Why am I getting an access violation trying to access a method on my C++/WinRT object?

devblogs.microsoft.com/oldnewthing/20201030-00

October 30, 2020

Raymond Chen

A customer had a C++/WinRT program that used the `Windows.Devices.PointOfService.ReceiptPrintJob` object to print a receipt on a point-of-service printer. They wanted to skip five lines, so they did the natural thing:

```
receiptPrintJob.FeedPaperByLine(5);
```

This worked great until it didn't.

On one system, the call to `FeedPaperByLine()` crashed with a null pointer exception, even though the `receiptPrintJob` variable was non-null.

Closer investigation revealed that the problem is that the failing system was running an old build of Windows 10.

The `FeedPaperByLine` method was added in Windows 10 Version 1903, but the system in question was running Windows 10 Version 1809.

Windows Runtime objects are represented by a pointer to their default interface. Any methods on nondefault interfaces require first obtaining that nondefault interface, and then calling the method on that other interface.

The `FeedPaperByLine` method is part of the interface `IReceiptPrintJob2`, so calling it goes like this at the ABI layer:

```
IReceiptPrintJob2* job2;
receiptPrintJob->QueryInterface(IID_PPV_ARGS(&job2));
HRESULT hr = job2->FeedPaperByLine(5);
job2->Release();
if (FAILED(hr)) throw hresult_error(hr);
```

If the object doesn't support the `IReceiptPrintJob2` interface, then the `QueryInterface` fails, and the output pointer `job2` is set to `nullptr`. The next line then tries to use the null pointer and crashes.

For performance reasons, the C++/WinRT library intentionally neglects to throw an exception when the `QueryInterface` fails, because that avoids both the code to throw the exception as well as the code to unwind from the exception. It instead relies on the crash on the next line.

The C++/WinRT library considers this an acceptable trade-off because the Windows Runtime metadata says that the interface is indeed supported, so the `QueryInterface` should always succeed.[1]

The customer's problem stemmed from running the program on a version of Windows 10 that was older than the version of the SDK that was used to compile the program. They must have specified their minimum OS version as including an older version of Windows 10 (perhaps by mistake), which puts them into the tricky world of "using a newer SDK but desiring to run on older versions of the operating system." In those cases, you need to probe for features that may not be supported on older systems before trying to use them.

You can do this from the Windows Runtime system by checking for the presence of a method:

```
if (ApiInformation.IsMethodPresent(
      winrt::name_of<ReceiptPrintJob>(),
      L"FeedPaperByLine")) {
  receiptPrintJob.FeedPaperByLine(5);
} else {
  // Print five blank lines.
  // Not as smooth, but it works.
  for (int i = 0; i < 5; i++) {
    receiptPrintJob.PrintLine(L"");
  }
}
```

Or you can do it from the C++/WinRT system by probing for the interface you want:

```
if (auto job2 = receiptPrintJob.try_as<IReceiptPrintJob2>(); job2) {
  job2.FeedPaperByLine(5);
} else {
  // Print five blank lines.
  // Not as smooth, but it works.
  for (int i = 0; i < 5; i++) {
    receiptPrintJob.PrintLine(L"");
  }
}
```

This second technique is faster, but it requires you to know which interface contains the method you are interested in.

[1] It could fail under low memory conditions or if there is a server crash. But if you have low memory or a server crash, you're probably not going to be able to recover from that anyway.

Raymond Chen

Follow