Deconstructing function pointers in a C++ template, the noexcept complication

devblogs.microsoft.com/oldnewthing/20200714-00

July 14, 2020

Raymond Chen

Last time, we put together a little traits class to decompose a function pointer into its components. But one thing missing from our class is the `noexcept` qualifier.

For the remainder of the discussion, I've removed the `FirstArg` and `LastArg` type aliases, since I came to the conclusion that they aren't really needed. What's left is this:

```
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> struct FunctionTraits;

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

But it falls apart when we give it a `noexcept` function pointer. (Note that `noexcept` did not become part of the function pointer type until C++17.)

```
void f()
{
  using T = int(*)() noexcept;
  using R = FunctionTraits<T>::RetType; // error
}
```

There is no match for `T` because none of our specializations support `noexcept` function pointers.

So let’s add `noexcept` to our signatures. Let’s try this version, which takes advantage of the fact that `noexcept` takes a Boolean parameter that says whether the `noexcept` applies. Saying `noexcept` with no parameters is shorthand for `noexcept(true)`, and omitting `noexcept` is the same as `noexcept(false)`.

```
template<typename R, typename... Args, bool Nonthrowing>
struct FunctionTraits<R(*)(Args...) noexcept(Nonthrowing)>
    : FunctionTraitsBase<R, Args...>
{
  using Pointer = R(*)(Args...) noexcept(Nonthrowing);
  static constexpr bool IsNoexcept = Nonthrowing;
};
```

The Microsoft compiler doesn’t like it:

```
// MSVC
error C2057: expecting constant expression
    struct FunctionTraits<R(*)(Args...) noexcept(Nonthrowing)>
                                                 ^^^^^^^^^^^
error C27027: 'Nonthrowing': template parameter not used or deducible
```

icc also doesn’t like it, but for a different reason: It’s perfectly happy to match the partial specialization to a non-`noexcept` function, but thinks it doesn’t apply to a `noexcept` function.

```
    // icc is okay with this
    using Test1 = FunctionTraits<int(*)(float) noexcept>;

    // but not this. "error: incomplete type is not allowed"
    using Test2 = FunctionTraits<int(*)(float) noexcept>;
```

On the other hand, gcc and clang are okay with it and deduce `Nonthrowing` appropriately. I’m not sure who is right. (I didn’t check icc.)

Well that’s a bummer. The parameter to `noexcept` is not deducible by the Microsoft compiler. We’ll just have to add a separate specialization.

```
template<typename R, typename... Args>
struct FunctionTraits<R(*)(Args...)>
    : FunctionTraitsBase<R, Args...>
{
  using Pointer = R(*)(Args...);
  constexpr static bool IsNoexcept = false;
};

template<typename R, typename... Args>
struct FunctionTraits<R(*)(Args...) noexcept>
    : FunctionTraitsBase<R, Args...>
{
  using Pointer = R(*)(Args...);
  constexpr static bool IsNoexcept = true;
};
```

Okay, so that takes care of the `noexcept` wrinkle. We'll look at another attribute next time.

Update: Paragraph [temp.deduct.type]/8 of the C++ specification lists the deducible contexts, and the `noexcept` specifier is not on the list. Therefore, MSVC is correct to reject it, and gcc and clang's behavior are nonstandard extensions. This was tracked as Core Working Group issue number CWG2355, with a vote to revise the standard passing in January 2022 and accepted on May 21, 2022. MSVC implemented the language change in February 2020.

Raymond Chen

Follow