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

devblogs.microsoft.com/oldnewthing/20200717-00

July 17, 2020

Raymond Chen

Last time, we tried to extend our traits class for function pointers so that it accept functions in all different calling conventions. This worked out great, except that on some architectures, many of the calling conventions are actually identical, and this causes our various partial specializations to result in duplicate definitions, and the compiler gets mad, and everything falls apart.

What we want to do is remove the partial specializations for calling conventions which are identical to cdecl. (In practice, it is just cdecl that the other calling conventions collapse into.)

We start by detecting whether a particular specialization should be enabled.

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

    template<typename T>
    static constexpr bool IsCdecl = std::is_same_v<Cdecl, T>;

    static constexpr bool HasStdcall = !IsCdecl<Stdcall>;
};
```

We check whether various calling conventions are present by seeing if a function declared with that convention is the same as a function declared with `cdecl`. (This is why I made the various calling conventions be represented by a function pointer type.)

The usual way to remove things is to use `std::enable_if`, so let's try that. Enable the specialization if `cdecl` and `stdcall` are different.

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

This fails to compile. First we get “default template arguments may not be used in partial specialization” on the `typename = std::enable_if_t<...>`. This error message is self-explanatory.

And then we get “redefinition of `FunctionTraits<R(*)(Args...)>`.” The problem is that in the case where `cdecl` and `stdcall` are the same, the partial specializations both try to define `FunctionTraits<R(*)(Args...)>`, so the compiler doesn’t know which one to choose.

Okay, so let’s put the `enable_if` in the second part.

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

This fails with "template argument 2 (`std::enable_if_t<...>`) is invalid." This is a fair complaint, because it is indeed invalid. SFINAE does not apply here because there is no substitution going on. We flat-out passed an invalid type as a template parameter, and that's an error.

But this gave me an idea. The problem with the first attempt is that we ended up with two partial specializations that ended up being the same thing. The problem with the second attempt is that we tried to create a bad partial specialization.

But maybe I can combine both bad ideas!

We'll try that next time.

Raymond Chen

Follow