Document number:	P2548R0
Date:	2022-07-12
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};` ✓	`copyable_function<void(void)> func0{lambda};` ✓
`const auto & ref{func};`	`const auto & ref0{func0};`
`func();` ✓	`func0();` ✓
`ref();` ✓	`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};` ✓	`copyable_function<void(void)> func{lambda};` ✓
`const auto & ref{func};`	`const auto & ref{func};`
`func();` ✓	`func();` ✓
`ref(); //operator() is const!` ⁉✓	`ref(); //operator() is NOT const!` ✗
`//this is the infamous constness-bug`	
	`copyable_function<void(void) const> tmp{lambda};` ✗

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 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 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_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, 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;
```

[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 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<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
  };
}
```

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.

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 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`.

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

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

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

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

(20.2) — is_copy_constructible_v<VT> is true.

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

22.10.17.5.4 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 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.