Document number:	P0288R0
Date:	2016–02–13
To:	SC22/WG21 LEWG
Reply to:	David Krauss (david_work at me dot com)
Revises:	N4543

A polymorphic wrapper for all Callable objects (rev. 2)

A new template `unique_function` is proposed. It is just like `std::function`, minus the copy constructor and copy assignment operator. This allows it to wrap function objects containing non-copyable resources. It also helps to express the idea of an operation that can only be performed once.

1. Motivation

In the beginning, `boost::function` was designed as a generalization of function pointers — as opposed to a radically broad notion of function values. Function objects were small and stateless. Users with a penchant for adventure and compiler diagnostics could use `boost::bind`, and given a few arguments, it would push `function` into the heap allocation regime. More prudent engineering would call for a manually-defined local class, filtered through `boost::ref` to squash the inefficient value semantics.

Times have changed. Lambda-capture syntax like `[u = std::move(u)](io_response r) {r.send_next(u);}` is not only trendy, but safe and convenient. Functional programming patterns are actually gaining traction, which means that real-world function objects are expected to do whatever other objects do, and to encapsulate whatever might be found in a local scope. Non-copyable objects are not uncommon, and non-copyability is viral.

Separately, since `function` is useful as an interface type, it can delegate resource ownership to a library. Before a library frees a resource, it may still be safely referenced locally. Such cases require a guarantee that the target object used by the library is the original one and not a copy.

Finally, when performance analysis finds that a copying a particular class causes a bottleneck, one may wish to delete its copy constructor, to prevent the problem from returning. Likewise, copy constructors of target objects that are never copied are template bloat.

1.1. Difficulty of workarounds

An event dispatching system, for example, might wish to manage ownership of event handler objects via `std::function`. This would require that the user provide copyable objects even though each will 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.

With `unique_function`

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 ) });
}
```

Without `unique_function`

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.

2. Proposal

A new template `unique_function` is introduced. Its members and their behavior are identical to `std::function`, except:

- Its copy constructor and copy assignment operator are defined as deleted.
- It does not use (nor require the existence of) copy constructors of target objects.
- Zero-overhead converting constructors and assignment operators from the corresponding `std::function` specialization are provided.

There are no changes to `std::function` whatsoever. The new template can be placed in `namespace std` or within `std::experimental`. Let its feature test macro be called `__cpp_lib_[experimental_]unique_function`.

The Fundamentals TS already specifies a class `experimental::function` with polymorphic allocation policies. Its changes are orthogonal to this proposal, but to this author's knowledge no current public implementation exists. This proposal's prototype also implements P0043 *Function wrappers with allocators and `noexcept`*, which generalizes the allocation features of `experimental::function` and thus could be used as the basis for a shipping implementation of it. Bearing `std::function` interoperability in mind, though, direct adoption into the standard is preferable.

2.1. Rationale

This is a minimalistic proposal. Other problems exist in `std::function`, but they are better solved separately.

Introducing a new template

A new primary template is introduced, as opposed to a specialization of `std::function`. 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.

Naming

The name `unique_function` is chosen because it only permits one instance of the target value. Like `unique_ptr`, it does not generate duplicate copies. While it is possible for two function objects to have identical invocation behavior, this does not necessarily contradict uniqueness: Behavioral equivalence is an impossible problem. On the other hand, it is intuitive to think of *resources* managed by e.g. `unique_ptr` as unique. When a reader sees `unique_function`, it may be assumed that it holds, and is, such a resource.

Another possible name is `move_only_function`. This would confusingly refer to the behavior of the wrapper itself as opposed to qualities of the wrapped object. The target may be copyable, or (given in-place construction) non-movable.

When such a utility has been implemented (see §6 *Implementations* below), `unique_function` has been the more popular name.

Interoperability

When `std::function` is converted to `unique_function`, the target is transferred just as if it were a copy- or move-construction. No wrapping overhead will be added when an interface migrates `std::function` parameters to `unique_function`.

No in-place construction

In-place construction has been removed from this proposal since N4543. It may be added as a uniform interface with `variant`, `any`, and other type-erasure facilities.

`const` safety

One known defect of `std::function` is that it offers a const-qualified call operator which invokes the target by a non-const access path. This problem is not addressed by this proposal. It is addressed by P0045R0 §2.1, which is pending revision. The solution is threefold:

1. Introduce a wrapper which performs const access: `function<void() const>`.
2. Add a const-unqualified `operator()` overload to the wrappers which already exist.
3. Deprecate the const-qualified call operator in unqualified wrappers.

Ignoring the first step, the second two steps are already conforming, and do not require any proposal.

Rather than introduce `unique_function` with a soon-to-be deprecated call operator signature, we could simply never provide it in the first place. However, this would render `const unique_function` uncallable, with no user recourse except to switch back to `function` or to use `const_cast`. (For example, a `unique_function` nonstatic member would not be callable from a `const&` reference to a class.)

3. Usage

`unique_function` is available as a solution when `std::function` balks at a non-copyable target. In the broad middle ground of usage where wrappers may be copied but aren't, the choice between `unique_function` and `std::function` comes down to aesthetics.

Interfaces

Non-template interfaces taking function objects are encouraged to accept `unique_function` instead of `std::function`. Like `function`, it should canonically be passed by value. Moving should incur minimal overhead, and there is no potentially expensive copy or heap allocation.

When returning polymorphic function objects to the user, it is still better to use `std::function` if possible, for the sake of flexibility.

Passing by value

`std::ref` (the factory function for `std::reference_wrapper`) is used to non-destructively pass `unique_function` using the by-value convention, for example to the standard Algorithms library, when it will not be retained.

This is also a good practice for `std::function`, as it achieves equivalent behavior without incurring a potentially expensive copy.

4. Standardese

Differences are given relative to the working draft N4567, following the contingency that the class is added to the standard as opposed to a Fundamentals TS edition.

First, adjust [func.wrap] §20.9.12 to broader scope.

> ¶1 This subclause describes ~~a~~ polymorphic wrapper class templates that ~~encapsulates~~ encapsulate arbitrary callable objects.

To avoid extensive text duplication, it is proposed to specify `unique_function` in the same clause as `function`.

> **20.9.12.2 ~~Class template function~~ Call wrapper classes** **[func.wrap.func]**

In [func.wrap.func] §20.9.12.2, add a `unique_function` synopsis after that of `function`.

The exact text as follows may be out of date: It should reflect any adopted changes to `function`. Currently, P0090 proposes to remove the result and argument typedefs, and `noexcept` could be restored to the move constructor and/or `swap`. In the following text block only, comments after declarations are editorial notes, not part of the proposed synopsis. Informative comments should be inserted to match `function`.

```
template<class> class unique_function;

template<class R, class... ArgTypes >
class unique_function<R(ArgTypes...)> {
public:
    typedef R result_type;          // Annotate these four members as per function.
    typedef T1 argument_type;                           // Omit all if P0090 is accepted.
    typedef T1 first_argument_type;
    typedef T2 second_argument_type;

    // construct/copy/destroy:
    unique_function() noexcept;
    unique_function(nullptr_t) noexcept;
    unique_function(const unique_function&) = delete;
    unique_function(unique_function&&);   // noexcept equivalently to function.
    template<class F> unique_function(F);
    template<class A> unique_function(allocator_arg_t, const A&)
        noexcept;
    template<class A> unique_function(allocator_arg_t, const A&,
        nullptr_t) noexcept;
    template<class A> unique_function(allocator_arg_t, const A&,
        unique_function&&);
```

```
    template<class F, class A> unique_function(allocator_arg_t,
        const A&, F);

    unique_function& operator=(const unique_function&) = delete;
    unique_function& operator=(unique_function&&);
    unique_function& operator=(nullptr_t) noexcept;
    template<class F> unique_function& operator=(F&&);
    template<class F> unique_function& operator=(reference_wrapper<F>)
        noexcept;

    ~unique_function();

    // function modifiers:
    void swap(unique_function&) noexcept;

    // function capacity:
    explicit operator bool() const noexcept;

    // function invocation:
    R operator()(ArgTypes...) const;

    // function target access:
    const std::type_info& target_type() const noexcept;
    template<class T>       T* target() noexcept;
    template<class T> const T* target() const noexcept;
};

// Null pointer comparisons:
template <class R, class... ArgTypes>
bool operator==(const unique_function<R(ArgTypes…)>&, nullptr_t)
    noexcept;
template <class R, class... ArgTypes>
bool operator==(nullptr_t, const unique_function<R(ArgTypes...)>&)
    noexcept;
template <class R, class... ArgTypes>
bool operator!=(const unique_function<R(ArgTypes...)>&, nullptr_t)
    noexcept;
template <class R, class... ArgTypes>
bool operator!=(nullptr_t, const unique_function<R(ArgTypes...)>&)
    noexcept;

// specialized algorithms:
template <class R, class... ArgTypes>
void swap(unique_function<R(ArgTypes...)>&,
    unique_function<R(ArgTypes...)>&);   // noexcept equivalently to function.

template<class R, class... ArgTypes, class Alloc>
struct uses_allocator<unique_function<R(ArgTypes...)>, Alloc>
    : true_type { };
```

Adjust the high-level description following the synopsis.

¶1 The `function` ~~class template provides~~ and `unique_function` class templates provide polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary callable objects (20.9.1), given a call signature (20.9.1), allowing functions to be first-class objects.

¶3 ~~The~~ A *polymorphic call wrapper* is a specialization of the `function` or `unique_function` class template. Each such specialization is a call wrapper (20.9.1) whose call signature (20.9.1) is `R(ArgTypes...)`.

Add a paragraph to clarify the present method of description.

¶4 The following clauses describe the templates `function` and `unique_function`. The identifier `PolymorphicCallWrapper` denotes either `function` or `unique_function`. In descriptions of class members, `PolymorphicCallWrapper` refers to the enclosing class.

Adjust the constructor specifications in [func.wrap.func.con] §20.9.12.2.1. Note that a new paragraph is inserted before ¶10.

¶1 When any ~~function~~ polymorphic call wrapper constructor that takes a first argument …

~~`function`~~ `PolymorphicCallWrapper() noexcept;`
`template <class A>` ~~`function`~~ `PolymorphicCallWrapper(allocator_arg_t,`
`    const A& a) noexcept;`

¶2 *Postconditions:* `!*this`.

`function(nullptr_t) noexcept;`
`template <class A>` ~~`function`~~ `PolymorphicCallWrapper(allocator_arg_t,`
`    const A& a, nullptr_t) noexcept;`

¶3 *Postconditions:* `!*this`.

`function(`~~`function`~~ `PolymorphicCallWrapper&& f);`
`template <class A>` ~~`function`~~ `PolymorphicCallWrapper(allocator_arg_t,`
`    const A& a, function&& f);`

¶6 *Effects:* …

`template<class F>` ~~`function`~~ `PolymorphicCallWrapper(F f);`
`template <class F, class A>` ~~`function`~~ `PolymorphicCallWrapper`
`    (allocator_arg_t, const A& a, F f);`

¶7 *Requires:* For `function` constructors, `F` shall be `CopyConstructible`. For `unique_function` constructors, `F` shall be `MoveConstructible`.

¶9.3 — `F` is an instance of ~~the function~~ a polymorphic call wrapper class template, and `!f`.

¶10 Otherwise, if `F` is a polymorphic call wrapper class with template parameters `R` and `Args...`, the target of `*this` is move-constructed from the target of `f`.

~~¶10~~ ¶11 *Throws:* …

~~`function`~~ `PolymorphicCallWrapper& operator=`
`    (`~~`function`~~ `PolymorphicCallWrapper&& f);`

~~¶14~~ ¶15 *Effects:* …

~~`function`~~ `PolymorphicCallWrapper& operator=(nullptr_t) noexcept;`

~~¶16~~ ¶17 *Effects:* …

`template<class F>` ~~`function`~~ `PolymorphicCallWrapper& operator=(F&& f);`

~~¶19~~ ¶20 *Effects:* ~~`function`~~ `PolymorphicCallWrapper(std::forward<F>(f))`
`    .swap(*this);`

`template<class F>` ~~`function`~~ `PolymorphicCallWrapper& operator=`
`    (reference_wrapper<F> f) noexcept;`

~~¶22~~ ¶23 *Effects:* ~~`function`~~ `PolymorphicCallWrapper(f).swap(*this);`

`~`~~`function`~~ `PolymorphicCallWrapper();`

~~¶24~~ ¶25 *Effects:* …

Likewise adjust `swap` in [func.wrap.func.mod].

`void swap(`~~`function`~~ `PolymorphicCallWrapper& other) noexcept;`

¶1 *Effects:* …

Likewise adjust the comparison operators in [func.wrap.func.nullptr].

`template <class R, class... ArgTypes>`
`bool operator==(const` ~~`function`~~ `PolymorphicCallWrapper<R(ArgTypes...)>&`
`    f, nullptr_t) noexcept;`
`template <class R, class... ArgTypes>`
`bool operator==(nullptr_t, const` ~~`function`~~ `PolymorphicCallWrapper`
`    <R(ArgTypes...)>& f) noexcept;`

¶1 *Returns:* `!f`.

`template <class R, class... ArgTypes>`
`bool operator!=(const` ~~`function`~~ `PolymorphicCallWrapper<R(ArgTypes...)>&`
`    f, nullptr_t) noexcept;`
`template <class R, class... ArgTypes>`
`bool operator!=(nullptr_t, const` ~~`function`~~ `PolymorphicCallWrapper`
`    <R(ArgTypes...)>& f) noexcept;`

¶2 *Returns:* `(bool) f`.

And `swap` again in [func.wrap.func.alg].

`template<class R, class... ArgTypes>`
`void swap(`~~`function`~~ `PolymorphicCallWrapper<R(ArgTypes...)>& f1,`
`    `~~`function`~~ `PolymorphicCallWrapper<R(ArgTypes...)>& f2);`

¶1 *Effects:* `f1.swap(f2);`

5. Future directions

Given in-place construction, `unique_function` would support non-movable target objects. This feature was removed since the previous revision, N4543, and it will be proposed again separately.

It may typically be easier to implement SFINAE, not a hard error, when a `std::function` constructor encounters a non-copyable target type. If `function` and `unique_function` obtain their constructors from a common template, `unique_function` cannot evaluate `is_copy_constructible` if that metafunction may instantiate a copy constructor. Let's keep an eye on this issue, but it's not a defect yet.

It is possible, without added overhead, to convert a `unique_function` value to `function`, provided it was initialized by conversion from `function`. This could be implemented as an explicit conversion, with an exception thrown upon failure.

6. Implementations

Matt Calabrese and Geoffrey Romer implemented a `unique_function` together with further extensions. They worked to combat bloat and developed the principle of minimizing constructor ODR-use.

In mid 2014, Agustín "K-ballo" Bergé implemented a `unique_function` within the HPX library.

In early 2015, StackOverflow user "Yakk" implemented a `move_only_function` to answer a question.[1] S/he included support of value categories and `const`-qualification as well.

In mid 2015, I attempted to implement this proposal within the libc++ `function` implementation. Due to difficulties in achieving interoperability of target objects, I gave up and started from scratch.

My *cxx_function* library[2] implements this proposal together with P0042R0 *`std::recover`: undoing type erasure*, P0043R0 *Function wrappers with allocators and `noexcept`*, P0045R0 *Overloaded and qualified `std::function`*, and in-place construction. It adds little compile-time overhead and it outperforms libc++ and libstdc++ at runtime.

In early 2016, the *function2* library[3] by Denis Blank (Naios) likewise implements a `unique_function` together with other enhancements including rvalues and cv-qualifiers.

It is likely that other implementations exist. This idea is ripe for standardization.

---

[1] http://stackoverflow.com/questions/28179817/how-can-i-store-generic-packaged-tasks-in-a-container

[2] https://github.com/potswa/cxx_function

[3] http://naios.github.io/function2/