Passing by address versus passing by reference, a puzzle

devblogs.microsoft.com/oldnewthing/20070326-00

March 26, 2007

Raymond Chen

Commenter Mike Petry asked via the Suggestion Box:

> Why can you dereference a COM interface pointer and pass it to a function with a Com interface reference.
>
> The call.
>
> ```
> OutputDebugString(_T("IntfByRef::Execute - Begin\n"));
> BadBoy badone;
> CComPtr<IDoer> Doer;
> Doer.CoCreateInstance(CLSID_Doer, NULL, CLSCTX_INPROC_SERVER);
> // created a raw pointer - maybe the
> // smart pointer was effecting it some how.
> IDoer* Doer2;
> Doer.CopyTo(&Doer2);
> badone.stupid_method(*Doer2);
> Doer2->Release();
> // no still works.
> ```
>
> The function called.
>
> ```
> void stupid_method(IDoer& IDoerRef)
> {
>  IDoerRef.Do();
>  CComQIPtr<IDispatch> WatchIt(&IDoerRef);
>  if( WatchIt )
>   OutputDebugString(_T("QI the address of the ")
>                     _T("ref works - this is weird\n"));
>  else
>   OutputDebugString(_T("At least trying to QI the ")
>                     _T("address of the ref fails\n"));
> }
> ```
>
> I found some code written like this during a code review. It is wrong but it seems to work.

You already know the answer to this question. You merely got distracted by the use of a COM interface. Let me rephrase the question, using an abstract C++ class instead of a COM interface. (The virtualness isn't important to the discussion.) Given this code:

```
class Doer {
 public: virtual void Do() = 0;
};
void caller(Doer *p)
{
 stupid_method(*p);
}
void stupid_method(Doer& ref)
{
 ref.Do();
}
```

How is this different from the pointer version?

```
void caller2(Doer *p)
{
 stupid_method2(p);
}
void stupid_method2(Doer *p)
{
 p->Do();
}
```

The answer: From the compiler's point of view, it's the same. I could prove this by going into what references mean, but you'd just find that boring, but instead I'll show you the generated code. First, the version that passes by reference:

```
; void caller(Doer *p) { stupid_method(*p); }
  00000 55               push    ebp
  00001 8b ec            mov     ebp, esp
  00003 ff 75 08         push    DWORD PTR _p$[ebp]
  00006 e8 00 00 00 00   call    stupid_method
  0000b 5d               pop     ebp
  0000c c2 04 00         ret     4
; void stupid_method(Doer& ref) { ref.Do(); }
  00000 55               push    ebp
  00001 8b ec            mov     ebp, esp
  00003 8b 4d 08         mov     ecx, DWORD PTR _ref$[ebp]
  00006 8b 01            mov     eax, DWORD PTR [ecx]
  00008 ff 10            call    DWORD PTR [eax]
  0000a 5d               pop     ebp
  0000b c2 04 00         ret     4
```

Now the version that passes by address:

```
; void caller2(Doer *p) { stupid_method2(p); }
  00000 55               push    ebp
  00001 8b ec            mov     ebp, esp
  00003 ff 75 08         push    DWORD PTR _p$[ebp]
  00006 e8 00 00 00 00   call    stupid_method2
  0000b 5d               pop     ebp
  0000c c2 04 00         ret     4
; void stupid_method2(Doer *p) { p->Do(); }
  00000 55               push    ebp
  00001 8b ec            mov     ebp, esp
  00003 8b 4d 08         mov     ecx, DWORD PTR _p$[ebp]
  00006 8b 01            mov     eax, DWORD PTR [ecx]
  00008 ff 10            call    DWORD PTR [eax]
  0000a 5d               pop     ebp
  0000b c2 04 00         ret     4
```

Notice that the code generation is identical.

If you're still baffled, go ask your local C++ expert.

Mind you, dereferencing an abstract object is highly unusual and will probably cause the people who read your code to scratch their heads, but it is nevertheless technically legal, in the same way it is technically legal to give a function that deletes an item the name `add_item`.

Raymond Chen

Follow