How do I do an interlocked exchange of a hat pointer?

devblogs.microsoft.com/oldnewthing/20170210-00

February 10, 2017

Raymond Chen

If you work with C++/CX, then you spend a lot of time working with types that wear hats. These are basically super-smart pointers. In addition to performing automatic `AddRef` and `Release`, they also perform automatic `QueryInterface` if you call a method that belongs to an interface other than the default one.

How do you perform an atomic exchange of these things?

The trick is realizing that a hat pointer is the same size as a raw pointer because it's physically represented as a pointer to the default interface of the underlying type. Therefore, you can perform the interlocked operation on the raw pointer, provided of course that the thing you are exchanging in can legally be placed in such a pointer.

Even if a hat pointer weren't the size of a raw pointer, what's important are that (1) it's the size of something that can be atomically exchanged, and (2) it is self-contained, without dependencies on other memory.

```
template<typename T, typename U>
T^ InterlockedExchangeRefPointer(T^* target, U value)
{
  static_assert(sizeof(T^) == sizeof(void*),
    "InterlockedExchangePointer is the wrong size");
  T^ exchange = value;
  void** rawExchange = reinterpret_cast<void**>(&exchange);
  void** rawTarget = reinterpret_cast<void**>(target);
  *rawExchange = static_cast<IInspectable*>(
    InterlockedExchangePointer(rawTarget, *rawExchange));
  return exchange;
}
```

Okay, what's going on here?

First, we verify our assumption: Namely, that a hat pointer is the same size as a raw pointer, because we're about to exchange the contents of the two things, and we need to be sure that we're exchanging the correct number of bytes.

Next, we convert the `value` to a compatible pointer. This allows you to pass anything convertible to `T^` as the second parameter, rather than having to pass something that is exactly a `T^`. If the function had bee prototyped as

```
template<typename T>
T^ InterlockedExchangeRefPointer(T^* target, T^ value)
```

then you would have gotten type inference errors if you pass a second parameter that is not literally a `T^`, but which can be converted to one (for example, `nullptr`) because the compiler can't figure out what `T` should be.

Once we've converted the `value` into a `T^` we can proceed with the raw exchange of contents. The `rawExchange` variable points to the variable `exchange`, but viewing it as a raw pointer rather than a hat pointer. Similarly, the `rawTarget` variable points to the target as a raw pointer.

We then ask `InterlockedExchangePointer` to do the dirty work of exchanging the values. We put the previous value of the target back into `exchange` (via the alias known as `rawExchange`).

Putting the answer back into `exchange` lets us return the smart version of the variable back to our caller.

So that's it. This is really just a fancy way of writing

```
THING InterlockedExchangeThing(Thing* thing, Thing newThing)
{
 newThing = InterlockedExchangeSizeOfThing(thing, newThing);
 return newThing;
}
```

Raymond Chen

Follow