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) func(lambda);</td>
<td>copyable_function&lt;void(void) 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(func); //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) func(lambda);</td>
<td>copyable_function&lt;void(void) 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>func();</td>
</tr>
<tr>
<td>ref(); //operator() is const!</td>
<td>ref(func); //operator() is NOT const!</td>
</tr>
<tr>
<td>//this is the infamous constness-bug</td>
<td></td>
</tr>
</tbody>
</table>

Revisions
R0: Initial version

Motivation
C++11 added function, a type-erased function wrapper that can represent any copyable callable matching the function signatures 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.

---

1 RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at
Concurrently, [P0792R10] introduced function_ref, a type-erased non-owning reference to any call-able matching a function signature in the form of \( \text{R(Args...) const\_noexcept\_op} \). 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 qualifi-ers 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 con-stness-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 copya-ble\_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

**Open Questions**

**Conversion to move\_only\_function**

Given that copyable\_function is a strict superset of move\_only\_function, should it provide conversion operators to move\_only\_function?

**Deprecation of function**

As copyable\_function aims to supersede function, should the latter (including bad\_func-tion\_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.
In [functional.syn], in the synopsis, add the proposed class template:

```cpp
// 22.10.17, polymorphic function wrappers
class bad_function_call;
template<class> class function; // not defined
template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

// 22.10.17.3.8, specialized algorithms
template<class R, class... ArgTypes>
void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;

// 22.10.17.3.7, null pointer comparison operator functions
template<class R, class... ArgTypes>
bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

// 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;
```

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

```cpp
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 &&, 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&&...);
    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&&) 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;
```
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.

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

Preconditions: Each small object optimization can only be applied to a type that is not

Constraints: 

Postconditions: 

Mandates: 

Declarations: 

In-place construction: 

Explicit copyable function:

Implements: 

Direct non-list-initialized construction:

Reference wrapper:

Forward construction:

Object optimization: 

Copyable function:

Copyable function (copy-constructible):

Copyable function (move-constructible):

Copyable function (null):

Copyable function (any):
is_copy_constructible_v<VT> 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 a target object of type VT direct-non-list-initialized with ilist, std::forward<Args>(args)... 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.

Preconditions:

Postconditions:

* this has a target object of type VT direct-non-list-initialized with ilist, std::forward<Args>(args)...

Postconditions: 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.