Deduce Function Type from Callable.

Document #:	DXXXXR0
Date:	2022-01-04
Project:	Programming Language C++
Audience:	LEWG

Reply-to: Bjørn Reese <breese@stofanet.dk>

1 Introduction

This proposal adds a transformation trait to deduce the function type (signature) from a callable.

2 Motivation

2.1 Function Wrapper

Consider a function template that takes a callable object as input, and that uses a pre-existing function wrapper to handle the callable object. The function wrapper typically requires a template parameter with the function type, so we need to map the callable type into a function type.

As an example, suppose we have to implement our own variation of `std::async` that enqueues a callable to have it executed on our own thread pool. We use `std::packaged_task` to wrap the callable. The function type required by `std::packaged_task` cannot be deduced from the callable type using CTAD, so the outline below uses the proposed `function_type`. Some of the implementation has been left as comments.

```
// Exposition-only
template <typename F, typename... Args>
auto my_async(F&& fn, Args&&... args) {
  using signature = function_type_t<F, Args...>;
  packaged_task<signature> task(forward<F>(fn));
  // get future
  // enqueue task to thread pool
  // return future
}
```

FiXme Note: `packaged_task` does not take *const* or *noexcept* qualifiers.

3 Function Type Trait

3.1 `function_type`

Transforms a callable type into a function type.

The purpose of `function_type` is to determine the function type (call signature) of a given callable type, such as a function pointer, a function reference, a member function pointer, a member function reference, or a call operator of a function object or a lambda expression.

Function objects and generic lambda expressions may have overloaded call operators or a template function call operator in which case the user must specify the call operator arguments. In all other cases, the optional argument types are ignored as the callable already has been disambiguated by the user.

Table 1:

Template	Comments
`template<class F, class... Args>` `struct function_type`	The member typedef `type` names the function type of `F`. The optional `Args...` types are used to disambiguate call operators that are overloaded or template functions. These types are ignored when there is no ambiguity. *Mandates:* `F` is a callable type.

```
template<class F, class... Args>
  using function_type_t = typename function_type<F, Args...>::type;
```

Informally, the `function_type<T', Args...>` transformation entails the following. Assume `T` is `remove_cvref_t<T'>`, `FT` is the function type resulting from the transformation, and `R` is the deduced return type.

1. Call operator: Let `FT` be `funtion_type<U, Args...>` if `T` has a matching `R operator()(Args...)` where `U` is the first matching:
   (a) `static_cast<R (T::*)(Args...) const noexcept>(&T::operator())`
   (b) `static_cast<R (T::*)(Args...) noexcept>(&T::operator())`
   (c) `static_cast<R (T::*)(Args...) const>(&T::operator())`
   (d) `static_cast<R (T::*)(Args...)>(&T::operator())`
2. Member function pointer: Let `FT` be `U` if `T` matches `U C::*`.
3. Function pointer: Let `FT` be `funtion_type<U, Args...>` if `T` matches `U*`. where `C` is a class type.
4. Function: Let `FT` be `T` if `is_function_v<T>` is `true`.

Examples of `function_type` are:

Example	*Result*
`function_type_t<void(*)(bool)>`	`void(bool)`
`function_type_t<void(cls::*)(bool)>`	`void(bool)`
`function_type_t<decltype([] (bool) {})>`	`void(bool) const`
`function_type_t<decltype([] (auto) {}), int>`	`void(int) const`

4 Related Work

N3579 [2] proposes an `std::signature` type traits to obtain the function type of a callable object.

5 References

[1] Richard Smith, *Working Draft, Standard for Programming Language C++*
`https://github.com/cplusplus/draft`

[2] Mike Spetus, *A type trait for signatures*
`https://wg21.link/N3579`