Document number:	P2548R1
Date:	2022-10-12
Project:	Programming Language C++
Audience:	LEWG
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*/ { … }};`	`auto lambda{[&]() /*const*/ { … }};`
`function<void(void)> func{lambda};` ✔ `const auto & ref{func};`	`copyable_function<void(void)> func0{lambda};` ✔ `const auto & ref0{func0};`
`func();` ✔ `ref();` ✔	`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 { … }};`	`auto lambda{[&]() mutable { … }};`
`function<void(void)> func{lambda};` ✔ `const auto & ref{func};`	`copyable_function<void(void)> func{lambda};` ✔ `const auto & ref{func};`

`func();` ✔
`ref(); //operator() is const!` ⁉✔
`//this is the infamous constness-bug`
	`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`.

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...)` $\texttt{const}_{\text{op}}$ $\texttt{(\&|\&\&)}_{\text{op}}$ $\texttt{noexcept}_{\text{op}}$ 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.

Concurrently, [P0792R10] introduced `function_ref`, a type-erased non-owning reference to any callable matching a function signature in the form of `R(Args...)` $\texttt{const}_{\text{op}}$ $\texttt{noexcept}_{\text{op}}$. 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 (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 YYYYMML //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 **[func.wrap.copy]**

22.10.17.5.1 General **[func.wrap.copy.general]**

1 The header `<functional>` 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*.

22.10.17.5.2 Class template `copyable_function` **[func.wrap.copy.class]**

```
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>, T&&);
    template<class T, class... Args>
      explicit copyable_function(in_place_type_t<T>, Args&&...);
    template<auto f, class T, class... Args>
      explicit 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&&);
    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, conversions
    explicit operator move_only_function<R(ArgTypes...) cv ref noexcept(noex)>() const &;
    operator move_only_function<R(ArgTypes...) cv ref noexcept(noex)>() && noexcept;

    // 22.10.17.5.6, 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... T>
      static constexpr bool is-invocable-using = see below;    //exposition only
    template<class VT>
      static constexpr bool is-callable-from = see below;      //exposition only
    template<auto f, class VT>
      static constexpr bool is-callable-as-if-from = see below; //exposition only
  };
}
```

1 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.

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_constructible_v<T>` is true. — *end note*]

22.10.17.5.3 Constructors, assignment, and destructor [func.wrap.copy.ctor]

```
template<class... T>
  static constexpr bool is-invocable-using = see below;
```

1 If *noex* is `true`, *is-invocable-using*`<T...>` is equal to:
`is_nothrow_invocable_r_v<R, T..., ArgTypes...>`
Otherwise, *is-invocable-using*`<T...>` is equal to:
`is_invocable_r_v<R, T..., ArgTypes...>`

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

2 *is-callable-from*`<VT>` is equal to:
`is_invocable_r_v<R, T..., ArgTypes...>` &&
is-invocable-using`<VT cv ref>` &&
is-invocable-using`<VT inv-quals>`

```
template<auto f, class VT>
  static constexpr bool is-callable-as-if-from = see below;
```

3 *is-callable-as-if-from*`<f, VT>` is equal to:
is-invocable-using`<decltype(f), VT inv-quals>`

```
copyable_function() noexcept;
copyable_function(nullptr_t) noexcept;
```

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

```
template<auto f> copyable_function(nontype_t<f>) noexcept;
```

5 *Constraints*: *is-invocable-using*`<decltype(f)>` is `true`.

6 *Postconditions*: `*this` has a target object. Such an object and `f` are template-argument-equivalent [temp.type].

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

7 *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`.

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

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

9 *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);
```

10 Let `VT` be `decay_t<F>`.

11 *Constraints*:

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

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

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

12 *Mandates*:

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

(12.2) — `is_copy_constructible_v<VT>` is `true`.

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

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

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

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

(14.3) — `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)`.

15 *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<auto f, class T> copyable_function(nontype_t<f>, T&& x);
```

16 Let `VT` be `decay_t<T>`.

17 *Constraints*: *`is-callable-as-if-from`*`<f, VT>` is `true`.

18 *Mandates*:

(18.1) — `is_constructible_v<VT, T>` is `true`, and

(18.2) — `is_copy_constructible_v<VT>` is `true`.

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

20 *Postconditions*: `*this` has a target object `d` of type `VT` direct-non-list-initialized with `std::forward<T>(x)`. `d` is hypothetically usable in a call expression, where `d(`*`call-args`*`...)` is expression equivalent to `invoke(f, d,` *`call-args`*`...)`.

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

22 Let `VT` be `decay_t<T>`.

23 *Constraints*:

(23.1) — `is_constructible_v<VT, Args…>` is `true`, and

(23.2) — *`is-callable-from`*`<VT>` is `true` for the first form or *`is-callable-as-if-from`*`<f, VT>` is `true` for the second form.

24 *Mandates*:

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

(24.2) — `is_copy_constructible_v<VT>` is `true`.

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

26 *Postconditions*: `*this` has a target object `d` of type `VT` direct-non-list-initialized with `std::forward<Args>(args)...`. With the second form, `d` is hypothetically usable in a call expression, where `d(`*`call-args`*`...)` is expression equivalent to `invoke(f, d,` *`call-args`*`...)`.

27 *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);
template<auto f, class T, class U, class... Args>
  explicit copyable_function(nontype_t<f>, in_place_type_t<T>, initializer_list<U> ilist, Args&&... args);
```

28 Let `VT` be `decay_t<T>`.

29 *Constraints*:

(29.1) — `is_constructible_v<VT, initializer_list<U>&, Args…>` is `true`, and

(29.2) — *`is-callable-from`*`<VT>` is `true` for first form or *`is-callable-as-if-from`*`<f, VT>` is `true` for the second form.

30 *Mandates*:

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

(30.2) — `is_copy_constructible_v<VT>` is `true`.

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

32 *Postconditions*: `*this` has a target object `d` of type `VT` direct-non-list-initialized with `ilist, std::forward<Args>(args)...`. With the second form, `d` is hypothetically usable in a call expression, where `d(`*`call-args`*`...)` is expression equivalent to `invoke(f, d,` *`call-args`*`...)`.

33 *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);
```

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

35 *Returns*: `*this`.

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

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

37 *Returns*: `*this`.

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

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

39 *Returns*: `*this`.

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

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

41 *Returns*: `*this`.

```
~copyable_function();
```

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

22.10.17.5.4 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`.

22.10.17.5.5 Conversions **[func.wrap.copy.conv]**

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

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

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

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

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

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

22.10.17.5.6 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.