Document number:	P0045R1
Date:	2017–02–06
To:	SC22/WG21 LEWG
Reply to:	David Krauss (david_work at me dot com)

Qualified `std::function` signatures

`std::function` implements type-erasure of the behavior of the call operator on an object, but it cannot emulate every function signature exactly, resulting in a couple rough edges. The qualified signatures added by this proposal improve `const` safety of the call operator (e.g. `function<int() const>`) and enable one-shot function types (e.g. `function<int() &&>`). This feature also meshes with proposals for non-copyable (P0288R1) and overloaded call wrappers. A quality prototype implementation is provided.[1]

1. Background

Upon reviewing N4159, LEWG resolved (issue 34) to pursue several extensions to `std::function`, including accepting target objects that can only be called as rvalue expressions, e.g. `std::move( target ) ( arg1, arg2 )`.

N4159 also raises the issue of const safety in `function`. For example, this works:

```
struct delay_buffer {
    int saved = 42;
    int operator () ( int i ) { return std::exchange( saved, i ); }
};
                                       // Small object optimization — no heap allocation.
const std::function< int( int ) > f = delay_buffer{};
assert ( f( 1 ) == 42 );
assert ( f( 5 ) == 1 );                         // A const object has changed state.
```

This example is troublesome because the object `f` is truly `const` (it's not only getting accessed that way), and the target object being allocated within it should also be truly `const`.

2. Qualified signatures

The crux of this proposal is to use the template type parameter of `std::function` as the type of its call operator function, including `&`, `&&`, `const`, and `noexcept` qualifiers. When a target object is adopted, it is checked for compatibility ([func.wrap.func] §20.14.12.2/2 [2]), with the additional constraint that any *cv-qualifier-seq* or *ref-qualifier* in the signature is applied to the function object. If no *ref-qualifier* is present, then `&` is used as in the current Lvalue-Callable constraint.

[1] https://github.com/potswa/cxx_function; no relation to CxxFunctionBenchmark by Tongari J.

[2] Citations in "[cross.ref] §1.2/3" format refer to the working draft N4618.

The critical expression used for constructor SFINAE becomes (with a default *ref-qualifier* of `&`):

```
INVOKE(declval<F cv-qualifier-seq_opt ref-qualifier >(), declval<ArgTypes>()..., R)
```

This proposal changes classic specializations such as `std::function<void()>` so that the call operator loses `const` qualification. This renders types such as `const function<void()> &` uncallable. This breakage is remedied by an additional, `[[deprecated]]` call operator with `const` qualification and a `const_cast` inside. The fix applies only to wrappers with unqualified signatures, so e.g. `const function<void()&> &` is not callable.

```
std::function< void() > f = []{};                                       // OK
f = []() mutable {};                                                    // OK
f();                                                                    // OK
auto const & fcr = f;
fcr();                                 // Deprecation warning: Legacy, loss of const safety.

std::function< void() const > fc = []{};                                // OK
auto const & fccr = fc;
fccr();                                                                 // OK
fc = []() mutable {};           // Error: Target does not support the const-qualified signature.
```

2.1. `const` compatibility

The const-unsafe behavior happens when a user passes a `std::function` by `const&` reference instead of by value or forwarding. This must be supported for now, but it should be discouraged by deprecation. Whenever the const-correctness issue is diagnosed, users should be informed of three solutions. Adapted from the documentation of the prototype library:

1. Pass the function by value or forwarding so it is not observed to be const.

   This has always been the canonical usage of `std::function` (and all Callable objects). This fix can be applied per call site, usually without affecting any external interface.

2. Add a const qualifier to the signature. This explicitly solves the problem through greater exposition and tighter constraints. It requires that the target object be callable as `const`.

   This is usually a good idea in any case, for functions that are not stateful. Ideally, `function< void() const >` should represent a function that does the same thing on each call. Having `function< void() > const` means that a call may change some state, but the wrapper doesn't have permission to do so. (This is the crux of the issue.)

   If the target needs to change, but only in a way that doesn't affect its observable behavior, consider using `mutable` instead. Note that lambdas allow `mutable`, but the keyword is somewhat abused: the members of the lambda are not mutable. It only makes the call operator non-const. A class must be explicitly defined with a `mutable` member.

3. Consistently remove `const` from the reference which gets called. Non-const references are the best way to share mutable access to values. More `const` is not necessarily better. Again, this reflects greater exposition and tighter constraints.

If a user *must* use a `const_cast`, do so within the target call handler, not on the `function` wrapper which gets called. This allows the hack to be applied once and for all, and affecting the narrowest set of objects. The validity of `const_cast` depends on whether or not the affected object is truly `const`, and even when the wrapper is, the target may not be.

2.2. Double wrapping

When a value is converted from one polymorphic wrapper type to another, the user may intend to transfer the original target object, or simply to preserve its behavior. Instead, the given wrapper value becomes the target of the new wrapper. The small-object optimization cannot apply in this situation, so the heap must be used. Unintended conversions of this sort potentially cause performance and memory usage issues.

```
void printint( int i ) { std::cout << i << '\n'; }
std::function< void( int ) > fi = &printint;
std::function< void( long ) > fl = fi;              // OK, but double wrapping.
assert ( fl.target_type() == typeid( &printint ) );                  // Fails.
```

Qualified signatures open a new avenue to the problem. The user might initialize a one-shot wrapper from a wrapper with an unqualified signature, or add const qualification where there was none. Interoperation with classic wrapper types is likely to be common.

```
std::function<void(int) &&> one_shot = fi;                // OK, no overhead.
std::function<void(int) const> const_safe = fi;  // Deprecated, but no overhead.
```

Fortunately, double wrapping is unnecessary for the proposed types. The type-erasure object containing the target can be made interoperable with wrapper objects of any function qualifiers, simply by ignoring them. Safety is guaranteed by the converting constructor or assignment operator.

```
fi = one_shot;                          // Error: cannot invoke one_shot as lvalue.
```

Elimination of double wrapping also nicely reduces the number internal target specializations, and helps ensure some commonality between wrapper templates.

2.3. `volatile`

Objects of `volatile`-qualified class type exist, albeit rarely. For example, there are volatile-qualified member overloads in `std::atomic`. This proposal permits such signatures for `function`, serving as an annotation that the target may modify a device register.

Calling through a `volatile` access path invokes no special semantics. There is no volatile-qualified assignment operator so a `volatile` wrapper can never be reassigned.

2.4. `noexcept`

Noexcept-qualified call signatures indicate that invocation must not throw. Since default-constructed wrappers always throw, a wrapper with such a call signature is not default-constructible.

```
std::function<void() noexcept> fn1;                // Error: not default-constructible.
std::function<void() noexcept> fn2 = []{};         // Error: target is not noexcept.
std::function<void() noexcept> fn3 = []() noexcept {};                       // OK.
```

3. Standardese

These changes are relative to N4618. Currently in [func.def] §20.14.1/2, *call signature* constructs a restricted form of function type. Loosen the restrictions, and let it properly be the type.

¶2 A *call signature* is ~~the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.~~ a function type [dcl.fct] which describes the behavior of a function object type [function.objects], by serving as the type of a function call operator [over.call]. A function type whose *parameter-type-list* ends with an ellipsis may not be a call signature. The *call-ref-qualifier* of a call signature is its *ref-qualifier*, if any, or otherwise `&`.

Remove the specified class template partial specialization from the synopsis of [func.wrap.func] §20.14.12.2. Specify the behavior of the primary template instead.

```
template<class Sig> class function ~~; // undefined~~
~~template<class R, class... ArgTypes>~~
~~class function<R(ArgTypes...)>~~ {
public:
```

Subsequently in the same synopsis, modify the declaration of the call operator. Add the deprecated signature as well.

```
// 20.14.12.2.4, function invocation
R operator()(ArgTypes...) ~~const~~ qualifiers;
                    // R, ArgTypes..., and qualifiers are the return type, the parameter-type-list,
                           // and the sequence “cv-qualifier-seq_opt ref-qualifier_opt noexcept-specifier_opt”
                                                      // of the function type Sig, respectively.

[[deprecated]] R operator()(ArgTypes...) const;
                                                  // only if qualifiers is an empty sequence
```

Likewise change the template parameters of the free functions.

```
// 20.14.12.2.6, Null pointer comparisons
template <~~class R, class... ArgTypes~~ typename Sig>
  bool operator==(const function<~~R(ArgTypes...)~~ Sig>&, nullptr_t)
                                                              noexcept;
```

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

// 20.14.12.2.7, specialized algorithms

```
template <~~class R, class... ArgTypes~~ typename Sig>
 void swap(function<~~R(ArgTypes...)~~ Sig>&, function<~~R(ArgTypes...)~~
                                                     Sig>&) noexcept;
```

Change the target object viability test in ¶2 and the wrapper capability description in ¶3 to reflect the change to *call signature*.

¶2 A callable type `F` is ~~*Lvalue*~~ *Callable* for ~~argument types `ArgTypes` and return type `R`~~ a call signature `R(ArgTypes...)` *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *noexcept-specifier*$_{opt}$ if the expression `noexcept(`*`INVOKE`*`(declval<F`~~`&`~~ *cv-qualifier-seq*$_{opt}$ *call-ref-qualifier*`>(), declval<ArgTypes>()..., R))`~~, considered as an unevaluated operand ([expr]),~~ is well formed ~~([func.require])~~, where *call-ref-qualifier* is the call-ref-qualifier of the signature ([func.def]), and if the *noexcept-specifier* is present, the expression evaluates to true.

¶3 ~~The `function` class template~~ A specialization `function<Sig>` is a call wrapper ([func.def]) whose call signature is ~~`R(ArgTypes...)`~~ `Sig`.

Modify [func.wrap.func.con] §20.14.12.2.1 to prohibit constructing null-valued, `noexcept`-qualified wrappers.

`function() noexcept;`

¶1 *Postconditions:* […]

¶? *Remarks:* This constructor shall be defined as deleted if `Sig` is a `noexcept` function type.

`function(nullptr_t) noexcept;`

¶2 *Postconditions:* […]

¶? *Remarks:* This constructor shall be defined as deleted if `Sig` is a `noexcept` function type.

Update the converting constructor specification and modify it to prevent double wrapping.

`template<class F> function(F f);`

¶7 *Requires:* […]

¶8 *Remarks:* This constructor template shall not participate in overload resolution unless `f` is ~~Lvalue~~ Callable ([func.require]) for ~~argument types `ArgTypes...` and return type `R`~~ the call signature `Sig`.

¶9 *Postconditions:* […]

¶? Otherwise, if `F` is a specialization of the `function` class template, and the return and parameter types of its call signature are respectively identical to those of `Sig`, then the target of `*this` is the target of `f` or a move-constructed object of the same type.

¶10 Otherwise, …

Likewise update the converting assignment operator specification.

¶21 *Remarks:* This assignment operator shall not participate in overload resolution unless `decay_t<F>` is ~~Lvalue~~ Callable ([func.require]) for ~~argument types `ArgTypes...` and return type `R`~~ the call signature `Sig`.

Modify [func.wrap.func.inv] §20.14.12.2.4 to reflect the new feature.

`R operator()(ArgTypes...)` ~~`const`~~ *`qualifiers`*`;`

¶? `R`, `ArgTypes...`, and *`qualifiers`* are the return type, the parameter-type-list, and the sequence *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *noexcept-specifier*$_{opt}$ of the call signature `Sig`, respectively.

¶1 *Returns:* *`INVOKE`*`(std::forward<F` *cv-qualifier-seq*$_{opt}$ *call-ref-qualifier* `>(f), std::forward<ArgTypes>(args)..., R)` (20.9.2), where `f` ~~is~~ names the target object (20.9.1) of `*this`, `F` is its type, and *cv-qualifier-seq*$_{opt}$ and *call-ref-qualifier* come from the call signature ([func.def]).

¶2 *Throws:* `bad_function_call` if `!*this`; otherwise, any exception thrown by the wrapped callable object.

`[[deprecated]] R operator()(ArgTypes...) const;`

¶? *Remarks:* This member shall participate in overload resolution only if *`qualifiers`* is an empty sequence. Its use is deprecated, including use in an unevaluated *`INVOKE`* expression while determining that the class is Callable for a different call signature.

¶? [*Note:* This overload allows non-`const` access to the target object of a `const`-qualified `function` object, which may be unsafe ([dcl.type.cv]). Several strategies may help to improve usage of `const`, to avoid calling this overload:

1. Pass the `function` object by value, by forwarding reference, or by non-`const` reference, to avoid forming a reference to `const`-qualified type. Then, each invocation may legitimately modify the target object.

2. Add a `const` qualifier to the call signature `Sig`, to forbid modification of the target object. It may be necessary, in turn to add `const` qualifiers to target class call operators.

   Ideally, qualification like `function<void() const>` should always be used when the same effect is expected over successive invocations, and `function<void()>` should only be used when the object's value or effect may vary.

3. As a last resort, add `const_cast` at the site of the deprecated call. Ensure that the referent object is not `const` ([dcl.type.cv]). — *end note*]

¶? *Returns:* `INVOKE(f, std::forward<ArgTypes>(args)..., R)` ([func.require]), where `f` names the target object ([func.def]) of `*this`.

¶? *Throws:* `bad_function_call` if `!*this`; otherwise, any exception thrown by the wrapped callable object.

Change the free function template signatures in [func.wrap.func.nullptr] §20.14.12.2.6 to avoid decomposing the call signature.

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

¶1 *Returns:* `!f`.

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

¶1 *Returns:* `(bool)f`.

Likewise in [func.wrap.func.alg] §20.14.12.2.7.

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

¶1 *Effects:* As if by: `f1.swap(f2);`

Finally, add a feature test macro, `__cpp_lib_qualified_call_signature`.

4. Future directions

Several problems remain to be potentially addressed in future proposals.

4.1. Unqualified wrappers ignore rvalue target behavior

If and when a reflective facility allows identification of the overload chosen for a given call expression, call wrappers without a ref-qualified call signature may require that the target object behaves the same for `&` and `&&` qualified call signatures.

4.2. Unsafe but useful conversions between specializations

Conversion of `function<void()>` to `function<void() noexcept>` or `function<void() const>` is disabled by SFINAE. The workaround is to manually double-wrap, e.g. through a lambda expression. Such conversions should be defined but `explicit`, following the usual pattern that the inverse of an implicit conversion is allowed through explicit syntax. Double-wrapping would then be avoided.

5. Conclusion

Qualified call wrapper signatures help to support fundamental principles: `const` safety since early C++ and rvalue propagation since C++11. They are overdue.

5.1. Kudos

Kudos to Geoff Romer for pursuing and advocating the overall direction.

5.2. Revision history

P0045R0 — Initial revision, as "Overloaded and qualified `std::function`."

P0045R1 — Remove overloading proposal.
Require the `[[deprecated]]` attribute.
Remove discussion of implementation issues and concurrency.
Add `noexcept` signatures.
Add normative wording.
Update future directions.