Pointers to member functions are very strange animals

devblogs.microsoft.com/oldnewthing/20040209-00

February 9, 2004

Raymond Chen

Pointers to member functions are very strange animals.

Warning: The discussion that follows is specific to the way pointers to member functions are implemented by the Microsoft Visual C++ compiler. Other compilers may do things differently.

Well, okay, if you only use single inheritance, then pointers to member functions are just a pointer to the start of the function, since all the base classes share the same "this" pointer:

```
class Simple { int s; void SimpleMethod(); };
class Simple2 : public Simple
  { int s2; void Simple2Method(); };
class Simple3 : public Simple2
  { int s3; Simple3Method(); };
```

```
p  →  Simple::s
      Simple2::s2

      Simple3::s3
```

Since they all use the same "this" pointer (p), a pointer to a member function of Base can be used as if it were a pointer to a member function of Derived2 without any adjustment necessary.

> The size of a pointer-to-member-function of a class that uses only single inheritance is just the size of a pointer.

But if you have multiple base classes, then things get interesting.

```
class Base1 { int b1; void Base1Method(); };
class Base2 { int b2; void Base2Method(); };
class Derived : public Base1, Base2
  { int d; void DerivedMethod(); };
```

p	→	Base1::b1
q	→	Base2::b2 Derived::d

There are now two possible “this” pointers. The first (p) is used by both Derived and Base1, but the second (q) is used by Base2.

A pointer to a member function of Base1 can be used as a pointer to a member function of Derived, since they both use the same “this” pointer. But a pointer to a member function of Base2 cannot be used as-is as a pointer to a member function of Derived, since the “this” pointer needs to be adjusted.

There are many ways of solving this. Here’s how the Visual Studio compiler decides to handle it:

A pointer to a member function of a multiply-inherited class is really a structure.

Address of function
Adjustor

> The size of a pointer-to-member-function of a class that uses multiple inheritance is the size of a pointer plus the size of a size_t.

Compare this to the case of a class that uses only single inheritance.

> The size of a pointer-to-member-function can change depending on the class!

Aside: Sadly, this means that Rich Hickey’s wonderful technique of Callbacks in C++ Using Template Functors cannot be used as-is. You have to fix the place where he writes the comment

```
// Note: this code depends on all ptr-to-mem-funcs being same size
```

Okay, back to our story.

To call through a pointer to a member function, the “this” pointer is adjusted by the Adjustor, and then the function provided is called. A call through a function pointer might be compiled like this:

```
void (Derived::*pfn)();
Derived d;
(d.*pfn)();
  lea  ecx, d       ; ecx = "this"
  add  ecx, pfn[4]  ; add adjustor
  call pfn[0]       ; call
```

When would an adjustor be nonzero? Consider the case above. The function Derived::Base2Method() is really Base2::Base2Method() and therefore expects to receive “q” as its “this” pointer. In order to convert a “p” to a “q”, the adjustor must have the value sizeof(Base1), so that when the first line of Base2::Base2Method() executes, it receives the expected “q” as its “this” pointer.

“But why not just use a thunk instead of manually adding the adjustor?” In other words, why not just use a simple pointer to a thunk that goes like this:

```
Derived::Base2Method thunk:
    add ecx, sizeof(Base1)  ; convert "p" to "q"
    jmp Base2::Base2Method  ; continue
```

and use that thunk as the function pointer?

The reason: Function pointer casts.

Consider the following code:

```
void (Base2::*pfnBase2)();
void (Derived::*pfnDerived)();
pfnDerived = pfnBase2;
  mov  ecx, pfnBase2            ; ecx = address
  mov  pfnDerived[0], ecx
  mov  pfnDerived[4], sizeof(Base1) ; adjustor!
```

We start with a pointer to a member function of Base2, which is a class that uses only single inheritance, so it consists of just a pointer to the code. To assign it to a pointer to a member function of Derived, which uses multiple inheritance, we can re-use the function address, but we now need an adjustor so that the pointer “p” can properly be converted to a “q”.

Notice that the code doesn’t know what function pfnBase2 points to, so it can’t just replace it with the matching thunk. It would have to generate a thunk at runtime and somehow use its psychic powers to decide when the memory can safely be freed. (This is C++. No garbage collector here.)

Notice also that when pfnBase2 got cast to a pointer to member function of Derived, its size changed, since it went from a pointer to a function in a class that uses only single inheritance to a pointer to a function in a class that uses multiple inheritance.

> Casting a function pointer can change its size!

I bet that you didn't know that before reading this entry.

There's still an awful lot more to this topic, but I'm going to stop here before everybody's head explodes.

Exercise: Consider the class

```
class Base3 { int b3; void Base3Method(); };
class Derived2 : public Base3, public Derived { };
```

How would the following code be compiled?

```
void (Derived::*pfnDerived)();
void (Derived2::*pfnDerived2();
pfnDerived2 = pfnDerived;
```

Answer to appear tomorrow.

Raymond Chen

Follow