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

devblogs.microsoft.com/oldnewthing/20200720-00

July 20, 2020

Raymond Chen

Last time, we tried to create a partial specialization for a template type that has a function pointer template parameter with a specific calling convention, but have the partial specialization disappear if the calling convention is identical to `cdecl`, so that the compiler won't complain about a duplicate definition.

I couldn't make the partial specialization disappear, but I could try the next best thing: Move it to somewhere harmless.

I can give the `FunctionTraits` a second template parameter. If the calling conventions are the same, then I use a different second template parameter so that the partial specializations end up different. But if the calling conventions are different, then I use the same second template parameter, so that the function signature is the thing that makes the partial specializations different.

```
template<typename F, int = 0>
struct FunctionTraits;

template<typename R, typename... Args>
struct FunctionTraits<R(CC_CDECL*)(Args...), 0>
    : FunctionTraitsBase<R, Args...>
{
  using Pointer = R(CC_CDECL*)(Args...);
  using CallingConvention = CallingConventions::Cdecl;
};

template<typename R, typename... Args>
struct FunctionTraits<R(CC_STDCALL*)(Args...),
    std::is_same_v<
        CallingConventions::Cdecl,
        CallingConventions::Stdcall> ? 1 : 0>
    : FunctionTraitsBase<R, Args...>
{
  using Pointer = R(CC_STDCALL*)(Args...);
  using CallingConvention = CallingConventions::Stdcall;
};
```

If `cdecl` and `stdcall` are different, then the two partial specializations are

```
struct FunctionTraits<R(CC_CDECL*)(Args...), 0>
```

and

```
struct FunctionTraits<R(CC_STDCALL*)(Args...), 0>
```

Since `cdecl` and `stdcall` are different, the two function pointer types are not the same, and we therefore are defining two different partial specializations.

On the other hand, if `cdecl` and `stdcall` are the same, then the two partial specializations come out to

```
struct FunctionTraits<R(*)(Args...), 0>
```

and

```
struct FunctionTraits<R(*)(Args...), 1>
```

Since `cdecl` and `stdcall` are the same, the two function pointer types are identical, but the integers are different, so we are still defining two different partial specializations.

Since the default value for the second template parameter is the integer `0`, the `, 1` partial specialization will never be used. But that's okay, because it's identical to the `, 0` specialization anyway.

Repeat for the other calling conventions, using unique integers for each, and also repeat for the `noexcept` and variadic versions. This means we write out the same partial specialization a total of twenty times, each slightly different. We'll use a macro to generate them.

```
#if defined(__GNUC__) || defined(__clang__)
  #define CC_CDECL __attribute__((cdecl))
  #define CC_STDCALL __attribute__((stdcall))
  #define CC_FASTCALL __attribute__((fastcall))
  #define CC_VECTORCALL __attribute__((vectorcall))
  #if defined(__INTEL_COMPILER)
    #define CC_REGCALL __attribute__((regcall))
  #else
    #define CC_REGCALL CC_CDECL
  #endif
#elif defined(_MSC_VER) || defined(__INTEL_COMPILER)
  #define CC_CDECL __cdecl
  #define CC_STDCALL __stdcall
  #define CC_FASTCALL __fastcall
  #define CC_VECTORCALL __vectorcall
  #if defined(__INTEL_COMPILER)
    #define CC_REGCALL __regcall
  #else
    #define CC_REGCALL CC_CDECL
  #endif
#else
  #define CC_CDECL
  #define CC_STDCALL
  #define CC_FASTCALL
  #define CC_VECTORCALL
  #define CC_REGCALL
#endif

struct CallingConventions
{
    using Cdecl = void(CC_CDECL*)();
    using Stdcall = void(CC_STDCALL*)();
    using Fastcall = void(CC_FASTCALL*)();
    using Vectorcall = void(CC_VECTORCALL*)();
    using Regcall = void(CC_REGCALL*)();
};

template<typename R, typename... Args>
struct FunctionTraitsBase
{
   using RetType = R;
   using ArgTypes = std::tuple<Args...>;
   static constexpr std::size_t ArgCount = sizeof...(Args);
   template<std::size_t N>
   using NthArg = std::tuple_element_t<N, ArgTypes>;
};

template<typename F, int = 0>
struct FunctionTraits;

#define MAKE_TRAITS2(Unique, CC, CCName, Noexcept, ArgList)   \
template<typename R, typename... Args>                        \
struct FunctionTraits<R(CC*)ArgList noexcept(Noexcept),       \
      std::is_same_v<CallingConventions::Cdecl,               \
                     CallingConventions::CCName>              \
                     ? Unique : 0>                            \
    : FunctionTraitsBase<R, Args...>                          \
{                                                             \
  using Pointer = R(CC*)ArgList noexcept(Noexcept);           \
  using CallingConvention = CallingConventions::CCName;       \
  constexpr static bool IsNoexcept = Noexcept;                \
  constexpr static bool IsVariadic =                          \
    !std::is_same_v<void(*)ArgList, void(*)(Args...)>;        \
}

#define MAKE_TRAITS(Unique, CC, CCName) \
        MAKE_TRAITS2(Unique, CC, CCName, true, (Args...)); \
        MAKE_TRAITS2(Unique, CC, CCName, true, (Args..., ...)); \
        MAKE_TRAITS2(Unique, CC, CCName, false, (Args...)); \
        MAKE_TRAITS2(Unique, CC, CCName, false, (Args..., ...))

MAKE_TRAITS(0, CC_CDECL, Cdecl);
MAKE_TRAITS(1, CC_STDCALL, Stdcall);
MAKE_TRAITS(2, CC_FASTCALL, Fastcall);
MAKE_TRAITS(3, CC_VECTORCALL, Vectorcall);
MAKE_TRAITS(4, CC_REGCALL, Regcall);

#undef MAKE_TRAITS
#undef MAKE_TRAITS2
```

Only the Intel compiler supports `regcall`, so for the other compilers, I treat it like `cdecl`, which lets the existing machinery kick in and remove `regcall` from the active partial specializations.

We pull a sneaky trick here by assigning the unique value `0` to `cdecl`: The value of `0` is the "Please use it" value. The macro recognizes that `Cdecl` is the same as `cdecl` and switches to the uniquifier to render the specialization moot, but the provided uniquifier of zero is in fact the value that makes the specialization active.

But wait, this isn't the only way to solve the problem. Next time, I'll rewind and try a different direction.

Raymond Chen

Follow