The spaceship (<=>) operator

```
/*

        Topics:
            . The spaceship operator
                .The basics
                    . Basic comparisons
                    . Spaceship comparisons
                    . weak ordering
                    . strong ordering
                    . partial ordering

                .Defaulted equality operator

                .Custom equality operator

                .Default ordering with spaceship

                .Members without spacehip operator

                .Custom spaceship operator for ordering

                .Logical operators simplified

*/
```

The spaceship (<=>) operator

```
//The basics
//Regular comparison on std::string
std::string m1{"Hello"};
std::string m2{"World"}; // World comes after Hello in alphabetical order so it's
                         // considered to be greater.

auto result = m1.compare(m2);
if(result > 0){
    fmt::println( "m1 > m2" );
}else if(result == 0){
    fmt::println( "m1 == m2" );
}else{
    fmt::println( "m1 < m2" );
}

// Three way comparison operator : spaceship operator ⟺
int n1{5};
int n2{5};
//int n3{0};

auto result_1 = ( n1 ⟺ n2);   //  The return value isn't a boolean but a three way comparison
                              //  category type. It can be less, greater or equivalent.

fmt::println( "n1 > n2 : {}" , ((n1 ⟺ n2) > 0) );
fmt::println( "n1 ⩾ n2 :{} ", ((n1 ⟺ n2) ⩾ 0) );
fmt::println( "n1 == n2 : {}" , ((n1 ⟺ n2) == 0) );
fmt::println( "n1 < n2 : {}" , ((n1 ⟺ n2) < 0) );
fmt::println( "n1 ⩽ n2 : {}" , ((n1 ⟺ n2) ⩽ 0) );
```

The spaceship (<=>) operator

```
//The basics: Ordering
// Strong ordering: When two types are equal, they are indistinguishable.
int n4{5};
int n5{5};

fmt::println( "n4 > n5 : {}" , (n4 > n5) ); // false
fmt::println( "n4 == n5 : {}" , (n4 == n5) ); //true :  Absolute equality
fmt::println( "n4 < n5 : {}" , (n4 < n5) ); // false

// Weak ordering: When two types are equivalent, they are not necessarily indistinguishable.
// Here, the two strings contain the same message, but they are not equal.
std::string m3{"Hello"};
std::string m4{"HELLO"};

fmt::println( "m3 > m4 :{} " , (m3 > m4) ); //
fmt::println( "m3 == m4 :{} " , (m3 == m4) ); // equivalence
fmt::println( "m3 < m4 :{} " , (m3 < m4) ); //

// Partial ordering: Some values of the types are not comparable to others.
double d1{ 33.9 };
double d2{ std::numeric_limits<double>::quiet_NaN() };

fmt::println("d1 > d2 : {}", (d1 > d2));// false
fmt::println("d1 == d2 : {}", (d1 == d2));// false
fmt::println("d1 < d2 : {}", (d1 < d2));// false
```

The spaceship (<=>) operator

```
//The basics: Ordering
// Strong ordering: When two types are equal, they are indistinguishable.
int n4{5};
int n5{5};

fmt::println( "n4 > n5 : {}" , (n4 > n5) ); // false
fmt::println( "n4 == n5 : {}" , (n4 == n5) ); //true :  Absolute equality
fmt::println( "n4 < n5 : {}" , (n4 < n5) ); // false

// Weak ordering: When two types are equivalent, they are not necessarily indistinguishable.
// Here, the two strings contain the same message, but they are not equal.
std::string m3{"Hello"};
std::string m4{"HELLO"};

fmt::println( "m3 > m4 :{} " , (m3 > m4) ); //
fmt::println( "m3 == m4 :{} " , (m3 == m4) ); // equivalence
fmt::println( "m3 < m4 :{} " , (m3 < m4) ); //

// Partial ordering: Some values of the types are not comparable to others.
double d1{ 33.9 };
double d2{ std::numeric_limits<double>::quiet_NaN() };

fmt::println("d1 > d2 : {}", (d1 > d2));// false
fmt::println("d1 == d2 : {}", (d1 == d2));// false
fmt::println("d1 < d2 : {}", (d1 < d2));// false
```

Explicitly defaulted equality operator (C++20)

```cpp
//Defaulted equality operator
export class Item
{
public:
    Item(int i) : Item(i, i, i) {}
    Item(int a_param, int b_param, int c_param) : a(a_param), b(b_param), c(c_param) {}

    //#2: Defaulted equality operator
    // Equality, default : member wise comparison for equality
    bool operator==(const Item &right_operand) const = default;

    private:
    int a{ 1 };
    int b{ 2 };
    int c{ 3 };
};
```

```cpp
/*
        . What you do:
            . explicitly default operator==
        . What you get:
            . a free operator!=
            . implicit conversions from literals left and right
            . simplicity: operator== set up as a member
*/
```

Explicitly defaulted equality operator (C++20)

```cpp
spaceship_ops_2::Item i1{ 1, 2, 3 };
spaceship_ops_2::Item i2{ 1, 2, 33 };

fmt::println("i1 == i2 : {}", (i1 == i2));
fmt::println("i1 ≠ i2 : {}", (i1 ≠ i2));
fmt::println("i1 == 12 : {}", (i1 == 12));
fmt::println("36 == i2 : {}", (36 == i2));// Watch out : i2==36
fmt::println("i1 ≠ 12 : {}", (i1 ≠ 12));
fmt::println("36 ≠ i2 : {}", (36 ≠ i2));
```

```
//Rewrite rules
| Expression     | Rewritten as     | Alternate Rewrite           |
| a == b         | b == a           |                             |
| a ≠ b          | !(a == b)        |  !(b == a)                  |
```

Custom equality operator (C++20)

```cpp
export class Point
{


public:
Point() = default;
Point(double x, double y) : m_x{ x }, m_y{ y } {}

Point(double x_y) : Point{ x_y, x_y } {}
// Operators
bool operator==(const Point &other) const;
double x() const { return m_x; }
double y() const { return m_y; }
double length() const;// Function to calculate distance from the point(0,0)
private:
double m_x{};
double m_y{};
};

//Implementations
double Point::length() const {
    return sqrt(pow(m_x - 0, 2) + pow(m_y - 0, 2) * 1.0);
}

bool Point::operator==(const Point &other) const {
    return (this->length() == other.length());
}
```

```cpp
/*
        . What you do:
            . set up a custom operator==
        . What you get:
            . a free operator!=
            . implicit conversions from literals left and right
            . simplicity: operator== set up as a member
*/
```

Custom equality operator (C++20)

```cpp
spaceship_ops_3::Point point1(10.0, 10.0);
spaceship_ops_3::Point point2(20.0, 20.0);

std::cout << "point1: " << point1 << "\n";
std::cout << "point2: " << point2 << "\n";

fmt::println("point1 == point2 : {}", (point1 == point2));
fmt::println("point1 ≠ point2 : {}", (point1 ≠ point2));
fmt::println("10.5 == point1 : {}", (10.5 == point1));
fmt::println("point1 == 10.5 : {}", (point1 == 10.5));
```

```
//Rewrite rules
| Expression     | Rewritten as     | Alternate Rewrite          |
| a == b         | b == a           |                            |
| a ≠ b          | !(a == b)        |  !(b == a)                 |
```

Default ordering with the spaceship operator (C++20)

```cpp
//Default ordering with the spaceship operator
export class Item
{
public:
    Item() = default;
    Item(int i) : Item(i, i, i) {}
    Item(int a_param, int b_param, int c_param) : a(a_param), b(b_param), c(c_param) {}

    //Defaulted spaceship operator
    // Default ordering : compiler generates >, < , ⩾, ⩽ and also puts in the == operator
    auto operator⟺(const Item &right_operand) const = default;

    private:
    int a{ 1 };
    int b{ 2 };
    int c{ 3 };
};
```

```
/*
        . What you do:
            . explicitly default operator⟺
        . What you get:
            . free logical comparison operators <,⩽, >, ⩾
            . a free operator==
            . a free operator≠ in terms of operator==
*/
```

Default ordering with the spaceship operator (C++20)

```cpp
spaceship_ops_4::Item i1{ 1, 2, 5 };
spaceship_ops_4::Item i2{ 1, 2, 4 };

//auto result1 = (i1 > i2);
auto result1 = ( (i1 <=> i2) > 0); // A possible option for
fmt::println(" i1 > i2 : {}" , result1 );

//auto result2 = (i1 >= i2);
auto result2 = ( (i1 <=> i2) >= 0); // A possib
fmt::println(" i1 >= i2 : {}" , result2 );

auto result3 = (i1 == i2);
fmt::println(" i1 == i2 : {}" , result3 );

auto result4 = (i1 != i2);
fmt::println(" i1 != i2 : {}" , result4 );

//auto result5 = (i1 < i2);
auto result5 = ( (i1 <=> i2) < 0); // A possibl
fmt::println(" i1 < i2 : {}" , result5 );

//auto result6 = (i1 <= i2);
auto result6 = ( (i1 <=> i2) <= 0); // A possible option for
fmt::println(" i1 <= i2 : {}" , result6 );


// Implicit conversions
auto result7 = (i1 > 20);
auto result8 = (20 < i1);// ( 20 <=> i1) < 0
auto result9 = (i2 != 12);
auto result10 = (12 != i2);
```

```
//Rewrite rules
| Expression     | Rewritten as     | Alternate Rewrite              |
|----------------|------------------|--------------------------------|
| a > b          | (a <=> b) > 0    | (b <=> a) < 0                  |
| a < b          | (a <=> b) < 0    | (b <=> a) > 0                  |
| a >= b         | (a <=> b) >= 0   | (b <=> a) <= 0                 |
| a <= b         | (a <=> b) <= 0   | (b <=> a) >= 0                 |
| a == b         | b == a           |                                |
| a != b         | !(a == b)        |  !(b == a)                     |
```

Members without the spaceship operator

```
export class BigItem
{
public:
BigItem() = default;
BigItem(int n) : BigItem(n, n, n) {}
BigItem(int a_param, int b_param, int c_param) : a(a_param), b(b_param), c(c_param) {}

// Ordering : compiler generates >, < , ⩾, ⩽ and also puts in the == operator
//auto operator⟺(const BigItem & right_operand) const = default;
std::strong_ordering operator⟺(const BigItem &right_operand) const = default;

private:
int a{ 1 };
int b{ 2 };
int c{ 3 };
Integer d;
};
```

Members without the spaceship operator

```
export struct Integer
{

Integer() = default;
Integer(int n) : m_wraped_int{ n } {}
int get() const { return m_wraped_int; }

bool operator==(const Integer &right) const { return (m_wraped_int == right.m_wraped_int); }
bool operator<(const Integer &right) const { return (m_wraped_int < right.m_wraped_int); }

private:
int m_wraped_int{};
};
```

Members without the spaceship operator (C++20)

```cpp
spaceship_ops_5::BigItem i1{ 1, 2, 5 };
spaceship_ops_5::BigItem i2{ 1, 2, 4 };

// auto result1 = (i1 > i2);
auto result1 = ((i1 <=> i2) > 0);// A possible option for the compiler magic
fmt::println(" i1 > i2: {}", result1);

// auto result2 = (i1 >= i2);
auto result2 = ((i1 <=> i2) >= 0);// A possible option for the compiler magic
fmt::println(" i1 >= i2: {}", result2);

auto result3 = (i1 == i2);
fmt::println(" i1 == i2: {}", result3);

auto result4 = (i1 != i2);
fmt::println(" i1 != i2: {}", result4);

// auto result5 = (i1 < i2);
auto result5 = ((i1 <=> i2) < 0);// A possible option for the compiler magic
fmt::println(" i1 < i2: {}", result5);

// auto result6 = (i1 <= i2);
auto result6 = ((i1 <=> i2) <= 0);// A possible option for the compiler magic
fmt::println(" i1 <= i2: {}", result6);


// Implicit conversions: The rewrite rules apply
auto result7 = (i1 > 20);
auto result8 = (20 < i1);// ( 20 <=> i1) < 0
auto result9 = (i2 != 12);
auto result10 = (12 != i2);
```

Custom spaceship operator for ordering (C++20)

```cpp
export class Vector{
    public:
    Vector(double x, double y) : m_x{ x }, m_y{ y } {}

    Vector(double x_y) : Vector{ x_y, x_y } {}
    // Operators
    bool operator==(const Vector&other) const;
    std::partial_ordering operator<=>(const Vector&right) const;
    double x() const { return m_x; }
    double y() const { return m_y; }
    double length() const;// Function to calculate distance from the
    private:
    double m_x{};
    double m_y{};
};

bool Vector::operator==(const Vector &other) const {
    return (this->length() == other.length());
}

std::partial_ordering Vector::operator<=>(const Vector &right) const
{
if (length() > right.length())
    return std::partial_ordering::greater;
else if (length() == right.length())
    return std::partial_ordering::equivalent;
else if (length() < right.length())
    return std::partial_ordering::less;
else
    return std::partial_ordering::unordered;
}
```

```cpp
/*
        . What you do:
            . set up a custom operator <=>
            . set up a custom operator==
        . What you get:
            . free logical comparison operators <,⩽, >, ⩾
            . a free operator≠ in terms of operator==
*/
```

Custom spaceship operator for ordering (C++20)

```cpp
spaceship_ops_6::Vector point1(10.0, 10.0);
spaceship_ops_6::Vector point2(20.0, 20.0);

std::cout << "point1: " << point1 << "\n";
std::cout << "point2: " << point2 << "\n";

auto result1 = (point1 > point2);
fmt::println("point1 > point2 : {}", result1);

auto result2 = (point1 ⩾ point2);
fmt::println("point1 ⩾ point2 : {}", result2);

auto result3 = (point1 == point2);
fmt::println("point1 == point2 : {}", result3);

auto result4 = (point1 ≠ point2);
fmt::println("point1 ≠ point2 : {}", result4);

auto result5 = (point1 < point2);
fmt::println("point1 < point2 : {}", result5);

auto result6 = (point1 ⩽ point2);
fmt::println("point1 ⩽ point2 : {}", result6);


// Implicit conversions
fmt::println("point1 > 20.1 : {}", (point1 > 20.1));
// fmt::println( "20.1 > point1 : {}" , (20.1 > point1) );
fmt::println("20.1 > point1 : {}", ((point1 ⟺ 20.1) < 0));
//((point1 ⟺ 20.1) < 0)
```

Simplifying logical operators (C++20)

```cpp
// Logical operators simplified.
export class Number {
    friend std::ostream &operator<<(std::ostream &out, const Number &number);

public:
    Number(int value); // We enable implicit conversions here.

    // getter
    int get_wrapped_int() const { return m_wrapped_int; }

    auto operator⟺(const Number &right) const = default;

private:
    int m_wrapped_int{0};
};

// Implementations

Number::Number(int value) : m_wrapped_int(value) {}

std::ostream &operator<<(std::ostream &out, const Number &number) {
    out << "Number : [" << number.m_wrapped_int << "]";
    return out;
}
```

Simplifying logical operators (C++20)

```
spaceship_ops_7::Number n1(10);
spaceship_ops_7::Number n2(20);


fmt::println("n1 > n2 : {}", (n1 > n2));
fmt::println("15 > n2 : {}", (15 > n2));
fmt::println("n1 > 25 : {}", (n1 > 25));

fmt::println("n1 ⩾ n2 : {}", (n1 ⩾ n2));
fmt::println("15 ⩾ n2 : {}", (15 ⩾ n2));
fmt::println("n1 ⩾ 25 : {}", (n1 ⩾ 25));

fmt::println("n1 == n2 : {}", (n1 == n2));
fmt::println("15 == n2 : {}", (15 == n2));
fmt::println("n1 == 25 : {}", (n1 == 25));

fmt::println("n1 < n2 : {}", (n1 < n2));
fmt::println("15 < n2 : {}", (15 < n2));
fmt::println("n1 < 25 : {}", (n1 < 25));

fmt::println("n1 ⩽ n2 : {}", (n1 ⩽ n2));
fmt::println("15 ⩽ n2 : {}", (15 ⩽ n2));
fmt::println("n1 ⩽ 25 : {}", (n1 ⩽ 25));
```

The three way comparison infrastructure: Summing up.

```
/*
    . If you want equality, and default member-wise lexicographical comparison
      is acceptable, default operator==. You'll get a free operator!=
    . If want equality, and defalt member-wise comparison isn't acceptable,
      then set up a custom operator==, You'll get a free operator!=
    . If you want ordering (>, <, >=, <=) and member-wise comparison is acceptable,
      then explicitly default operator<=>, you'll get all 4 logical
      operators for free, plus operator== as a bonus!
    . If you want ordering (>, <, >=, <=) and member-wise comparison isn't acceptable,
      then set up a custom operator<=>, you'll get all 4 logical
      operators for free. You won't get operator==, though.
  */
```