## **Evaluation Order in C++17**

Now let's try and understand evaluation order in C++ 17!

## WE'LL COVER THE FOLLOWING

- ^
- The example case
- What about Operator Overloading?

## The example case #

```
#include <iostream>
class Query
 public:
    Query& addInt(int i)
      std::cout << "addInt: " << i << '\n';
      return *this;
    Query& addFloat(float f)
      std::cout << "addFloat: " << f << '\n';</pre>
      return *this;
    }
};
float computeFloat()
  std::cout << "computing float... \n";</pre>
  return 10.1f;
}
float computeInt()
  std::cout << "computing int... \n";</pre>
  return 8;
}
int main()
  Query q;
  q.addFloat(computeFloat()).addInt(computeInt());
}
```







[]

You probably expect that using C++14 computeInt() happens after addFloat. Unfortunately, that might not be the case. For instance here's an output from GCC 4.7.3:

```
computing int...
computing float...
addFloat: 10.1
addInt: 8
```

The chaining of functions is already specified to work from left to right (thus addInt() happens after addFloat()), but the order of evaluation of the inner expressions can differ. To be precise:

The expressions are indeterminately sequenced with respect to each other.

With C++17, function chaining will work as expected when they contain inner expressions, i.e., they are evaluated from left to right: In the expression:

```
a(expA).b(expB).c(expC)
```

expA is evaluated before calling b().

Compiling the previous example with a conformant C++17 compiler, yields the following result:

```
computing float...
addFloat: 10.1
computing int...
addInt: 8
```

## What about Operator Overloading? #

Another result of this change is that when using operator overloading, the order of evaluation is determined by the order associated with the corresponding built-in operator.

For avample.

Tor champic.

```
std::cout << a() << b() << c();
```

The above code contains operator overloading and expands to the following function notation:

```
operator<<(operator<<(std::cout, a()), b()), c());</pre>
```

Before C++17, a(), b() and c() could be evaluated in any order. Now, in C++17, a() will be evaluated first, then b() and then c().

Here are more rules described in the paper P0145R3:

The following expressions are evaluated in the order a, then b:

- 1. a.b
- 2. a->b
- 3. a->\*b
- 4. a(b1, b2, b3) // b1, b2, b3 in any order
- 5. b @= a // '@' means any operator
- 6. a[b]
- 7. a << b
- 8. a >> b

If you're not sure how your code might be evaluated, then it's better to make it simple and split it into several clear statements. You can find some guides in the Core C++ Guidelines, for example ES.44 and ES.44.

Extra Info: The change was proposed in: P0145R3.

Now that you've understood evaluation order let's take a look at the concept of Copy Elision in the next lesson.