The mystery of the crash that seems to be on a std::move operation

devblogs.microsoft.com/oldnewthing/20220120-00

January 20, 2022

Raymond Chen

A customer was encountering a crash that appeared only on the ARM version of their program. Here's a simplified version:

```
void polarity_test(std::shared_ptr<Test> test)
{
    test->harness->callAndReport([test2 = std::move(test)]() mutable
    {
        test2->reverse_polarity();
        ::resume_on_main_thread([test3 = std::move(test2)]()
        {
            test3->reverse_polarity();
        });
    });
}
```

They reported that the first line was crashing on the `std::move`:

```
    test->harness->callAndReport([test2 = std::move(test)]() mutable
```

Now, `std::move` doesn't actually generate any code. It just changes the reference from an lvalue reference to an rvalue reference, which is an operation that takes place entirely in the computer's mind. There is no code generation to accompany it.

The problem is somewhere else.

Since the problem occurred only on one CPU architecture, it's possible that there was a bug in the back-end code generator. But just to be safe, they contacted the compiler front-end team, the back-end team (for code generation), and the libraries team (for `shared_ptr`).

I stepped in and pointed out that there was an order-of-evaluation dependency.

```
    test->harness->callAndReport([test2 = std::move(test)]() mutable
```

The left part of the statement reads from `test`. The lambda capture modifies `test` (by moving it to the captured variable `test2`).

Historically, the order of evaluation of subexpressions is left unspecified for the most part, although a handful of operations define an order, most notably that short-circuiting expressions evaluate the first operand before the second (if at all).[1]

The traditional expression ordering rules do not require that deciding which function to call must occur before evaluating the parameters.

The traditional dependency chart looks like this:[2]

`test`	
↓	
`operator->`	
↓	
`harness`	
↓	
`operator->`	`test2 = std::move(test)`
↓	↓
`callAndReport`	lambda constructed
↓	↓
function call	

Since there is no dependency between the read of `test` on the left hand side and the modification of `test` on the right hand side, the operations could occur in either order.

And then C++17 happened.

C++17 added additional order-of-evaluation rules beyond the traditional ones: In the below expressions, `a` is evaluated before `b`:

Operation	Description
`a(b)`	Function call
`a[b]`	Subscript operator

`a.*b`
`a->*b`
	Pointer to member

`a << b`
`a >> b`
	Shifting

`b = a`
`b op= a`
	Assignment
	(note: right to left)

Personally, I find it interesting that the standard chose to evaluate the function before the arguments. In practice, it is more convenient to calculate the arguments first if the function is identified via pointer-chasing, since that can be done without disturbing many registers.

Anyway, since function calls now evaluate the function before arguments, the order of evaluation starting in C++17 now requires that the `test` on the left hand side be evaluated before the `test2 = std::move(test)` in the lambda.

So the question came down to what version of the language the customer is compiling with.

The customer came back and said that they were using Visual Studio 2019, but in C++14 mode.

So that explains it.

Next time, we'll look at potential fixes (beyond "upgrade to C++17").

[1] The compiler's freedom to evaluate function arguments in any order leads to a case where the underlying architecture influences the order of operations. Stack-based parameters are more likely to be evaluated before register-based parameters: Once you evaluate the value of a register-based parameter, you have to find a place to keep it while you evaluate the other parameters. You can try to keep it in the register that will be used to pass the parameter (good), or you can try to keep it in another register temporarily (okay), or you can spill it and reload it just before the call (bad). If the other parameters are complicated to calculate, you may be forced to spill. On the other hand, a stack-based parameter is going to be spilled to the stack *anyway*, so you may as well just calculate it and spill it, and you're done. You don't have to burn a register to hold the parameter until the call.

This means that even if the only thing you take into account is the calling convention, the optimal order of evaluation can vary between x86-32 (no register parameters, except possibly for `this`), x86-64/arm (four register parameters), and arm64 (eight register parameters).

[2] Even though I showed `harness` as coming after the preceding `operator->` in the traditional ordering, that is not really a rule of the language, but rather an artifact of mental inlining. What really happened is

```
test
```

↓	
`operator->`	`&Test::harness`
↓	↓
`operator->*` (produces `harness`)	

But `&Test::harness` has no dependencies on anything,

Raymond Chen

Follow