# Fixing the crash that seems to be on a std::move operation

devblogs.microsoft.com/oldnewthing/20220121-20

January 21, 2022

Raymond Chen

Last time, we looked at a crash that was root-caused to an order of evaluation bug if compiled as C++14. One solution to the problem is to switch to C++17 mode, but presumably the customer isn't willing to make that drastic a change to their product yet. Maybe there's something we can do that is less disruptive.

```
// std::shared_ptr<Test> test;
test->harness->callAndReport([test2 = std::move(test)]() ...);
```

The problem here is that we move the pointer out of the `test` when building the lambda, while at the same time fetching the pointer from the `test` in order to locate the function to call. In C++14, these two operations are not sequenced, so there is no guarantee on the order of evaluation, even though our code depends on it:

| | |
|---|---|
| `test` | ← need this to happen |
| ↓ | |
| `operator->` | |
| ↓ | |
| `harness` | before this happens |
| ↓ | ↓ |
| `operator->` | `test2 = std::move(test)` |
| ↓ | ↓ |
| `callAndReport` | lambda constructed |
| ↓ | ↓ |
| function call | |

We’ll have to break the large expression (with unspecified order of evaluation) into two expressions that force the desired order of evaluation.

We need the retrieval of the object referenced by `test` to happen before the evaluation of `test2 = std::move(test)`. So let’s write it out explicitly in the order we want the operations to happen, as two separate statements to enforce sequencing.

There are a few ways of doing this, depending on where we want to break the chain in the above diagram.

We could break it as soon as possible:

```
auto& original_test = *test; // get this before we std::move(test)
original_test.harness->callAndReport([test2 = std::move(test)]() ...);
```

An equivalent, perhaps less weird version, is this:

```
auto test_ptr = test.get(); // get this before we std::move(test)
test_ptr->harness->callAndReport([test2 = std::move(test)]() ...);
```

Both of these force the sequencing of the “figure out what `test` points to” to happen before the rest of the expression:

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

Perhaps even less weird-looking is sequencing after the acquisition of the `harness`.

```
    auto& harness = test->harness; // get this before we std::move(test)
    harness->callAndReport([test2 = std::move(test)]() ...);
```

This moves the explicit sequencing a little later in the evaluation of the left column:

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

It doesn't really matter where we put it, as long as it definitely sequences the read of `test` ahead of its modification by move-assignment.

But maybe we're trying too hard.

The problem with the original code was that it was being too clever, using `std::move` to transfer the `std::shared_ptr` into the lambda and avoid bumping and then dropping the reference count. But what did we really save?

Let's look at the entire (repaired) function and evaluate its effect on the `std::shared_ptr`:

```
void polarity_test(std::shared_ptr<Test> test)
{
    auto& harness = test->harness; // get this before we std::move(test)
    harness->callAndReport([test2 = std::move(test)]() mutable
    {
        ...
    });
}
```

The `test` is destructed at the end of the function, and that's unavoidable.

The `test2` is copy-constructed from `test`, and it destructs when the lambda is destroyed.

I checked gcc, clang, MSVC, and icc, and none of them optimize out the `test` destructor, not even in simple cases like this:

```
void simple(std::shared_ptr<int> p)
{
    p.reset();
    // run the destructor now
}
```

For gcc, clang, and icc, the reason is that the call site is responsible for destructing parameters, and the call site doesn't know what the ultimate fate of the `shared_ptr` is. (That changes if the call is inlined, though.)

So all you're really saving by moving the `test` into the lambda is the increment of the reference count. It turns out incrementing the reference count of a `shared_ptr` isn't that bad. It's a null pointer test and an interlocked increment.

This doesn't appear to be a performance-sensitive code path. Indeed, it looks like a test! The simplest solution is probably just to avoid the optimization and copy the `shared_ptr`.

```
    test->harness->callAndReport([test2 = test]() ...);
```

**Reminder**: C++17 strengthened the order of evaluation rules and now requires that for function calls, determining the function to call must be performed before the arguments are evaluated.

Raymond Chen

**Follow**