Document number:	P2548R6
Date:	2023-06-15
Project:	Programming Language C++
Audience:	LWG
Reply-to:	Michael Florian Hava[1] <mfh.cpp@gmail.com>

copyable_function

Abstract

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

Tony Table

Before	Proposed

`auto lambda{[&]() /*const*/ { … }};`
 	
`function<void(void)> func{lambda};` ✓
`const auto & ref{func};`
 	
`func();` ✓
`ref();` ✓
	`auto lambda{[&]() /*const*/ { … }};`
	
	`copyable_function<void(void)> func0{lambda};` ✓
	`const auto & ref0{func0};`
	
	`func0();` ✓
	`ref0(); //operator() is NOT const!` ✗
	
	`copyable_function<void(void) const> func1{lambda};` ✓
	`const auto & ref1{func1};`
	
	`func1();` ✓
	`ref1(); //operator() is const!` ✓

`auto lambda{[&]() mutable { … }};`
 	
`function<void(void)> func{lambda};` ✓
`const auto & ref{func};`
 	
`func();` ✓
`ref(); //operator() is const!` ⁉✓
`      //this is the infamous constness-bug`
	`auto lambda{[&]() mutable { … }};`
	
	`copyable_function<void(void)> func{lambda};` ✓
	`const auto & ref{func};`
	
	`func();` ✓
	`ref(); //operator() is NOT const!` ✗
	
	`copyable_function<void(void) const> tmp{lambda};` ✗

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.

		To			
		`function`	`move_only_function`	`copyable_function`	`function_ref`
From	`function`		✅	✅	✅
	`move_only_function`	❌		❌	✅
	`copyable_function`	✅	✅		✅
	`function_ref`	✅	✅	✅	

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 YYYYMML //also in <functional>
```

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

[functional.syn]

22.10.2 Header `<functional>` synopsis **[functional.syn]**

```
namespace std {
…
  // [func.wrap.move], 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

  // [func.wrap.copy], 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

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

[func.wrap.general]

22.10.17.1 General **[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 `x` 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*:

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

- end example]

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

22.10.17.2 Class `bad_function_call` **[func.wrap.badcall]**

[func.wrap.copy]

[DRAFTING NOTE: Add a new section in *[func.wrap]*]

22.10.17.?? Copyable wrapper **[func.wrap.copy]**

22.10.17.??.1 General **[func.wrap.copy.general]**

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

(1.1) — *cv* is either const or empty,

(1.2) — *ref* is either &, &&, or empty, and

(1.3) — *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:

(2.1) — If *ref* is empty, let *`inv-quals`* be *cv*&,

(2.2) — otherwise, let *`inv-quals`* be *`cv ref`*.

22.10.17.??.2 Class template `copyable_function` **[func.wrap.copy.class]**

```
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(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();

    // [func.wrap.copy.inv], invocation
    explicit operator bool() const noexcept;
    R operator()(ArgTypes...) cv ref noexcept(noex);

    // [func.wrap.copy.util], 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
  };
}
```

1 The copyable_function class template provides polymorphic wrappers that generalize the notion of a callable object (***[func.def]***). 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.

2 *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 T for which is_nothrow_move_constructible_v<T> is true. — *end note*]

22.10.17.??.3 Constructors, assignment, and destructor [func.wrap.copy.ctor]

```
template<class VT>
  static constexpr bool is-callable-from = see below;
```

1 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;
```

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

```
copyable_function(const copyable_function& f);
```

3 *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.

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

```
copyable_function(copyable_function&& f) noexcept;
```

5 *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);
```

6 Let VT be decay_t<F>.

7 *Constraints*:

(7.1) — remove_cvref_t<F> is not the same as copyable_function, and

(7.2) — remove_cvref_t<F> is not a specialization of in_place_type_t, and

(7.3) — *is-callable-from*<VT> is true.

8 *Mandates*:

(8.1) — is_constructible_v<VT, F> is true, and

(8.2) — is_copy_constructible_v<VT> is true.

9 *Preconditions*: VT meets the *Cpp17Destructible* and *Cpp17CopyConstructible* requirements.

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

(10.1) — f is a null function pointer value, or

(10.2) — f is a null member function pointer value, or

(10.3) — 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).

11 *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.

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

12 Let VT be decay_t<T>.

13 *Constraints:*

(13.1) — is_constructible_v<VT, Args...> is true, and

(13.2) — *is-callable-from*<VT> is true.

14 *Mandates*:

(14.1) — VT is the same type as T, and

(14.2) — is_copy_constructible_v<VT> is true.

15 *Preconditions*: VT meets the *Cpp17Destructible* and *Cpp17CopyConstructible* requirements.

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

17 *Throws*: 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.

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

18 Let VT be decay_t<T>.

19 *Constraints*:

(19.1) — is_constructible_v<VT, initializer_list<U>&, Args...> is true, and

(19.2) — *is-callable-from*<VT> is true.

20 *Mandates*:

(20.1) — VT is the same type as T, and

(20.2) — is_copy_constructible_v<VT> is true.

21 *Preconditions*: VT meets the *Cpp17Destructible* and *Cpp17CopyConstructible* requirements.

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

23 *Throws*: 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.

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

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

25 *Returns*: *this.

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

26 *Effects*: Equivalent to: copyable_function(std::move(f)).swap(*this);

27 *Returns*: *this.

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

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

29 *Returns*: *this.

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

30 *Effects*: Equivalent to: copyable_function(std::forward<F>(f)).swap(*this);

31 *Returns*: *this.

```
~copyable_function();
```

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

22.10.17.??.4 Invocation [func.wrap.copy.inv]

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

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

```
R operator()(ArgTypes... args) cv ref noexcept(noex);
```

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

3 *Effects*: Equivalent to:

```
    return INVOKE<R>(static_cast<F inv-quals>(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 Utility [func.wrap.copy.util]

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

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

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

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

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

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

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.