Expression Evaluation Order

Let's take a look at how C++ 14 and C++ 17 address Expression Evaluation

WE'LL COVER THE FOLLOWING

Expression Evaluation: C++ Older Version

Until C++17 the language hasn't specified any evaluation order for function parameters. *Period.*

For example, that's why in C++14 `make_unique` is not just syntactic sugar, but it guarantees memory safety.

Let's have a look at the following example:

```
foo(unique_ptr<T>(new T), otherFunction()); // first case
```

And with explicit `new`:

```
foo(make_unique<T>(), otherFunction()); // second case
```

Considering the first case, in C++14, we only know that `new T` is guaranteed to happen before the `unique_ptr` construction, but that's all. For example, `new T` might be called first, then `otherFunction()`, and then the constructor `unique_ptr` is invoked.

For such evaluation order, when `otherFunction()` throws, then `new T` generates a leak (as the unique pointer is not yet created).

When you use `make_unique`, as in the second case, the leak is not possible as you wrap memory allocation and creation of unique pointer in one call.

Expression Evaluation: C++17

C++17 addresses the issue shown in the first case. Now, the evaluation order of function arguments is “practical” and predictable. In our example, the compiler won’t be allowed to call `otherFunction()` before the expression `unique_ptr<T>(new T)` is fully evaluated.

In an expression:

```
f(a, b, c);
```

The order of evaluation of `a`, `b`, `c` is still unspecified, but any parameter is fully evaluated before the next one is started. It’s especially crucial for complex expressions like this:

```
f(a(x), b, c(y));
```

If the compiler chooses to evaluate `a(x)` first, then it must evaluate `x` before processing `b`, `c(y)` or `y`.

This guarantee fixes the problem with `make_unique` vs `unique_ptr<T>(new T())`. A given function argument must be fully evaluated before other arguments are evaluated.

Let’s take a look at evaluation order in C++ 17 in detail in the next lesson.