The early history of redundant function pointer casts: MakeProcInstance

devblogs.microsoft.com/oldnewthing/20180423-00

April 23, 2018

Raymond Chen

If you look through old code, you see a lot of redundant function pointer casts. (If you're writing new code, you should get rid of as many function pointer casts as possible, because a function pointer cast is a bug waiting to happen.) Why does old code have so many redundant function pointer casts?

Because back in the old days, they weren't redundant.

In the days of 16-bit Windows, function prologues were required to take very specific forms in order to make stack walking work, and stack walking was necessary in order to simulate an MMU on a CPU that didn't have one.

Another rule for prologues has to do with state management. The full prologue for a far function looks like this:

```
mov     ax, ds
nop
inc     bp
push    bp
mov     bp, sp
push    ds
mov     ds, ax
```

Before we can dig into those instructions, we need to know a bit about how code segments worked in real-mode 16-bit Windows. In real-mode 16-bit Windows, there was a single address space for all applications because the CPU had no concept of per-process address spaces. The kernel simulated separate address spaces by managing *instances*. The instance (represented by an *instance handle*) specified the location of the data segment the code should operate on. If you have two copies of a program running, the code is shared, but each program has its own data. The instance handle tells you where that data is.

And the instance handle is kept in the `ds` register.

Therefore, it is essential that every function have its `ds` register set to the instance handle that describes where the code should find its data. You can think of it as a "global `this` pointer for the process."

Okay, so let's look at the function prologue again. First, it copies `ds` to `ax` via a two-byte `mov ax, ds` instruction. Then there is a `nop`. This pads the prologue size to three bytes.

The next four instructions build the stack frame: The `inc bp` marks the stack frame as a far frame. The `push bp` and `mov bp, sp` build the `bp` chain. And the `push ds` saves the original `ds` register, which also provides breathing room for return address patching.

And then we move `ax` back into `ds`. The instance handle just took a little tour of the `ax` register and then returned back home. What was the point of that?

Recall that in 16-bit Windows, every far function called from another segment was listed in the module's Entry Table.

When a far function is placed in the exported function table, the loader patches the first three bytes of the function to three `nop` instructions. Non-exported functions remain unchanged. This means that non-exported functions do the redundant `ds` rigamarole. It's a little extra work, but it's ultimately harmless.

The effect of patching out the initial `mov ax, ds` is that the function ends up doing this:

- Build a far stack frame, which includes saving the original `ds`.
- Set `ds` to whatever was passed in the `ax` register.

The second step means that the code, when it executes, operates on the data associated with the handle passed in the `ax` register.

Okay, great, but this means that you can't call an exported function directly, because it will set the `ds` register to whatever value is passed in the `ax` register. Since the `ax` register is not part of the calling convention, its value is garbage.

But that's okay. We made things worse so we can make them better.

The `MakeProcInstance` function creates a stub function that loads the `ax` register with the instance handle you provide, and then jumps to the function you provide. Really. That's all it did. (When you're done, you call `FreeProcInstance` to free the memory back to the system.)

This stub function was known as a *procedure instance thunk*, or a *proc instance* for short. Hence the name `MakeProcInstance`.

Okay, finally the punch line. The `MakeProcInstance` function didn't care what kind of function pointer you passed it. Whatever you passed in, it returned the same kind of pointer back out, because all the stub did was twiddle the `ax` register and then jump to the real function. The parameters on the stack didn't change, the cleanup convention didn't change, nothing else changed.

The `MakeProcInstance` function was declared as returning a `FARPROC`, which is a typedef for a far function that takes no parameters and returns nothing. The parameters and return value are irrelevant; it just had to be *something*.

But what this means is that when you take your function, like a window enumeration callback, and create a procedure instance for it, the thing you get back has been type-erased to a generic function pointer. To make it useful again, you need to cast it back to what it was originally.

For example, if what you passed was a `WNDENUMPROC`, then you need to cast the procedure instance back to a `WNDENUMPROC`. If you passed a `TIMERPROC`, then you need to cast the procedure instance back to a `TIMERPROC`. You could anachronistically express this as

```
template<typename R, typename ...Args>
auto MakeProcInstanceT(R (FAR *func)(Args...), HINSTANCE inst)
{
  return (decltype(func))MakeProcInstance((FARPROC)func, inst);
}
```

Of course, you didn't have this fancy template deduction in 1983-era C, so you had to cast the return value manually.

And that brings us to today. Even though `MakeProcInstance` has been obsolete for decades, some people imprinted on the "gotta cast your function pointers to get them to compile" pattern, either because they wrote code when the cast was required and fell into the habit, or or (more likely) they learned from code that was written by someone who inherited this habit from somebody else. And yes, this inherited folk wisdom can even be found in MSDN.

The redundant function pointer cast is now a type of folklore, passed down from developer to developer, even though it's no longer needed and in fact will mask problems caused by mismatched prototypes.

Raymond Chen

Follow