Debugging walkthrough: Diagnosing a __purecall failure

devblogs.microsoft.com/oldnewthing/20101029-00

October 29, 2010

Raymond Chen

Prerequisite: Understanding what __purecall means.

I was asked to help diagnose an issue in which a program managed to stumble into the `__purecall` function.

```
XYZ!_purecall:
00a14509 a100000000      mov     eax,dword ptr ds:[00000000h]
ds:0023:00000000=????????
```

The stack at the point of failure looked like this:

```
XYZ!_purecall
XYZ!CViewFrame::SetFrame+0x14d
XYZ!CViewFrame::SetPresentation+0x355
XYZ!CViewFrame::BeginView+0x1fe
```

The line at `XYZ!CViewFrame::SetFrame` that called the mystic `__purecall` was a simple `AddRef` :

```
  pSomething->AddRef(); // crashes in __purecall
```

From what we know of `__purecall` , this means that somebody called into a virtual method on a derived class after the derived class's destructor has run. Okay, well, let's see if we can find the object in question. Since the method being called is a COM method, the `__stdcall` calling convention applies, which means that the `this` pointer is on the stack.

```
0:023> dd esp+4 l1
0529f76c  06a88d58
```

Using our knowledge of the layout of a COM object, we can navigate through memory to find the vtable.

```
0:023> dps 06a88d58
06a88d58  009b2eac XYZ!CRegistrationSink::`vftable'
06a88d5c  06b20058
06a88d60  00000002
06a88d64  00998930 XYZ!CObjectWithBrush::`vftable'
06a88d68  00000000
06a88d6c  009c9c80 XYZ!CBrowseSite::`vftable'
06a88d70  009c9c70 XYZ!CBrowseSite::`vftable'
06a88d74  00000000
....
0:023> dps 009b2eac
009b2eac  00a14509 XYZ!_purecall // virtual QueryInterface() = 0
009b2eb0  00a14509 XYZ!_purecall // virtual AddRef() = 0
009b2eb4  00a14509 XYZ!_purecall // virtual Release() = 0
009b2eb8  009cb1e4 XYZ!CRegistrationSink::Register
009b2ebc  009b3d2d XYZ!CRegistrationSink::Unregister
```

We see that the object has been destructed down to the `CRegistrationSink` base class, and the attempt to increment its reference count has led us into the abyss of `__purecall`.

But what was this object before it descended into madness?

Well, we know that the object was something derived from `CRegistrationSink`. And the other values in memory tell us that the object most likely also derived from `CObjectWithBrush` and `CBrowseSite`. Just for fun, here's the `CObjectWithBrush` vtable, to confirm that we destructed down to that point:

```
00998930  00a14509 XYZ!_purecall // virtual QueryInterface() = 0
00998934  00a14509 XYZ!_purecall // virtual AddRef() = 0
00998938  00a14509 XYZ!_purecall // virtual Release() = 0
0099893c  0099880d XYZ!CObjectWithBrush::SetBrush
00998940  00a319ee XYZ!CObjectWithBrush::GetBrush
00998944  00a13fd9 XYZ!CObjectWithBrush::`scalar deleting destructor'
```

Ooh, it looks like `CObjectWithBrush` has a virtual destructor. Probably to destroy the brush.

A check of the source code tells us that nobody derives from `CBrowseSite`, so that is almost certainly the original object type.

As a cross-check, we check whether what we have matches the memory layout of a `CBrowseSite`:

```
0:023> dt XYZ!CBrowseSite 06a88d58
   +0x000 __VFN_table : 0x009b2eac
   +0x004 m_prgreg         : 0x06a88d58 Registration
   +0x008 m_creg           : 2
   +0x00c __VFN_table : 0x00998930
   +0x010 m_hbr            : (null)
   +0x014 __VFN_table : 0x009c9c80
   +0x018 __VFN_table : 0x009c9c70
   +0x01c m_cRef           : 0
```

Looks not unreasonable. (Well, aside from the fact that we have a bug…) The object has most likely begun its destruction because its reference count (`_cRef`) went to zero.

At this point, there was enough information to ask the developers responsible for `CViewFrame` and `CBrowseSite` to work out how the `CViewFrame` ended up running around with a pointer to an object that has already been destructed.

Raymond Chen

Follow