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(func0);</td>
</tr>
<tr>
<td>func();</td>
<td>func0();</td>
</tr>
<tr>
<td>ref();</td>
<td>ref0();</td>
</tr>
<tr>
<td></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(func1);</td>
</tr>
<tr>
<td>func();</td>
<td>func1();</td>
</tr>
<tr>
<td>ref(); //operator() is const!</td>
<td>ref1();</td>
</tr>
<tr>
<td>//this is the infamous constness-bug</td>
<td>//operator() is NOT const!</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.

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]).

---

1 RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at
[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...) constexpr &|&& noexcept 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(Args...) constexpr noexcept. 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 (including the changes proposed in [P2511R2]) 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.
Proposed Wording
Wording is relative to [N4910]. 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

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:

1. If ref is empty, let inv-quals be cv.
2. 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<auto f> copyable_function(nontype_t<f>) noexcept;
        template<class F> copyable_function(F&&);
        template<auto f, class T> copyable_function(nontype_t<f>, 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&&...);
        template<auto f, class T, class U, class... Args>
          explicit copyable_function(nontype_t<f>, in_place_type_t<T>, initializer_list<U>, Args&&...);

        copyable_function& operator=(const copyable_function&);
        copyable_function& operator=(copyable_function&&) noexcept;
        template<class F> copyable_function& operator=(nullptr_t);
        template<class F> copyable_function& operator=(F&&);
    }
}
```
```cpp
```
— remove `cvref_t<T>` 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 or a specialization of `reference_wrapper`.

```cpp
template<auto f, class T, class... Args>
explicit copyable_function(nontype_t<f>, T&& x);

template<auto f, class T, class... Args>
explicit copyable_function(nontype_t<f>, in_place_type_t<T>, initialize_list<U> ilist, Args&&... args);

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

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

Returns: `*this`.

```cpp
template<class F, copyable_function& f>
copyable_function& operator=(F&& f);
```

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

Returns: `*this`.

```cpp
copyable_function& operator=(nullptr_t) noexcept;
```

Effects: Equivalent to:

```cpp
*this = f;
```

Returns: `*this`.

```cpp
copyable_function& operator=(const copyable_function& f);
```

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

Returns: `*this`.

```cpp
copyable_function& operator=(copyable_function&& f);
```

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

Returns: `*this`.

```cpp
copyable_function& operator=(nullptr_t) noexcept;
```

Effects: Equivalent to:

```cpp
*this = f;
```

Returns: `*this`.

```cpp
template<class F> copyable_function& operator=(F&& f);
```

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

Returns: `*this`.

```cpp
~copyable_function();
```

Postconditions: `*this` has a target object of type `VT` direct-non-list-initialized with `std::forward<F>(f)`.

Effects: Any exception thrown by the initialization of the target object. May throw `bad_alloc` unless `VT` is a pointer or a specialization of `reference_wrapper`.

```cpp
explicit copyable_function(std::forward<F>(f)).swap(*this);
```

Effects: Equivalent to:

```cpp
call(f, d, call-args...);
```

Returns: `*this`.

```cpp
copyable_function f1, f2;
```

Effects: Destroys the target object of `*this` if any.

Returns: `*this`.

```cpp
template<class F> copyable_function& operator=(F&& f);
```

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

Returns: `*this`.

```cpp
~copyable_function();
```

Effects: Destroys the target object of `*this` if any.
22.10.17.5.4 Invocation

```cpp
explicit operator bool() const noexcept;
```

**Returns**: `true` if `*this` has a target object, otherwise `false`.

```cpp
operator()() const cv ref noexcept(noex);
```

**Preconditions**: `*this` has a target object.

**Effects**: Equivalent to:

```
return INVOKE<R>(static_cast<F invquals>(f), std::forward<ArgTypes>(args));
```

where `f` is an lvalue designating the target object of `*this` and `F` is the type of `f`.

22.10.17.5.5 Conversions

```cpp
explicit operator move_only_function<R(ArgTypes...) cv ref noexcept(noex)>() const &;
```

**Returns**: An object containing a copy of the target object of `*this`, if any.

**Throws**: Any exception thrown by copying the target object. May throw `bad_alloc`.

```cpp
operator move_only_function<R(ArgTypes...) cv ref noexcept(noex)>() && noexcept;
```

**Postconditions**: `*this` has no target object.

**Returns**: An object containing the target object `*this` had before the call, if any.

22.10.17.5.6 Utility

```cpp
void swap(copyable_function& other) noexcept;
```

**Effects**: Exchanges the target objects of `*this` and `other`.

```cpp
friend void swap(copyable_function& f1, copyable_function& f2) noexcept;
```

**Effects**: Equivalent to `f1.swap(f2)`.

```cpp
friend bool operator==(const copyable_function& f, nullptr_t) noexcept;
```

**Returns**: `true` if `f` has no target object, otherwise `false`.

---

**Acknowledgements**

Thanks to [RISC Software GmbH](https://www.riscsoft.de) for supporting this work. Thanks to Peter Kulczycki for proof reading and discussions.