SpaceShip Operator

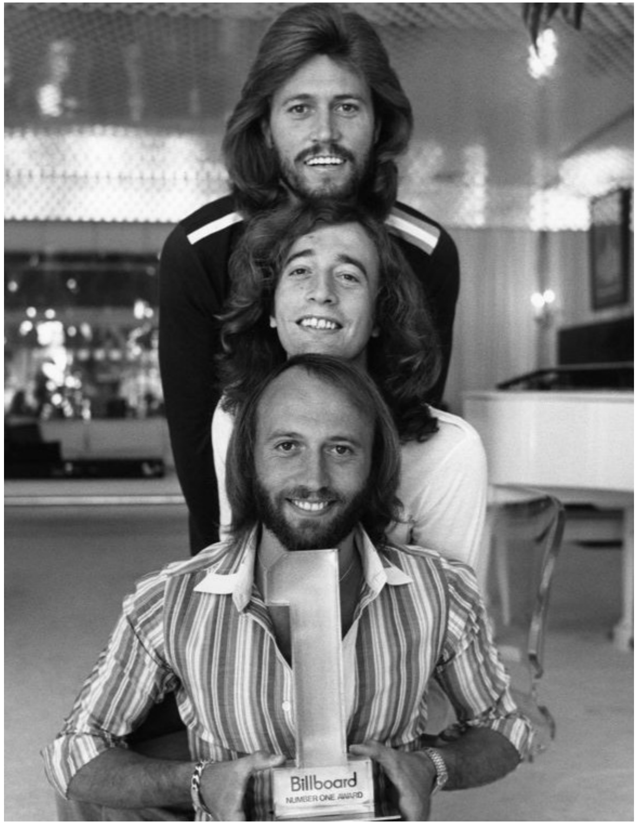
Lieven de Cock
www.codeblocks.org

Interactive Session

- Who are these ?

Interactive Session

- Who are these ?



Interactive Session

- Who are these ?

Interactive Session

- Who are these ?

Interactive Session

- Who are these ?

Siblings in Music

- Struct : SirName, FirstName
- Vector of these
- Sort

Siblings in Music : Solution

```
std::vector<SiblingsInMusic> siblings
{
   {"Jackson", "Tito"},
   {"Jackson", "Jackie"},
   {"Jackson", "Michael"},
   {"Jackson", "Jermaine"},
   {"Jackson", "Marlon"},
   {"Gallagher", "Noel"},
   {"Gallagher", "Liam"},
   {"Jackson", "Janet"},
   {"Gibb", "Robin"},
   {"Gibb", "Barry"},
   {"Gibb", "Maurice"},
   {"Minogue", "Kylie"},
   {"Minogue", "Dannii"}
};

std::ranges::sort(siblings);


std::ranges::for_each(siblings, [](const auto& sibling)
      {
         std::cout << sibling.sirName << " " << sibling.firstName << std::endl;
      });
```

```
struct SiblingsInMusic
{
   std::string sirName;
   std::string firstName;
   auto operator<=>(const SiblingsInMusic&) const = default;
};
```

```
Gallagher Liam
Gallagher Noel
Gibb Barry
Gibb Maurice
Gibb Robin
Jackson Jackie
Jackson Janet
Jackson Jermaine
Jackson Marlon
Jackson Michael
Jackson Tito
Minogue Dannii
Minogue Kylie
```

The SpaceShip has landed

- That’s it folks !

Operators

- Equality
- Relational

The Past

- Our test class : MagicInt
- **Implicit** constructor
- Compare
- Sortable

- Problem:
  mi == 242 : OK
  242 == mi : WOOPS

```
class MagicInt
{
public:
    MagicInt(int val) : mValue{val} {}
    bool operator== (const MagicInt& rhs) const
    {
        return mValue == rhs.mValue;
    }
    bool operator< (const MagicInt& rhs) const
    {
        return mValue < rhs.mValue;
    }
private:
    int mValue;
};
```

The Past : fix the compare problem

- Honor mathematics
- Scott Meyers : act as an int
- So “==” ==> 2 methods needed
- Free method which swaps operands

```
class MagicInt
{
public:
    MagicInt(int val) : mValue{val} {}
    bool operator== (const MagicInt& rhs) const
    {
        return mValue == rhs.mValue;
    }
private:
    int mValue;
};
bool operator== (int lhs, const MagicInt& rhs)
{
    return rhs == lhs;
}
```

The Past : fix the compare problem → hidden friends

- 1 (free) method
- Both arguments can be implicitly converted

```cpp
class MagicInt
{
public:
    MagicInt(int val) : mValue{val} {}

    friend bool operator== (const MagicInt& lhs, const MagicInt& rhs)
    {
        return lhs.mValue == rhs.mValue;
    }

private:
    int mValue;
};
```

The Past : all 6

- ==
- <
- <= (== or <)
- != (not ==)
- > (not <=)
- >= (not <)

The Past : all 6

```
friend bool operator== (const MagicInt& lhs, const MagicInt& rhs)
{
   return lhs.mValue == rhs.mValue;
}

friend bool operator!= (const MagicInt& lhs, const MagicInt& rhs)
{
   return !(lhs.mValue == rhs.mValue);
}

friend bool operator< (const MagicInt& lhs, const MagicInt& rhs)
{
   return (lhs.mValue < rhs.mValue);
}

friend bool operator<= (const MagicInt& lhs, const MagicInt& rhs)
{
   return (lhs.mValue < rhs.mValue) || (lhs.mValue == rhs.mValue);
}

friend bool operator>= (const MagicInt& lhs, const MagicInt& rhs)
{
   return !(lhs.mValue < rhs.mValue);
}
friend bool operator> (const MagicInt& lhs, const MagicInt& rhs)
{
   return !(lhs.mValue <= rhs.mValue);
}
```

The Past : be complete

- noexcept
- constexpr
- [[nodiscard]]

```
[[nodiscard]] friend constexpr bool operator== (const MagicInt& lhs, const MagicInt& rhs) noexcept
{
    return lhs.mValue == rhs.mValue;
}

[[nodiscard]] friend constexpr bool operator!= (const MagicInt& lhs, const MagicInt& rhs) noexcept
{
    return !(lhs.mValue == rhs.mValue);
}

[[nodiscard]] friend constexpr bool operator< (const MagicInt& lhs, const MagicInt& rhs) noexcept
{
    return (lhs.mValue < rhs.mValue);
}

[[nodiscard]] friend constexpr bool operator<= (const MagicInt& lhs, const MagicInt& rhs) noexcept
{
    return (lhs.mValue < rhs.mValue) || (lhs.mValue == rhs.mValue);
}

[[nodiscard]] friend constexpr bool operator>= (const MagicInt& lhs, const MagicInt& rhs) noexcept
{
    return !(lhs.mValue < rhs.mValue);
}

[[nodiscard]] friend constexpr bool operator> (const MagicInt& lhs, const MagicInt& rhs) noexcept
{
    return !(lhs.mValue <= rhs.mValue);
}
```

C++20 : Rewrites

- Equality operator "=="
- A a; B b;
- a != b : compiler will try
  - a != b
  - !(a == b)
  - !(b == a)
- Note : it will NOT try (b != a)
- a == b : compiler will try
  - a == b
  - b == a

C++20 : Rewrites

```cpp
class MagicInt
{
public:
    MagicInt(int val) : mValue{val} {}

    bool operator== (const MagicInt& rhs) const
    {
        return mValue == rhs.mValue;
    }

private:
    int mValue;
};
```

```cpp
const MagicInt f1{242};
const MagicInt f2{100};

std::cout << (f1 == f2) << std::endl;     /// OK
std::cout << (f1 != f2) << std::endl;     /// rewritten as !(a==b)

std::cout << (f1 == 242) << std::endl;  /// OK
std::cout << (242 == f1) << std::endl;  /// rewritten as (b==a)

std::cout << (f1 != 242) << std::endl;   /// rewritten as !(a==b)
std::cout << (242 != f1) << std::endl;   /// rewritten as !(b==a)
```

```
0
1
1
1
0
0
```

- Note : no hidden friend here !!!

C++20 : Rewrites : Endless Recursion

```
class MagicInt
{
public:
    MagicInt(int val) : mValue{val} {}


    bool operator== (const MagicInt& rhs) const
    {
        return mValue == rhs.mValue;
    }
private:
    int mValue;
};

bool operator== (int lhs, const MagicInt& rhs)
{
    return rhs == lhs;
}
```

```
const MagicInt f1{242};

std::cout << (f1 == 242) << std::endl;

std::cout << (242 == f1) << std::endl;

// gcc 11.3/12 warns, on the above function

//     in c++20 this comparison calls the current function recursively with reversed arguments

//     infinite recursion detected [-Winfinite-recursion]

// clang 14

//     all paths through this function will call itself [-Winfinite-recursion]
```

- Danger : your codebase can break
- prefers to call itself with swapped operands, which is a better match than the member function which needs a type conversion

C++20 : Rewrites : Endless Recursion → solutions

- Only C++20 → remove this method, no longer needed
- return rhs.operator==(lhs)
- return rhs == MagicInt{lhs}

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

```
bool operator== (int lhs, const
MagicInt& rhs)
{
    return rhs == MagicInt{lhs};
}
```

SpaceShip : operator<=>

- Rules of the game different depending on:
  - Defaulted
  - Not defaulted
- Don’t call it yourself (it is an implementation detail), use the normal operators (which it brings to the table)
- Depending on the members’ support:
  - constexpr
  - noexcept
- similar rewriting if the first operand can be "implicitly type converted"

SpaceShip : operator<=>

- takes preference over all other relational operators
- Return type : something that can be compared to 0
- #include <compare>

```
class MagicInt
{
public:
    MagicInt(int val) : mValue{val} {}

    auto operator<=> (const MagicInt& rhs) const = default;

private:
    int mValue;
};
```

```
0
1
0
1
0
1
1
0
1
1
0
```

```
const MagicInt f1{242};
const MagicInt f2{100};

std::cout << (f1 < f2) << std::endl;
std::cout << (f1 > f2) << std::endl;
std::cout << (f1 <= f2) << std::endl;
std::cout << (f1 >= f2) << std::endl;
std::cout << (f1 == f2) << std::endl;
std::cout << (f1 != f2) << std::endl;

const MagicInt f3{242};
std::cout << (f1 == f3) << std::endl;
std::cout << (f1 != f3) << std::endl;

std::vector vec{f1, f2};
std::sort(vec.begin(), vec.end());
```

SpaceShip : operator<=> : **DEFAULTED**

- It gives:
  - 4 relational operators
  - **2 equality operators**
- Compares an object member by member => order matters

- NOT defaulted => no equality operators !

3-way comparison

- a OP b
  - a OP b < 0
  - a OP b == 0
  - a OP b > 0
- eg.: std::strcmp
- a <=> b
  - a <=> b < 0
  - a <=> == 0
  - a <=> b > 0

Rewrites

- x <= y → if no matching definition, try
  - (x <=> y) <= 0
  - 0 <= (y <=> x)
- x <=> y equal to 0 → x and y equal or equivalent
- x <=> y less than 0 → x is less than y
- x <=> y greater than 0 → x is greater than y
- operator != never rewritten to call operator<=> (it might call an operator== generated from a defaulted operator<=>)

SpaceShip : operator<=> : constexpr

```
struct Coordinate
{
    double x{};
    double y{};
    double z{};

    auto operator<=> (const Coordinate&) const = default;
};
```

```
constexpr Coordinate co{1.0, 2.0, 3.0};
static_assert(co < Coordinate{1.1, 0.0, 0.0});
```

SpaceShip : Not Default

- No “==” → if needed → add it explicitly

```
struct Person
{
    std::string firstName;
    std::string lastName;

    auto operator<=> (const Person& rhs) const
    {
        return lastName <=> rhs.lastName;
    }

    /// we need to add this one !!!
    bool operator== (const Person& rhs) const
    {
        return lastName == rhs.lastName;
    }
};
```

```
std::vector<Person> vec{ {"Eric", "Cartman"},
{"Stan", "Marsh"}, {"Kyle", "Broflovski"},
{"Kenny", "McCormick"}  };

std::sort(vec.begin(), vec.end());

Person author1{"Trey", "Parker"};
Person author2{"Matt", "Stone"};

std::cout << (author1 == author2) << std::endl;
```

SpaceShip : Multiple Members, and some don’t matter

- Struct Person
  - LastName
  - FirstName
  - Age
  - SomeData (which we don’t care about)
- → not default
- → add “==”
- → implement by calling <=> on the members

SpaceShip : Multiple Members, and some don't matter

```
struct Person
{
    std::string firstName{};
    std::string lastName{};
    int age{};
    std::vector<int> someData{};

    bool operator== (const Person& rhs) const noexcept
    {
        return firstName == rhs.firstName &&
               lastName == rhs.lastName &&
               age == rhs.age;
    }
```

```
auto operator<=> (const Person& rhs) const noexcept
{
    auto cmp = lastName <=> rhs.lastName;   /// 1st criterion
    if(cmp != 0)
    {
        return cmp;
    }
    cmp = firstName <=> rhs.firstName;   /// 2nd criterion
    if(cmp != 0)
    {
        return cmp;
    }
    return age <=> rhs.age; /// 3rd criterion
}
```

Compare / order ???

- Can we always compare or order ?
- Equal versus Equivalent
- Hello ↔ hello
- Nan: Not A Number
- std::less

```
0
0
0
0
10
24.2
nan
```

```
const double d1{24.2};
const double d2{10.0};
const double d3{std::numeric_limits<double>::quiet_NaN() };
// ALL FALSE
std::cout << (d1 == d3) << std::endl;
std::cout << (d1 < d3) << std::endl;
std::cout << (d1 > d3) << std::endl;
std::cout << (d3 == d3) << std::endl;

// BUT
std::vector vec{d1, d3, d2};
std::sort(vec.begin(), vec.end());

for(const auto& value : vec)
{
    std::cout << value << “ “;
}
```

Different Comparison Categories

- Strong Ordering
- Weak Ordering
- Partial Ordering
- Stronger ordering can convert to a weaker one, but not the other way around (implicit type conversions)
- Is the return type of the <=> operator → different return types

Strong Ordering (Total Ordering)

- Any value of a given type is : less than, or equal, or greater than any other value of this type
- Examples : int, std::string
- std::strong_ordering
  - std::strong_ordering::less
  - std::strong_ordering::equal (std::strong_ordering::equivalent)
  - std::strong_ordering::greater

Weak Ordering

- Any value of a given type is : less than, or **equivalent**, or greater than any other value of this type
- Equivalent does not mean they have to be equal
- Examples : case insensitive strings
- std::weak_ordering
  - std::weak_ordering::less
  - std::weak_ordering::equivalent
  - std::weak_ordering::greater

Partial Ordering

- Any value of a given type **COULD BE** : less than, or *equivalent*, or greater than any other value of this type
- It may not be possible to specify an order between 2 values at all → unordered
- Examples : floating point types
- std::partial_ordering
  - std::partial_ordering::less
  - std::partial_ordering::equivalent
  - std::partial_ordering::greater
  - std::partial_ordering::**unordered**

Avoid

- if (x <=> y == std::strong_ordering::equal)
- → might not compile
- → if (x <=> y == 0)

What in case of multiple ordering criteria of different comparison categories

```
struct Person
{
    std::string firstName{};
    std::string lastName{};
    double age{};
    std::vector<int> someData{};

    bool operator== (const Person& rhs) const noexcept
    {
        return firstName == rhs.firstName &&
               lastName == rhs.lastName &&
               age == rhs.age;
    }
```

- std::string → strong
- double → partial
- What is return type of operator<=> ?
- auto → nope : different return types will be deduced

Go for the common ground → weaker type

```
std::partial_ordering operator<=> (const Person& rhs) const noexcept
{
    auto cmp = lastName <=> rhs.lastName;   /// 1st criterion
    if(cmp != 0)
    {
        return cmp;
    }
    cmp = firstName <=> rhs.firstName;   /// 2nd criterion
    if(cmp != 0)
    {
        return cmp;
    }
    return age <=> rhs.age; /// 3rd criterion
}
```

- First 2 → strong → implicit conversion to partial
- Third one → partial

Map to the stronger type

```
    std::strong_ordering operator<=> (const Person& rhs) const noexcept
    {
        auto cmp = lastName <=> rhs.lastName;   /// 1st criterion
        if(cmp != 0)
        {
            return cmp;
        }
        cmp = firstName <=> rhs.firstName;   /// 2nd criterion
        if(cmp != 0)
        {
            return cmp;
        }
        const auto res = age <=> rhs.age; /// 3rd criterion
```

- First mappings trivial
- What to do with ‘unordered’ ?

```
        if(res == std::partial_ordering::less)
        {
            return std::strong_ordering::less;
        }
        else if (res == std::partial_ordering::equivalent)
        {
            return std::strong_ordering::equal;
        }
        else if (res == std::partial_ordering::greater)
        {
            return std::strong_ordering::greater;
        }
        /// aka -->  std::partial_ordering::unordered  --> let's just choose something, eg less
        /// or when we assume that an std::partial_ordering::unordered should not happen, we could throw
        return std::strong_ordering::less;
    }
```

Map to the stronger type (there is help for double)

```
std::strong_ordering operator<=> (const Person& rhs) const noexcept
{
    auto cmp = lastName <=> rhs.lastName;   /// 1st criterion
    if(cmp != 0)
    {
        return cmp;
    }
    cmp = firstName <=> rhs.firstName;   /// 2nd criterion
    if(cmp != 0)
    {
        return cmp;
    }
    return std::strong_order(age, rhs.age); /// 3rd criterion
}
```

- Also works for NaN
- std::compare_three_way function object for operator<=> (similar like std::less function object for operator<)

What if we don’t know the types (generic code) ?

- What could be the comparison categories for those unknown types ?
- Who is stronger, who is weaker ?
- determine the common ground (aka the 'greatest common divisor'):
  - std::common_comparison_category<T1, T2>
  - computes the strongest comparison category

What if we don't know the types (generic code) ?

```
struct Person
{
    std::string name{};
    double value{};
    std::vector<int> someData{};

    auto operator<=> (const Person& rhs) const noexcept
        -> std::common_comparison_category_t<decltype(name <=> name), decltype(value <=> value)>
    {
        const auto cmp = name <=> rhs.name;
        if(cmp != 0)
        {
            return cmp;
        }
        return value <=> rhs.value;
    }
};
```

Rewrites / Overload Resolution : Equality operators

- x != y
- Compiler will try: note: a rewritten expression never tries to call a member operator!=
  - x.operator!=(y)
  - operator!=(x, y)

  - ! x.operator==(y)
  - ! operator==(x, y)
  - ! y.operator==(x)

  - ! x.operator==(y) (generated from x.operator<=>)
  - ! y.operator==(x) (generated from y.operator<=>)

Rewrites / Overload Resolution : Relational operators

- Rewritten statements fall back on operator<=> and compare the result with 0
- Example : x <= y
- Compiler will try:
  - x.operator<=(y)
  - operator<=(x, y)

  - x.operator<=>(y) <= 0
  - operator<=>(x, y) <= 0
  - 0 <= y.operator<=>(x)

Concepts

- Require all 6 operators
- Standard concept : three_way_comparable

```
template <typename T>
requires std::three_way_comparable<T>
void foo(const T& /*t*/)
{
}
```

```
class MagicInt
{
public:
    MagicInt(int val) : mValue{val} {}


    auto operator<=> (const MagicInt& rhs) const = default;

private:
    int mValue;
};
```

```
class NonDefaultedInt
{
public:
    NonDefaultedInt(int val) : mValue{val} {}


    auto operator<=> (const NonDefaultedInt& rhs) const
    {
        return mValue <=> rhs.mValue;
    }

    bool operator== (const NonDefaultedInt& rhs) const
    {
        return mValue == rhs.mValue;
    }

private:
    int mValue;
};
```

Concepts

```
foo(242);

const MagicInt f1{242};
foo(f1);            /// will fail it the <=> is not there

const NonDefaultedInt f2{242};
foo(f2);            /// will fail it the <=> AND == are not there
```

Concepts

- Hand made : require <=>
- Note : <=> does not always bring ==
- NonDefaultInt now has no == in this test

```
template <typename T>
concept MySpaceshippable = requires (T t)
{
    t <=> t;
};

template <typename T>
requires MySpaceshippable<T>
void foo(const T& /*t*/)
{
}
```

```
foo(242);

const MagicInt f1{242};
foo(f1);          /// will fail it the <=> is not there

const NonDefaultedInt f2{242};
foo(f2);          /// will fail it the <=> is not there
(== is not required, just <=>)
```

Concepts

- Hand made and we want all 6

```
template <typename T>
concept MySpaceshippable6 = requires (T t)
{
    t <=> t;
    t == t;
};

template <typename T>
requires MySpaceshippable6<T>
void foo(const T& /*t*/)
{
}
```

Concepts

- Hand made and we want all 6
- Use a standard template (equality_comparable) and something we add

```
template <typename T>
concept MySpaceshippable6 = std::equality_comparable<T> && requires (T t)
{
    t <=> t;
};

template <typename T>
requires MySpaceshippable6<T>
void foo(const T& /*t*/)
{
}
```

Inheritance

- First compares the base classes, going from left to right
- Defaulted on Derived, as such requires:
  - Operators on the its members
  - Operators on the base class
- So it does not automatically defaults the base too, the base class has it’s own independent life as usual

Inheritance

```
struct Base
{
    int x{};

    auto operator<=> (const Base& rhs) const = default;
};

struct Derived : Base
{
    int y{};

    auto operator<=> (const Derived& rhs) const = default;
};
```

Inheritance

```
Derived d1 {500, 50};  /// x is 500, y is 50 ==> base equal, derived not
Derived d2 {500, 40};

std::cout << (d1 == d2) << std::endl; // false

Derived d3 {500, 50};
Derived d4 {600, 50};

std::cout << (d3 == d4) << std::endl; // false

Derived d5 {500, 50};
Derived d6 {500, 50};

std::cout << (d5 == d6) << std::endl; // true
```

QUESTIONS