Deconstructing function pointers in a C++ template

devblogs.microsoft.com/oldnewthing/20200713-00

July 13, 2020

Raymond Chen

I always forget how to deconstruct a function pointer type in a C++ template, so I'm writing it down.

```
template<typename F> struct FunctionTraits;

template<typename R, typename... Args>
struct FunctionTraits<R(*)(Args...)>
{
    using Pointer = R(*)(Args...);
    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>;
    using FirstArg = NthArg<0>;
    using LastArg = NthArg<ArgCount - 1>;
};
```

We start by saying that `FunctionTraits` is a template class that takes a single type parameter, but don't actually provide any class definition yet. The class definition will come from the specializations.

We then define a specialization that takes a function pointer, where `R` is the return type, and where `Args...` are the argument types. This specialization defines a bunch of stuff.

This gets us started, but there are a bunch of weird things that cause this definition to fail.

One is the case of a function with no parameters. In that case, the `FirstArg` and `LastArg` will not compile, since there are no arguments to be the first or last of. We'll need to specialize for that case.

```
template<typename R>
struct FunctionTraits<R(*)()>
{
    using Pointer = R(*)();
    using RetType = R;
    using ArgTypes = std::tuple<>;
    static constexpr std::size_t ArgCount = 0;
};
```

The specialization for the no-parameter case doesn't define `NthArg`, `FirstArg` or `LastArg`, since those things don't exist.

We can factor the common pieces into a base class to avoid repeating ourselves quite so much.

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

template<typename F> struct FunctionTraits;

template<typename R, typename... Args>
struct FunctionTraits<R(*)(Args...)>
    : FunctionTraitsBase<R, Args...>
{
  using Pointer = R(*)(Args...);
  template<std::size_t N>
  using NthArg = std::tuple_element_t<N, ArgTypes>;
  using FirstArg = NthArg<0>;
  using LastArg = NthArg<ArgCount - 1>;
};

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

Unfortunately, this doesn't work because `ArgCount` is not not recognized as dependent names. We have to give the compiler a little help:[1]

```
template<typename R, typename... Args>
struct FunctionTraits<R(*)(Args...)>
    : FunctionTraitsBase<R, Args...>
{
  using base = FunctionTraitsBase<R, Args...>;
  using Pointer = R(*)(Args...);
  template<std::size_t N>
  using NthArg = std::tuple_element_t<N, ArgTypes>;
  using FirstArg = NthArg<0>;
  using LastArg = NthArg<base::ArgCount - 1>;
};
```

That looks pretty good. But we're not done yet, not by a longshot. Next time, we'll investigate one of the categories of function pointers that this template cannot recognize.

[1] For dependent names that represent members, you can use `this->` to mark them as dependent names if you are using them in an instance member function. But for dependent names that are types, or for use where there is no `this`, there doesn't seem to be an easy way to identify them as dependent.

Raymond Chen

Follow