Chapter 6

Three-way comparisons: Simplify your comparisons

The previous chapters have shown some of the enormous improvements provided by C++20. In this chapter, I present the new spaceship operator, which helps us write less code when defining comparisons. Writing comparisons becomes easier and, by default, more correct.

The name spaceship comes from how this operator looks: `<=>`. The operator looks like one of those star-fighters in a famous space-movie series. The operator is not something entirely new; something like it is available in other languages. For C++, it helps to follow the principle of writing less code and let the compiler do the work.

C++ is a language that allows a developer to write less code. For example, we need not write `this` before every method or for every member access. Calls to the constructors and destructors happen automatically in the background according to some rules. With C++11's `=default`, we can request the compiler's default implementation for special member functions, even if under normal circumstances, the compiler would not do so. Not having to write special member functions, as simple as a default constructor can be, is something I consider highly valuable.

There was at least one dark corner where the compiler did not help us to the same extent. Whenever we had in the past a class that required comparison operators, we

were required to provide an implementation. Sure, if there was some special business going on, then it was sensible to do so, the compiler could not know that. However, consider a basic case where you needed to provide an `operator==` which did something special. For consistency reasons, you usually would have provided `operator!=` as well. But what was its implementation? Well, in all cases I can remember, the implementation was this:

```
bool operator!=(const T& t) { return !(*this == t); }
```

Isn't that a little sad? That is not special code. It is absolutely trivial, but has to be written, reviewed and maintained. Let's see how C++20 tackles this corner.

6.1 Writing a class with equal comparison

Let's start with imagining we have to write some kind of medical application. Whenever you enter a hospital you are identified by a unique Medical Record Number (MRN). This number is used during your entire stay to identify you, because your full name may not be unique. That is even true for my name, which is not common in German. The same goes for my brother. We focus on the implementation of an MRN class. At this point, we do not care about all the access functions, just a class with a data member holding the actual value. It shall be default-constructible and constructible by a `uint64_t` which is the internal type of the MRN where the value is stored. Implementing a class `MedicalRecordNumber` can be like this:

Listing 6.1

```
class MedicalRecordNumber {
public:
  MedicalRecordNumber() = default;
  explicit MedicalRecordNumber(uint64_t mrn)
  : mMRN{mrn}
  {}

private:
  uint64_t mMRN;
};
```

Of course, we want two MRNs to be comparable to each other. They can either be equal, which means it is the same patient, or not equal, if the two numbers belong to different patients. There should be no ordering between different MRNs, since they are generated in an unknown order to prevent malicious actions. Objects of the class, as defined so far, cannot be compared with anything. The following trivial code, which tests whether two objects represent the same person, fails to compile:

```
MedicalRecordNumber mrn0{};
MedicalRecordNumber mrn1{3};
const bool sameMRN = mrn0 == mrn1;
```

The compiler does not know how to compare `MedicalRecordNumber` with `MedicalRecordNumber`. Adding the member function `operator==` inside the class makes the previous example work.

Listing 6.2

```
bool operator==(const MedicalRecordNumber& other) const
{
  return other.mMRN == mMRN;
}
```

With that, the former comparison works. But to make not equal (`!=`) work as well, we have to provide an operator for that too:

Listing 6.3

```
bool operator!=(const MedicalRecordNumber& other) const
{
  return !(other == *this);
}
```

6.1.1 Comparing different types

What we have done so far, adding and implementing `operator==`, is still simple. Now, let's say that the MRN should also be comparable to a plain `uint64_t`. This requires us to write yet another pair of equality operators. However, this is not all. Let's think about which comparison combinations we can have. We would like to compare an MRN object with the plain type `uint64_t`. How about the other way around? Sure, that should work as well. This is consistent behavior. Speaking in code, the following should compile:

```
const bool sameMRNA = mrn0 == 3ul;
const bool sameMRNB = 3ul == mrn0;
```

That means we need to add 4 more (2 for ==, 2 for !=) functions to our class, adding up to 6 total methods exclusively for equality comparisons. Two for == and two for !=. We end up with this:

Listing 6.4

```
A The initial member functions
bool operator==(const MedicalRecordNumber& other) const
{
  return mMRN == other.mMRN;
}

bool operator!=(const MedicalRecordNumber& other) const
{
  return !(*this == other);
}

B The additional overloads for uint64_t
friend bool operator==(const MedicalRecordNumber& rec,
                       const uint64_t&            num)
{
  return rec.mMRN == num;
}

friend bool operator!=(const MedicalRecordNumber& rec,
                       const uint64_t&            num)
{
  return !(rec == num);
}

C The additional overloads with swapped arguments for uint64_t
friend bool operator==(const uint64_t&            num,
                       const MedicalRecordNumber& rec)
{
  return (rec == num);
}
```

```
31
32 friend bool operator!=(const uint64_t&            num,
33                         const MedicalRecordNumber& rec)
34 {
35   return !(rec == num);
36 }
```

Listing 6.4

In addition to the two original methods (A), we needed two additional overloads (B), defined as `friend`-functions, plus two more overloads for swapped arguments (C). The boiler-plate code just increased by a lot.

The `friend`-trick

The reason for the `friend`-functions here is not simply to make `==` and `!=` work, but also the comparison to `uint64_t`. Without this `friend`-trick, the following would compile:

```
1 const bool sameMRN = mrn0 == 3ul;
```

but the other way around wouldn't:

```
1 const bool sameMRN = 3ul == mrn0;
```

With the operators as friends taking two arguments, they are considered during Argument Dependent Lookup (ADL), because one of the comparison objects is of type `MedicalRecordNumber`.

6.1.2 Less hand-written code with operator reverse, rewrite and `=default`

Do you enjoy writing such code? At this point, we have 6 comparison functions from which 4 are simple redirects. C++20 enables us to apply `=default`, which we gained with C++11, here as well, requesting the compiler to fill in the blanks. It makes this code much shorter:

```
1 bool operator==(const MedicalRecordNumber& other) const =
2   default;
3 bool operator==(const uint64_t& other) const
4 {
5   return other == mMRN;
6 }
```

Listing 6.5

From six functions down to two, where we can request the compiler to provide one of the two functions for use by using `=default`. That is a reduction I consider

absolutely worthwhile. But wait, did I cheat? What about the `operator!=`, they are missing, so I clearly cheated, and we should need four functions instead, which would not be a big reduction. Plus the `friend`-trick is gone, so clearly this code should not work as before. The good news is, I did not cheat, and the code works exactly as before. In C++20, we only need the two functions provided in Listing 6.5 on page 179, period. The reasons for it are two new abilities of the compiler operator reverse and rewrite, which are explained in detail in §6.6 on page 192. Without knowing more about these two abilities, you are already good to go and write your reduced equality comparisons.

6.2 Writing a class with ordering comparison, pre C++20

In the sections before we looked at equality comparison, which is one part. Sometimes we need more than to check for equality, we like to order things. Every time we need to sort unique objects of a class, it involves the operators `<`, `>`, `<=`, `>=` and the equality comparisons `==` and `!=` to establish an order between the objects which are sorted.

Sticking with the example to write a medical application, think about a class that represents a patient name. Names are comparable to each other. They can be equal (or not) and sorted alphabetically. In our case, the names are stored in a class `String` which is a wrapper around a `char` array, and it should offer ordering comparison. It does its job by storing a pointer to the actual string containing the name and its length. Our class is called `String` class with a constructor that takes a `char` array. The length is determined by a constructor template. To fulfill the requirements of the patient name, this class should be comparable for equality and provide ordering such that for two `String`-instances, we can figure out which is the greater one, or if they are the same.

As you can see and might have experienced yourself, we need to implement all six comparison functions in order to achieve this. A common approach here is to have one function which does the actual comparison, let's name it `Compare`. It returns `Ordering`, a type with three different values, less (−1), equal (0), or greater (1). These three values are the reason for another name of the spaceship operator, this is

sometimes referred to as three-way comparison. In fact, the standard uses the term three-way comparison for the spaceship operator.

Back to the `String` class. All six comparison operators `==`, `!=`, `<`, `>`, `<=`, `>=` call `Compare` and create their result based on the return-value of `Compare`. At this point, `String` does not have a C++20 spaceship operator, but the result `Compare` gives and the fact that `String` provides all six comparisons makes `String` work as if with C++20 and the spaceship operator. Here it is emulated with pre C++20 code, the way we had to do it for many years.

```
class String {
public:
  template<size_t N>
  explicit String(const char (&src)[N])
  : mData{src}
  , mLen{N}
  {}

  A Helper functions which are there for completeness.
  const char* begin() const { return mData; }
  const char* end() const { return mData + mLen; }

  B The equality comparisons.
  friend bool operator==(const String& a, const String& b)
  {
    if(a.mLen != b.mLen) {
      return false;  C Early exit for performance
    }

    return Ordering::Equal == Compare(a, b);
  }

  friend bool operator!=(const String& a, const String& b)
  {
    return !(a == b);
  }

```

Listing 6.6

```
28    (D) The ordering comparisons.
29    friend bool operator<(const String& a, const String& b)
30    {
31      return Ordering::LessThan == Compare(a, b);
32    }
33
34    friend bool operator>(const String& a, const String& b)
35    {
36      return Ordering::GreaterThan == Compare(a, b);
37    }
38
39    friend bool operator<=(const String& a, const String& b)
40    {
41      return Ordering::GreaterThan != Compare(a, b);
42    }
43
44    friend bool operator>=(const String& a, const String& b)
45    {
46      return Ordering::LessThan != Compare(a, b);
47    }
48
49  private:
50    const char*  mData;
51    const size_t mLen;
52
53    (E) The compare function which does the actual comparison.
54    static Ordering Compare(const String& a, const String& b);
55  };
```

Listing 6.6

`String` has the equality comparisons (== and !=) (B) and the ordering comparisons (<, >, <=, >=) (D). The `Compare` method is private (E). `Compare` (E) returns `Ordering`, which as said before, can be seen as a class enum with the three values `Equal`, `LessThan`, and `GreaterThan`. The possibly hardest part is the implementation of `Compare` itself. The rest is just noise. We ignore implementing `Compare` at this point and focus on the comparison operators and how much code we have to write to enable comparisons for this, and any other class.

What if `String` should also to be comparable to a `std::string`? The number of comparison operators increases by 12! The reason is that we need to provide both operator pairs:

- `const String& a, const std::string& b`
- `const std::string& a, const String& b`

This becomes a lot more boiler-plate code. In fact, for each type we like this class to be comparable with, the number of operators increases by 12. Say that we want it also be comparable to a C-style string, we end up with 30 operators! All of them simply redirect to the `Compare` function. Okay, the `std::string` version would call `.c_str` on the object.

This is where the spaceship operator comes in, for consistent comparisons.

6.3 Writing a class with ordering comparison in C++20

The spaceship operator in C++ is written as `operator<=>` and has a dedicated return-type which can be expressed as less than (−1), equal to (0), or greater than (1), more or less the same as the `Ordering` returned by `Compare` in Listing 6.6 on page 181. This type, defined in the header `<compare>`, is not required for just the equality comparison functions `==` and `!=`, but is as soon as we want ordering. We will discuss the different comparison types in §6.4 on page 186. With `=default`, we can request a default implementation from the compiler for the spaceship operator, like for the special member functions and the equality operators we saw before. With respect to the `String` example, this means throwing all the `operator@@` out, replacing it by a single `operator<=>`, and including the `compare` header like this:

Listing 6.7

```
1 #include <compare>
2
3 auto operator<=>(const String& other) const = default;
```

Isn't that great? From six functions down to one, and we are done.

The two operators (A) define a weak order. However, as it was written before C++20, it doesn't have a spaceship operator. Which at times was totally fine, of course. Now, if we have to write a new class `ShinyCpp20Class` which has a field of type `Legacy`. It should be ordering comparable, preferably by using the spaceship operator. The question is, how do we do that? What is the most efficient and painless way to write it? We can implement a spaceship operator who does the three-way comparison for each of the struct's fields. This is writing boiler-plate code. Thanks to a last-minute update to the standard, we can just default the spaceship operator:

Listing 6.17

```
class ShinyCpp20Class {
  Legacy mA;
  Legacy mB;

public:
  ShinyCpp20Class(int a, int b);

  std::weak_ordering  (A)
  operator<=>(const ShinyCpp20Class&) const = default;
};
```

The only special thing is that we have to be specific about the return-type of `operator<=>`. We cannot just say `auto`. (A) shows that in this case we apply `weak_ordering` as the return type. This is what `Legacy` allows us. This gives us the ability to incorporate pre-C++20 code into new, code and apply the spaceship operator there.

A word about upgrading to C++20

When we say consistent comparisons, we mean it. This is a marvelous thing. Consistent comparisons makes the language stronger and reduces the places where we can make mistakes.
C++20 does clean up some bad code we could have written before C++20, sadly leading to some interesting situations. Have a look at the following code:

```
1  struct A {
2    bool operator==(B&) const { return true; }  (A)
3  };
4
5  struct B {
6    bool operator==(const A&) const { return false; }  (B)
7  };
```

Listing 6.18

The piece of code in Listing 6.18 is questionable. Not because of the silly return true and false. Have a look at the parameter signatures. (A) takes a non-`const` parameter, which is quite questionable. On the other hand, (B) looks like a signature as it ought to be. Despite the missing `const` on (A), this code works just fine, as long as it is not invoked with a const object of type `B`. What are the odds for that? Well, if there is a `const B` object, the code stops compiling. Assume that it compiled in C++17, it implies that it worked *correctly* in C++17, missing `const` or not.

Remember that in C++20 we have consistent comparisons? They are consistent even without the spaceship operator. Why? Because for the equality operator, it allows the compiler to reverse the arguments. When it does this with the code in Listing 6.18, (A) becomes the better match in the overload set, because it does not have the `const`. Using (B) requires an internal `const`-cast of the compiler for object `A`. These additional actions make (A) the better match, as there is none required.

Now, the code has worked in C++17, but for the wrong reasons. There are more corner-case examples like this. All of them reveal inconsistencies. By upgrading your code to C++20, the compiler will point all these inconsistencies out to you. The assumption is that such things happen only very, very rarely.

Cliff notes

- Consistent comparisons are a valuable feature that can take a lot of boiler-plate code from our plate. At the same time it ensures that our comparisons are heterogeneous and with that consistent.
- The compiler may reverse every comparison of the form `a.operator@(b)`: the compiler may change it to `b.operator@(a)`, if this is the best match.
- For that reason you no longer need the `friend`-trick, just declare your comparison operators as member functions.

- The compiler also performs *rewrites* where `a.operator!=(b)` can be rewritten to `!a.operator==(b)`.
- We can `=default` all comparison operators who take the class itself as an argument.
- `constexpr` can be added or removed when using `=default`.
- Remember that `=default` performs a member-wise comparison. This is the same meaning for the comparison operators as for the special member functions like the copy constructor.
- Member-wise comparison goes through all members in declaration order from top to bottom.
- When defaulting comparison operators, prefer the *primary* operators. Since they provide the full complement of all comparisons and the compiler can rewrite them, this is all you need.
- When you default `operator<=>`, you automatically also get the equality comparison operators. However, if you provide your own `operator<=>` implementation, also provide your own `operator==` version. Refrain from defaulting `operator==` in this case, as it still defaults to member-wise comparison.
- You need to include the `<compare>` header to get the new std-types.
- Invoke `<=>` when you ask for comparison, use `==` when you ask for equality.