Introduction to the C++20 spaceship operator

CMP · Follow
7 min read · Aug 28, 2023

C++20 introduced the three-way comparison operator, also known as the "spaceship operator" due to its appearance: `<=>` . The purpose is to streamline the process of comparing objects.

The Basics

Below is a simple example that uses this new spaceship operator:

```
#include <compare>

int main() {
  int a = 1;
  int b = 2;

  auto result = a <=> b;

  if (result < 0) {
    std::cout << "a is less than b" << std::endl;
  } else if (result == 0) {
    std::cout << "a is equal to b" << std::endl;
  } else { // result > 0
    std::cout << "a is greater than b" << std::endl;
  }

  return 0;
}
```

Note that the `compare` header must be included.

For integral types such as `int`, the type of the value returned by the spaceship operator is `std::strong_ordering`, which can have one of three values:

- `std::strong_ordering::less`: If the left operand (`a`) is less than the right operand (`b`).
- `std::strong_ordering::equal`: If `a` is equal to `b`.
- `std::strong_ordering::greater`: If `a` is greater than `b`.

For floating-point types such as `double`, the spaceship operator returns one of four possible values:

- `std::partial_ordering::less`: If `a` is less than `b`.
- `std::partial_ordering::equivalent`: If `a` is "equivalent" to `b`. This is essentially the same as "equal", but also includes the case of `-0 <=> +0`.
- `std::partial_ordering::greater`: If `a` is greater than `b`.
- `std::partial_ordering::unordered`: If `a` or `b` is `NaN`.

You can also directly use the spaceship operator with some other types, such as `std::vector` and `std::string`.

Objects

Let's first look at ordering in a custom data structure *before* C++20:

```
struct Foo {
    int value;

    bool operator==(const Foo& rhs) const {
        return value == rhs.value;
    }
```

```
    bool operator!=(const Foo& rhs) const {
        return !(value == rhs.value);
    }
    bool operator<(const Foo& rhs) const {
        return value < rhs.value;
    }
    bool operator>(const Foo& rhs) const {
        return value > rhs.value;
    }
    bool operator<=(const Foo& rhs) const {
        return value <= rhs.value;
    }
    bool operator>=(const Foo& rhs) const {
        return value >= rhs.value;
    }
};

int main() {
  Foo a{1};
  Foo b{2};

  std::cout << std::boolalpha << (a == b) << std::endl; // prints false
  std::cout << std::boolalpha << (a != b) << std::endl; // prints true
  std::cout << std::boolalpha << (a < b) << std::endl; // prints true
  std::cout << std::boolalpha << (a > b) << std::endl; // prints false
  std::cout << std::boolalpha << (a <= b) << std::endl; // prints true
  std::cout << std::boolalpha << (a >= b) << std::endl; // prints false
}
```

In this example compiled with C++17, if we do not define all of the comparison operators in the `Foo` structure, then the compiler will generate errors when those missing operators are used. All of the boilerplate code can be highly simplified in C++20 by using the spaceship operator:

```
#include <compare>

struct Foo {
    int value;

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

int main() {
```

```
    Foo a{1};
    Foo b{2};

    std::cout << std::boolalpha << (a == b) << std::endl; // prints false
    std::cout << std::boolalpha << (a != b) << std::endl; // prints true
    std::cout << std::boolalpha << (a < b) << std::endl; // prints true
    std::cout << std::boolalpha << (a > b) << std::endl; // prints false
    std::cout << std::boolalpha << (a <= b) << std::endl; // prints true
    std::cout << std::boolalpha << (a >= b) << std::endl; // prints false
}
```

There are a few things to note here. First, the operator no longer has a return type of `bool` and instead has `std::strong_ordering` in this case, which can be deduced by the compiler when using `auto`. Second, C++20 also added the ability to default comparison operators, eliminating the need to write out `return value <=> rhs.value;`. However, if you took the first C++17 example and simply replaced the operator definitions with `default`, that would not work! This is due to the concept of *rewriting*, which is explained later in this article.

Primary vs Secondary Operators

Look at the following table from Barry Revzin's article Comparisons in C++20:

	Equality	Ordering
Primary	`==`	`<=>`
Secondary	`!=`	`<`, `>`, `<=`, `>=`

In C++20, there are two categories of operators: Equality and Ordering. The Primary Equality operator is `==` and the Primary Ordering operator is `<=>`. The Secondary Equality operator is `!=`, and the Secondary Ordering operators are `<`, `>`, `<=`, and `>=`.

Primary operators can always be defaulted, and you can default the Secondary operators *if* the corresponding Primary operator is defined. For example, the following example compiles correctly:

```
#include <compare>

struct Foo {
    int value;

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

int main() {
  Foo a{1};
  Foo b{2};

  std::cout << std::boolalpha << (a < b) << std::endl; // prints true
}
```

However, if the `<=>` spaceship operator is *not* defined, then this code will not compile:

```
#include <compare>

struct Foo {
  int value;

  bool operator<(const Foo& rhs) const = default;
};

int main() {
  Foo a{1};
  Foo b{2};

  std::cout << std::boolalpha << (a < b) << std::endl; // Object of type 'Foo'
}
```

C++20 has two new features related to comparison operators:

- **Primary operators can be reversed:** Take the following example:

```
#include <compare>

struct Foo {
  int value;

  explicit Foo(int value) : value(value) {}

  bool operator==(const int otherValue) const {
    return value == otherValue;
  }
};

int main() {
  Foo a{10};

  std::cout << std::boolalpha << (a == 10) << std::endl; // prints true
}
```

It is clear that `a == 10` will return `true` because the constructor sets `a.value` to `10`, and the expression gets evaluated as `a.operator==(10)`. However, up until C++20, the expression `10 == a` would give a compiler error because there is no such operator. In C++20, Primary operators can be *reversed*, meaning that `10 == a` would compile because the language knows that `a == 10` and `10 == a` functionally mean the same thing.

- **Secondary operators can be rewritten:** In C++20, Secondary operators can be *rewritten* in terms of their Primary operator. For example, `a < b` is rewritten as `(a <=> b) < 0`. This is what allows the spaceship operator to replace the other Ordering operators. Look at the following example:

```
#include <compare>

struct Foo {
    int value;
```

```
    explicit Foo(int value) : value(value) {}

    auto operator<=>(const int otherValue) const {
      return value <=> otherValue;
    }
};

int main() {
  Foo a{1};

  std::cout << std::boolalpha << (a < 10) << std::endl; // prints true
}
```

Here, when evaluating `a < 10`, the compiler would first look for `operator<` and fail, then look for the rewritten version of the Primary operator. So, this expression would be evaluated as `a.operator<=>(10) < 0`, allowing us to use `<` without explicitly defining that operator in the `Foo` structure.

Likewise, the same principle applies to the other Primary operator: `==` and its Secondary operator: `!=`. For these Equality operators, `a != b` would be rewritten as `!(a == b)`.

Important note: Equality and Ordering operators are logically separated, meaning that `<=>` will never invoke `==` and vice versa. This means that you should always define **both** `<=>` and `==` Primary operators, and **only** those Primary operators (Secondary operators will be rewritten, so no need to define them). A caveat is that if you default the `<=>` operator, then the `==` operator will be implicitly defaulted. To make this clear, here are some examples of "good" and "bad" uses of the comparison operators in C++20:

- Good: `<=>` defaulted, which implicitly defaults `==`

```
struct Foo {
  int value;

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

```
int main() {
  Foo a{1};
  Foo b{2};

  std::cout << std::boolalpha << (a < b) << std::endl; // No error. <=> is defa
  std::cout << std::boolalpha << (a == b) << std::endl; // No error. == is impl
}
```

- Bad: Only defaulting `==`

```
struct Foo {
  int value;

  bool operator==(const Foo& rhs) const = default;
};

int main() {
  Foo a{1};
  Foo b{2};

  std::cout << std::boolalpha << (a < b) << std::endl; // Error. <=> is not def
  std::cout << std::boolalpha << (a == b) << std::endl; // No error. == is defa
}
```

- Good: Defining `<=>` and defining `==`

```
struct Foo {
  int value;

  auto operator<=>(const Foo& rhs) const {
    return value <=> rhs.value;
  }

  bool operator==(const Foo& rhs) const {
    return value == rhs.value;
  }
};
```

```
int main() {
  Foo a{1};
  Foo b{2};

  std::cout << std::boolalpha << (a < b) << std::endl; // No error. <=> is defi
  std::cout << std::boolalpha << (a == b) << std::endl; // No error. == is defi
}
```

- Bad: Only defining `<=>`

```
struct Foo {
  int value;

  auto operator<=>(const Foo& rhs) const {
    return value <=> rhs.value;
  }
};

int main() {
  Foo a{1};
  Foo b{2};

  std::cout << std::boolalpha << (a < b) << std::endl; // No error. <=> is defi
  std::cout << std::boolalpha << (a == b) << std::endl; // Error. == is not def
}
```

- Good: Defining `<=>` but also defaulting `==`

```
struct Foo {
  int value;

  auto operator<=>(const Foo& rhs) const {
    return value <=> rhs.value;
  }

  bool operator==(const Foo& rhs) const = default;
};

int main() {
```

```
  Foo a{1};
  Foo b{2};

  std::cout << std::boolalpha << (a < b) << std::endl; // No error. <=> is defi
  std::cout << std::boolalpha << (a == b) << std::endl; // No error. == is defa
}
```

Conclusion

Whether you actually use `<=>` directly for comparisons like in the first example, the spaceship operator is a useful feature for reducing the amount of boilerplate code when defining custom types and is an important addition to C++20 that modern C++ programmers should know about. To summarize the rules and best practices:

- The `<=>` operator returns a type of `*_ordering` rather than a `bool`.
- The Equality operators (`==` and `!=`) and the Ordering operators (`<`, `>`, `<=`, and `>=`) are logically separate
- The Primary operators (`==` and `<=>`) can be reversed
- The Secondary operators (`!=`, `<`, `>`, `<=`, and `>=`) can be rewritten in terms of their Primary operator
- When defining a custom type, you should simply default the `<=>` operator without defining any of the Secondary operators. If you cannot default it and need to define it in a more complex way, then you must also define/default the `==` operator

Cpp Cpp20 Cplusplus System Programming Programming

Follow

Written by CMP

74 Followers

Software engineer specializing in operating systems, navigating the intracicies of the C++ language and systems programming.

More from CMP

CMP

CMP

std::lock_guard vs std::unique_lock vs std::scoped_lock

So many C++ locking mechanisms… which one should you choose?

4 min read · Jun 16, 2023

35 1

CMP

Using Attributes in C++

A useful way to communicate with the compiler and other developers through code

3 min read · Jan 16

10 1

CMP

const vs constexpr in C++

Both const and constexpr are keywords used to specify that a value cannot be modified, but they have some differences in functionality and...

4 min read · Sep 15, 2023

37 5

See all from CMP

Recommended from Medium

Jakub Neruda

Getting started with C++ modules

Modules. A concept common in many programming languages (with Rust having the most similar implementation), but only standardized in C++20...

7 min read · Jan 18

1 1

Paul J. Lucas

Undefined Behavior in C and C++

What undefined behavior is, why it exists, and how to avoid it.

8 min read · Aug 31, 2023

11 1

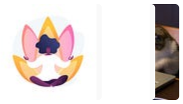
Lists

General Coding Knowledge

20 stories · 851 saves

Coding & Development

11 stories · 409 saves



Stories to Help You Grow as a Software Developer

19 stories · 761 saves

ChatGPT

21 stories · 428 saves

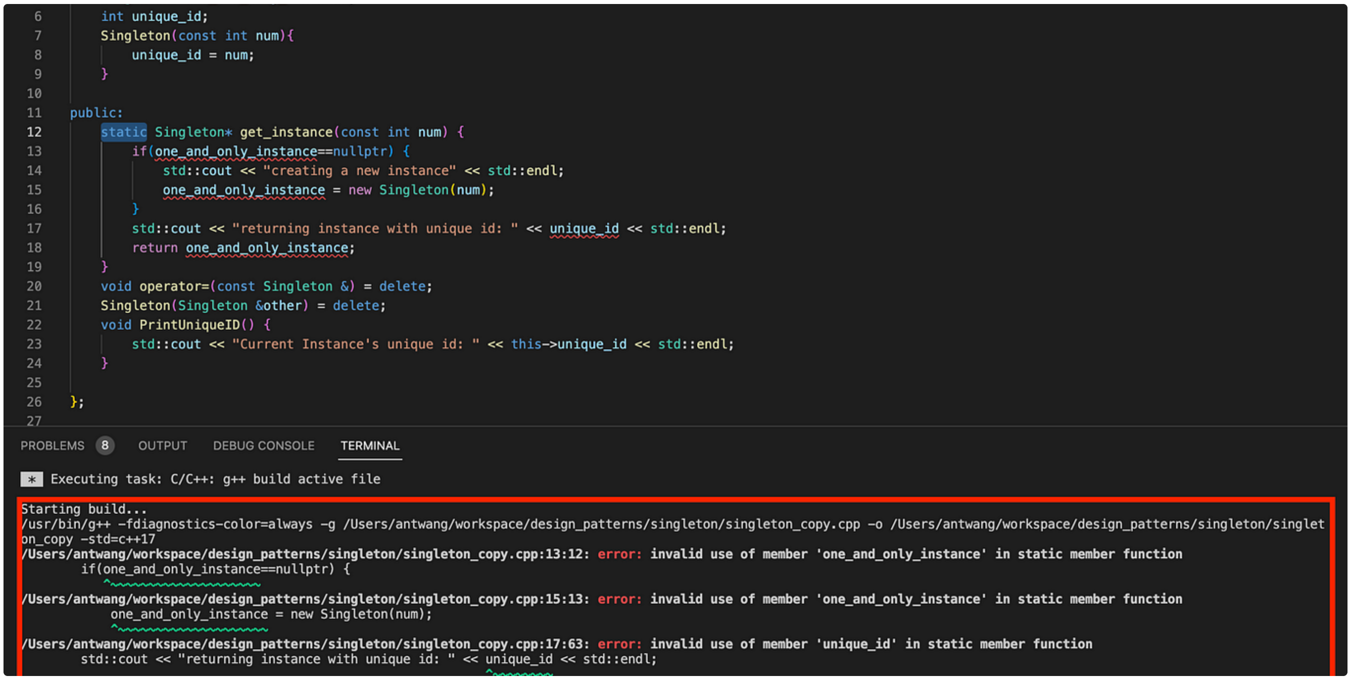
brk

Mastering Delegating Constructors in C++: Harnessing the Power of Delegating Constructors

Delegating constructors are a powerful feature in C++ that allow you to simplify and streamline your code by reusing the logic of one...

3 min read · Oct 9, 2023

12



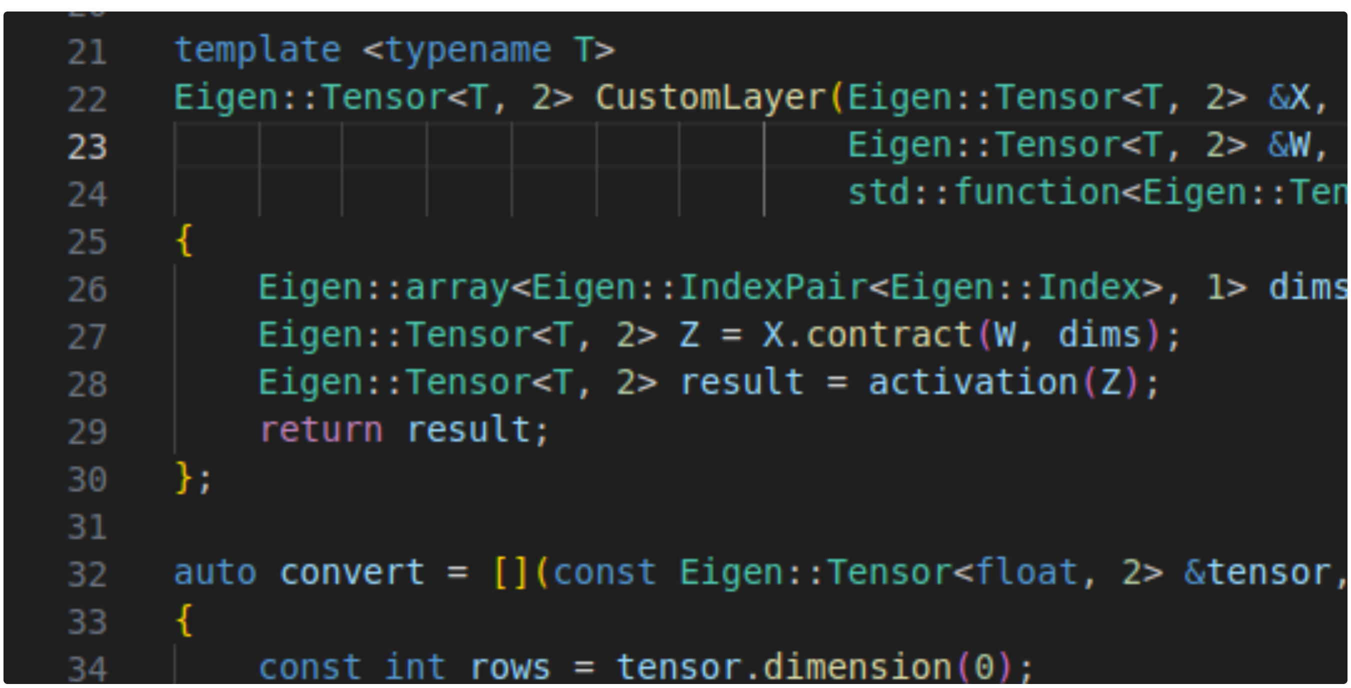
Anthony Wang

Singleton in C++

Continuing with the series where I share my learnings on design pattern from Refactoring Guru, after reading the tutorial on Singleton, I...

5 min read · Aug 23, 2023

96 2



Luiz doleron in Towards AI

Automatic Differentiation with Python and C++ for Deep Learning

This story explores automatic differentiation, a feature of modern Deep Learning frameworks that automatically calculates the parameter...

11 min read · Aug 30, 2023

91

CMP

Modern C++ memory allocation

Memory allocation is an important concept for systems programmers to understand, especially when working in environments where resources...

8 min read · Aug 1, 2023

77 1

See more recommendations