Document number: P2721R1
Date: 2026-06-17
Project: Programming Language C++
Audience: LEWG
Reply-to: Michael Florian Hava[1] <mfh.cpp@gmail.com>

Deprecating `function` in C++29

Abstract
This paper proposes deprecating `function` (and associated entities) as it has unresolvable API design issues and has been superseded by `copyable_function`.

Revisions
R0: Initial version

R1: Retargeting paper to C++29, added proposed wording.

Motivation
C++11 added `function`, a type-erased function wrapper that can represent any *copyable* callable matching a given function signature. Since its introduction, there have been identified several issues with its design (see [N4159]) – including the famous constness-bug:

```
//the constness bug of std::function:

//consider:
auto lambda = [&]() mutable { … };
l(); ✔️
const auto & r{lambda};
r(); ❌ //lambda::operator() is mutable => can't be called via const &!

//but:
function<void(void)> func{lambda};
func(); ✔️
const auto & cref{func};
cref(); ✔️⚡ //func::operator() is const => can invoke mutable lambda through const &!
             // this breaks the fundamental guarantee that concurrently calling const member functions is safe!
```

As `function` was incompatible (and could not be made compatible) with *non-copyable* functors, [P0288] introduced `move_only_function`. The design of which not only drops the *copyable* requirement but also fixes bugs present in `function`, removes RTTI dependence and adds support for `const`-, `noexcept`- and ref-qualifiers.

Semi-concurrently [P0792] introduced `function_ref` as a non-owning reference to a functor. Like `move_only_function` it does not depend on RTTI and supports `const`- and `noexcept`-qualifiers[2].

After `move_only_function` was approved for C++23 and with `function_ref` targeting C++26, there were serious inconsistencies between the *polymorphic function wrappers* of the standard library. Whilst some of the new features[3] could have been back-ported to `function`, it was impossible to reach feature parity without an API break. To improve the situation, [P2548]

---

[1] RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at

[2] There is no support for ref-qualifiers as `function_ref` itself is a reference type.

[3] Primarily `noexcept`- and ref-qualifier support.

introduced `copyable_function` (design consistent with `move_only_function`) as a replacement of `function`.

With the approval of `copyable_function` and `function_ref` for C++26 there remains little reason to keep `function` in the blessed part of the standard library. Furthermore it has been pointed out multiple times (both by [P3023] and externally) that the current state of *polymorphic function wrappers* in the standard library is complicated - growing from one to four distinct classes within two standard cycles.

Deprecating `function` would lead to a more unified standard library design and would send a clear guidance to users which types to use in new codes.

Previous Polls

R0 of this proposal was originally submitted in 2024 for consideration in the C++26 cycle and was reviewed in Tokyo (2024-03-22). The following two polls were taken:

> **Poll:** We are interested in deprecating std::function at some future point.
>
> | SF | F | N | A | SA |
> |---|---|---|---|---|
> | 10 | 5 | 3 | 2 | 2 |
>
> **Outcome:** Consensus in favor

> **Poll:** Forward “P2721R0: Deprecating function” to LWG for C++26 (to be confirmed by an electronic poll).
>
> | SF | F | N | A | SA |
> |---|---|---|---|---|
> | 6 | 3 | 3 | 6 | 4 |
>
> **Outcome:** No consensus for change

Impact on the Standard

`function` and `bad_function_call` are moved to Annex D without a change in functionality.

Proposed Wording

Wording is relative to [N5046]. Additions are presented like this, removals like ~~this~~ and drafting notes like **this**.

[functional.syn]

??.??.? Header <functional> synopsis **[functional.syn]**

```
…

namespace std {
  …
  // [func.wrap], polymorphic function wrappers
  // [func.wrap.badcall], class bad_function_call
  class bad_function_call;

  // [func.wrap.func], class template function
  template<class> class function;        // not defined
  template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

  // [func.wrap.func.alg], function specialized algorithms
  template<class R, class... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;

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

  // [func.wrap.move], move-only wrapper
  …
}
```

…

[func.wrap.general]

??.??.??.? 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`, `move_only_function`, or `function_ref`, such that the target object `x` of `t` has a type that is a specialization of ~~`function,`~~ `copyable_function`, `move_only_function`, or `function_ref`. 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.

[depr.func]

Create a new entry in Annex D with the following content.

D.?? Deprecated `function` **[depr.func]**

D.??.1 Header <functional> synopsis **[depr.func.syn]**

The header `<functional>` has the following additions:

```
namespace std {
  // [depr.func.badcall], class bad_function_call
  class bad_function_call;

  // [depr.func.wrap], class template function
  template<class> class function;        // not defined
  template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

  // [depr.func.alg], function specialized algorithms
  template<class R, class... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;

  // [depr.func.nullptr], function null pointer comparison operator functions
  template<class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
}
```

D.??.2 General **[depr.func.general]**

1 In **[func.wrap.general]/2** both `t` and `x` can also be of a type that is a specialization of `function`.
[DRAFTING NOTE: The intention here is to replicate the behavior from before the deprecation.]

[func.wrap.badcall] and [func.wrap.func]

Change these two sections to make them subsections of [depr.func] and place them after [depr.func.general].

~~??.??.??.?~~D.??.3 Class `bad_function_call` **~~[func.wrap.badcall]~~**[depr.func.badcall]

1 An exception of type `bad_function_call` is thrown by `function::operator()` (**~~[func.wrap.func.inv]~~**[depr.func.wrap.inv]) when the function wrapper object has no target.

```
namespace std {
  class bad_function_call : public exception {
  public:
    // see ***[exception]*** for the specification of the special member functions
    const char* what() const noexcept override;
  };

const char* what() const noexcept override;
```

2 *Returns*: An implementation-defined NTBS.

~~??.??.??.?~~D.??.4 Class template `function` **~~[func.wrap.func]~~**[depr.func.wrap]

~~??.??.??.?.1~~D.??.4.1 General **~~[func.wrap.func.general]~~**[depr.func.wrap.general]

```
  template<class R, class... ArgTypes>
  class function<R(ArgTypes...)> {
  public:
    using result_type = R;

    // ~~[func.wrap.func.con]~~[depr.func.wrap.con], construct/copy/destroy
    function() noexcept;
    function(nullptr_t) noexcept;
    function(const function&);
    function(function&&) noexcept;
    template<class F> function(F&&);

    function& operator=(const function&);
    function& operator=(function&&);
    function& operator=(nullptr_t) noexcept;
    template<class F> function& operator=(F&&);
    template<class F> function& operator=(reference_wrapper<F>) noexcept;

    ~function();

    // ~~[func.wrap.func.mod]~~[depr.func.wrap.mod], function modifiers
    void swap(function&) noexcept;

    // ~~[func.wrap.func.cap]~~[depr.func.wrap.cap], function capacity
    explicit operator bool() const noexcept;

    // ~~[func.wrap.func.inv]~~[depr.func.wrap.inv], function invocation
    R operator()(ArgTypes...) const;

    // ~~[func.wrap.func.targ]~~[dep.func.wrap.targ], function target access
    const type_info& target_type() const noexcept;
    template<class T>       T* target() noexcept;
    template<class T> const T* target() const noexcept;
  };

  template<class R, class... ArgTypes>
    function(R(*)(ArgTypes...)) -> function<R(ArgTypes...)>;

  template<class F> function(F) -> function<see below>;
}
```

1 The `function` class template provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary callable objects (**[func.def]**), given a call signature (**[func.def]**).

2 The `function` class template is a call wrapper (**[func.def]**) whose call signature (**[func.def]**) is `R(ArgTypes...)`.

3 [*Note 1*: The types deduced by the deduction guides for `function` might change in future revisions of C++. — *end note*]

~~??.??.??.?.2~~D.??.4.2 Constructors and destructor **~~[func.wrap.func.con]~~**[depr.func.wrap.con]

```
function() noexcept;
```

1 *Postconditions*: `!*this`.

```
function(nullptr_t) noexcept;
```

2 *Postconditions*: `!*this`.

```
function(const function& f);
```

3 *Postconditions*: `!*this` if `!f`; otherwise, the target object of `*this` is a copy of the target object of `f`.

4 *Throws*: Nothing if `f`'s target is a specialization of `reference_wrapper` or a function pointer. Otherwise, may throw `bad_alloc` or any exception thrown by the copy constructor of the stored callable object.

5 *Recommended practice*: Implementations should avoid the use of dynamically allocated memory for small callable objects, for example, where `f`'s target is an object holding only a pointer or reference to an object and a member function pointer.

```
function(function&& f) noexcept;
```

6 *Postconditions*: If `!f`, `*this` has no target; otherwise, the target of `*this` is equivalent to the target of `f` before the construction, and `f` is in a valid state with an unspecified value.

7 *Recommended practice*: Implementations should avoid the use of dynamically allocated memory for small callable objects, for example, where `f`'s target is an object holding only a pointer or reference to an object and a member function pointer.

```
template<class F> function(F&& f);
```

8 Let `FD` be `decay_t<F>`.

9 *Constraints*:

(9.1) — `is_same_v<remove_cvref_t<F>, function>` is `false`, and

(9.2) — `is_invocable_r_v<R, FD&, ArgTypes...>` is `true`.

10 *Mandates*:

(10.1) — `is_copy_constructible_v<FD>` is `true`, and

(10.2) — `is_constructible_v<FD, F>` is `true`.

11 *Preconditions*: `FD` meets the *Cpp17CopyConstructible* requirements.

12 *Postconditions*: `!*this` is `true` if any of the following hold:

(12.1) — `f` is a null function pointer value.

(12.2) — `f` is a null member pointer value.

(12.3) — `remove_cvref_t<F>` is a specialization of the `function` class template, and `!f` is `true`.

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

14 *Throws*: Nothing if `FD` is a specialization of `reference_wrapper` or a function pointer type. Otherwise, may throw `bad_alloc` or any exception thrown by the initialization of the target object.

15 *Recommended practice*: Implementations should avoid the use of dynamically allocated memory for small callable objects, for example, where `f` refers to an object holding only a pointer or reference to an object and a member function pointer.

```
template<class F> function(F) -> function<see below>;
```

16 *Constraints*: `&F::operator()` is well-formed when treated as an unevaluated operand and either

(16.1) — `F::operator()` is a non-static member function and `decltype(&F::operator())` is either of the form `R(G::*)(A...)` *cv* $\&_{opt}$ $\texttt{noexcept}_{opt}$ or of the form `R(*)(G, A...)` $\texttt{noexcept}_{opt}$ for a type `G`, or

(16.2) — `F::operator()` is a static member function and `decltype(&F::operator())` is of the form `R(*)(A...)` $\texttt{noexcept}_{opt}$.

17 *Remarks*: The deduced type is `function<R(A...)>`.

18 [*Example 1*:

```
void f() {
  int i{5};
  function g = [&](double) { return i; };       // deduces function<int(double)>
}
```

— *end example*]

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

19 *Effects*: As if by `function(f).swap(*this);`

20 *Returns*: `*this`.

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

21 *Effects*: Replaces the target of `*this` with the target of `f`.

22 *Returns*: `*this`.

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

23 *Effects*: If `*this != nullptr`, destroys the target of `this`.

24 *Postconditions*: `!(*this)`.

25 *Returns*: `*this`.

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

26 *Constraints*: `is_invocable_r_v<R, decay_t<F>&, ArgTypes...>` is `true`.

27 *Effects*: As if by: `function(std::forward<F>(f)).swap(*this);`

28 *Returns*: `*this`.

```
template<class F> function& operator=(reference_wrapper<F> f) noexcept;
```

29 *Effects*: As if by: `function(f).swap(*this);`

30 *Returns*: `*this`.

```
~function();
```

31 *Effects*: If `*this != nullptr`, destroys the target of `this`.

~~??.??.??.?.3~~D.??.4.3 **Modifiers** ~~**[func.wrap.func.mod]**~~[depr.func.wrap.mod]

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

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

~~??.??.??.?.4~~D.??.4.4 **Capacity** ~~**[func.wrap.func.cap]**~~[depr.func.wrap.cap]

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

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

~~??.??.??.?.5~~D.??.4.5 **Invocation** ~~**[func.wrap.func.inv]**~~[depr.func.wrap.inv]

```
R operator()(ArgTypes... args) const;
```

1 *Returns*: `INVOKE<R>(f, std::forward<ArgTypes>(args)...)` (**[func.require]**), where `f` is the target object (**[func.def]**) of `*this`.

2 *Throws*: `bad_function_call` if `!*this`; otherwise, any exception thrown by the target object.

~~??.??.??.?.6~~D.??.4.6 **Target access** ~~**[func.wrap.func.targ]**~~[depr.func.wrap.targ]

```
const type_info& target_type() const noexcept;
```

1 *Returns*: If `*this` has a target of type T, `typeid(T)`; otherwise, `typeid(void)`.

```
template<class T>       T* target() noexcept;
template<class T> const T* target() const noexcept;
```

2 *Returns*: If `target_type() == typeid(T)` a pointer to the stored function target; otherwise a null pointer.

~~??.??.??.?.7~~D.??.4.7 **Null pointer comparison operator functions** ~~**[func.wrap.func.nullptr]**~~[depr.func.wrap.nullptr]

```
template<class R, class... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>& f, nullptr_t) noexcept;
```

1 *Returns*: `!f`.

~~??.??.??.?.8~~D.??.4.8 **Specialized algorithms** ~~**[func.wrap.func.alg]**~~[depr.func.wrap.alg]

```
template<class R, class... ArgTypes>
  void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2) noexcept;
```

1 *Effects*: As if by: `f1.swap(f2);`

Acknowledgements

Thanks to RISC Software GmbH for supporting this work. Thanks to Zhihao Yuan for providing feedback on an initial draft. Thanks to Peter Kulczycki for proof reading R0.