Deconstructing function pointers in a C++ template, the calling convention conundrum

devblogs.microsoft.com/oldnewthing/20200716-00

July 16, 2020

Raymond Chen

We continue our attempt to build a traits class for function pointers by incorporating another attribute of function pointers: The calling convention.

Calling conventions are not formally part of the standard, but they are a common extension, supported by the Microsoft compiler, gcc, clang, and icc. (Possibly others; I didn't research it too deeply.)

None of the compilers can deduce the calling convention, so we'll have to write things out multiple times, just like we did for `noexcept`.

Let's start by defining our own macros for calling conventions, since the compilers express them differently. And for simplicity, let's start with just two of the available calling conventions: `cdecl` and `stdcall`. There are also `fastcall` and `vectorcall`, but we'll deal with those analogously.

```
#if defined(__GNUC__) || defined(__clang__)
  #define CC_CDECL __attribute__((cdecl))
  #define CC_STDCALL __attribute__((stdcall))
#elif defined(_MSC_VER) || defined(__INTEL_COMPILER)
  #define CC_CDECL __cdecl
  #define CC_STDCALL __stdcall
#else
  #define CC_CDECL
  #define CC_STDCALL
#endif
```

The order of the tests is significant because the Intel compiler sets both `__INTEL_COMPILER` and `__GNUC__` when running in gcc mode, and in that case, we want to use the gcc-style attributes.

I'll also create a type for each calling convention. I'll have each calling convention be represented by a pointer to a simple function that uses the indicated calling convention. This'll come in handy later.

```
struct CallingConventions
{
    using Cdecl = void(CC_CDECL*)();
    using Stdcall = void(CC_STDCALL*)();
};
```

And now to create all the partial specializations. We have two calling conventions, times two `noexcept` flavors, times two variadic-ness-es, for a total of eight partial specializations. This is starting to get annoying, especially since there are plenty more calling conventions to come. But we're still trying to figure out what we're doing, so let's look just at the case of a non-variadic non-`noexcept` function, in varying calling conventions.

```
template<typename R, typename... Args>
struct FunctionTraits<R(CC_CDECL*)(Args...)>
    : FunctionTraitsBase<R, Args...>
{
  using Pointer = R(CC_CDECL*)(Args...);
  using CallingConvention = CallingConventions::Cdecl;
};

template<typename R, typename... Args>
struct FunctionTraits<R(CC_STDCALL*)(Args...)>
    : FunctionTraitsBase<R, Args...>
{
  using Pointer = R(CC_STDCALL*)(Args...);
  using CallingConvention = CallingConventions::Stdcall;
};
```

That wasn't so hard, albeit repetitive. (Especially when we add the other three variations.) Now we can deduce the calling convention from a function pointer.

Well, except that this fails to compile on architectures where some of the calling conventions end up the same. For example, even though the four calling conventions are distinct on x86-32, they all collapse into one calling convention on x86-64. If you compile the above code on x86-64, you get errors because the compiler sees duplicate definitions.

Fixing this will require quite a bit more work, which we'll look at next time.

Raymond Chen