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></th>
<th>Before</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto lambda{<a href="">&amp;</a> {</td>
<td>auto lambda{<a href="">&amp;</a> {</td>
<td>auto lambda{<a href="">&amp;</a> {</td>
</tr>
<tr>
<td>/&quot;const&quot;/</td>
<td>}</td>
<td>/&quot;const&quot;/</td>
</tr>
<tr>
<td>} (</td>
<td>const auto &amp; ref(func);</td>
<td>const auto &amp; ref(func);</td>
</tr>
<tr>
<td>}</td>
<td>func();</td>
<td>func0();</td>
</tr>
<tr>
<td>}</td>
<td>ref();</td>
<td>ref0();</td>
</tr>
<tr>
<td>}</td>
<td>func();</td>
<td>func0();</td>
</tr>
<tr>
<td>}</td>
<td>ref(); //operator() is NOT const!</td>
<td>ref0(); //operator() is NOT const!</td>
</tr>
<tr>
<td>}</td>
<td>//this is the infamous constness-bug</td>
<td></td>
</tr>
<tr>
<td>auto lambda{<a href="">&amp;</a></td>
<td>auto lambda{<a href="">&amp;</a> mutable {</td>
<td>auto lambda{<a href="">&amp;</a> mutable {</td>
</tr>
<tr>
<td>mutable {</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>} (</td>
<td>const auto &amp; ref(func);</td>
<td>const auto &amp; ref(func);</td>
</tr>
<tr>
<td>}</td>
<td>func();</td>
<td>func1();</td>
</tr>
<tr>
<td>}</td>
<td>ref(); //operator() is const!</td>
<td>ref1(); //operator() is const!</td>
</tr>
<tr>
<td>}</td>
<td></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.
- Removed open question on the deprecation of function.
- Replaced previously proposed conversion operators to move_only_function.

---

1 RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at
- Added section on conversions between standard library polymorphic function wrappers.
- Added section on potential allocator support.

**Motivation**

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

<table>
<thead>
<tr>
<th>From</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>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>move_only_function</td>
<td>❌</td>
<td></td>
<td>🔴</td>
<td>✔️</td>
</tr>
<tr>
<td>copyable_function</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>function_ref</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.

Impact on the Standard
This proposal is a pure library addition. It introduces a new class template and adds an optimization requirement to an existing class template.

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]. Additions are presented like `this`, removals like `this`.

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

```cpp
#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:

```cpp
// 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:

```cpp
22.10.17.5 Copyable wrapper
------------------------------- [func.wrap.copy]
22.10.17.5.1 General [func.wrap.copy.general]
The header provides partial specializations of copyable_function for each combination of the possible replacements of the placeholders cv, ref, and noex where:

1. cv is either const or empty,
2. ref is either & or &&, or empty, and
3. 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
```
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&&...);
        template<class T, class U, class... Args>
            explicit copyable_function(in_place_type_t<T>, initializer_list<U>, Args&&...);
        copyable_function& operator=(const copyable_function&);
        copyable_function& operator=(copyable_function&&);
        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;
        R operator()(ArgTypes...) cv ref noexcept(noex);

        // 22.10.17.5.5, utility
        void swap(copyable_function&) 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_callable_from = 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-args 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: Such small-object optimization can only be applied to a type T for which is_nothrow_constructible_v<T> is true.)

22.10.17.5.3 Constructors, assignment, and destructor

The copyable_function class template (see below) is callable-from = see above.

If noex is true, is_callable-from<VT> is equal to
is_nothrow_invocable_r_v<R, VT cv ref, ArgTypes...> &&
is_nothrow_invocable_r_v<R, VT inv-quals, ArgTypes...>
Otherwise, is_callable-from<VT> is equal to
is_invocable_r_v<R, VT cv ref, ArgTypes...> &&
is_invocable_r_v<R, VT inv-quals, ArgTypes...>

copyable_function() noexcept;
copyable_function(nullptr_t) noexcept;
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 V be decay<Ex<F>>
Constraints:
1. remove_cvref_t<F> is not the same as copyable_function and
2. remove_cvref_t<F> is not a specialization of in_place_type_t, and
3. is_callable_from<V> is true.
Mandates:
4. is_constructible_v<V, F> is true, and
5. is_copy_constructible_v<V, F> is true.
Postconditions: f meets the C++17Destructible requirements, and if is_move_constructible_v<V, F> is true, V meets the C++17MoveConstructible 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(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 std::bad_alloc unless VT is a function pointer or a specialization of reference_wrapper.

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

```cpp
template<class T, class... Args>
explicit copyable_function(in_place_type_t<T>, initializer_list<Args> ilist, Args&&... args);
```

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

Effects: Equivalent to: copyable_function(f).swap(*this);
Returns: *this

copyable_function& operator=(const copyable_function& f);
Effects: Equivalent to: copyable_function(f).swap(*this);
Returns: *this

copyable_function& operator=(copyable_function&& f);
Effects: Equivalent to: copyable_function(std::move(f)).swap(*this);
Returns: *this

copyable_function& operator=(nullptr_t) noexcept;
Effects: Destroys the target object of *this, if any.
Returns: *this

copyable_function& operator=(copyable_function& f);
Effects: Equivalent to: copyable_function(std::forward<ArgTypes>(args)...) swap(*this);
Returns: *this

copyable_function& operator=(nullptr_t) noexcept;
Effects: Destroys the target object of *this, if any.
Returns: *this
```
Returns: true if f has no target object, otherwise false.

[func.wrap.movector]
In [func.wrap.movector], insert the following:

```cpp
template<class F> move_only_function(F&& f);

Let VT be decay_t<F>.

Constraints:
(5.1) — remove_cvref_t<F> is not the same as move_only_function, and
(5.2) — remove_cvref_t<F> is not a specialization of in_place_type_t, and
(5.3) — is callable from VT is true.

Mandates: is_constructible_v<VT, F> is true.

Preconditions: VT meets the Cpp17Destructible requirements, and if is_move_constructible_v<VT> is true, VT meets the Cpp17MoveConstructible requirements.

Postconditions: *this has no target object if any of the following hold:
(I.1) — f is a null function pointer value, or
(I.2) — f is a null member function pointer value, or
(I.3) — remove_cvref_t<F> is a specialization of the move_only_function class template, and f has no target object.

Otherwise, *this has a target object of type VT direct-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.<F is copyable_function<Arg1...> cv Ref noexcept(noexcept)>.
```

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.