Deconstructing function pointers in a C++ template, vexing variadics

devblogs.microsoft.com/oldnewthing/20200715-00

July 15, 2020

Raymond Chen

Last time, we taught our little traits class about `noexcept` functions. One of many oddball cases in the world of function pointers is that of the variadic function, a classic example of which is `printf`.

Recall that we had 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...);
  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;
};
```

But it falls apart when we give it a function like `printf` because none of our specializations handle that case. Let's fix that.

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

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

We have to repeat this for the `noexcept` version, so things are getting pretty repetitive now that we're up to four flavors of this structure. We saw with `noexcept` that the varieties cannot be deduced via templating, but we'll reduce the repetitiveness with a temporary macro.

```
#define MAKE_TRAITS(Noexcept, Variadic, ...)               \
template<typename R, typename... Args>                     \
struct FunctionTraits<R(*)(__VA_ARGS__) noexcept(Noexcept)> \
    : FunctionTraitsBase<R, Args...>                       \
{                                                          \
  using Pointer = R(*)(__VA_ARGS__) noexcept(Noexcept);    \
  constexpr static bool IsNoexcept = Noexcept;             \
  constexpr static bool IsVariadic = Variadic;             \
}

MAKE_TRAITS(false, false, Args...);
MAKE_TRAITS(false,  true, Args..., ...);
MAKE_TRAITS( true, false, Args...);
MAKE_TRAITS( true,  true, Args..., ...);

#undef MAKE_TRAITS
```

The `MAKE_TRAITS` macro takes three-ish parameters.

- `Noexcept` is `true` to create the `noexcept` version, or `false` to create the regular potentially-throwing version.
- `Variadic` is `true` to mark the result as variadic, or `false` if not.
- The "third" parameter is either `Args...` to create the non-variadic version or `Args..., ...` to create the variadic version.

The treatment of `Noexcept` takes advantage of the optional argument to the `noexcept` specifier, discussed earlier.

The way we interpret the "third" parameter is a workaround for preprocessor limitations.

Passing a comma in a macro parameter is complicated, because it is normally interpreted as a parameter separator. In order to protect it, you need to enclose the comma in parentheses, as we discussed some time ago.

The hack is to use a variadic macro. The remaining parameters are all captured into the pseudo-parameter `__VA_ARGS__`, and you can spit them back out in the macro expansion.

If you fail to pass anything for the variadic parameter, then the result is emptiness, but that creates a problem, because we want to include a comma prior to the `...` if it is present, but omit it if we are generating the non-variadic version. C++20 adds the `__VA_OPT__` pseudo-macro which expands its argument only if the `__VA_ARGS__` is nonempty. I'm trying to stick with C++17 for now, so I can't use that.

Instead, I make the caller pass the prior parameter `Args...` as well, so that the comma is "baked into" the `__VA_ARGS__`.

I can simplify the above macro a bit, by inferring whether the result is variadic by comparing the function pointer against an explicitly non-variadic version.

```
#define MAKE_TRAITS(Noexcept, ...)                          \
template<typename R, typename... Args>                      \
struct FunctionTraits<R(*)(__VA_ARGS__) noexcept(Noexcept)> \
    : FunctionTraitsBase<R, Args...>                        \
{                                                           \
  using Pointer = R(*)(__VA_ARGS__) noexcept(Noexcept);     \
  constexpr static bool IsNoexcept = Noexcept;              \
  constexpr static bool IsVariadic =                        \
    !std::is_same_v<void(*)(__VA_ARGS__), void(*)(Args...)>;\
}

MAKE_TRAITS(false, Args...);
MAKE_TRAITS(false, Args..., ...);
MAKE_TRAITS( true, Args...);
MAKE_TRAITS( true, Args..., ...);

#undef MAKE_TRAITS
```

I could also avoid the whole `__VA_ARGS__` nonsense by requiring that the argument list be parenthesized:

```
#define MAKE_TRAITS(Noexcept, ArgsList)                \
template<typename R, typename... Args>                 \
struct FunctionTraits<R(*)ArgsList noexcept(Noexcept)> \
    : FunctionTraitsBase<R, Args...>                   \
{                                                      \
  using Pointer = R(*)ArgsList noexcept(Noexcept);     \
  constexpr static bool IsNoexcept = Noexcept;         \
  constexpr static bool IsVariadic =                   \
    !std::is_same_v<void(*)ArgsList, void(*)(Args...)>;\
}

MAKE_TRAITS(false, (Args...));
MAKE_TRAITS(false, (Args..., ...));
MAKE_TRAITS( true, (Args...));
MAKE_TRAITS( true, (Args..., ...));

#undef MAKE_TRAITS
```

Are we done?

Nope, there's still more that needs to be done to cover function pointers. We'll look at another complication next time.

In case you were wondering: I'm not talking about abominable functions. Those things are evil, and I'm going to pretend they simply don't exist. There's nothing you can do with them anyway.

Raymond Chen

Follow