copyable_function

Abstract
This paper proposes a replacement for function in the form of a copyable variant of move_only_function.

Tony Table

<table>
<thead>
<tr>
<th>Before</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto lambda(<a href="void">&amp;</a> /<em>const</em>/ { ... });</td>
<td>auto lambda(<a href="void">&amp;</a> /<em>const</em>/ { ... });</td>
</tr>
<tr>
<td>function&lt;void(void)&gt; func(lambda); const auto &amp; ref(func);</td>
<td>copyable_function&lt;void(void)&gt; func0(lambda);</td>
</tr>
<tr>
<td>func(); ref();</td>
<td>func0();</td>
</tr>
<tr>
<td>//operator() is NOT const!</td>
<td></td>
</tr>
<tr>
<td>auto lambda(<a href="void">&amp;</a> mutable { ... });</td>
<td>auto lambda(<a href="void">&amp;</a> mutable { ... });</td>
</tr>
<tr>
<td>function&lt;void(void)&gt; func(lambda); const auto &amp; ref(func);</td>
<td>copyable_function&lt;void(void)&gt; func1(lambda);</td>
</tr>
<tr>
<td>func(); ref(); //operator() is const! //this is the infamous constness-bug</td>
<td>func1(); ref(); //operator() is NOT const!</td>
</tr>
<tr>
<td>copyable_function&lt;void(void) const&gt; tmp(lambda);</td>
<td></td>
</tr>
</tbody>
</table>

Revisions
R0: Initial version

R1:
- Incorporated the changes proposed for move_only_function in [P2511R2].
- Added wording for conversions from copyable_function to move_only_function.

R2:
- Removed changes adopted from [P2511R2] as that proposal didn’t reach consensus in the 2022-10 LEWG electronic polling.

R3: Updates after LEWG Review on 2022-11-08:
- Fixed requirements on callables in the design section – copy-construct-ability is sufficient.

1 RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at
• Removed open question on the deprecation of function.
• Replaced previously proposed conversion operators to move_only_function.
• Added section on conversions between standard library polymorphic function wrappers.
• Added section on potential allocator support.

R4: Updates after LEWG Review on 2022-11-11:
• Removed mandatory optimization for conversion to move_only_function.

R5: Updates after LEWG Review on 2023-03-07:
• Added section on naming of this class.
• Extended wording with recommended practice to avoid double wrapping of type-erased function wrappers.
• Fixed some wording bugs.

R6: Updates after LWG Review on 2023-06-14:
• Wording for double wrapping of type-erased function wrappers.
• Fixed some wording bugs.

Motivation
C++11 added function, a type-erased function wrapper that can represent any copyable callable matching the function signature R(Args...). Since its introduction, there have been identified several issues – including the infamous constness-bug – with its design (see [N4159]).

[P0288R9] introduced move_only_function, a move-only type-erased callable wrapper. In addition to dropping the copyable requirement, move_only_function extends the supported signature to R(Args...) const_op (&|&)& op noexcept_op and forwards all qualifiers to its call operator, introduces a strong non-empty precondition for invocation instead of throwing bad_function_call and drops the dependency to typeid/RTTI (there is no equivalent to function’s target_type() or target()).

Concurrently, [P0792R10] introduced function_ref, a type-erased non-owning reference to any callable matching a function signature in the form of R(Args...) const_op noexcept_op. Like move_only_function, it forwards the noexcept-qualifier to its call operator. As function_ref acts like a reference, it does not support ref-qualifiers and does not forward the const-qualifier to its call operator.

As a result, function is now the only type-erased function wrapper not supporting any form of qualifiers in its signature. Whilst amending function with support for ref/noexcept-qualifiers would be a straightforward extension, the same is not true for the const-qualifier due to the long-standing constness-bug. Without proper support for the const-qualifier, function would still be inconsistent with its closest relative.

Therefore, this paper proposes to introduce a replacement to function in the form of copyable_function, a class that closely mirrors the design of move_only_function and adds copyability as an additional affordance.
Design space
The main goal of this paper is consistency between the move-only and copyable type-erased function wrappers. Therefore, we follow the design of move_only_function very closely and only introduce three extensions:

1. Adding a copy constructor
2. Adding a copy assignment operator
3. Requiring callables to be copy-constructible

Conversions between function wrappers
Given the proliferation of proposals for polymorphic function wrappers, LEWG requested an evaluation of the “conversion story” of these types. Note that conversions from function_ref always follow reference semantics for obvious reasons.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>function</th>
<th>move_only_function</th>
<th>copyable_function</th>
<th>function_ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>function</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>move_only_function</td>
<td>move_only_function</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>copyable_function</td>
<td>copyable_function</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>function_ref</td>
<td>function</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

It is recommended that implementors do not perform additional allocations when converting from a copyable_function instantiation to a compatible move_only_function instantiation, but this is left as quality-of-implementation.

Concerning allocator support
After having reviewed R2, LEWG requested a statement about potential allocator support. As this proposal aims for feature parity with move_only_function (apart from the extensions mentioned above) and considering the somewhat recent removal of allocator support from function [P0302], we refrain from adding allocator support to copyable_function. We welcome an independent paper introducing said support to both classes.

Naming discussion
During the review of R4, there were questions raised for the rationale for the name copyable_function, especially as it was perceived inconsistent with move_only_function. Our rationale for the name is as follows: copyable_function is a copyable function call wrapper that requires the target object to be copyable, so the copyable-prefix references both aspects. Furthermore, there isn’t actually an inconsistency with move_only_function, as the move_only-prefix only applies to the wrapper; the wrapper is move-only, but there is no reason to require the target object to be as well.

Impact on the Standard
This proposal is a pure library addition.

Implementation Experience
The proposed design has been implemented at https://github.com/MFHava/P2548.
Proposed Wording

Wording is relative to [N4928]. Additions are presented like this, removals like this and drafting notes like this.

[version.syn]

`#define __cpp_lib_copyable_function YYYYMM //also in <functional>`

[DRAFTING NOTE: Adjust the placeholder value as needed to denote this proposal's date of adoption.]

[functional.syn]

```cpp
namespace std {
    // [func.wrap.move], move only wrapper
    template<class... S> class move_only_function;
        class move_only_function<R(ArgTypes...)> cv ref noexcept(noex); // see below

    // [func.copy], copyable wrapper
    template<class... S> class copyable_function;
        class copyable_function<R(ArgTypes...)> cv ref noexcept(noex); // see below

    // [func.search], searchers
    template<class ForwardIterator, class BinaryPredicate = equal_to<>>
        class default_searcher;
}
```

[func.wrap.general]

1. Subclause [func.wrap] describes polymorphic wrapper classes that encapsulate arbitrary callable objects.

2. Let `t` be an object of a type that is a specialization of `function`, `copyable_function`, or `move_only_function`, such that the target object of `t` has a type that is a specialization of `function`, `copyable_function`, or `move_only_function`, each argument of the invocation of `x` evaluated as part of the invocation of `t` may alias an argument in the same position in the invocation of `t` that has the same type, even if the corresponding parameter is not of reference type.

   **Example 1:**
   ```cpp
class move_only_function<void(T)> f{copyable_function<void(T)>{[
    ];}};
T t;
f(t); // It is unspecified how many copies of T are made
```

   **Recommended practice:** Implementations should avoid double wrapping when constructing polymorphic wrappers from one another.

[func.wrap.copy]

1. The header provides partial specializations of `copyable_function` for each combination of the possible replacements of the placeholders `cv`, `ref`, and `noex` where:

   - `cv` is either `const` or empty,
   - `ref` is either `&`, `&&`, or empty, and
   - `noex` is either true or false.

2. For each of the possible combinations of the placeholders mentioned above, there is a placeholder `inv-quals` defined as follows:

   - If `ref` is empty, let `inv-quals` be `cv`.
   - Otherwise, let `inv-quals` be `cv ref`.

3. **Class template copyable_function**

   ```cpp
   namespace std {
    template<class R, class... ArgTypes>
        class copyable_function<R(ArgTypes...)> cv ref noexcept(noex);

    public:
        using result_type = R;

    // [func.wrap.copy.ctor], constructors, assignments, and destructors
        copyable_function() noexcept;
        copyable_function(nullptr_t) noexcept;

    copyable_function(const copyable_function&);
        copyable_function(const copyable_function&&);
    }
```
template<class T, class... Args>
    explicit copyable_function(in_place_type_t<T>, Args&&... args);

friend constexpr bool
    is_invocable_r_v<R, VT
    is_nothrow_invocable_r_v<R, VT

Copyable function

Recommended practice: Implementations should avoid the use of dynamically allocated memory for a small contained value.

Constraints

Let
    is_constructible_v<VT, Args...>
    is_copy_constructible_v<VT
    is_move_constructible_v<VT
    is_constructible_v<VT, Args...>
    is_nothrow_constructible_v<VT, Args...>
    is_constructible_v<VT, F>
    is_nothrow_constructible_v<VT, F>
    is_constructible_v<remove_cvref_t<F>, Args...>
    is_nothrow_constructible_v<remove_cvref_t<F>, Args...>
    is_constructible_v<remove_cvref_t<F>, F>
    is_nothrow_constructible_v<remove_cvref_t<F>, F>
    is_constructible_v<remove_cvref_t<F>, Args...>
    is_nothrow_constructible_v<remove_cvref_t<F>, Args...>

Postconditions

- This has no target object.

Postconditions: At this point, f has no target object. Otherwise, the target object of this is a copy of the target object of f.

Throws: Any exception thrown by the initialization of the target object. May throw bad_alloc.

Postconditions: The target object of this has target object f had before construction, and f is in a valid state with an unspecified value.

Constraints

- remove_cvref(t( )) is not the same as copyable function, and
- remove_cvref(t( )) is a specialization of in_place_type_t<T>
- is_callable_from_VT is true

Constraints

- is_constructible_v<VT, F> is true, and
- is_copy_constructible_v<VT, F> is true,
- Preconditions: VT meets the Cpp17Destructible and Cpp17CopyConstructible requirements.

Postconditions: *this has no target object if any of the following hold:

- f is a null function pointer value, or
- f is a null member function pointer value, or
- remove_cvref(t( )) is not a specialization of in_place_type_t<T>
- Otherwise, this has a target object of type VT direct-non-list-initialized with std::forward<F(f())>

Throws: Any exception thrown by the initialization of the target object. May throw bad_alloc unless VT is a function pointer or a specialization of reference wrapper.
Acknowledgements

Thanks to RISC Software GmbH for supporting this work. Thanks to Peter Kulczycki for proof reading and discussions. Thanks to Matt Calabrese for helping to get conversions to move_only_function to work. Thanks to Tomasz Kamiński for coming up with wording for the double wrapping avoidance.