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

1 RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at
Motivation

C++11 added function, a type-erased function wrapper that can represent any copyable callable matching the function signature \( R(\text{Args}...) \). 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 \( R(\text{Args}...) \text{ const} \oplus \text{&}|\&\oplus \text{noexcept}_\oplus \) 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.

Concurrently, [P0792R10] introduced function_ref, a type-erased non-owning reference to any callable matching a function signature in the form of \( R(\text{Args}...) \text{ const} \oplus \text{noexcept}_\oplus \). 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 copyable

Additionally, as copyable_function is a strict superset of move_only_function, we provide conversion operators from the former to the latter. We prefer conversion operators in copyable_function to converting constructors in move_only_function as the latter is a more fundamental type that shouldn’t have to know about the more specialized one.

Open Questions

Deprecation of function
As copyable_function aims to supersede function, should the latter (including bad_function_call) be moved to Annex D with the adoption of this paper?

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](https://github.com/MFHava/P2548).

Proposed Wording
Wording is relative to [N4910](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4910.html). Additions are presented like this, removals like this.

[version.syn]
In [version.syn], add:

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

Adjust the placeholder value as needed to denote this proposal’s date of adoption.

[functional.syn]
In [functional.syn], in the synopsis, add the proposed class template:

```
// 22.10.17.4, move only wrapper
template<class... S> class move_only_function; // not defined
template<class R, class... ArgTypes>
    class move_only_function<R(ArgTypes...) cv ref noexcept(noex)>; // see below

// 22.10.17.5, copyable wrapper
template<class... S> class copyable_function; // not defined
template<class R, class... ArgTypes>
    class copyable_function<R(ArgTypes...) cv ref noexcept(noex)>; // see below

// 22.10.18, searchers
template<class ForwardIterator, class BinaryPredicate = equal_to<>>
class default_searcher;
```

[func.wrap]
In [func.wrap], insert the following section at the end of Polymorphic function wrappers:

```
22.10.17.5 Copyable wrapper
22.10.17.5.1 General
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.
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

22.10.17.5.2 Class template copyable_function
namespace std {
    template<class... S> class copyable_function; // not defined
    template<class R, class... ArgTypes>
        class copyable_function<R(ArgTypes...) cv ref noexcept(noex)>;
        public:
            using result_type = R;

            // 22.10.17.5.3, constructors, assignments, and destructors
            copyable_function() noexcept;
            copyable_function(nullptr_t) noexcept;
            copyable_function(const copyable_function&);
            copyable_function(copyable_function&&) noexcept;
            template<class F> copyable_function(F&&);
            template<class T, class... Args>
                explicit copyable_function(in_place_type_t<T>, Args&&...);
                explicit copyable_function(in_place_type_t<T>, initializer_list<Args>);

                copyable_function& operator=(const copyable_function&);
                copyable_function& operator=(copyable_function&&) noexcept;
                copyable_function& operator=(nullptr_t) noexcept;
                template<class F> copyable_function& operator=(F&&);

                ~copyable_function();
```
// 22.10.17.5.4, invocation
explicit operator bool() const noexcept;
    & operator[](Args...) cv ref noexcept(noeox);

// 22.10.17.5.5, conversion
explicit operator move_only_function<Args...>() const &
    operator move_only_function<Args...>() cv ref noexcept(0ex) && noexcept;

// 22.10.17.5.6, utility
void swap(copyable_function& f) noexcept;
friend void swap(copyable_function&, copyable_function&) noexcept;
friend bool operator==(const copyable_function&, nullptr_t) noexcept;

private:
    template<class VT
        static constexpr bool is_copy_constructible_v<VT>
            = is_constructible_v<VT, Args...> &&
            is_nothrow_constructible_v<VT> &&
            is_nothrow_invocable_r_v<R, VT> &&
            is_nothrow_invocable_r_v<R, VT, ArgTypes...> &&
            is_nothrow_invocable_r_v<R, VT inv-quals, ArgTypes...>;
    template<class F
        static constexpr bool is_copy_constructible_v<VT, F>
            = is_constructible_v<VT, F> &&
            is_nothrow_constructible_v<VT, F> &&
            is_nothrow_invocable_r_v<R, VT, ArgTypes...> &&
            is_nothrow_invocable_r_v<R, VT inv-quals, ArgTypes...>;

    template<class F
        copyable_function(F&& f);
    private:
        template<class VT
            static constexpr bool isCopyConstructibleFrom = see below; // exposition only

The copyable function class template provides polymorphic wrappers that generalize the notion of a callable object [22.10.3]
These wrappers can store, copy, move, and call arbitrary callable objects, given a call signature. Within this subclause, call-arg
is an argument pack with elements that have types ArgTypes&& ... respectively.
Recommended practice: Implementations should avoid the use of dynamically allocated memory for a small contained value.

Note 1: Such small object optimization can only be applied to a type VT for which is_nothrow_constructible_v<VT> is true.

22.10.17.5.3 Constructors, assignment, and destructor

[func.wrap.copy_ctor]

## template<class VT

    static constexpr bool isCopyConstructibleFrom = see below;
    if (noex is true, isCopyConstructibleFrom) is equal to:
        is nothrow invokeable r v c f, VT cv ref, ArgTypes... &
        is nothrow invokeable r v c f, VT inv-quals, ArgTypes... &
    Otherwise, isCopyConstructibleFrom is equal to:
        is_comparable r v c f, VT cv ref, ArgTypes... &&
        is_comparable r v c f, VT inv-quals, ArgTypes... &&

        copyable_function() noexcept;
    copyable_function(nullptr_t) noexcept;
    copyable_function(R(ArgTypes...), VT c f);
    Postconditions: *this has no target object.
    copyable_function(const copyable_function& f)
    Postconditions: *this has no target object if f had 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.
    copyable_function(copyable_function& f) noexcept;
    Postconditions: The target object of *this is the target object f had before construction, and f is in a valid state with an
unspecified value.

## template<class F

    copyable_function(F& f);
    let f be decay, tv f;
    Constraints:
        — remove cv ref tcv f is not the same as copyable function, and
        — remove cv ref t f is not a specialization of in place type t, and
        — is_comparable from tv f is true.
    Mandates:
        — is constructible v cv f, F is true, and
        — is copy constructible v cv f is true.
    Preconditions: f meets the Cpp17MoveConstructible requirements, and if is move constructible v cv f is true, VT meets the
Cpp17MoveConstructible 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 cv ref t f is a specialization of the copyable function class template, and f has no target object;
    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.
Of a specialization of reference wrapper.

## template<class T, class... Args

    explicit copyable_function(in place_type tcv f, Args&&... args);
    let f be decay, tv f;
    Constraints:
        — is constructible v cv f, Args... is true, and
        — is_comparable from tv f is true.
    Mandates:
        — VT is the same type as T, and
        — is copy constructible v cv f is true.
    Preconditions: f meets the Cpp17MoveConstructible requirements, and if is move constructible v cv f is true, VT meets the
Cpp17MoveConstructible requirements.
Acknowledgements

Thanks to RISC Software GmbH for supporting this work. Thanks to Peter Kulczycki for proof reading and discussions.