const * const a1, ipv4 const & a2);
    friend bool operator>(char const * const a1, ipv4 const & a2);
    friend bool operator<=(char const * const a1, ipv4 const & a2);
    friend bool operator>=(char const * const a1, ipv4 const & a2);
private:
    std::array<unsigned char, 4> data;
};
```

std::rel_ops

```cpp
namespace rel_ops
{
    template <class T>
    bool operator!=( const T& lhs, const T& rhs )
    { return !(lhs == rhs); }

    template <class T>
    bool operator>( const T& lhs, const T& rhs )
    { return rhs < lhs; }

    template <class T>
    bool operator<=( const T& lhs, const T& rhs )
    { return !(rhs < lhs); }

    template <class T>
    bool operator>=( const T& lhs, const T& rhs )
    { return !(lhs < rhs); }
}
```

```cpp
struct ipv4
{ /* ... */ };

friend bool
operator==(ipv4 const & a1, ipv4 const & a2) noexcept
{ return a1.data == a2.data; }

friend bool
operator<(ipv4 const & a1, ipv4 const & a2) noexcept
{ return a1.to_ulong() < a2.to_ulong(); }

int main()
{
    ipv4 a1 { 192, 169, 0, 1 };
    ipv4 a2 { 168, 52, 77, 101 };

    using namespace std::rel_ops;
    assert(a1 != a2);
    assert(a1 > a2);
}
```

std::rel_ops

```cpp
namespace rel_ops
{
    template <class T>
    bool operator!=( const T& lhs, const T& rhs )
    { return !(lhs == rhs); }

    template <class T>
    bool operator>( const T& lhs, const T& rhs )
    { return rhs < lhs; }

    template <class T>
    bool operator<=( const T& lhs, const T& rhs )
    { return !(rhs < lhs); }

    template <class T>
    bool operator>=( const T& lhs, const T& rhs )
    { return !(lhs < rhs); }
}
```
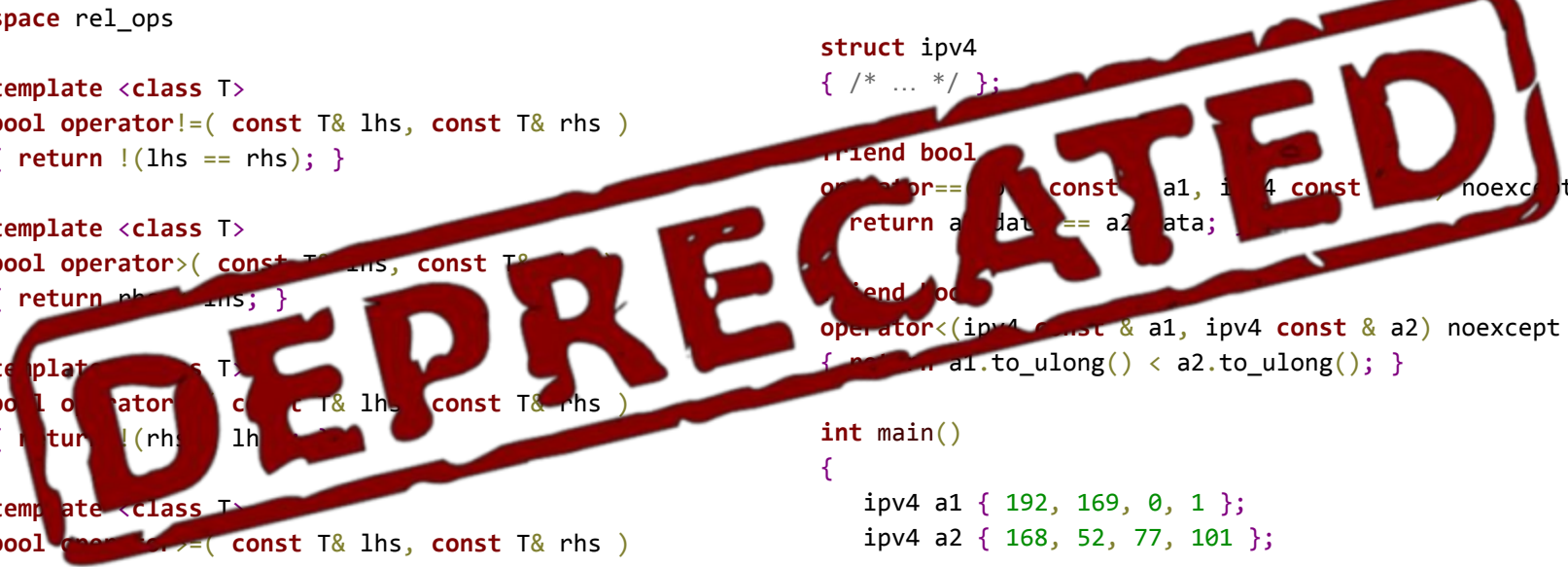
```cpp
struct ipv4
{ /* ... */ };

    friend bool
    operator==(ipv4 const & a1, ipv4 const & a2) noexcept
    { return a1.data == a2.data; }

    friend bool
    operator<(ipv4 const & a1, ipv4 const & a2) noexcept
    { return a1.to_ulong() < a2.to_ulong(); }

int main()
{
    ipv4 a1 { 192, 169, 0, 1 };
    ipv4 a2 { 168, 52, 77, 101 };

    using namespace std::rel_ops;
    assert(a1 != a2);
    assert(a1 > a2);
}
```

DEPRECATED

spaceship operator <=>

https://scifi.stackexchange.com

TIE fighter operator |=|

http://disney.wikia.com/wiki/TIE_Fighter

three-way comparison operator

Three-way comparison operator...

- Is overloadable
- Has precedence higher than < and lower than <<
- Returns a type comparable against 0
  - Other return types are allowed
- In the standard return a standard comparison category type
- Is for type implementers only
- Supported (partially) in Clang 7, VC++ 2019

ipv4 rewritten in C++20

```cpp
struct ipv4
{
    explicit ipv4(unsigned char const a = 0,
                  unsigned char const b = 0,
                  unsigned char const c = 0,
                  unsigned char const d = 0) noexcept;
    unsigned long to_ulong() const noexcept;

    auto operator<=>(ipv4 const & rhv) const noexcept = default;

    std::strong_ordering operator<=>(unsigned long const a) const noexcept
    {
        return to_ulong() <=> a;
    }


    std::strong_ordering operator<=>(char const * const a) const
    {
        return *this <=> from_string(a);
    }

private:
    std::array<unsigned char, 4> data;
};
```

ipv4 rewritten in C++20

```cpp
struct ipv4
{
    explicit ipv4(unsigned char const a = 0,
                  unsigned char const b = 0,
                  unsigned char const c = 0,
                  unsigned char const d = 0) noexcept;
    unsigned long to_ulong() const noexcept;


    auto operator<=>(ipv4 const & rhv) const noexcept = default;

    std::strong_ordering operator<=>(unsigned long const a) const noexcept
    {
        return to_ulong() <=> a;
    }


    std::strong_ordering operator<=>(char const * const a) const
    {
        return *this <=> from_string(a);
    }


private:
    std::array<unsigned char, 4> data;
};
```

```cpp
struct ipv4
{
    explicit ipv4(unsigned char const a = 0,
                  unsigned char const b = 0,
                  unsigned char const c = 0,
                  unsigned char const d = 0) noexcept;
    unsigned long to_ulong() const noexcept;

    friend bool operator==(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator!=(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator<(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator>(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator<=(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator>=(ipv4 const & a1, ipv4 const & a2) noexcept;
    friend bool operator==(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator!=(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator<(ipv4 const & a1, unsigned long const a2) noexcept;
    friend auto operator>(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator<=(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator>=(ipv4 const & a1, unsigned long const a2) noexcept;
    friend bool operator==(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator!=(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator<(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator>(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator<=(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator>=(unsigned long const a1, ipv4 const & a2) noexcept;
    friend bool operator==(ipv4 const & a1, char const * const a2);
    friend bool operator!=(ipv4 const & a1, char const * const a2);
    friend bool operator<(ipv4 const & a1, char const * const a2);
    friend bool operator>(ipv4 const & a1, char const * const a2);
    friend bool operator<=(ipv4 const & a1, char const * const a2);
    friend bool operator>=(ipv4 const & a1, char const * const a2);
    friend bool operator==(char const * const a1, ipv4 const & a2);
    friend bool operator!=(char const * const a1, ipv4 const & a2);
    friend bool operator<(char const * const a1, ipv4 const & a2);
    friend bool operator>(char const * const a1, ipv4 const & a2);
    friend bool operator<=(char const * const a1, ipv4 const & a2);
    friend bool operator>=(char const * const a1, ipv4 const & a2);
private:
    std::array<unsigned char, 4> data;
};
```

Overload the three-way comparison operator as member functions.

Even if you want the first operand to be of a type that is not this class

Overload the three-way comparison operator as non-member functions.

Only if you want implicit conversion on both arguments
(compare two objects neither of which is of this type)

```cpp
struct foo
{
    foo(int const x)
        : data{ x } {}
    foo(std::string_view x)
        : data{ std::stoi(x.data()) } {}

private:
    int data;

    friend auto operator<=>(foo const & lhv, foo const & rhv) noexcept;
};

inline auto operator<=>(foo const & lhv, foo const & rhv) noexcept
{
    return lhv.data <=> rhv.data;
}

assert(foo{ 42 } == foo{ "42" });
assert(42 == std::string_view{ "42" });
```

```cpp
struct foo
{
    foo(int const x)
        : data{ x } {}
    foo(std::string_view x)
        : data{ std::stoi(x.data()) } {}

private:
    int data;

    friend auto operator<=>(foo const & lhv, foo const & rhv) noexcept;
};

inline auto operator<=>(foo const & lhv, foo const & rhv) noexcept
{
    return lhv.data <=> rhv.data;
}

assert(foo{ 42 } == foo{ "42" });
assert(42 == std::string_view{ "42" });
```

```cpp
struct A
{
    int i;
};

struct B
{
    B(A a) : i(a.i) { }

    int i;
};

inline auto
operator<=>(B const& lhs, B const& rhs) const noexcept
{
    return lhs.i <=> rhs.i;
}

assert(A{ 2 } == A{ 2 });
assert(A{ 2 } < A{ 1 });
```

Implement only the three-way operator.
Use only the two-way operators.

Memberwise comparison

Automatically implemented with strongest comparison

```cpp
struct foo {
    auto operator<=>(int const&) const noexcept = default;
};
```

Automatically implemented with user-specified comparison

```cpp
struct foo {
    std::strong_ordering operator<=>(int const&) const noexcept = default;
};
```

Non-memberwise comparison

```cpp
class Project : public CostUnit
{
    int       id;
    int       type;
    std::string name;

public:
    std::strong_ordering operator<=>(const Project& other) const noexcept
    {
        if (auto cmp = (CostUnit&)(*this) <=> (CostUnit&)other; cmp != 0) return cmp;
        if (auto cmp = name <=> other.name; cmp != 0) return cmp;
        if (auto cmp = type <=> other.type; cmp != 0) return cmp;
        return id <=> other.id;
    }
};
```

```cpp
Project p1, p2;
if(p1 == p2) { }

if(p1 < p2) { }

std::set<Project> s;
s.insert(p1);
```

Comparison categories

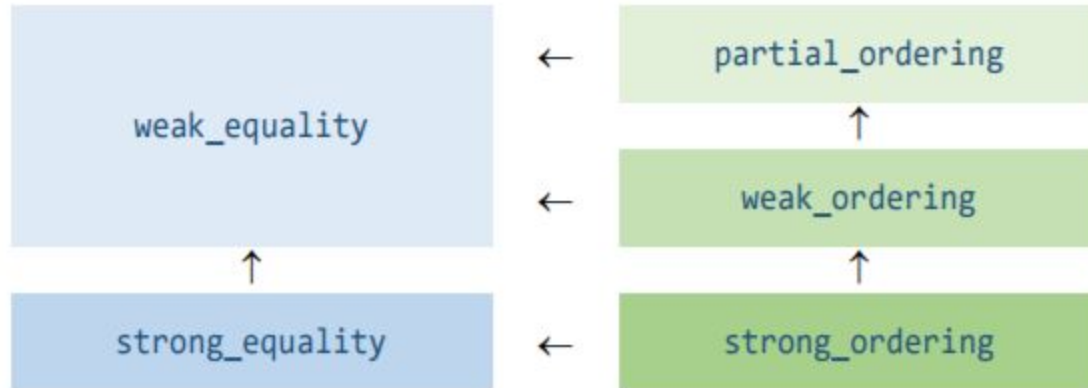| Category	Operators	Substitutability	Comparable values
`strong_ordering`	`==, !=, <, <=, >, >=`	Yes	Yes
`weak_ordering`	`==, !=, <, <=, >, >=`	No	Yes
`partial_ordering`	`==, !=, <, <=, >, >=`	No	No
`strong_equality`	`==, !=`	Yes	
`weak_equality`	`==, !=`	No	

Substitutability
- If `a==b`, then `f(a)==f(b)`, where f is a function that reads only comparison-salient state accessible via the argument's public const members

Comparable values
- Yes: at least one of `a < b`, `a == b`, or `a > b` must be true
- No: `a < b`, `a == b`, and `a > b` may be all false

Conversions

Herb Sutter, P0515R2

Valid values

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

operator<=> should return

Substitutability? a==b ⇒ f(a)==f(b)	Comparable values? a<b, a==b, or a>b	Support **a < b** ?	
		Yes	No
Yes	Yes	std::strong_ordering	std::strong_equality
No	Yes	std::weak_ordering	std::weak_equality
	No	std::partial_ordering	

Partial ordering example

```cpp
struct Employee
{
    bool is_managed_by(Employee const&) const;
    bool is_manager_of(Employee const&) const;
    bool is_same(Employee const&) const;

    std::partial_ordering operator<=>(Employee const& other) const noexcept
    {
        if(is_same(other))
            return std::partial_ordering::equivalent;
        if(is_managed_by(other))
            return std::partial_ordering::less;
        if(is_manager_of(other))
            return std::partial_ordering::greater;
        return partial_ordering::unordered;
    }
};
```

The comparison operators should implement a total ordering.

For any other ordering implement a named predicate.

Implement operator `<=>` if your type should have *full ordering* (return `std::strong_ordering`) or *equality* (return `std::strong_equality`).

Otherwise, implement comparison as a named function.

Utilities

`std::compare_3way()`

- Compares values using `operator <=>` and produces the strongest applicable comparison category
- Falls back to two-way comparison operators

`std::lexicographical_compare_3way()`

- Lexicographic comparison of two ranges using `operator <=>`

`std::common_comparison_category`

- Alias for the strongest comparison category to which all of the template arguments can be converted to

std::pair's <=> possible implementation

```cpp
template<class T1, class T2>
struct pair
{
    T1 first;
    T2 second;

    auto operator<=> (pair const& other) const
        -> std::common_comparison_category_t<
                decltype(std::compare_3way(first, other.first)),
                decltype(std::compare_3way(second, other.second)>
    {
        if (auto cmp = std::compare_3way(first, other.first); cmp != 0) return cmp;
        return std::compare3_way(second, other.second);
    }
}
```

Lexicographical comparison for ranges

```cpp
std::vector<int> v {1, 1, 2, 3, 5, 8, 13};
std::array<int, 7> a {1, 1, 2, 3, 5, 8, 13};

if(0 == std::lexicographical_compare_3way(
    std::cbegin(v), std::cend(v),
    std::cbegin(a), std::cend(a)))
{
    // are equal
}
```

Containers

```cpp
template<typename T>
strong_ordering operator<=>(vector<T> const& lhs, vector<T> const& rhs)
{
    size_t min_size = min(lhs.size(), rhs.size());
    for (size_t i = 0; i != min_size; ++i)
    {
        if (auto const cmp = compare_3way(lhs[i], rhs[i]); cmp != 0)
        {
            return cmp;
        }
    }

    return lhs.size() <=> rhs.size();
}
```

```cpp
template<typename T>
bool operator==(vector<T> const& lhs, vector<T> const& rhs)
{
    // short-circuit on size early
    const size_t size = lhs.size();
    if (size != rhs.size()) {
        return false;
    }

    for (size_t i = 0; i != size; ++i) {
        // use ==, not <=>, in all nested comparisons
        if (lhs[i] != rhs[i]) {
            return false;
        }
    }

    return true;
}
```

Source
P1185R1, Barry Revzin

Containers

- Container types (eg. vector) should implemented operator == to short-circuit comparison (by checking sizes)
- other types containing such types must explicitly implement <=>, ==, and !=
  - because operator == and != invoke <=>

```cpp
struct S
{
    vector<string> names;
    auto operator<=>(S const&) const = default;
    bool operator==(S const& rhs) const { return names == rhs.names; }
    bool operator!=(S const& rhs) const { return names != rhs.names; }
};
```

<=> != == (aka P1185R1)

```cpp
struct A {
    auto operator<=>(A const&) const = default;
};
```

<=> != == (aka P1185R1)

```cpp
struct A {
    auto operator<=>(A const&) const = default;
};
struct A {
    bool operator==(A const&) const = default;
    auto operator<=>(A const&) const = default;
};
```

<=> != == (aka P1185R1)

```
struct A {
    auto operator<=>(A const&) const = default;
};
struct A {
    bool operator==(A const&) const = default;
    auto operator<=>(A const&) const = default;
};


struct B {
    std::strong_equality operator<=>(B const&) const = default;
};
```

<=> != == (aka P1185R1)

```cpp
struct A {
    auto operator<=>(A const&) const = default;
};
struct A {
    bool operator==(A const&) const = default;
    auto operator<=>(A const&) const = default;
};


struct B {
    std::strong_equality operator<=>(B const&) const = default;
};

struct B {
    bool operator==(B const&) const = default;
};
```

Two-way comparison generation

For an expression a@b, where @ is a two-way comparison operator:

- Perform name lookup for a@b, a<=>b, and b<=>a
  - If @ is == then `a == b` and `b == a`
  - If @ is != then `a != b`, `!(a == b)`, `!(b == a)`
- Each potential candidate <=> found is included in overload resolution if:
  - <=> returns `std::*_ordering` and @ is one of `== != < > <= >=`
  - <=> returns `std::*_equality` and @ is one of `== !=`
- Select best match with normal overload resolution rules
  - prefer a@b over a<=>b over b<=>a in case of ambiguity
- Finally
  - If a<=>b is the best match, then rewrite a@b to a<=>b@0
  - If b<=>a is the best match, then rewrite b@a to 0@b<=>a

Built-in <=> comparisons

Type	Category	Comments
void		Not allowed.
bool	strong_ordering	If only one of the operands is a bool the program is ill-formed.
Integral	strong_ordering	If a narrowing conversion is required, other than from an integral type to a floating point type, the program is ill-formed.
Floating point	partial_ordering	-0 <=> +0 yields equivalent NaN <=> anything yields unordered
Object pointer	strong_ordering	Unspecified result if pointers do not point into the same object or array.
Function pointer, pointer to member, nullptr_t	strong_equality	
enumerations	same as underlying type's <=>	If multiple enumerators have the same value then strong_ordering => weak_ordering strong_equality => weak_equality
(copyable) arrays (T[N])	same as T's <=>	Only when comparing members of a class. Otherwise, ill-formed. Lexicographic elementwise comparison.

Standard library overloads

Non-member `operator<=>`

    std::pair, std::tuple

    std::string, std::string_view

    std::vector, std::array, ...

    std::shared_ptr, std::unique_ptr

    std::complex

    std::optional

    ...

Further readings

Spaceship Operator

https://blog.tartanllama.xyz/spaceship-operator/

Mathematics behind Comparison

https://foonathan.net/blog/2018/06/20/equivalence-relations.html

Implementing the spaceship operator for optional

https://medium.com/@barryrevzin/implementing-the-spaceship-operator-for-optional-4de89fc6d5ec

Consistent comparison

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0515r2.pdf

<=> != ==

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1185r1.html

Q&A