Document number: N4159
Date: 2014-10-10
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Geoffrey Romer <gromer@google.com>,
Roman Perepelitsa <romanp@google.com>

`std::function` and Beyond

Introduction

`std::function`'s design dates back to the implementation of `boost::function`, circa 2001, and was preserved largely intact as it passed through TR1 and into the standard library. Although well-suited to the C++98 environment, this design has been overtaken by subsequent changes to the core language, and has several significant shortcomings in modern C++ usage:

- It is not const-correct, and consequently is subject to data races.
- It can't wrap non-copyable function objects.
- It can only wrap function objects that are callable as lvalues.

This document discusses these problems, and explores potential solutions to them.

Definitions

For purposes of this paper, a type is *lvalue-callable*[1] (for some set of argument types) if a non-const lvalue of that type can be used as the postfix-expression of a function call. *rvalue-callable* is defined correspondingly. A type is *const-callable* if both a const lvalue and a const rvalue of that type can be used as the postfix-expression of a function call (*const-rvalue-callable* and *const-lvalue-callable* can be used to distinguish the two cases). Note that const-callable implies lvalue-callable and rvalue-callable.

The Trouble with std::function

Const-correctness and data races

`std::function`'s `operator()` is a const method, and yet it invokes the target function in a non-const context, and so it may invoke a non-const `operator()` when the target is a function object. The standard is not as clear about this as one might wish, but no other interpretation is tenable:

- The standard does not treat the target function as a member of `std::function`, and in fact it can't be implemented as a direct member, so the fact that `std::function`'s `operator()` is const-qualified does not license us to assume that the target is const-qualified within it.
- [func.wrap.func]/p1 states that a `std::function` can "store, copy, and call arbitrary callable objects", which evidently includes function objects that have no const-qualified `operator()`. This is possible only if `std::function`'s `operator()` treats the target

[1] See LWG 2393 for a proposal to standardize this term.

function as non-const. This text is not strictly normative (and, as we will see below, it is false for other reasons), but it at least indicates intent.
- In practice, all `std::function` implementations that we are aware of treat the target object as non-const.

Whether or not the target function is *formally* a member of the `std::function` wrapper, it is *intuitively* a member: it is accessible only via the wrapper, it is copied when the wrapper is copied, and destroyed when the wrapper is destroyed. As a result, `std::function` violates users' intuitive expectations about the meaning of `const`.

Worse, this situation directly contravenes [res.on.data.races]/p3, which states that "A C++ standard library function shall not directly or indirectly modify objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's non-const arguments, including `this`." One might argue that `std::function`'s `operator()` can satisfy this requirement using internal synchronization, but this does not solve the problem: `std::function` exposes a pointer to the target function via the `target()` method, which means that the target function is always potentially accessible to any thread, regardless of any internal synchronization.

As a historical note, Peter Dimov reports that this behavior was implemented intentionally, but only to support function objects that were logically const, but had accidentally omitted const from the `operator()` declaration (which tended to work because function objects are typically passed by value). The fact that it also supports genuinely non-const function calls was an accident.

We have encountered the argument that `std::function` should instead have "shallow const" semantics, so that a const `std::function` is prohibited only from changing its target to a different object, not from mutating the state of its current target. Shallow constness is a property of reference types, by which we mean types whose values refer to objects outside themselves. This includes ordinary C++ references, but also pointers[2], smart pointers, iterators, and `reference_wrapper`. Reference types are identifiable by a cluster of related attributes:
- They have shallow-copy semantics: copies of a reference object refer to the same underlying object, and mutations via one copy are visible via the others.
- Their operations access and manipulate their means of referring to the referenced object, but do not access the referenced object itself[3]. For example, none of `unique_ptr<T>`'s member functions accesses the `T` value; only the underlying `T` pointer.
- Their reference-ness is fundamental to their meaning. This is reflected in their API contracts, and also in their names, which typically mention "reference", "ptr", or "iterator".
- They have shallow-const semantics, meaning that if you can get non-const access to the referenced object at all, you can get it even via a const reference object.

[2] But not function pointers, since functions are not objects (see [intro.object]/p1). More fundamentally, the concept of a reference type is meaningless when the underlying entity is immutable.
[3] `reference_wrapper`'s `operator()` is a notable exception, but we would argue this is merely a duplicate (perhaps an imitation) of the corresponding defect in `std::function`.

`std::function` has none of these attributes except for shallow constness, and even that applies only to `operator()`; the `target()` member has deep-const semantics. Thus, `std::function` is quite clearly not a reference type, and users cannot be expected to treat it as a reference type, so the shallow const model is inapplicable.

Non-copyable function objects

`std::function` is copyable, and its copy constructor has the postcondition that "`*this` targets a copy of `f.target()`", which implies that the target type must be copyable. Unlike with many other class templates, this requirement applies even if the copy constructor is never actually invoked, and in fact this requirement is explicitly specified in `std::function`'s constructor from a function object (this is because the implementation must instantiate all the operations it can potentially apply to the target, in order to implement type erasure).

Consequently, `std::function` is unusable with move-only function objects. Such objects arise quite naturally in many settings, such as when a lambda captures a move-only object by value, so this is a very significant limitation.

By the same token, `std::function` is unusable with types that are neither movable nor copyable. Such types can still satisfy the requirements of a function object type, but are much less common in practice.

Non-lvalue-callable function objects

[function.objects]/p1 states that "A *function object type* is an object type that can be the type of the *postfix-expression* in a function call". This definition can apply even if the type is not lvalue-callable, i.e. if the type's only `operator()` overloads are rvalue-reference qualified. However, `std::function::operator()` is fairly clearly specified to treat the target function as an lvalue, and so it will not work with such function objects.

It is tempting to dismiss such function objects as too unnatural to worry about, but this is not the case. An rvalue-ref-qualified `operator()` is the natural form for a function that can only be invoked once. This is a very common constraint for code written in continuation-passing style, and also arises naturally due to move semantics. For example, if `std::bind` were extended to support binding move-only arguments to move-only parameters, the resulting function object would effectively be subject to that constraint. Given the importance of this constraint, it is a serious deficiency that `std::function` cannot wrap functions that express that constraint directly in the type system.[4]

Repairing `std::function`

`std::function`'s lack of support for non-copyable and non-lvalue-callable function objects could plausibly be treated as a feature request, but the const-correctness issue is an outright defect:

[4] There is also arguably a missing language feature here, to allow users to create lambdas whose call operators are rvalue-ref-qualified, but that is beyond the scope of this paper.

`std::function` simply cannot be implemented as specified without violating the standard library's guarantees on data race avoidance.

Any adapter type faces a tradeoff between maximizing the number of clients it supports, and maximizing the number of types it can adapt: if you want to increase the power of the adaptor interface to support more clients, you will usually have to impose more requirements on the adapted type. Conversely, if you want to relax your requirements to adapt more types, you will usually have to remove features from the adaptor interface that were supported by those requirements. Clever adapter design can mitigate this tradeoff, but not avoid it altogether.

`std::function`, in trying to avoid this tradeoff, has promised its clients an API that it cannot implement in terms of the requirements it imposes on the target type. The only way to balance the books is to remove functionality from the interface, or tighten requirements on the target. This could take several forms:

Most obviously, we could **add internal synchronization**. This is only feasible if we remove the `target()` member function. `target()` is somewhat out of place to begin with, since it exists solely to break the abstraction boundary that `std::function` provides, so removing it is tempting. However, internal synchronization could impose a substantial overhead on every `operator()` call, and creates a very real risk of deadlock.

The core problem is the mismatch between `std::function`'s const-qualified `operator()` and the (potentially) non-const-qualified `operator()` of the target, so one obvious fix is to remove the `const` qualifier from `std::function::operator()`. However, this is likely to break huge amounts of existing code, including plenty of code that has no const-correctness issues. Instead, we could **add a non-const overload for `operator()`**, and revise the const overload to throw an exception (or terminate the program) if the target function is not const-callable. This change would explicitly break only use cases that are inherently const-incorrect. However, the breakage would manifest only at run-time, making it much more difficult for users to have confidence that their code is safe. In addition, it would potentially change the behavior of any `std::function` whose target has separate const and non-const `operator()` overloads.

The obvious converse of the above solution is to **require the target type to be const-callable**. This is the most principled and conceptually simplest fix, but it has the potential to break substantial amounts of existing code at compile time (albeit code that, if not actually thread-unsafe, is at least tacitly relying on `std::function`'s const-incorrectness). It's worth noting that user code can already do this on an opt-in basis, by adding another wrapper type between the `std::function` and the intended target which `static_asserts` that the target is const-callable. However, because this constraint would be type-erased, code that receives `std::function` objects from other libraries could not rely on it. Under this approach, it might still make sense to add a non-const `operator()` overload to `std::function`. This would permit client code to implement any of the other solutions on an opt-in basis, by wrapping the target in a suitable adapter before passing it to `std::function`. For example, there could be a call wrapper

which throws on non-const calls if the target is const-only, and one that uses internal synchronization, and one that just implements the current `std::function` behavior and hopes for the best.

If we cannot tolerate breaking any existing users, we can simply **standardize the status quo**, and document that `operator()` is exempted from [res.on.data.races]/p3, or document that the target function is required to support concurrent non-const calls. This amounts to passing the buck to our users, which makes it by far the most convenient solution for platform vendors and the committee, but whether it is the most convenient solution for users is less certain. It would silently delegate responsibility for thread-safety to users, while depriving them of one of the most important tools for doing so (namely const-correctness).

Beyond `std::function`

The fundamental reason that `std::function` lacks support for non-copyable and non-lvalue-callable function objects is that type erasure can only erase the implementation; it cannot erase the interface. The “callable” concept encompasses surprisingly many degrees of interface freedom, which type-erasing adapters must somehow accomodate.

Possible solutions to this problem will tend to fall on a spectrum. At one extreme, we could define a separate type-erasing wrapper for every possible use case. At the other, we could define a single “compromise” interface for the wrapper, support sufficiently-similar interfaces through explicit and/or implicit adaptation, and exclude the rest (in the hope that they are sufficiently marginal). The essential tradeoff is that with more wrapper types we can cover the space more exactly, at the cost of reduced interoperability (since separate components are less likely to use compatible wrappers) and a higher cognitive burden on users.

Exploring the solution space

We begin by examining the extreme of defining a separate wrapper for every distinct possible use case. Consider a `std::function`-like wrapper `wrapper<R(ArgTypes…)>`, which will wrap some callable type `target`. We would like wrapper to implement as many as possible of the following members[5]:

```
wrapper(const wrapper&);
wrapper& operator=(const wrapper&);
wrapper(wrapper&&);
wrapper& operator=(wrapper&&);

R operator()(ArgTypes…) &;
R operator()(ArgTypes…) const &;
R operator()(ArgTypes…) &&;
```

[5] We assume that wrapper will not provide volatile-qualified members. Standard library types almost never do (`std::atomic` and `std::atomic_flag` are the only exceptions), and we see no use cases to warrant departing from that convention.

```
R operator()(ArgTypes…) const &&;
```

(in some circumstances, we might want to replace a pair of ref-qualified operators with a single ref-unqualified operator, but that does not change the analysis).

The properties of `target` determine whether and how `wrapper` implements these members[6]:

- The copy constructor must be implemented in terms of `target`'s copy constructor.
- The copy assignment operator can be implemented in terms of `target`'s copy assignment operator, or its copy constructor.
- The move constructor can be implemented in terms of `target`'s move constructor, or its copy constructor, or it can be implemented without any reliance on `target` by allocating the object on the heap.
- The move assignment operator can be implemented in all the same ways as the move constructor.
- The const `operator()`s can be implemented by calling the target through a const reference of the appropriate value category, and so rely on `target` to be const-lvalue-callable or const-rvalue-callable (respectively).
- The non-const `operator()`s can be implemented by calling the `target` object through a non-const reference of the appropriate category, and so rely on `target` to be lvalue-callable or rvalue-callable (respectively).

Thus, although a variety of factors can influence how each member is implemented, only the following properties of `target` affect which members can be provided at all:

- Whether it is copy-constructible
- Whether it is copy-assignable
- Whether it is lvalue-callable, and if so whether it is const-lvalue-callable
- Whether it is rvalue-callable, and if so whether it is const-rvalue-callable

There are 2×2×3×3 = 36 possible combinations of these properties, and thus we need 36 wrapper interfaces if we want to support every conceivable use case.

We can reduce this number substantially, if we are willing to exclude some marginal use cases:

- Cases where the **target type is not copy-constructible, and the client requires copy-assignment**. Such cases seem extremely unlikely: copy-assignability almost always implies copy-constructibility, because the copy constructor can be trivially implemented in terms of the copy-assignment operator (although this is seldom the best way).
- Cases where the **target type is not lvalue-callable, and the client requires copyability**. As discussed earlier, types that are callable but not lvalue-callable express the constraint that the function should be called only once, which is incompatible with copyability: shallow-copy semantics would be pointless (if only one of the copies can

[6] Note that the mechanics of type erasure will typically enable a single wrapper type to use different implementation strategies for different target types.

ultimately be called, why bother making the other copies?), and deep-copy semantics would nullify the call-only-once constraint (just capture a copy before every call, and you can call as much as you want).

- Cases where the **target type is not const-lvalue-callable, but the client requires const-rvalue-callability**. It is difficult to see a plausible use case for such types: deliberately operating on const rvalues is rare, and if the function call can be executed without logically mutating internal state, it can almost certainly be implemented the same way for lvalues as for rvalues. Note that call-only-once functions should not be const-rvalue-callable: calling one is certainly a state mutation, logically if not physically, because it changes *whether it can be called*.
- Cases where the **target type is not rvalue-callable**. This is more debatable, because there are some arguably legitimate use cases for such types. For example, if a function object's purpose was to compute some aggregate from the values passed into its call operator, calling it as an rvalue would be useless (because it would be destroyed before the result could be extracted), so it might provide only lvalue-ref-qualified call operators, to help its clients catch mistakes. However, while this is not obviously wrong, neither is it obviously right: the operations thus forbidden are merely useless, and correlated with certain kinds of errors; they are not harmful in themselves. Thus, even if this technique provides value, it may not provide enough value to justify all the extra wrappers needed to support it. Furthermore, non-rvalue-callable types may be an interoperability hazard, because every API that takes a function wrapper would have to choose between disallowing lvalue-only arguments and disallowing rvalue-only arguments. Requiring rvalue-callability gives us a useful lowest-common-denominator callable interface.
- Cases where the **client doesn't use the call operator at all**, because such cases don't need a function wrapper in the first place.

If we apply all of these exclusions, we are left with five distinct wrapper interfaces:

1. Copyable, const-callable
2. Copyable, lvalue-callable and rvalue-callable
3. Move-only, const-callable
4. Move-only, lvalue-callable and rvalue-callable
5. Move-only, rvalue-callable

We could perhaps argue that we don't need a move-only, const-callable wrapper, because a const-callable function object is effectively stateless, and so can always be made copyable using reference-counting (either explicitly in a user-supplied wrapper, or implicitly as part of the wrapper implementation). Even more tenuously, we could argue that we don't need a copyable, lvalue-callable wrapper, because if the function accumulates API-visible state, you generally want that state to reflect the whole computation, so you want to use a single object throughout.

It is difficult to see how we can push the reduction any further than this without simply abandoning the static type system and moving to run-time type checking (see below for more on this approach).

Naming the wrapper types

There will be plenty of time to argue over naming later, but it's instructive to consider how we might name the 5 wrapper interfaces identified in the previous section. One possibility is to use a fixed, predictable naming scheme, e.g. `std::copyable_const_function`, `std::copyable_mutable_function`, `std::movable_rvalue_function`. However, if we go that route, we might as well use actual template parameters, rather than naming conventions (see the next section).

Instead, we could use more meaningful names that try to capture the distinct use cases for these wrappers, thereby guiding API designers toward the appropriate wrapper for their use case. For example, we could call #1 `std::copyable_function` and #4 `std::movable_function`, giving them relatively generic names because we expect them to be the most common general-purpose wrappers. #5 could be called `std::continuation`, a term which is suggestive of the stateful, call-only-once nature of the wrapper. One reviewer has suggested `std::generator` for #2, since the combination of copyability and mutability is well-suited to objects like random number generators. Alternatively, we could name the const versions `std::copyable_pure_function` and `std::moveable_pure_function`, since a const constraint on a functor is normally used to signify functional purity.

However, it is evident that there is a lot of guesswork involved here, which does not bode well for our chances of choosing well, however many rounds of bikeshedding we go through. Furthermore, many of the possible names have misleading connotations- continuations typically don't return, generators typically take no arguments (and, indeed, some generators may not be copyable), and a const function may still be 'impure'.

A parameterized wrapper template

If we have relatively few wrapper types, it may be practical to name and specify them as independent types, but if we have more than about 5, this will become unwieldy. We can manage the complexity by defining a single function wrapper template, which takes additional template parameters to specify its interface. The choice of parameterization depends on which use cases we wish to support. As an illustrative example, if we want to support only the 5 major use cases discussed above, we could define roughly it as follows (many details omitted):

```
enum call_type { rvalue_callable_only, lvalue_callable, const_callable };

template<class, bool Copyable, call_type CallType>
class basic_function;

template<class R, class... ArgTypes, bool Copyable, call_type CallType>
class basic_function<R(ArgTypes...), Copyable, CallType> {
  basic_function() noexcept;
  basic_function(basic_function&&);
```

```
  basic_function& operator=(basic_function&&);
  ~basic_function();
  explicit operator bool() const noexcept;
  const std::type_info& target_type() const noexcept;
  template<class T> T* target() noexcept;
  template<class T> const T* target() const noexcept;

  R operator()(ArgTypes...) &&;

  // Requires:
  // - 'Copyable' is false, or F is CopyConstructible,
  // - F is rvalue-callable with 'Args',
  // - CallType < lvalue_callable or F is lvalue-callable with 'Args',
  // - and CallType < const_callable or F is const-callable with 'Args'
  template <class F, class... Args>
  void emplace(Args&&...);

  // Shall not participate in overload resolution unless all of
  // the above, and F is MoveConstructible.
  template<class F> basic_function(F&&);
  template<class F> basic_function& operator=(F&&);

  // Shall not participate in overload resolution if !Copyable
  basic_function(const basic_function&);
  basic_function& operator=(const basic_function&);

  // Shall not participate in overload resolution if
  // CallType < lvalue_callable
  R operator()(ArgTypes...) &;

  // Shall not participate in overload resolution if
  // CallType < const_callable
  R operator()(ArgTypes...) const &;
  R operator()(ArgTypes...) const &&;
};
```

Some notes about this API:

- We have added an `emplace()` member, to support non-movable function types.
- The parameterization permits the user to specify a wrapper that is copyable but not lvalue-callable, which we saw earlier is unlikely to be useful. This costs us little, but if necessary we could make it ill-formed.

One particularly notable feature of the above example is that it provides all `operator()` overloads that are consistent with the given parameter: with `lvalue_callable`, it provides separate lvalue and rvalue overloads rather than a single ref-unqualified overload, and with `const_callable`, we retain both non-const overloads, and add separate `const &` and `const &&` overloads for `operator()`, rather than collapsing them to a single const overload. This permits us to forward calls to the corresponding overloads of the target (if present). This may be useful in some situations, has no obvious technical drawbacks, and slightly simplifies the SFINAE logic. However, it may be simpler to retain the property that an engaged function wrapper always wraps exactly one function, never a nontrivial overload set.

Of course, a series of bool/enum template parameters becomes unwieldy if we want to support configuration along more axes. Various other options are available, such as having a bit-set or policy class as a single configuration parameter, or 'named parameters', as implemented by e.g. the Boost Parameter Library.

Under this scheme, the type names will be more verbose and less memorable than a set of explicitly named wrappers would be. However, the two are not mutually exclusive: the named wrappers can be defined as template aliases[7]. This gives us the best of both worlds: a small, focused set of named wrappers for the common cases, and a parameterized wrapper to ensure the uncommon cases aren't out in the cold.

One obvious question is whether `std::function` itself can be one of these aliases. This depends on which of the solutions we adopt for the const-correctness issue, because some of those solutions leave `std::function` in too inconsistent a state to be an instance of the parameterized wrapper. Even for the more consistent solutions, making `std::function` an alias may require additional breaking changes, depending on the exact API. If we cannot make `std::function` an alias, it would probably be better to deprecate it, because the inconsistency between it and the other function wrappers will be an ongoing source of confusion.

Qualified function signatures

One special case of the parameterized wrapper approach deserves special attention: cv- and ref-qualifiers are actually part of a function's type, and can therefore be embedded in the existing template parameter, which specifies the function signature:

```
std::function_wrapper<void() &&> f1;     // rvalue-callable
std::function_wrapper<void()> f2;        // lvalue- and rvalue-callable
std::function_wrapper<void() const> f3;  // const-callable
```

In effect, the template argument would specify the qualifiers of the wrapper's `operator()`, as well as the arguments, and that operator would call the "corresponding overload"[8] of the target.

[7] The name `basic_function` is intended to suggest an analogy with `basic_string`.
[8] More precisely, it would invoke the target in such a way that an overload with identical qualifiers would be selected, if it were present.

Of course, we cannot specify copyability in such a way, so we still need a parameter for that (or, perhaps, two separate templates).

This approach has some clear advantages: it lets us express most of the parameter space using a concise, pre-existing notation, whose meaning is more or less deducible from the rest of the language. With that simplification, we may not even need to supplement the parameterized system with named aliases, permitting us to get away with only one or two new library names.

However, it also has some significant drawbacks: the use of qualifiers may give the impression that these types can only target member functions, and although the syntax leverages existing concepts in the standard, they are relatively new and advanced concepts, and so may be confusing to many programmers. One particular source of confusion is that cv- and ref-qualifiers on functions have conversion behavior that is opposite to their top-level behavior: whereas `T&` is convertible to `T const&`, but not the reverse, `T(Arg)const&` is convertible to `T(Arg)&`, but not the reverse[9].

It may thus still be beneficial to provide named aliases for important use cases, as discussed above. However, aliases which actually transform the signature parameter may be more confusing than aliases which simply bind values to the parameters, and will definitely be less useful:

```
template <class F>
using clever_alias = wrapper<add_const_to_function_t<F>>;

template <class F>
void Foo(clever_alias<F>);

clever_alias<void()> f;
Foo(f);  // Error: can't deduce F
```

An approach using signature qualifiers is also less convenient to implement: it seems to require a separate partial specialization for each possible combination of qualifiers, whereas approaches with separate parameters could probably make do with a single specialization and extensive use of SFINAE. This creates an interesting tension: we previously dismissed volatile-qualified functions because they are such a marginal use case that they're not worth including in the parameter space, despite the likely small implementation cost. In this approach, the shoe is on the other foot: from an interface point of view, we'd have to go out of our way to disable it, and yet the implementation cost is now substantial (4 more partial specializations).

[9] More formally, function types are contravariant in the function qualifiers. This is a corollary of the general fact that function types are contravariant in their parameter types, and function qualifiers are more or less part of the type of the implicit object parameter.

The biggest problem, though, is that we could not adopt this approach for `std::function` itself without major breaking changes: current uses of `std::function` do not have qualifiers in the template argument, which in this model would imply `operator()` should not be const-qualified. Adopting this approach for `std::function` would thus break many legitimate use cases.

We could, however, extend `std::function` to support this model when any qualifiers are supplied (`std::function` currently does not permit qualifiers, so this is a pure extension). This would leave no way to explicitly specify no qualifiers, but a single lvalue-reference qualifier might be an adequate substitute. It goes without saying that this could create substantial confusion.

‘`throw`’ing caution to the winds

A very different alternative would be to stick with std::function as the sole function wrapper, add the various possible operator() overloads to it, and have the operations that are not supported by the target simply throw exceptions (e.g. if the target is move-only, the copy constructor will throw). This permits us to retain a single, familiar wrapper, with no changes in syntax. However, this would entail a dramatic reduction in the safety of code using std::function: the object’s static type would tell you almost nothing about what you can safely do with it (although it could be augmented with query functions to make that determination).

It’s important to note that this is not merely an extension of std::function’s existing type erasure: type erasure is a form of dynamic polymorphism, which erases the differences between implementations of the same interface. This approach, by contrast, erases both implementation and interface, and is thus a full-blown dynamic type system. This sort of system design is legitimate, but decidedly at odds with C++’s general preference for strong static type safety.

Summary and recommendations

The following is a summary of the decisions the committee will need to make to resolve these issues, together with our tentative recommendations.

Should we deprecate `std::function`?

We do not recommend this. `std::function` is one of the most visible library additions in C++11, and one of the most prominent elements of the library overall, so deprecation would be extremely disruptive and disorienting. Furthermore, we could never really get rid of it (we can’t even complete the deprecation of `auto_ptr`), so this would leave an extremely valuable name trapped in limbo.

How should we make `std::function` thread-safe?

The options we see are:

1. Add internal synchronization
2. Require the target type to be const-callable
3. Add a non-const `operator()`
4. Standardize the status quo

We recommend requiring the target type to be const-callable, because it leaves `std::function` in the most useful, consistent state. This would probably break significant amounts of client code, but we think the broken code could be fixed with trivial local edits in most cases, and in any event, the alternatives look worse. Adding internal synchronization and adding a non-const `operator()` both carry an unacceptable risk of breaking existing code at run-time. Standardizing the status quo would be a disservice to our users unless we also deprecate `std::function`, which we think would be a bad outcome (see above).

Must the standard library support the following use cases at all?

1. **Target type is non-copyable**
   We strongly recommend "yes": non-copyable functions will be increasingly common, some use cases require uniqueness, and few require copyability.
2. **Target type is not lvalue-callable**
   We strongly recommend "yes": non-lvalue-callability lets us express call-only-once in the type system, and this is an important use case that wrappers should support.
3. **Target type is not rvalue-callable**
   We weakly recommend "no": there are plausible use cases for this, but the ones we're aware of are not compelling, and it's valuable to have every wrapper be rvalue-callable, both for conceptual simplicity and to maximize interoperability. However, this is a new area, so more convincing use cases may emerge with more experience.
4. **Target type is not lvalue-callable, and client requires copyability**
   We weakly recommend "no": we are aware of no use cases for this, and there are theoretical reasons to doubt that any exist. However, experience is lacking, so it's difficult to be certain.
5. **Target type is not const-lvalue-callable, but the client requires const-rvalue-callability**
   We weakly recommend "no", for the same reasons as #4.
6. **Target type is not copy-constructible, and client requires copy-assignment**
   We recommend "no": copy-assignable should imply copy-constructible.
7. **Client doesn't use the call operator at all**
   We recommend "no", obviously.

Should function wrappers have multiple operator() overloads?

We tentatively recommend "no". `std::function` has evidently opted not to, and we see no compelling need to change that decision. Furthermore, given that a type-erasing function wrapper cannot feasibly support overloading on the type or qualification level of function arguments, it seems simpler and more consistent not to support it for implicit-object qualification either.

Should we provide a parameterized wrapper type?

We recommend "yes": even if the parameter space is very small, the parameterized wrapper provides orthogonality, which will make the wrapper interfaces much easier to understand, reason about, and remember. Furthermore, a parameterized wrapper gives us leeway to

support marginal use cases (and gives us a safety margin in case we guess wrong about what's "marginal"), while minimizing the additional complexity exposed to users who don't need it.

Should we provide individually-named wrappers?

Should we parameterize using qualified function signatures?

These questions are nominally independent, but we regard them as tightly coupled in practice: the concision and semantic transparency of the qualified function signature approach is uniquely well-suited to being directly user-facing, and uniquely ill-suited to being a mostly-ignored library implementation detail. Any other approach to parameterization would probably be too verbose to survive without named aliases for the common-case uses.

We tentatively recommend parameterizing using qualified function signatures, and hence not providing named aliases. A parameterized type, despite its greater up-front complexity, is probably more user-friendly in the long run because once users learn the parameter semantics, they can immediately understand any function wrapper they encounter. Furthermore, we can always choose to provide named aliases later, if they prove to be worthwhile.

Future work

A more concrete design (and, eventually, standard language) will of course be needed once we have guidance from the committee.

The library standard generally does not distinguish const-callable from non-const-callable function objects, to say nothing of lvalue vs. rvalue callability, so these issues are likely not confined to `std::function`. For example, `std::reference_wrapper` and `std::result_of` may be subject to similar problems. We should also re-examine the standard's requirements on callable arguments (e.g. hashers, comparators, and predicates), to ensure that they are unambiguous, and not too strict, with respect to the different flavors of callability. LWG 2393 addresses one aspect of these issues.

Other parts of the library might benefit from the concept of using rvalue-ref-qualification to express the call-only-once constraint. For example, `std::bind` could be extended to support binding arbitrary arguments to arbitrary parameter types (currently, it lacks support for binding move-only arguments to by-value or rvalue-reference parameters). `std::packaged_task` already imposes a call-only-once constraint on its call operator, so in principle it could use rvalue-ref-qualification to enforce that requirement, but that change probably can't be made at this point, because it would probably break most existing code.

As noted earlier, it would be quite useful to be able to create a lambda whose call operator is rvalue-ref-qualified.

Library race-freedom guarantees

The meaning and implications of [res.on.data.races] need much more attention, as well as the whole "const = thread-safe" philosophy. The issues discussed in this paper are probably only the tip of the iceberg. Consider the following example, which contains a race on `n`:

```
#include <set>

int main() {
  int n = 0;

  // No ‘mutable’, so operator() is const
  auto cmp = [&](int a, int b) {
    ++n;
    return a < b;
  };
  std::set<int, decltype(cmp)> s(cmp);
  s.insert(1);
  auto result1 = std::async(
      std::launch::async, [&](){ return s.find(1); });
  auto result2 = std::async(
      std::launch::async, [&](){ return s.find(1); });
}
```

Or, consider the following example, which contains a race on `s.value`:

```
#include <functional>

struct S {
  void Mutate() { ++value; }
  int value = 0;
};

int main() {
  S s;
  const auto f = std::bind(&S::Mutate, &s);

  // Calls const operator()
  auto result1 = std::async(std::launch::async, f);
  auto result2 = std::async(std::launch::async, f);
  result1.wait();
  result2.wait();
}
```

In both cases, a standard library function with no non-const arguments appears to be modifying objects that are accessible by multiple threads, leading to a data race[10]. This violates the letter of [res.on.data.races]/p3, but it's not clear that the library is genuinely at fault in either case. [res.on.data.races]/p3 presumably needs to condition on user code obeying the same constraints. The latter case could possibly be resolved by broadening the notion of "arguments" to include bound arguments, in the case of call wrapper types.

[10] It's debatable whether `decltype(f)::operator()` is a standard library function, but this seems like the most reasonable interpretation.