Deconstructing function pointers in a C++ template, returning to enable_if

devblogs.microsoft.com/oldnewthing/20200721-00

July 21, 2020

Raymond Chen

Last time, we finally figured out a way to get partial specializations of function pointer traits to work with various calling conventions, even on systems where the calling conventions end up collapsing into the same underlying convention. The solution involved redirecting the unwanted duplicates to a non-default template parameter.

I noted that our second attempt at `enable_if` didn't work because the parameter to `enable_if` was always false, and therefore the result was always invalid. But what if we could make it a dependent type, so that the fact that it wasn't invalid wasn't immediately recognized?

We saw this before, when we wanted to create a type-dependent expression that is always false, and that led to our invention of `unconditional_t` and `unconditional_v`. We can dust those off and use them here.

```
template<typename F, typename = void>
struct FunctionTraits;

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...),
    std::enable_if_t<
      unconditional_v<bool,
        !std::is_same_v<
          CallingConventions::Cdecl,
          CallingConventions::Stdcall>,
        R>>
    >
    : FunctionTraitsBase<R, Args...>
{
  using Pointer = R(CC_STDCALL*)(Args...);
  using CallingConvention = CallingConventions::Stdcall;
};
```

This time, the use of `enable_if` works because there is a substitution going on: The substitution of `R` into `unconditional_v`. Now, it turns out that the substitution is meaningless, but the SFINAE rule doesn't care about meaningfulness.

Which leads us to the following version. I removed the dependency on `unconditional_v` by using the `sizeof(std::decay_t<T>*)` trick.

```
#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, typename = void>
struct FunctionTraits;

#define MAKE_TRAITS2(CC, CCName, Noexcept, ArgList)     \
template<typename R, typename... Args>                 \
struct FunctionTraits<R(CC*)ArgList noexcept(Noexcept), \
      std::enable_if_t<sizeof(std::decay_t<R>*) &&      \
        !std::is_same_v<CallingConventions::Cdecl,      \
                        CallingConventions::CCName>>>   \
    : 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(CC, CCName) \
        MAKE_TRAITS2(CC, CCName, true, (Args...)); \
        MAKE_TRAITS2(CC, CCName, true, (Args..., ...)); \
        MAKE_TRAITS2(CC, CCName, false, (Args...)); \
        MAKE_TRAITS2(CC, CCName, false, (Args..., ...))

MAKE_TRAITS(CC_CDECL, Cdecl);
MAKE_TRAITS(CC_STDCALL, Stdcall);
MAKE_TRAITS(CC_FASTCALL, Fastcall);
MAKE_TRAITS(CC_VECTORCALL, Vectorcall);
MAKE_TRAITS(CC_REGCALL, Regcall);

#undef MAKE_TRAITS
#undef MAKE_TRAITS2
```

Oh this comes so very close, but we do get redefinition errors if three or more calling conventions collapse. That's because the second and third ones generate identical partial specializations. The partial specializations are both ineffective because the `enable_if` fails, but the compiler doesn't know that.

There's another problem because this code sees that `Cdecl` is the same as itself, so it removes the partial specialization for `Cdecl`, even though we want that one to hang around.

We can rescue this by reintroducing the uniquifier, so that each `enable_if` is different and therefore doesn't get counted as a duplicate. We also use the uniquifier to identify which expansion is for `cdecl` so we keep that one enabled.

```
#define MAKE_TRAITS2(U, CC, CCName, Noexcept, ArgList)  \
template<typename R, typename... Args>                  \
struct FunctionTraits<R(CC*)ArgList noexcept(Noexcept), \
      std::enable_if_t<sizeof(R*) &&                    \
        (U == 0 ||                                      \
         !std::is_same_v<CallingConventions::Cdecl,     \
                        CallingConventions::CCName>)>>  \
    : 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(U, CC, CCName) \
        MAKE_TRAITS2(U, CC, CCName, true, (Args...)); \
        MAKE_TRAITS2(U, CC, CCName, true, (Args..., ...)); \
        MAKE_TRAITS2(U, CC, CCName, false, (Args...)); \
        MAKE_TRAITS2(U, 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);
```

Phew, that was quite an ordeal. Trust me, this'll come in handy someday. I just had to spend a bunch of time laying groundwork. The payoff will come some time later.

Bonus chatter: Even if you're not interested in function pointer traits, you can use this technique for other scenarios where you want to specialize for a variety of types, some of which might be the same.

Raymond Chen

Follow