Recognizing different types of exception objects that Windows platform libraries can throw

devblogs.microsoft.com/oldnewthing/20200821-00

August 21, 2020

Raymond Chen

Last time, we saw how to dig the exception object out of a `std::exception_ptr`. But what kind of exception objects might there be?

For C++/CX code, you are probably going to get a `Platform::Exception^`.

```
0:007> ?? p
class std::exception_ptr
   +0x000 _Data1           : 0x08a5885c Void
   +0x004 _Data2           : 0x08a58850 Void
```

We learned that the `_Data1` points to an `EXCEPTION_RECORD` [1]

```
0:007> .exr 0x08a5885c
ExceptionAddress: 00000000
   ExceptionCode: e06d7363 (C++ EH exception)
  ExceptionFlags: 00000001
NumberParameters: 3
   Parameter[0]: 19930520
   Parameter[1]: 08a61330
   Parameter[2]: b371fc08
  pExceptionObject: 08a61330
  _s_ThrowInfo    : b371fc08
```

And we noted that `Parameter[1]` is the object that was thrown.

```
0:007> dps 08a61330
08a61330  08a5b088
08a61334  fdfdfdfd
08a61338  ...
```

A hat pointer `^` is just a COM pointer in disguise, so let's dump the pointer to see what's there.

```
0:007> dps 08a5b088 l5
08a5b088  793d0640 vccorlib140d_app!Platform::Exception::`vftable'
08a5b08c  793d065c vccorlib140d_app!Platform::Exception::`vftable'
08a5b090  793d0680 vccorlib140d_app!Platform::Exception::`vftable'
08a5b094  793d06ac vccorlib140d_app!Platform::Exception::`vftable'
08a5b098  00e5c694
```

The vtable tells us that we have a `Platform::Exception`, which, to be fair, isn't particularly surprising.

```
0:007> dt  vccorlib140d_app!Platform::Exception 08a5b088
   +0x000 __VFN_table : 0x793d0640
   +0x004 __VFN_table : 0x793d065c
   +0x008 __VFN_table : 0x793d0680
   +0x00c __VFN_table : 0x793d06ac
   +0x010 __description    : 0x00e5c694 Void
   +0x014 __restrictedErrorString : 0x00e5da54 Void
   +0x018 __restrictedErrorReference : (null)
   +0x01c __capabilitySid  : (null)
   +0x020 __hresult        : 0n-2147483635
   +0x024 __restrictedInfo : 0x08a3843c Void
   +0x028 __throwInfo      : 0x793c6c24 Void
   +0x02c __size           : 0x20
   +0x030 __prepare        : Platform::IntPtr
   +0x034 __abi_reference_count : __abi_FTMWeakRefData
   +0x03c __abi_disposed   : 0
```

Okay, now we have something. The `__hresult` is `0n-2147483635`, which is the debugger's strange way of saying `0x8000000D` which is `E_ILLEGAL_STATE_CHANGE`.

If you're writing in C++/WinRT, then the thing that is thrown will probably be a `winrt::hresult_error`.

Go through the same exercise as before, but this time when we dump the thrown object we get

```
0:007> dps 0942b8e0
0942b8e0  00000000
0942b8e4  aabbccdd
0942b8e8  80004002
0942b8ec  09439b5c
0942b8f0  fdfdfdfd
```

The structure of an `hresult_error` is currently

```
bstr_handle m_debug_reference;
uint32_t m_debug_magic{ 0xAABBCCDD };
hresult m_code;
com_ptr<IRestrictedErrorInfo> m_info;
```

We can match those up to the dump above. The `m_debug_reference` is `nullptr`, the `m_debug_magic` is the expected value of `0xxAABBCCDD` (which is good, because it acts as confirmation that what we have really is an `hresult_error`), the `m_code` is `0x80004002`, which is `E_NOINTERFACE`, and the `m_info` is `0x09439b5c`, whatever that is.

If you use WIL, then you may have a `wil::ResultException`, which looks like this in memory:

```
struct ResultException : std::exception
{
    FailureType type; // 0 means FailureType::Exception
    HRESULT hr;
    long failureId;
    PCWSTR pszMessage;
    DWORD threadId;
    PCSTR pszCode;
    PCSTR pszFunction;
    PCSTR pszFile;
    uint32_t uLineNumber;
    ... more stuff ...
};
```

```
0:000> dps 000001e8a083edb0
000001e8`a083edb0  00007ff7`e2200298 Win32!wil::ResultException::`vftable'
000001e8`a083edb8  00000000`00000000 // std::exception
000001e8`a083edc0  00000000`00000000 // std::exception
000001e8`a083edc8  8007139f`00000000 // hr / type (0 = Exception)
000001e8`a083edd0  cccccccc`00000042 // padding / failureId
000001e8`a083edd8  00000000`00000000 // pszMessage
000001e8`a083ede0  cccccccc`00001de4 // padding / threadId
000001e8`a083ede8  000001e8`a0833a14 // pszCode
000001e8`a083edf0  000001e8`a0833a26 // pszFunction
000001e8`a083edf8  000001e8`a0833a2f // pszFile
000001e8`a083ee00  00000001`0000001e // cFailureCount / uLineNumber
```

From this, we can extract that the `HRESULT` was `0x8007139f` which is `E_NOT_VALID_STATE`, there is no custom message (`pszMessage` is null), and we have pointers to various strings plus a line number.

```
0:000> $ pszCode is the expression that failed
0:000> da 000001e8`a0833a14
000001e8`a0833a14  "TrySomething(a, b, c)"

0:000> $ pszFunction is the function that threw
0:000> da 000001e8`a0833a26
000001e8`a0833a26  "Foo::Bar"

0:000> $ pszFile is the file name
0:000> da 000001e8`a0833a2f
000001e8`a0833a2f  "C:\\Test\\Bar.cpp"
```

So we see that this exception was the result of a

```
THROW_IF_FAILED(TrySomething(a, b, c));
```

that can be found in the function `Foo::Bar` at line 30 (`0x001e`) of the file `C:\Test\Bar.cpp`. It was the `0x42` 'th error encountered by the program.

If you have symbols, you can ask the debugger to print this all nice and pretty for you.

```
0:000> ?? ((Contoso!wil::ResultException*)0x000001e8a083edb0)
class wil::ResultException * 0x000001e8`a083edb0
   +0x000 __VFN_table : 0x00007ff7`c9de0308
   +0x008 _Data            : __std_exception_data
   +0x018 m_failure        : wil::StoredFailureInfo
   +0x0b8 m_what           : wil::details::shared_buffer

0:000> ?? ((Contoso!wil::ResultException*)0x000001e8a083edb0)->m_failure
class wil::StoredFailureInfo
   +0x000 m_failureInfo    : wil::FailureInfo
   +0x090 m_spStrings      : wil::details::shared_buffer

0:000> ?? ((Contoso!wil::ResultException*)0x000001e8a083edb0)-
>m_failure.m_failureInfo
truct wil::FailureInfo
   +0x000 type             : 0 ( Exception )
   +0x004 hr               : 8007139f
   +0x008 failureId        : 0n66
   +0x010 pszMessage       : (null)
   +0x018 threadId         : 0x1de4
   +0x020 pszCode          : 0x000001e8`a0833a14  "TrySomething(a, b, c)"
   +0x028 pszFunction      : 0x000001e8`a0833a26  "Foo::Bar"
   +0x030 pszFile          : 0x000001e8`a0833a2f  "C:\Test\Bar.cpp"
   +0x038 uLineNumber      : 0x1e
   +0x03c cFailureCount    : 0n1
   +0x040 pszCallContext   : (null)
   +0x048 callContextOriginating : wil::CallContextInfo
   +0x060 callContextCurrent : wil::CallContextInfo
   +0x078 pszModule        : 0x000001e8`a0833a38  "Test.exe"
   +0x080 returnAddress    : 0x00007ff7`c9dcf993 Void
   +0x088 callerReturnAddress : 0x00007ff7`c9dd7b65 Void
```

You can see the other stuff in the `FailureInfo` structure, like the module name, and a few return addresses.

Of course, the exception could have come from something else, in which case you'll have to do your own decoding.

```
0:000> dps 0x00000090`5172f8a0 l4
00000090`5172f8a0  00007ffb`22408b98 std::time_put<...>::`vftable'+0x7b8
00000090`5172f8a8  0000026b`841b4b10
00000090`5172f8b0  00000000`00000001
```

This looks like somebody threw a `std::time_put`, but you're just being faked out by COMDAT folding. You can dig into the exception type data to see what it is, or you can just guess at pointers, in the hope that one of them will give you a clue.

Fortunately, many exceptions derive from `std::exception`, and in MSVC, the `std::exception` has a string pointer.

```
0:000> da 0000026b`841b4b10
0000026b`841b4b10  "invalid vector<T> subscript"
```

Aha, so this perticular guy was an invalid subscript exception from `std::vector`, also known as `std::out_of_range`.

Those are the four commonly-thrown exceptions I was planning to cover today. They are the ones that come from the libraries commonly used in Windows client code.[2]

[1] Note that this is an implementation detail that may change at any time. This information is provided for debugging purposes. The fields have opaque names to discourage people from relying on said implementation details. But we're debugging here, so we can deal with the possibility that the fields no longer mean what we think they mean. Debugging is an exercise in optimism.

[2] I left out MFC's `CException`. Sorry, MFC developers.

Raymond Chen

Follow