Document number:	N4543
Date:	2015–05–26
To:	SC22/WG21 LEWG
Reply to:	David Krauss (david_work at me dot com)
References:	N4159

A polymorphic wrapper for all Callable objects

1. Summary

This proposal describes `unique_function`, a variation of `std::function` supporting non-copyable target objects. Its interface removes the copy constructor, and adds in-place construction of target objects.

2. Motivation

Several factors may prevent copying a function object. It may have a non-copyable member. Other objects may depend on its mutable state or retain references to it. In the latter cases, the copy constructor might not actually be deleted. An event dispatching system, for example, might wish to manage ownership of handler objects via `std::function`. This would require that the user provide copyable objects even though each will always remain unique.

Current workarounds include using `reference_wrapper` as the function target type, trying to pass a unique `std::function` object always by reference or `reference_wrapper`, or defining an always-throwing copy constructor. These sacrifice overhead or user-friendly ownership semantics for artificial copyability.

For example, an event-handler map is trivial to implement if the library is willing to demand that the handlers be copyable. The end result is optimal, but inflexible.

```
std::map< std::string, std::function< void() > > commands;
                         // ^ Want unique_function here.

template< typename ftor >
void install_command( std::string name, ftor && handler ) {
    commands.insert({ std::move( name ),
                      std::forward< ftor >( handler ) });
}
```

Improving the external interface quality by allowing non-copyable types is fairly difficult. Efficiency is also reduced. In particular, we need two parallel type erasures.

```
struct owned_function {
    // Order of these members is significant, and this must remain an aggregate.
    std::function< void() > wrapper;
    std::unique_ptr< void *, void (*)( void * ) > alloc;
};
```

```
std::map< std::string, owned_function > commands;

template< typename ftor, typename ... a >
void install_command( std::string name, a && ... arg ) {
    auto ptr = std::make_unique<ftor>( std::forward< a >( arg ) ... );
    commands.insert( std::make_pair(
        std::move( name ), owned_function {
            std::ref( * ptr.get() ),
            { // unique_ptr constructor arguments
                ptr.release(), // Must call get() before release().
                [] (void *p) { delete static_cast< ftor * >( p ); }
            },
        }
    ) );
}

template< typename ftor >
void install_command( std::string name, ftor && handler ) {
    install_command< std::decay_t< ftor >, ftor && >
        ( std::move( name ), std::forward< ftor >( handler ) );
}
```

Plenty of other solutions exist, perhaps some simpler than this. Arriving at a simple solution is hard, though! The above has non-obvious aspects in overload resolution, order of evaluation, and `unique_ptr` deleter customization. It works around some unimplemented DRs and exposes some other bugs. Many solutions are less flexible or incorporate extraneous functionality such as data structures. None are easy or efficient enough, and certainly none are idiomatic.

3. Proposal

The motivating example painstakingly reimplemented some basic functionality. This functionality is added to `std::function`, yielding `unique_function`.

```
template< class Target >
class any_piecewise_construct_tag {};

template< class >
class unique_function;

template< class Ret, class ... ArgTypes >
class unique_function< Ret( ArgTypes ... ) > {
public:
    // 3.1, Parity with std::function:
    unique_function() noexcept;
    unique_function( unique_function && );
    unique_function( unique_function const & ) = delete;
    unique_function & operator = ( unique_function && );
    unique_function & operator = ( unique_function const & ) = delete;
    // Include operator() and other member function signatures of std::function.
```

```
    // 3.2, Target object transfers
    unique_function( function< Ret( ArgTypes ... ) > && );
    unique_function( function< Ret( ArgTypes ... ) > const & );
    unique_function & operator =
                            ( function< Ret( ArgTypes ... ) > && );
    unique_function & operator =
                       ( function< Ret( ArgTypes ... ) > const & );

    // 3.3, In-place construction:
    template< class F, class ... Args >
    unique_function( any_piecewise_construct_tag< F >, Args && ... );
    template< class A, class F, class ... Args >
    unique_function( allocator_arg_t, A const &,
        any_piecewise_construct_tag< F >, Args && ... );

    template< class F, class A, class ... Args >
    allocate_assign( A const &, Args && ... );
    template< class F, class ... Args >
    emplace_assign( Args && ... );
};

template< class Sig, class Target, class ... Args >
unique_function< Sig >
make_unique_function( Args && ... );

template< class Sig, class Target, class A, class ... Args >
unique_function< Sig >
allocate_unique_function( A const &, Args && ... );
```

A new template-name is introduced, as opposed to a specialization of `function`. There is little benefit to a user template being generic only across `function` specializations. Good generic code is written against an interface (e.g. Callable or availability of `target`), without naming an implementation (e.g. `function`). Existing templates which do hard-code `function` support may not be compatible with `unique_function` anyway.

The name `unique_function` is chosen because it only permits one instance of the target value. The address of `target` remains constant across ownership transfers if it does not implement move construction. These properties offer parity with `unique_ptr`.

3.1. Parity with `std::function`

Aside from the copy constructor and copy assignment operator, deleted for obvious reasons, the new template adopts the interface of `std::function`.

Non-movable target objects are supported; these must be managed by the allocator. They cannot be constructed directly into the wrapper.

3.2. Target object transfers from `std::function`

Initializing or assigning a `unique_function` from a `function` of the same signature initializes the new target object from that of the source wrapper, and does not result in double wrapping.

The reverse operations are impossible, since the target may not be copyable and the `unique_function` wrapper certainly isn't. No change to `function` is needed.

Interoperability may be achieved without allowing `unique_function` to use a copy constructor that may be available, or a throwing move constructor. ODR-use of any practically unused constructor should be forbidden, or at least strongly discouraged to prevent bloat.

3.3. In-place construction

A new tag type `any_piecewise_construct_tag` signals in-place construction and nominates the target type. The intent is that the interface can be replicated in other classes such as `any` and `function`. The name is subject to debate.

For the `allocate_assign` and `emplace_assign` member functions, the templated tag is unnecessary because the target type is supplied as an explicit template argument.

The new constructors are very ugly, but they represent the most efficient interface. Factory functions with terminology borrowed from `shared_ptr` offer more elegance.

4. Future directions

The in-place construction interface should be applicable to `function` and `any` as well as `unique_function`.

It may be useful to have a `unique_any`. Given multi-signature `functions` (pending proposal), since `any` is nearly equivalent to a `function` with an empty overload set, implementation of `unique_any` could be trivial.

Target object transfers from `function` to `any` may also be useful, but they would not be the default behavior. They could be more useful, and the reverse transfer more tractable, if the user could extend the erasure data accompanying the target.

5. Implementation and acknowledgements

Matt Calabrese and Geoffrey Romer independently invented this feature set, and implemented it together with further extensions. They worked to combat bloat and developed the principle of minimizing constructor ODR-use.

I have retrofitted some functionality into the libc++ `function` implementation. There is no particular conceptual difficulty, and `function` became aware of move constructors that it had ignored. It should be noted, though, that libc++ and libstdc++ both still need architectural changes to support C++11 type-erased function allocators. Although this proposal could be taken incrementally, in practice it would likely be implemented within wider-ranging revisions.