Document number: P0045
Date: 2015–09–27
To: SC22/WG21 LEWG
Reply to: David Krauss
(david_work at me dot com)
References: N4159

Overloaded and qualified `std::function`

`std::function` supports exactly one function signature at a time. Multiple, overloaded signatures have many uses: exposing several functionalities of a class, discriminating between access as `const`, non-`const`, rvalue, lvalue, etc. Several third-party libraries address the niche, but the role of `const` is controversial and *ref-qualifiers* receive little attention. This proposal replaces `std::function<R(P...)>` with `std::function<Sig...>`, where `Sig` is any type allowed for member functions. The `const` dilemma is resolved without breakage. A quality prototype implementation is provided.[1]

1. Background

Upon reviewing N4159 last November, 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 )`, and supporting non-copyable target types. This proposal covers the first; for the second please see N4543.

Several libraries in circulation provide polymorphic call wrappers with multiple signatures. There even exists the CxxFunctionBenchmark to track which is the fastest. Since `const` is more useful when overloaded against non-`const`, and `&&` is much more useful with an alternative `&`, this feature is treated simultaneously.

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 ); }
};
const std::function< int( int ) > f = delay_buffer{};   // No heap allocation.
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`. The normative problem is that the call operator is const-qualified ([func.wrap.func.inv] §20.9.12.2.4 [2]) but the Callable requirement is applied directly to the value of the target object, not to a const-qualified access path to it ([func.wrap.func.con] §20.9.12.2.1/8).

---

[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 N4527.

1.1. N4348, *Making `std::function` safe for concurrency*

N4348 proposes to mandate a const-qualified access path to the target during dispatching. The motivation is not as stated above. Instead, it focuses on [res.on.data.races] §17.6.5.9/3, a library-wide requirement that functions with no non-const parameters (e.g. `function::operator()`) must not cause the modification of any objects, even through a required callback invocation. Unfortunately, treating the target object as `const` is a non-solution:

1. An even bigger issue, [res.on.data.races]/2, is ignored. The difference is that ¶3 restricts *modifications* through access paths *starting with* the target object, whereas ¶2 restricts *all accesses* through access paths *not starting with* the target object. So, given N4348, `std::cout << "hello";` would be allowed, but still illegal within a target object. (Or even a pure accessor like `std::cout.good()`.)
2. Starting from a const-qualified target object access constrains the user very minimally. Wrapping a target with `std::reference_wrapper` strips qualification even if `function` adds it. Indeed, these two call wrappers are equivalent under [res.on.data.races] so any `const`-oriented solution should apply equally to both of them. Indirection through any ordinary pointer within the target undoes the solution, as the user obtains a means to unsafe modifications. So, many cases afoul of ¶3 will not be caught.
3. It is a breaking change. Well-established use cases must migrate to the new adaptor `std::const_unsafe_fun`, which simply encapsulates a `const_cast`. When used with `std::function`, it is (at least) a safe cast, despite the name. The adaptor is proposed to be immediately deprecated, but without any alternative but to do the `const_cast` internally. So, it foists busywork upon users.
4. The `const_cast` solution disregards actual race conditions in an attempt to get on the right side of an arbitrary rule. The purpose of [res.on.data.races] is to prevent data races, but `const_unsafe_fun` does nothing to encourage the user check for a race and add synchronization. Usage gets tagged as “unsafe” regardless of its synchronization (and probably falsely), and left at that. So, it distracts from the actual problem.

Most race conditions can only be solved by synchronization. Any sort of synchronization inside `std::function` would add too much overhead just by its presence, even if it never blocked. User-supplied callbacks and target objects must be exempt from library implementation rules. Users do deserve a solution, because accesses to `std::cout` in a multithreaded program do embody data races — regardless of `function`. Unfortunately, that requires a critical sections library, which is not presented here. This author did consider the problem and posted a critical section adaptor class for function objects on std-proposals. However, although that was tricky, developing guidelines and educating on the correct usage of critical sections is even harder. A collaborator with such experience is invited to help pursue `std::critical_section`.

2. Qualified signatures

A signature given to `std::function` is used verbatim to define its call operator. When a target object is adopted, it is checked for compatibility as per Callable ([func.wrap.func] §20.9.12.2/2), 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. The critical expression becomes:

INVOKE(declval<F *cv-qualifier-seq*$_{opt}$ *ref-qualifier*>(), declval<ArgTypes>()..., R)

This changes classic functions such as `std::function<void()>` so that the call operator loses `const` qualification, and rendering `std::function<void()> const&` uncallable. This breakage is remedied by an auxiliary call operator with the `const` qualifier, but potentially with a `[[deprecated]]` attribute. The fix applies only to signatures with no qualifiers, so `std::function<void()&>` does not implement an auxiliary `void() const&` signature. The auxiliary function essentially encapsulates a `const_cast<function*>(this)` expression.

```
                                          // Error, Target does not support given signature:
std::function< void() const > cf = []() mutable {};

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

std::function< void() const > cfgood = []{};                               // OK
auto const & fcrgood = cfgood;
fcrgood();                                                                 // OK
```

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 may be widespread. Therefore we must consider whether the legacy support should be normatively `[[deprecated]]`, §D-deprecated, or merely discouraged. Note that a conforming implementation cannot reject a program for making a deprecated call, because attributes are limited to hint-level semantics. Conversely, it is the implementation's prerogative to add a warning (such as by `[[deprecated("uh-oh")]]`) to any suspicious standard library function; this does not affect conformance.

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.

These recommendations should truly improve the quality of the affected code. Regardless, the status quo may be maintained indefinitely by selecting a lesser level of deprecation.

2.2. `volatile`

Objects of `volatile`-qualified class type exist, albeit rarely. There are volatile-qualified member overloads in `std::atomic`. Such signatures are allowed for `function` as well. Calling a wrapper by a `volatile` access path invokes no particular special semantics aside from using a likewise-qualified access path to the target object. The wrapper may run slightly slower if it inadvertently applies `volatile` to itself, but this should not be construed as a measure of safety. It is unlikely that the object model, the strict aliasing rule, and data race avoidance would permit asynchronously reconstructing the target object (even changing its type) across all platforms.

2.3. Conversions

Safety is generically guaranteed in conversions between `std::function` specializations differing only in qualifiers, but this operation consumes resources. As with any valid conversion between `std::function` specializations, the target becomes doubly wrapped and dispatch will proceed through two indirect calls (or any number, depending how many conversions are dynamically executed).

```
std::function< void(int) > a = [](int){};
std::function< void(long) > b;
for (;;) {                                          // Memory leak since C++11
    b = a;
    a = b;
}
```

Such a loop **cannot** happen between, say, `<void(int)>` and `<void(int) const>`, so this is not a pressing issue. Adding const safety per §2.1 will never lead to memory leaks. Strongly-typed `noexcept` (N4533) creates new function types but does not verify the exception safety of

function bodies, so cyclic wrapping of `noexcept(true)` and `noexcept(false)` is possible, if not particularly likely. Likewise between `<void(int)>` and `<void(int)&>`.

To allow implementations to combat such leaks, `target_type()` should be unspecified when it would refer to a different `std::function` specialization. A runtime check could traverse nested wrappers and eliminate any complete cycle. It would only need to run (or exist) in the corner case of a suspicious conversion. (The prototype library does not implement such measures.)

3. Overloading

Rather than a single template type parameter, `std::function` accepts a type parameter pack. The signature of each type defines one call operator overload, and the `function` specialization exposes the complete overload set. Target objects are checked against all the overload signatures as described in §2 *Qualified signatures*.

```
std::function< void( int ), void( std::string ) > fis;
fis = [] ( auto && msg ) { std::cout << msg << '\n'; };            // OK
fis( 42 );
fis( "hello, world!" );
fis({ -1ULL });          // Error: narrowing conversion. (Perfect forwarding is not applied.)

fis = []( int ){};         // Error: cannot convert std::string to int in call operator.

std::function< void( long ), void( std::string const & ) > flsr;
flsr = fis;                                                  // OK, but double wrapped.
std::function< void( int ) > fi = fis;          // OK, but maybe double wrapped.
```

The overload set must be valid if it was expressed as a sequence of member function declarations. As usual for its preconditions, the library is not required to diagnose this rule. For the prototype library under Clang and GCC:

```
function<void(), void() const &> x;        // Undiagnosed error: invalid overloading.
function<void() const &, void()> y;                          // Similarly undiagnosed.
y = x;               // Diagnosis here, revealing that the invalid second signatures were ignored.
```

Both compilers are apparently not conforming to [namespace.udecl] §7.3.3/15-16; otherwise the problem would have been diagnosed at the declarations of `x` and `y`.

The members `result_type`, `first_argument_type`, `second_argument_type` and `argument_type` are each defined if none of the signatures would give them conflicting definitions. So, `std::function<void(int), int(long, long), int(short, long)>` does not define `result_type` or `first_argument_type`, but has `argument_type` as `int` and `second_argument_type` as `long`. (Note, this functionality has not been implemented in the prototype.)

The class `std::function<>` is well-defined, though not particularly useful. Aside from minor differences, it is like `fundamentals_v1::any`. There is no particular reason to forbid it, and doing so could interfere with templates that mention it without actually creating an object.

3.1. Uniqueness

The behavior of a `function` specialization is determined by the *set* of given signatures, but its type identity is determined by the *sequence* of signatures. Ideally, permutations of the same overload set should be normalized to a canonical order. This may be considered separately for `function` itself and for the hidden internal classes which manage each target type.

Double-wrapping alleviation depends on detecting that two wrappers share compatible wrapped target object types. Like wrappers sharing similar signatures (§2.3 *Conversions*), ones with the same canonicalization should not double-wrap each other, and it would help the library to specify that double-wrapping is not guaranteed.

The language currently lacks a facility for sorting type-lists, but alias templates do provide the means to map multiple *template-ids* to the same type. In the long run, it may help if `std::function<T...>` is specified to be an alias to `std::canonical_function<U...>` (`with sizeof...(T) == sizeof...(U)`, since duplicates are not allowed), to an unspecified type, or even to some `std::type_erasure` modeled after Boost.TypeErasure. The most visible difference would be in extracting the signature type from a `function` specialization: introspection would need to be performed on `canonical_function` instead. However, most such cases are covered by `first_argument_type` etc. anyway, and minding the alias template would be only a minor consideration in the broad problem of general N-ary function signature introspection. Another minor difference is that a `friend class` declaration cannot name an alias template; a `friend typename` declaration is needed instead. Users should not do either. The impact of `canonical_function` should be fairly low.

3.2. `const` compatibility redux

Each unqualified signature is shadowed by a deprecated, `const`-qualified overload, per §2, but these overloads have lower priority and are hidden by ones generated by explicitly-specified signatures (regardless of the order of the template parameter list).

It is possible, but unlikely for an auxiliary overload to affect overload resolution. For example, given `function<void(int), void(long) const> const &r`, the call `r('x')` selects the `void(int)` signature. Actually seeing this would indicate an underlying problem. Anyway, the risk is mitigated as long as the deprecation warning accompanies any implicit `const_cast`.

4. Implementation

Overloading is difficult to implement within the architectures currently used by GCC libstdc++ and Clang libc++. The former stores a function pointer directly within the wrapper type. Extending this methodology would yield `function` specializations with `sizeof` proportional to the number of signatures. The latter stores the function pointer in a vtable by implementing a `virtual` dispatch function. Parameter packs play badly with polymorphic classes; the hidden manager for each target type would each generate an intermediate class per signature, with the vtables requiring $O(N^2)$ space in the number of signatures.

The prototype library achieves compile-time and runtime efficiency by using a global `tuple` to hold a descriptor table for each target type. Ordinary function pointers are smaller than the

PTMFs which populate Common ABI vtables; they (not coincidentally) dispatch faster because there is no base pointer adjustment; and they can be set to `nullptr` for various optimizations. Also, unlike a vtable, `tuple` supports data members so `target_type` may be implemented without an indirect call.

A tuple also provides a viable means of ABI versioning. For example, if a library wanted to add support for calling `swap` on target objects, that would require an additional internal function, which would ordinarily be an ABI bump. Given a dynamic ABI version field, new versions of the library headers could check whether a given target object was generated by the older headers lacking erased `swap`, and fall back on the old code. This legacy support may be dropped with the next "real" ABI version.

The prototype library ABI has other advantages over the open-source status quo. Wrapper overhead is reduced to one pointer, compared to two in libc++ and libstdc++, and padding is eliminated. It supports local storage of nontrivially-copyable and highly-aligned types. No special case is used to support the `nullptr` state. During its development, the author found bugs in both libstdc++ and libc++ relating to special cases for targets of pointer type. Having mature standard libraries as an initial foundation, the prototype avoided introducing special cases and workarounds.

Qualification presents another challenge. Because there is no template facility to inspect only the *ref-qualifier* or one *cv-qualifier* of a signature, liberally-applied macros generate the templates needed to dissect qualified member signatures, so they may be reassembled as non-members. Some compile time may be regained in the prototype by more consistently using non-member signatures, avoiding the macro technique.

The library simultaneously implements N4543 *A polymorphic wrapper for all Callable objects*, P0043 *Function wrappers with allocators and* `noexcept`, and (not complete as of this date) P0042 *`std::recover`: undoing type erasure*. It is compatible with the C++11 dialects of GCC and Clang, and it is distributed under a generous license with the intention of being used by such implementations as a baseline.

5. Conclusion

Overloading within `function` is eminently useful. Qualified signatures are overdue for uniform support of C++11 fundamentals. This is sure to be a popular addition to the library. Not only that, it will boost the popularity of ordinary single-signature `function`, by allowing organizations to move away from their own in-house multi-signature solutions.

5.1. Future directions

Besides highly-appropriate uses like enabling move semantics with `const&` and `&&` overloads, the overloading capability allows unrelated functionalities and accessors to be jumbled together, ideally (but usually not) sanitized by dispatch tags. A facility like Boost.TypeErasure would be more scalable and efficient. Ideally, target object interoperability will be guaranteed between wrappers of similar functionality.

It may be helpful to require that the lvalue- and rvalue-qualified forms represented by a signature with no ref-qualifier will both dispatch to the same function. The language provides no such facility, to the author's knowledge. The desired distinction between `&` and no ref-qualifier remains an open question at this time.

Some optimizations and features remain to be implemented in the prototype library. See its issue tracker.

5.2. Kudos

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