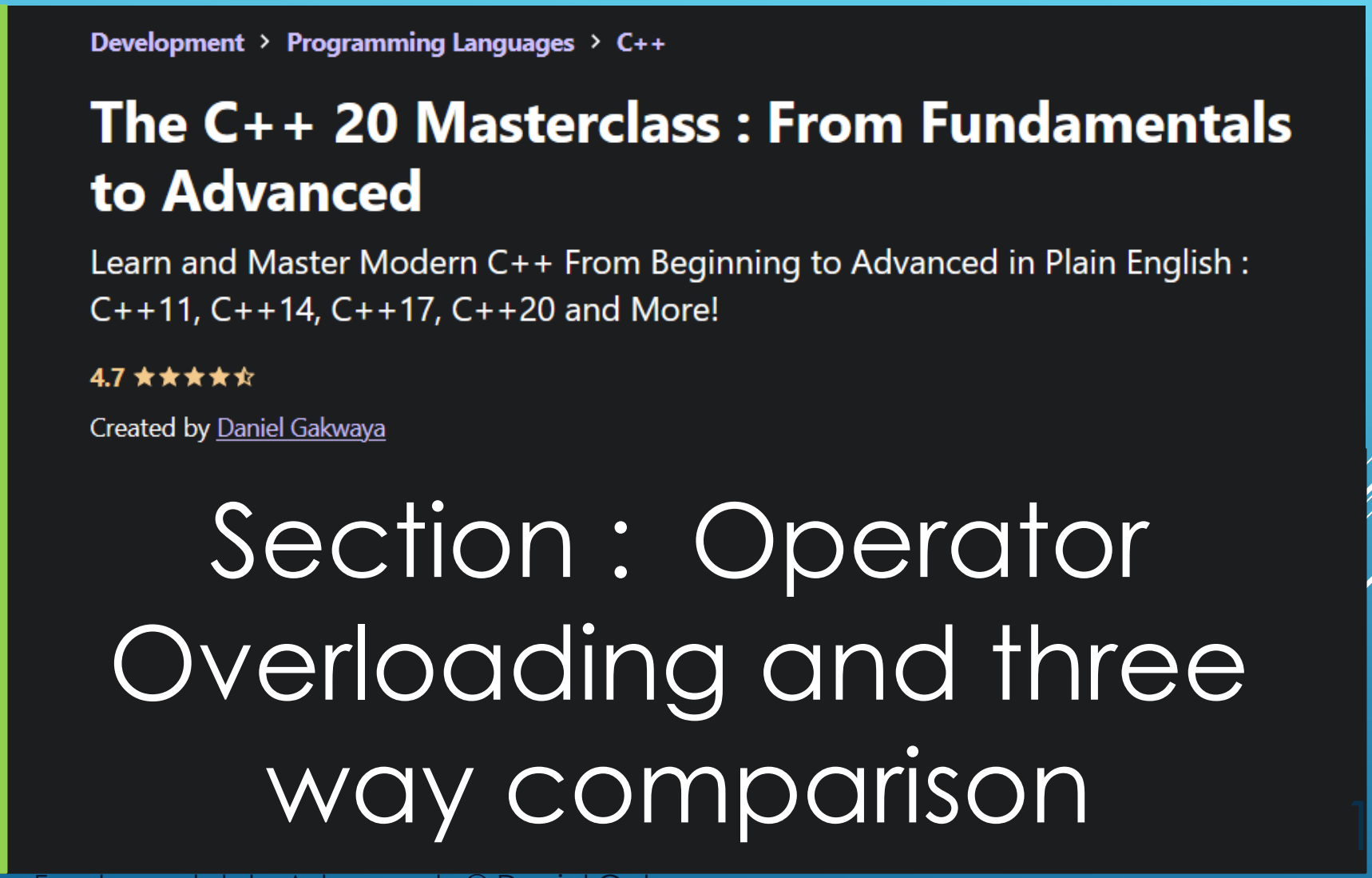
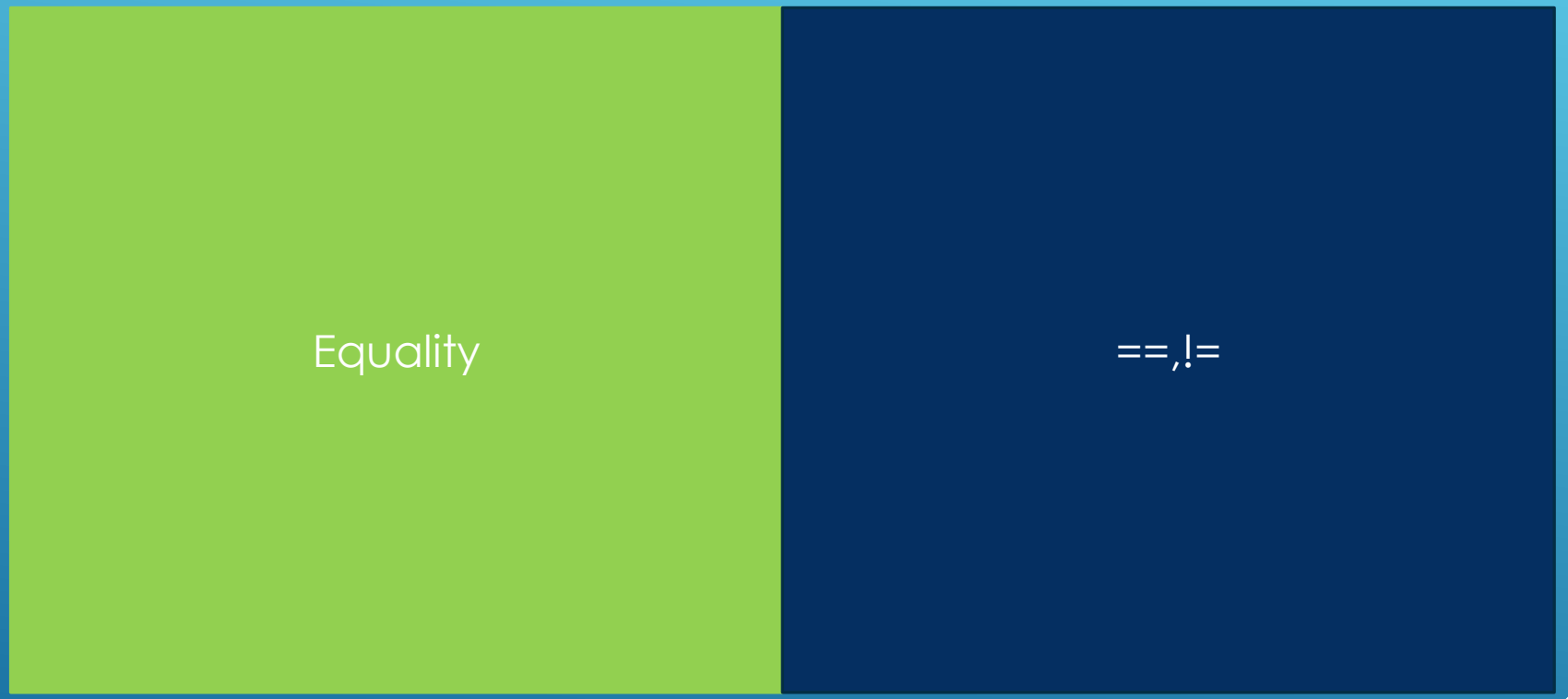
Slides



Section : Operator Overloading and three way comparison

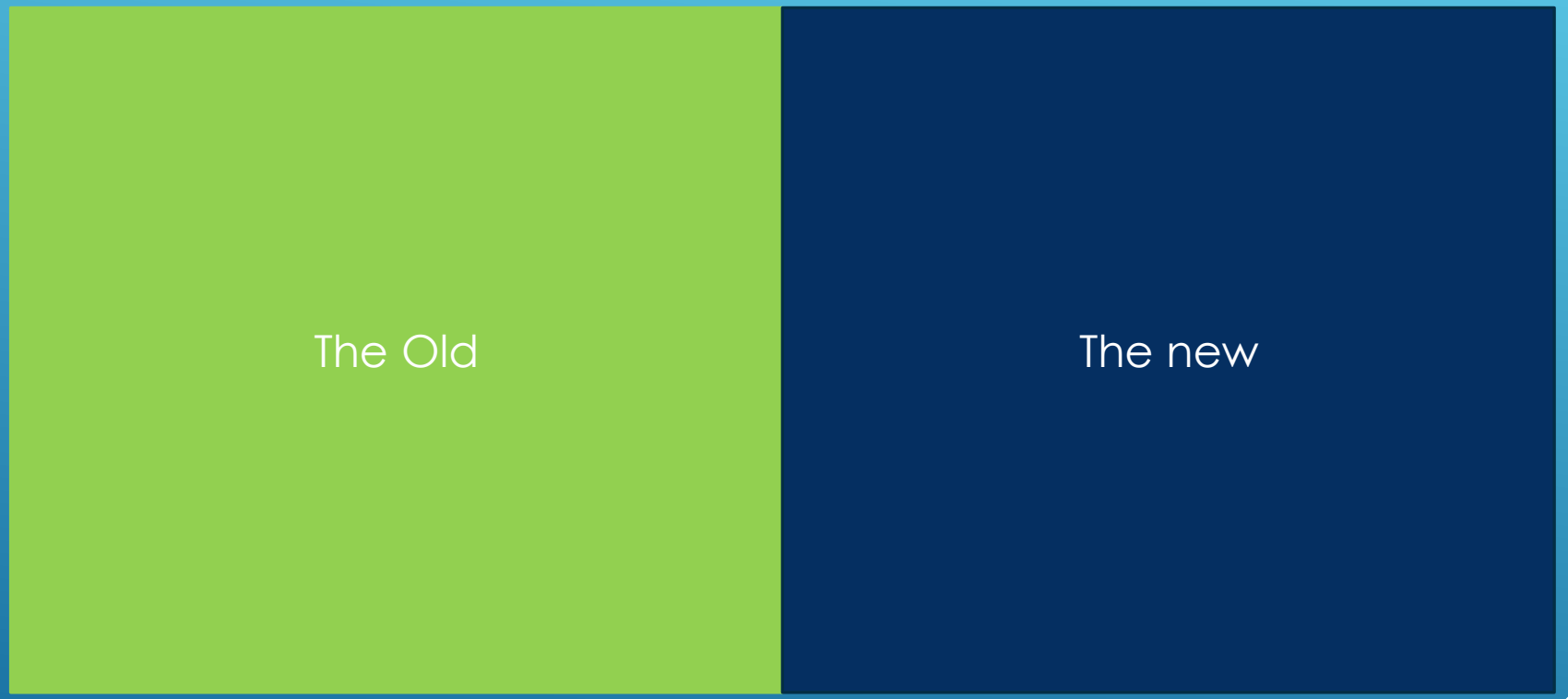
Slide intentionally left empty

Logical Operators and Three way comparison in C++20 : Introduction


Equality
==,!=


Ordering
>,>=,<,<=

Binary operators


The Old
The new

Slide intentionally left empty

Overloading all logical operators

```
class Point
{
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    ~Point() = default;
    void print_info()const{ ...

    bool operator> (const Point& other) const;
    bool operator< (const Point& other) const;
    bool operator>=(const Point& other) const;
    bool operator<=(const Point& other) const;
    bool operator==(const Point& other) const;
    bool operator!=(const Point& other) const;
private:
    double length() const;
private :
    double m_x{};
    double m_y{};
};
```

```
Point point1(10.0,10.0);
Point point2(20.0,20.0);

std::cout << "point1 : " << point1 << std::endl;

std::cout << "point2 : "  << point2 << std::endl;

std::cout << "point1 > point2 : " <<std::boolalpha <<(point1 > point2) << std::endl;
std::cout << "point1 < point2 : "  << (point1 < point2) << std::endl;
std::cout << "point1 >= point2 : " << (point1 >= point2) << std::endl;
std::cout << "point1 <= point2 : " << (point1 <= point2) << std::endl;
std::cout << "point1 == point2 : " << (point1 ==point2) << std::endl;
std::cout << "point1 != point2 : " << (point1 != point2) << std::endl;
```

Slide intentionally left empty

std::rel_ops namespace

In need of all logical operators for your type? Only overload 2 of them < and ==, and the compiler will generate the rest for you!

As of C++20, `std::rel_ops` are deprecated in favor of `operator<=>`.

*source : https://en.cppreference.com/w/cpp/utility/rel_ops/operator_cmp

```
class Point
{
    friend std::ostream& operator<< (std::ostream& out , const Point& p);
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    //Operators
    bool operator> (const Point& other) const;
    bool operator==(const Point& other) const;

private:
    double length() const;   // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```

```cpp
#include <iostream>
#include <utility>
#include "point.h"

using namespace std::rel_ops;

int main(int argc, char **argv)
{
    Point point1(10.0,10.0);
    Point point2(20.0,20.0);

    std::cout << std::boolalpha ;

    std::cout << "point1 > point2 : " <<(point1 > point2) << std::endl;
    std::cout << "point1 < point2 : " << (point1 < point2) << std::endl;
    std::cout << "point1 >= point2 : " << (point1 >= point2) << std::endl;
    std::cout << "point1 <= point2 : " << (point1 <= point2) << std::endl;
    std::cout << "point1 == point2 : " << (point1 ==point2) << std::endl;
    std::cout << "point1 != point2 : " << (point1 != point2) << std::endl;
    return 0;
}
```

Slide intentionally left empty

Logical Operators with Implicit conversions

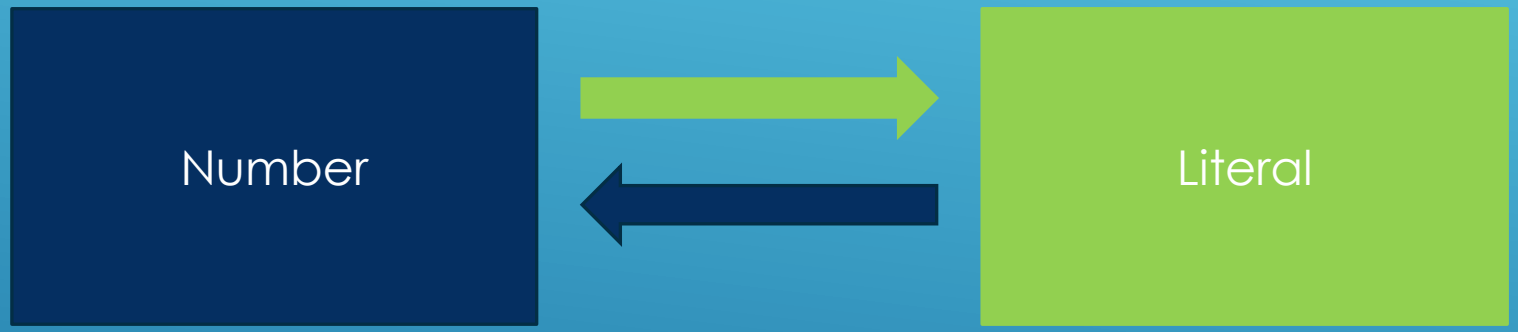
When the binary operator is set up as a member function, implicit conversions don't work for the left operand

```
class Number
{
    friend std::ostream& operator<<(std::ostream& out , const Number& number);

public:
    Number() = default;
    explicit Number(int value );
    //getter
    int get_wrapped_int() const{
            return m_wrapped_int;
    }

    bool operator<(const Number& right) const{
        return m_wrapped_int < right.m_wrapped_int;
    }

    ~Number();
private :
    int m_wrapped_int{0};
};
```

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

std::cout << std::boolalpha;
std::cout << "n1 < n2 : " << (n1 < n2) << std::endl;
std::cout << "15 < n2 : " << (15 < n2) << std::endl;
std::cout << "n1 < 25 : " << (n1 < 25) << std::endl;
```

std::rel_ops namespace ?

```
class Number
{
    friend std::ostream& operator<<(std::ostream& out , const Number& number);
    //Comparison operators
    friend bool operator<(const Number& left_operand, const Number& right_operand);
    friend bool operator<(int left_operand, const Number& right_operand);
    friend bool operator<(const Number& left_operand, int right_operand);

    friend bool operator==(const Number& left_operand, const Number& right_operand);
    friend bool operator==(int left_operand, const Number& right_operand);
    friend bool operator==(const Number& left_operand, int right_operand);

    friend bool operator>(const Number& left_operand, const Number& right_operand);
    friend bool operator>(int left_operand, const Number& right_operand);
    friend bool operator>(const Number& left_operand, int right_operand);
public:
    Number() = default;
    explicit Number(int value );
private :
    int m_wrapped_int{0};
};
```


Number
Literal

C++ 20 three way comparisons to the rescue

Slide intentionally left empty

Three way comparison operator

Know the result of a comparison : >, < or == in one go!

```cpp
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){
  std::cout << "m1 > m2" << std::endl;
}else if(result == 0){
  std::cout << "m1 == m2" << std::endl;
}else{
  std::cout << "m1 < m2" << std::endl;
}
```

std::strcmp

Defined in header <cstring>

```
int strcmp( const char *lhs, const char *rhs );
```

Compares two null-terminated byte strings lexicographically.

The sign of the result is the sign of the difference between the values of the first pair of characters (both interpreted as `unsigned char`) that differ in the strings being compared.

The behavior is undefined if `lhs` or `rhs` are not pointers to null-terminated strings.

Parameters

`lhs, rhs` - pointers to the null-terminated byte strings to compare

Return value

Negative value if `lhs` appears before `rhs` in lexicographical order.

Zero if `lhs` and `rhs` compare equal.

Positive value if `lhs` appears after `rhs` in lexicographical order.

```cpp
int n1{5};
int n2{5};

auto result = ( n1 <=> n2);

std::cout << std::boolalpha;
std::cout << "n1 > n2 : " << ((n1 <=> n2) > 0) << std::endl;
std::cout << "n1 >= n2 : "<< ((n1 <=> n2) >= 0) << std::endl;
std::cout << "n1 == n2 : " << ((n1 <=> n2) == 0) << std::endl;
std::cout << "n1 < n2 : " << ((n1 <=> n2) < 0) << std::endl;
std::cout << "n1 <= n2 : " << ((n1 <=> n2) <= 0) << std::endl;

// <=> does not return an int like std::string::compare() or strcmp . But a type whose
// value is comparable to literal 0 , not a int with a value of 0
```

Spaceship operator return values

std::strong_ordering
std::weak_ordering
std::partial_ordering

Std::strong_ordering

A type that can be used to describe absolute equality in comparisons. For example for the fundamental type “int” we can have absolute equality.

std::strong_ordering

Defined in header <compare>

```
class strong_ordering;      (since C++20)
```

The class type `std::strong_ordering` is the result type of a three-way comparison that

- admits all six relational operators (==, !=, <, <=, >, >=)
- implies substitutability: if a is equivalent to b, f(a) is also equivalent to f(b), where f denotes a function that reads only comparison-salient state that is accessible via the argument's public const members. In other words, equivalent values are indistinguishable.
- does not allow incomparable values: exactly one of a < b, a == b, or a > b must be true

Constants

The type `std::strong_ordering` has four valid values, implemented as const static data members of its type:

Member constant	Definition
less (inline constexpr) [static]	a valid value of the type `std::strong_ordering` indicating less-than (ordered before) relationship (public static member constant)
equivalent (inline constexpr) [static]	a valid value of the type `std::strong_ordering` indicating equivalence (neither ordered before nor ordered after), the same as `equal` (public static member constant)
equal (inline constexpr) [static]	a valid value of the type `std::strong_ordering` indicating equivalence (neither ordered before nor ordered after), the same as `equivalent` (public static member constant)
greater (inline constexpr) [static]	a valid value of the type `std::strong_ordering` indicating greater-than (ordered after) relationship (public static member constant)

Strong ordering

```
int n3{45};
int n4{56};
//Either of the comparisons below will always be true. Only one though
std::cout << "n3 > n4 : " << (n3 > n4) << std::endl;
std::cout << "n3 == n4 : " << (n3 == n4) << std::endl;
std::cout << "n3 < n4 : " << (n3 < n4) << std::endl;
```

Std::weak_ordering

A type that can be used to describe NON absolute equality(equivalence) in comparisons. For example two strings “Hello” and “HELLO” may be considered equivalent but not equal

std::weak_ordering

Defined in header <compare>

```
class weak_ordering;      (since C++20)
```

The class type `std::weak_ordering` is the result type of a three-way comparison that

- admits all six relational operators (==, !=, <, <=, >, >=)
- does not imply substitutability: if a is equivalent to b, f(a) may not be equivalent to f(b), where f denotes a function that reads only comparison-salient state that is accessible via the argument's public const members. In other words, equivalent values may be distinguishable.
- does not allow incomparable values: exactly one of a < b, a == b, or a > b must be true

Constants

The type `std::weak_ordering` has three valid values, implemented as const static data members of its type:

Member constant	Definition
less (inline constexpr) [static]	a valid value of the type `std::weak_ordering` indicating less-than (ordered before) relationship (public static member constant)
equivalent (inline constexpr) [static]	a valid value of the type `std::weak_ordering` indicating equivalence (neither ordered before nor ordered after) (public static member constant)
greater (inline constexpr) [static]	a valid value of the type `std::weak_ordering` indicating greater-than (ordered after) relationship (public static member constant)

Weak ordering

```
std::string msg1 {"Hello"};
std::string msg2 {"HELLO"};
//Either of the comparisons below will always be true. Only one though
std::cout << "msg1 > msg2 : " << (msg1 > msg2) << std::endl; // Greater
std::cout << "msg1 == msg2 : " << (msg1 == msg2) << std::endl; // Equivalent
std::cout << "msg1 < msg2 : " << (msg1 < msg2) << std::endl; // Less
```

Std::partial_ordering

A type that can be used to describe incomparable values for a certain type.

std::partial_ordering

Defined in header <compare>

```
class partial_ordering;      (since C++20)
```

The class type `std::partial_ordering` is the result type of a three-way comparison that

- admits all six relational operators (==, !=, <, <=, >, >=)
- does not imply substitutability: if a is equivalent to b, f(a) may not be equivalent to f(b), where f denotes a function that reads only comparison-salient state that is accessible via the argument's public const members. In other words, equivalent values may be distinguishable.
- admits incomparable values: a < b, a == b, and a > b may all be false

Constants

The type `std::partial_ordering` has four valid values, implemented as const static data members of its type:

Member constant	Definition
`less` (inline constexpr) [static]	a valid value of the type `std::partial_ordering` indicating less-than (ordered before) relationship (public static member constant)
`equivalent` (inline constexpr) [static]	a valid value of the type `std::partial_ordering` indicating equivalence (neither ordered before nor ordered after) (public static member constant)
`greater` (inline constexpr) [static]	a valid value of the type `std::partial_ordering` indicating greater-than (ordered after) relationship (public static member constant)
`unordered` (inline constexpr) [static]	a valid value of the type `std::partial_ordering` indicating relationship with an incomparable value (public static member constant)

Std::partial_ordering

```
double d1{12.9};
double d2{std::numeric_limits<double>::quiet_NaN()};

// d1 is neither > , < or == to d2 .
std::cout << std::boolalpha;
std::cout << "d1 > d2 : " << (d1 > d2) << std::endl; // false
std::cout << "d1 == d2 : " << (d1 == d2) << std::endl; // false
std::cout << "d1 < d2 : " << (d1 < d2) << std::endl; // false
```

Spaceship operator

- The < = > operator embodies the result of a comparison in C++ 20
- It returns a type whose value is comparable to literal 0, not an int whose value is 0
- It is meant to be used by the compiler to generate other operators (>,>=, <,<=), and it is rarely used in user facing code. Although it is possible to do so.
- The type of the return type describes the kind of comparison we support for our type. Options are strong_ordering, weak_ordering and partial_ordering
- The < = > operator will mostly be set up as a member function, and the compiler will put in the magic necessary to make generated comparison operators work with implicit conversions as much as possible


std::strong_ordering
std::weak_ordering
std::partial_ordering

Slide intentionally left empty

Defaulted Equality Operator

```
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){}

    //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 };
};
```

Default member wise comparison

```
bool operator== ( const Item& right_operand) const{
  if( (a == right_operand.a ) && (b == right_operand.b)
      && (c == right_operand.c)){
        return true;
  }
  return false;
}
```

Free != operator

In C++ 20 the compiler will use the == operator and synthesize a != operator for you FOR FREE

Free != operator

```
Item i1{1,2,3};
Item i2{1,2,3};

std::cout << std::boolalpha;
std::cout << "i1 == i2 : " << (i1 == i2) << std::endl;
std::cout << "i1 != i2 : " << (i1 != i2) << std::endl;
```

Implicit conversions

Implicit conversions work even for the first operand, when the operator is set up as a member

Implicit conversions

```
Item i1{1,2,3};
Item i2{1,2,3};

std::cout << std::boolalpha;
std::cout << "i1 == i2 : " << (i1 == i2) << std::endl;
std::cout << "i1 != i2 : " << (i1 != i2) << std::endl;
std::cout << "i1 == 12 : " << (i1 == 12) << std::endl;
std::cout << "36 == i2 : " << (36 == i2) << std::endl;
std::cout << "i1 != 12 : " << (i1 != 12) << std::endl;
std::cout << "36 != i2 : " << (36 != i2) << std::endl;
```

	Rewrite option1	Rewrite option2
a == b	b ==a	
a != b	!(a == b)	!(b == a)

Slide intentionally left empty

Custom Equality Operator

```
class Point
{
    friend std::ostream& operator<< (std::ostream& out , const Point& p);
public:
    Point() = default;
    Point(double x_y) : m_x{x_y},m_y{x_y}{}
    Point(double x, double y) : m_x{x}, m_y{y}{}
    //Operators
    bool operator==(const Point& other) const;
private:
    double length() const;   // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```

Custom comparison

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

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

std::cout << std::boolalpha;
std::cout << "point1 == point2 : " << (point1 ==point2) << std::endl;
std::cout << "point1 != point2 : " << (point1 != point2) << std::endl;
std::cout << "point1 == 10.0 : " << (point1 == 10.0) << std::endl; // Implicit conversion
std::cout << "10.0 == point1 : " << (10.0 == point1) << std::endl;
//In C++20, the compiler is smart enough to see that we are testing for equality
//and flip the operands, then call the member operator to give us the expected result
```

	Rewrite option1	Rewrite option2
a == b	b == a	
a != b	!(a == b)	!(b == a)

Slide intentionally left empty

Default ordering with < = >

Ordering : > , >= , < , <=

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

        //Ordering : compiler generates >, < , >=, <= .Defalut also puts in the == operator
        auto operator <=> (const Item& right_operand) const = default;

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

- The compiler will use the < = > operator to generate >, >=, <, <=
- If the operator is defaulted, the compiler will also generate a == operator for you
- The compiler then uses == to generate !=
- The operators will do member wise lexicographical comparison
- The return type of our <=> operator are auto deduced but it will be one of the three options : std::strong_ordering, std::weak_ordering, std::partial_ordering
- In this case std::strong_ordering will be deduced, because we will compare int members in the declaration order, and the int type supports strong_ordering by default

A possible > operator : member wise comparison

```
bool operator>(const Item& right) const{
        return ( (a > right.a) || (b > right.b)
                    || (c > right.c))
}
```

A possible == operator : member wise comparison

```
bool operator==(const Item& right) const{
        return ( (a == right.a) || (b == right.b)
                        || (c == right.c))
}
```

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

std::cout << std::boolalpha;

auto result1 = (i1 > i2);
//auto result1 = ( (i1 <=> i2) > 0); // A possible option for the compiler magic
std::cout << " i1 > i2 : " << (i1 > i2) << std::endl;

auto result2 = (i1 >= i2);
//auto result2 = ( (i1 <=> i2) >= 0); // A possible option for the compiler magic
std::cout << " i1 >= i2 : " << (i1 >= i2) << std::endl;

auto result5 = (i1 < i2);
//auto result5 = ( (i1 <=> i2) < 0); // A possible option for the compiler magic
std::cout << " i1 < i2 : " << (i1 < i2) << std::endl;

auto result6 = (i1 <= i2);
//auto result6 = ( (i1 <=> i2) <= 0); // A possible option for the compiler magic
std::cout << " i1 <= i2 : " << (i1 <= i2) << std::endl;
```

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

auto result3 = (i1 == i2);
std::cout << " i1 == i2 : " << (i1 == i2) << std::endl;

auto result4 = (i1 != i2);
std::cout << " i1 != i2 : " << (i1 != i2) << std::endl;
```

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

	Rewrite option1	Rewrite option2
a == b	b == a	
a != b	!(a == b)	!(b == a)
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

Slide intentionally left empty

Members without operator <=>

```
struct Integer{

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




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

```
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{};
};
```

```
class Item {
public :
    Item() = default;
    Item ( 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 Item& right_operand) const = default;
    //std::strong_ordering operator<=> (const Item& right_operand) const = default;
        /* ...
private:
    int a{ 1 };
    int b{ 2 };
    int c{ 3 };
    Integer d;
};
```

```
Item i1{1,2,5};
Item i2{1,2,5};

auto result1 = (i1 > i2);
std::cout << " i1 > i2 : " << (i1 > i2) << std::endl;

auto result2 = (i1 >= i2);
std::cout << " i1 >= i2 : " << (i1 >= i2) << std::endl;

auto result3 = (i1 == i2);
std::cout << " i1 == i2 : " << (i1 == i2) << std::endl;

auto result4 = (i1 != i2);
std::cout << " i1 != i2 : " << (i1 != i2) << std::endl;

auto result5 = (i1 < i2);
std::cout << " i1 < i2 : " << (i1 < i2) << std::endl;

auto result6 = (i1 <= i2);
std::cout << " i1 <= i2 : " << (i1 <= i2) << std::endl;
```

Slide intentionally left empty

Custom <=> operator for ordering

```
class Point
{
    friend std::ostream& operator<< (std::ostream& out , const Point& p);
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    //Operators
    bool operator==(const Point& other) const;
    std::partial_ordering operator<=>(const Point& right) const;

private:
    double length() const;   // Function to calculate distance from the point(0,0)

private :
    double m_x{};
    double m_y{};
};
```

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

std::partial_ordering Point::operator<=>(const Point& 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;
}
```

```
Point point1(10.0,10.0);
Point point2(10.0,10.0);

std::cout << std::boolalpha;
auto result1 = (point1 > point2);
std::cout << "point1 > point2 : " << result1 << std::endl;

auto result2 = (point1 >= point2);
std::cout << "point1 >= point2 : " << result2 << std::endl;

auto result3 = (point1 == point2);
std::cout << "point1 == point2 : " << result3 << std::endl;

auto result4 = (point1 != point2);
std::cout << "point1 != point2 : " << result4 << std::endl;

auto result5 = (point1 < point2);
std::cout << "point1 < point2 : " << result5 << std::endl;

auto result6 = (point1 <= point2);
std::cout << "point1 <= point2 : " << result6 << std::endl;
```

Implicit conversions

	Rewrite option1	Rewrite option2
a == b	b == a	
a != b	!(a == b)	!(b == a)
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

Slide intentionally left empty

Logical Operators Simplified

```
class Number
{
    friend std::ostream& operator<<(std::ostream& out , const Number& number);
    //Comparison operators

public:
    Number() = delete;
    explicit Number(int value );

    //getter
    int get_wrapped_int() const{
            return m_wrapped_int;
    }
    ~Number();
private :
    int m_wrapped_int{0};
};
```

```
friend bool operator<(const Number& left_operand, const Number& right_operand);
friend bool operator<(int left_operand, const Number& right_operand);
friend bool operator<(const Number& left_operand, int right_operand);

friend bool operator==(const Number& left_operand, const Number& right_operand);
friend bool operator==(int left_operand, const Number& right_operand);
friend bool operator==(const Number& left_operand, int right_operand);

friend bool operator>(const Number& left_operand, const Number& right_operand);
friend bool operator>(int left_operand, const Number& right_operand);
friend bool operator>(const Number& left_operand, int right_operand);

friend bool operator>=(const Number& left_operand, const Number& right_operand);
friend bool operator>=(int left_operand, const Number& right_operand);
friend bool operator>=(const Number& left_operand, int right_operand);

friend bool operator<=(const Number& left_operand, const Number& right_operand);
friend bool operator<=(int left_operand, const Number& right_operand);
friend bool operator<=(const Number& left_operand, int right_operand);

friend bool operator!=(const Number& left_operand, const Number& right_operand);
friend bool operator!=(int left_operand, const Number& right_operand);
friend bool operator!=(const Number& left_operand, int right_operand);
```

```
class Number
{
    friend std::ostream& operator<<(std::ostream& out , const Number& number);
    //Comparison operators
    /* ...
public:
    Number() = delete;
    explicit Number(int value );
    /* ...
    auto operator<=>(const Number& right) const = default;
    auto operator<=> (int n) const{
        return m_wrapped_int <=> n;
    }
    bool operator==(const Number& right) const{
        return m_wrapped_int == right.m_wrapped_int;
    }
    bool operator==(int n){
        return m_wrapped_int == n;
    }
    ~Number();
private :
    int m_wrapped_int{0};
};
```

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

std::cout << std::boolalpha;
std::cout << "n1 > n2 : " << (n1 > n2) << std::endl;
std::cout << "15 > n2 : " << (15 > n2) << std::endl; // How is this working?

std::cout << "15 > n2 : " << (n2 > 15) << std::endl;
std::cout << "n1 > 25 : " << (n1 > 25) << std::endl;

std::cout << "n1 >= n2 : " << (n1 >= n2) << std::endl;
std::cout << "15 >= n2 : " << (15 >= n2) << std::endl;
std::cout << "n1 >= 25 : " << (n1 >= 25) << std::endl;

std::cout << "n1 == n2 : " << (n1 == n2) << std::endl;
std::cout << "15 == n2 : " << (15 == n2) << std::endl;
std::cout << "n1 == 25 : " << (n1 == 25) << std::endl;

std::cout << "n1 < n2 : " << (n1 < n2) << std::endl;
std::cout << "15 < n2 : " << (15 < n2) << std::endl;
std::cout << "n1 < 25 : " << (n1 < 25) << std::endl;

std::cout << "n1 <= n2 : " << (n1 <= n2) << std::endl;
std::cout << "15 <= n2 : " << (15 <= n2) << std::endl;
std::cout << "n1 <= 25 : " << (n1 <= 25) << std::endl;
```

	Rewrite option1	Rewrite option2
a == b	b ==a	
a != b	!(a == b)	!(b == a)
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

Slide intentionally left empty

Operator<=> as non member

Do your <=> as a member. That's going to be a good choice in more than 90% of the cases

```
class Point{
    public :
    Point(double x, double y) : m_x{x},m_y{y}{}

    //Type conversion operator from Number to Point
    operator Number()const{
        return Number(static_cast<int>(m_x));
    }

    private :
        double m_x{};
        double m_y{};
};
```

```
Point p1{10,10};
Point p2{20,20};

bool result  = ( p1 > p2);
std::cout << std::boolalpha;
std::cout << "p1 > p2 : " << result << std::endl;
```

```
class Number
{
    friend std::ostream& operator<<(std::ostream& out , const Number& number);
public:
    Number() = delete;
    explicit Number(int value );
    //getter
    int get_wrapped_int() const{return m_wrapped_int;}

    auto operator<=>(const Number& right) const = default;
    auto operator<=> (int n) const{
        return m_wrapped_int <=> n;
    }
    bool operator==(const Number& right) const{
        return m_wrapped_int == right.m_wrapped_int;
    }
    bool operator==(int n){
        return m_wrapped_int == n;
    }
private :
    int m_wrapped_int{0};
};
```

```
class Number
{
public:
    Number() = delete;
    explicit Number(int value );
    int get_wrapped_int() const{return m_wrapped_int;}
private :
    int m_wrapped_int{0};
};

inline auto operator<=>(const Number& left,const Number & right){
    return (left.get_wrapped_int() <=> right.get_wrapped_int());
}
inline auto operator<=>(int left,const Number & right){
    return (left <=> right.get_wrapped_int());
}
inline bool operator==(const Number& left,const Number & right){
    return (left.get_wrapped_int() == right.get_wrapped_int());
}
inline bool operator==(int left,const Number & right){
    return (left == right.get_wrapped_int());
}
```

	Rewrite option1	Rewrite option2
a == b	b == a	
a != b	!(a == b)	!(b == a)
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

Slide intentionally left empty

Zooming in on Weak Ordering : Example 1

Comparing strings by size only

```
ComparableString cmp_str1("Dog");
ComparableString cmp_str2("Fog");

std::cout << std::boolalpha;
std::cout << "cmp_str1 > cmp_str2 : " << (cmp_str1 > cmp_str2) << std::endl;
std::cout << "cmp_str1 < cmp_str2 : " << (cmp_str1 < cmp_str2) << std::endl;

std::cout << "cmp_str1 == cmp_str2 : " << (cmp_str1 == cmp_str2) << std::endl;
```

```
class ComparableString {
public:
    ComparableString(const std::string& str): m_str{ str }{}

    std::weak_ordering operator <=> (const ComparableString& right_side) const {

        if (m_str.size() == right_side.m_str.size()) {
            return std::weak_ordering::equivalent;
        }
        else if (m_str.size() > right_side.m_str.size()) {
            return std::weak_ordering::greater;
        }
        else {
            return std::weak_ordering::less;
        }
    }

    bool operator== (const ComparableString& right_operand) const {
        return (m_str.size() == right_operand.m_str.size());
    }
private:
    std::string m_str;
};
```

	Rewrite option1	Rewrite option2
a == b	b == a	
a != b	!(a == b)	!(b == a)
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

Slide intentionally left empty

Zooming in on Weak Ordering : Example 2

Case insensitive strings

```
CaseInsensitiveString ci_str1("Hello");
CaseInsensitiveString ci_str2("HELLO");

std::cout << std::boolalpha;
std::cout << "ci_str1 <= ci_str2 : " << (ci_str1 <= ci_str2) << std::endl;

//You need to put in a == operator. Compiler won't generate one for you.
std::cout << "ci_str1 == ci_str2 : " << (ci_str1 == ci_str2) << std::endl;
```

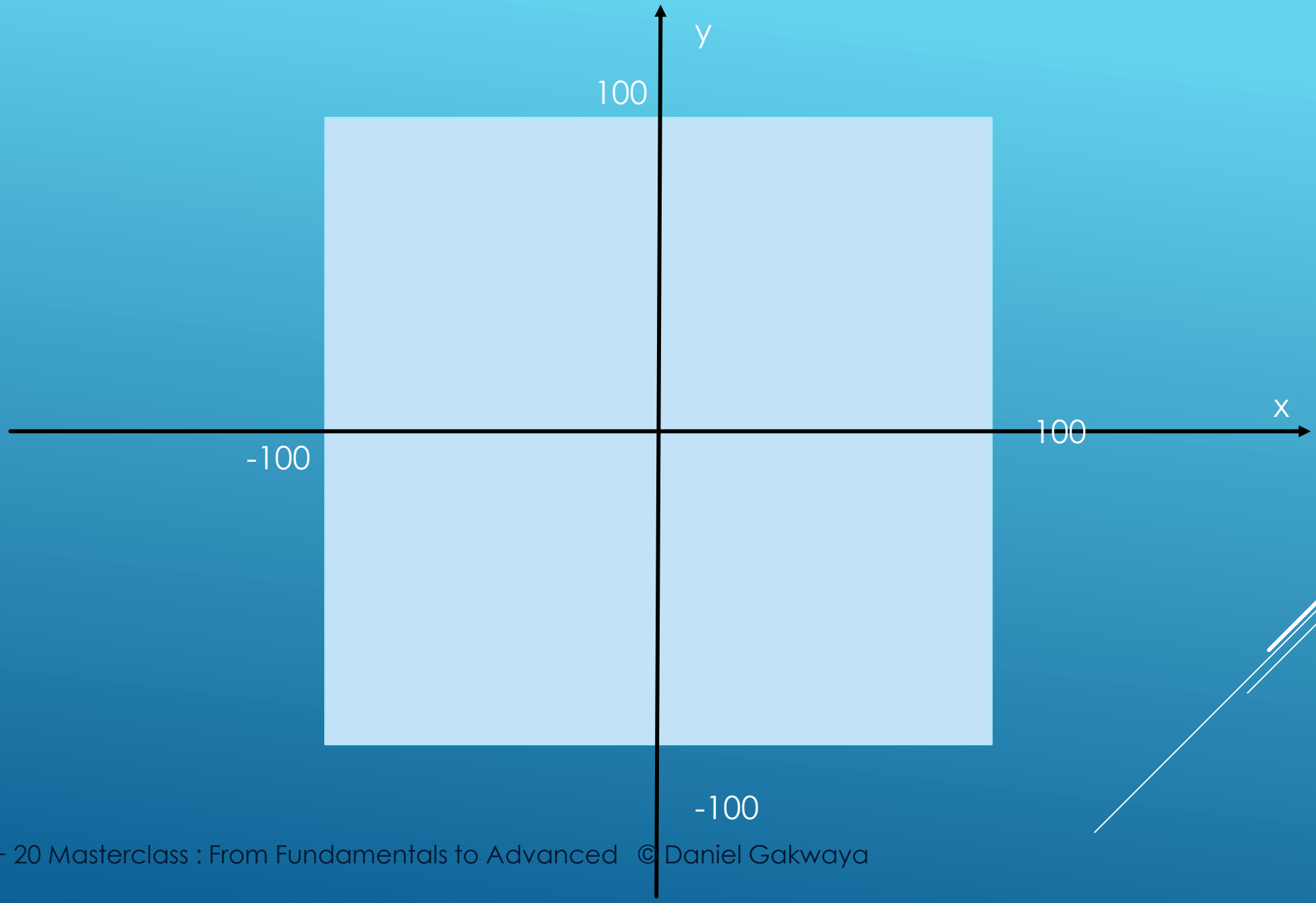
```
std::weak_ordering case_insensitive_compare(const char* str1, const char* str2) {

    //Turn them all to uppercase
    std::string string1(str1);
    std::string string2(str2);

    for (auto& c : string1) {
        c = toupper(c);
    }

    for (auto& c : string2) {
        c = toupper(c);
    }

    int cmp = string1.compare(string2);
    if (cmp > 0)
        return std::weak_ordering::greater;
    else if (cmp == 0)
        return std::weak_ordering::equivalent;
    else
        return std::weak_ordering::less;
}
```

```
class CaseInsensitiveString {
public:
    CaseInsensitiveString(const std::string& str): s(str){}

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

    bool operator==(const CaseInsensitiveString& right_operand)const{
        return (case_insensitive_compare(s.c_str(),right_operand.s.c_str())
            == std::weak_ordering::equivalent)? true : false;
    }
private:
    std::string s;
};
```

	Rewrite option1	Rewrite option2
a == b	b == a	
a != b	!(a == b)	!(b == a)
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

Slide intentionally left empty

Zooming in on partial ordering


y
100
x
100
-100
-100

```
//Valid comparable points are within the [100,100] bounds
class Point{
    public :
        Point (int x, int y)
            : m_x{x}, m_y{y}
            {}
        /* ...
    private :
        bool is_within_bounds(const Point& p) const{
            if((std::abs(p.m_x) < 100) && (std::abs(p.m_y) < 100))
                return true;
            return false;
        }
        double length() const{
            return sqrt(pow(m_x - 0, 2) +  pow(m_y - 0, 2) * 1.0);
        }
        int m_x{};
        int m_y{};
};
```

Operators

```
bool operator==(const Point& right) const {
    return length() == right.length();
}

std::partial_ordering operator<=>(const Point& right) const {

    if(is_within_bounds(*this) && is_within_bounds(right)){
        if(length() > right.length())
            return std::partial_ordering::greater;
        else if(length() < right.length())
            return std::partial_ordering::less;
        else
            return std::partial_ordering::equivalent;
    }
    return std::partial_ordering::unordered;

}
```

Operators

```
Point p1(101,20);
Point p2(20,30);

std::cout << std::boolalpha;
auto result1 = (p1 > p2);
std::cout << "p1 > p2 : " << result1 << std::endl;

auto result2 = (p1 >= p2);
std::cout << "p1 >= p2 : " << result2 << std::endl;

auto result3 = (p1 == p2);
std::cout << "p1 == p2 : " << result3 << std::endl;

auto result4 = (p1 != p2);
std::cout << "p1 != p2 : " << result4 << std::endl;

auto result5 = (p1 < p2);
std::cout << "p1 < p2 : " << result5 << std::endl;

auto result6 = (p1 <= p2);
std::cout << "p1 <= p2 : " << result6 << std::endl;
```

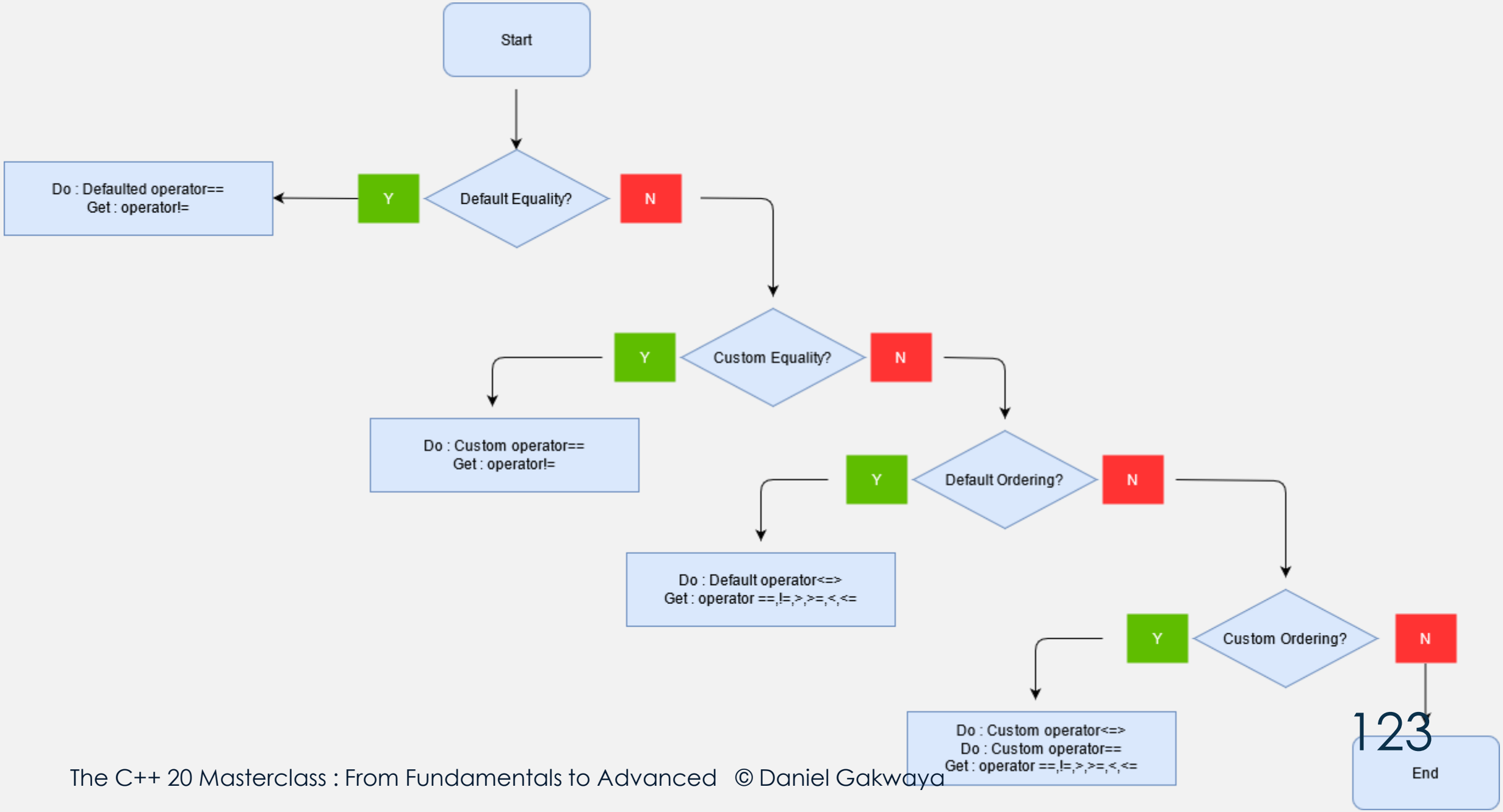
	Rewrite option1	Rewrite option2
a == b	b == a	
a != b	!(a == b)	!(b == a)
a > b	(a <=> b) > 0	(b <=> a) < 0
a >= b	(a <=> b) >= 0	(b <=> a) <= 0
a < b	(a <=> b) < 0	(b <=> a) > 0
a <= b	(a <=> b) <= 0	(b <=> a) >= 0

Slide intentionally left empty

Comparisons in C++20 : Summing up

- If you want equality, and the default member wise lexicographical comparison is ok for your design, then default the == operator, the compiler will generate the != for you
- If you want custom == , then set up your own non default operator== , and the compiler will use that to synthesize the != for you
- If you want ordering (>,>=,<,<=), and default member wise, lexicographical comparison is ok for your design, then default the <=> operator. The compiler will give you a free operator == . <=> will be used to generate the >,<=,<,<= operators and the == operator used to generate the != operator
- If you want ordering, and default member wise lexicographical comparison doesn't work for you, then you need to overload your own non default <=> operator. If you do this, the compiler won't generate a == operator for you though, you'll need to put that in yourself. The compiler will use these to generate all the 6 comparison operators

	Rewrite option1	Rewrite option2
a == b	b == a	
a != b	!(a == b)	!(b == a)
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


Start
Default Equality?
Y
Do : Defaulted operator==
Get : operator!=
N
Custom Equality?
Y
Do : Custom operator==
Get : operator!=
N
Default Ordering?
Y
Do : Default operator<=>
Get : operator ==,!=,>,>=,<,<=
N
Custom Ordering?
Y
Do : Custom operator<=>
Do : Custom operator==
Get : operator ==,!=,>,>=,<,<=
N
End

Slide intentionally left empty

Logical Operators and three way comparison in C++20 : Summary

All Logical Operators

```
class Point
{
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    ~Point() = default;
    void print_info()const{ ...

    bool operator> (const Point& other) const;
    bool operator< (const Point& other) const;
    bool operator>=(const Point& other) const;
    bool operator<=(const Point& other) const;
    bool operator==(const Point& other) const;
    bool operator!=(const Point& other) const;
private:
    double length() const;
private :
    double m_x{};
    double m_y{};
};
```

Rel_ops namespace

```
class Point
{
    friend std::ostream& operator<< (std::ostream& out , const Point& p);
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    //Operators
    bool operator> (const Point& other) const;
    bool operator==(const Point& other) const;

private:
    double length() const;   // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```

Rel ops namespace

```
#include <iostream>
#include <utility>
#include "point.h"

using namespace std::rel_ops;

int main(int argc, char **argv)
{
    Point point1(10.0,10.0);
    Point point2(20.0,20.0);

    std::cout << std::boolalpha ;

    std::cout << "point1 > point2 : " <<(point1 > point2) << std::endl;
    std::cout << "point1 < point2 : " << (point1 < point2) << std::endl;
    std::cout << "point1 >= point2 : " << (point1 >= point2) << std::endl;
    std::cout << "point1 <= point2 : " << (point1 <= point2) << std::endl;
    std::cout << "point1 == point2 : " << (point1 ==point2) << std::endl;
    std::cout << "point1 != point2 : " << (point1 != point2) << std::endl;
    return 0;
}
```

Comparison operators gone MAD!

```
friend bool operator<(const Number& left_operand, const Number& right_operand);
friend bool operator<(int left_operand, const Number& right_operand);
friend bool operator<(const Number& left_operand, int right_operand);

friend bool operator==(const Number& left_operand, const Number& right_operand);
friend bool operator==(int left_operand, const Number& right_operand);
friend bool operator==(const Number& left_operand, int right_operand);

friend bool operator>(const Number& left_operand, const Number& right_operand);
friend bool operator>(int left_operand, const Number& right_operand);
friend bool operator>(const Number& left_operand, int right_operand);

friend bool operator>=(const Number& left_operand, const Number& right_operand);
friend bool operator>=(int left_operand, const Number& right_operand);
friend bool operator>=(const Number& left_operand, int right_operand);

friend bool operator<=(const Number& left_operand, const Number& right_operand);
friend bool operator<=(int left_operand, const Number& right_operand);
friend bool operator<=(const Number& left_operand, int right_operand);

friend bool operator!=(const Number& left_operand, const Number& right_operand);
friend bool operator!=(int left_operand, const Number& right_operand);
friend bool operator!=(const Number& left_operand, int right_operand);
```

Defaulted Equality Operator

```
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){}

    //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 };
};
```

Custom Equality Operator

```
class Point
{
    friend std::ostream& operator<< (std::ostream& out , const Point& p);
public:
    Point() = default;
    Point(double x_y) : m_x{x_y},m_y{x_y}{}
    Point(double x, double y) : m_x{x}, m_y{y}{}
    //Operators
    bool operator==(const Point& other) const;
private:
    double length() const;   // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```

Defaulted three way comparison operator

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

        //Ordering : compiler generates >, < , >=, <= .Defalut also puts in the == operator
        auto operator <=> (const Item& right_operand) const = default;

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

Members without operator<=>

```
class Item {
public :
    Item() = default;
    Item ( 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 Item& right_operand) const = default;
    //std::strong_ordering operator<=> (const Item& right_operand) const = default;
        /* ...
private:
    int a{ 1 };
    int b{ 2 };
    int c{ 3 };
    Integer d;
};
```

Custom three way comparison operator

```
class Point
{
    friend std::ostream& operator<< (std::ostream& out , const Point& p);
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    //Operators
    bool operator==(const Point& other) const;
    std::partial_ordering operator<=>(const Point& right) const;

private:
    double length() const;   // Function to calculate distance from the point(0,0)

private :
    double m_x{};
    double m_y{};
};
```

C++ Simplifies comparison operators

```
class Number
{
    friend std::ostream& operator<<(std::ostream& out , const Number& number);
    //Comparison operators
    /* ...
public:
    Number() = delete;
    explicit Number(int value );
    /* ...
    auto operator<=>(const Number& right) const = default;
    auto operator<=> (int n) const{
        return m_wrapped_int <=> n;
    }
    bool operator==(const Number& right) const{
        return m_wrapped_int == right.m_wrapped_int;
    }
    bool operator==(int n){
        return m_wrapped_int == n;
    }
    ~Number();
private :
    int m_wrapped_int{0};
};
```

C++20 comparison operators as non members

```
class Number
{
public:
    Number() = delete;
    explicit Number(int value );
    int get_wrapped_int() const{return m_wrapped_int;}
private :
    int m_wrapped_int{0};
};

inline auto operator<=>(const Number& left,const Number & right){
    return (left.get_wrapped_int() <=> right.get_wrapped_int());
}
inline auto operator<=>(int left,const Number & right){
    return (left <=> right.get_wrapped_int());
}
inline bool operator==(const Number& left,const Number & right){
    return (left.get_wrapped_int() == right.get_wrapped_int());
}
inline bool operator==(int left,const Number & right){
    return (left == right.get_wrapped_int());
}
```

Examples

Slide intentionally left empty