Constexpr Callable Wrappers

Steve Downey <sdowney@gmail.com> <sdowney2@bloomberg.net>

Document #: P3574R0
Date: 2025-01-13
Project: Programming Language C++
Audience: LEWG

Abstract

Add `constexpr` in the interface of `copyable_function` and `move_only_function`.

Contents

Changes Since Last Version

— **Initial Version**

1 Comparison table

1.1 Copyable Function

```
std::copyable_function<int(int)> c;
```

```
constexpr std::copyable_function<int(int)> c;
```

2 Motivation

Enable type-erasure of functions for compile time programming, for many of the same reasons as normal programming.

3 Design

Add `constexpr` in the interface.

However, no existing compiler and type erased callable work with merely adding `constexpr`. There seem to be no true implementation barriers as consteval can now allocate and cast from void to the original type, which are the key capabilities.

4 Proposal

Add `constexpr` in the interface of `copyable_function` and `move_only_function`.

5 Wording

𝔉.1 Function objects [function.objects]

𝔉.𝔉.1 General [function.objects.general]

1 A *function object type* is an object type that can be the type of the *postfix-expression* in a function call.[1] A *function object* is an object of a function object type. In the places where one would expect to pass a pointer to a function to an algorithmic template, the interface is specified to accept a function object. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.

𝔉.𝔉.2 Header `<functional>` synopsis [functional.syn]

```
namespace std {
  // 𝔉.𝔉.5, invoke
  template<class F, class... Args>
    constexpr invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)            // freestanding
      noexcept(is_nothrow_invocable_v<F, Args...>);

  template<class R, class F, class... Args>
    constexpr R invoke_r(F&& f, Args&&... args)                                    // freestanding
      noexcept(is_nothrow_invocable_r_v<R, F, Args...>);

  // 𝔉.𝔉.6, reference_wrapper
  template<class T> class reference_wrapper;                                       // freestanding

  template<class T> constexpr reference_wrapper<T> ref(T&) noexcept;               // freestanding
  template<class T> constexpr reference_wrapper<const T> cref(const T&) noexcept;  // freestanding
  template<class T> void ref(const T&&) = delete;                                  // freestanding
  template<class T> void cref(const T&&) = delete;                                 // freestanding

  template<class T>
    constexpr reference_wrapper<T> ref(reference_wrapper<T>) noexcept;             // freestanding
  template<class T>
    constexpr reference_wrapper<const T> cref(reference_wrapper<T>) noexcept;      // freestanding

  // ??, common_reference related specializations
  template<class R, class T, template<class> class RQual, template<class> class TQual>
    requires see below
  struct basic_common_reference<R, T, RQual, TQual>;

  template<class T, class R, template<class> class TQual, template<class> class RQual>
    requires see below
  struct basic_common_reference<T, R, TQual, RQual>;

  // 𝔉.𝔉.7, arithmetic operations
  template<class T = void> struct plus;                                            // freestanding
  template<class T = void> struct minus;                                           // freestanding
  template<class T = void> struct multiplies;                                      // freestanding
  template<class T = void> struct divides;                                         // freestanding
  template<class T = void> struct modulus;                                         // freestanding
  template<class T = void> struct negate;                                          // freestanding
  template<> struct plus<void>;                                                    // freestanding
  template<> struct minus<void>;                                                   // freestanding
  template<> struct multiplies<void>;                                              // freestanding
  template<> struct divides<void>;                                                 // freestanding
  template<> struct modulus<void>;                                                 // freestanding
  template<> struct negate<void>;                                                  // freestanding

  // 𝔉.𝔉.8, comparisons
  template<class T = void> struct equal_to;                                        // freestanding
  template<class T = void> struct not_equal_to;                                    // freestanding
```

1) Such a type is a function pointer or a class type which has a member `operator()` or a class type which has a conversion to a pointer to function.

```
template<class T = void> struct greater;                                    // freestanding
template<class T = void> struct less;                                       // freestanding
template<class T = void> struct greater_equal;                              // freestanding
template<class T = void> struct less_equal;                                 // freestanding
template<> struct equal_to<void>;                                           // freestanding
template<> struct not_equal_to<void>;                                       // freestanding
template<> struct greater<void>;                                            // freestanding
template<> struct less<void>;                                               // freestanding
template<> struct greater_equal<void>;                                      // freestanding
template<> struct less_equal<void>;                                         // freestanding

// ??, class compare_three_way
struct compare_three_way;                                                   // freestanding

// 𝟐𝟑.𝟏𝟎.10, logical operations
template<class T = void> struct logical_and;                                // freestanding
template<class T = void> struct logical_or;                                 // freestanding
template<class T = void> struct logical_not;                                // freestanding
template<> struct logical_and<void>;                                        // freestanding
template<> struct logical_or<void>;                                         // freestanding
template<> struct logical_not<void>;                                        // freestanding

// 𝟐𝟑.𝟏𝟎.11, bitwise operations
template<class T = void> struct bit_and;                                    // freestanding
template<class T = void> struct bit_or;                                     // freestanding
template<class T = void> struct bit_xor;                                    // freestanding
template<class T = void> struct bit_not;                                    // freestanding
template<> struct bit_and<void>;                                            // freestanding
template<> struct bit_or<void>;                                             // freestanding
template<> struct bit_xor<void>;                                            // freestanding
template<> struct bit_not<void>;                                            // freestanding

// 𝟐𝟑.𝟏𝟎.12, identity
struct identity;                                                            // freestanding

// 𝟐𝟑.𝟏𝟎.13, function template not_fn
template<class F> constexpr unspecified not_fn(F&& f);                     // freestanding
template<auto f> constexpr unspecified not_fn() noexcept;                   // freestanding

// 𝟐𝟑.𝟏𝟎.14, function templates bind_front and bind_back
template<class F, class... Args>
  constexpr unspecified bind_front(F&&, Args&&...);                         // freestanding
template<auto f, class... Args>
  constexpr unspecified bind_front(Args&&...);                              // freestanding
template<class F, class... Args>
  constexpr unspecified bind_back(F&&, Args&&...);                          // freestanding
template<auto f, class... Args>
  constexpr unspecified bind_back(Args&&...);                               // freestanding

// 𝟐𝟑.𝟏𝟎.15, bind
template<class T> struct is_bind_expression;                                // freestanding
template<class T>
  constexpr bool is_bind_expression_v =                                     // freestanding
    is_bind_expression<T>::value;
template<class T> struct is_placeholder;                                    // freestanding
template<class T>
  constexpr int is_placeholder_v =                                          // freestanding
    is_placeholder<T>::value;

template<class F, class... BoundArgs>
  constexpr unspecified bind(F&&, BoundArgs&&...);                          // freestanding
```

```
template<class R, class F, class... BoundArgs>
  constexpr unspecified bind(F&&, BoundArgs&&...);                                  // freestanding

namespace placeholders {
  // M is the implementation-defined number of placeholders
  see below _1;                                                                     // freestanding
  see below _2;                                                                     // freestanding
             .
             .
             .
  see below _M;                                                                     // freestanding
}

// ?.?.16, member function adaptors
template<class R, class T>
  constexpr unspecified mem_fn(R T::*) noexcept;                                    // freestanding

// ?.?.18, polymorphic function wrappers
// ?.?.18.2, class bad_function_call
class bad_function_call;

// ?.?.18.3, class template function
template<class> class function;        // not defined
template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

// ?.?.18.3.8, function specialized algorithms
template<class R, class... ArgTypes>
  void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;

// ?.?.18.3.7, function null pointer comparison operator functions
template<class R, class... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

// ?.?.18.4, 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

// ?.?.18.5, 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

// ?.?.18.6, non-owning wrapper
template<class... S> class function_ref;                                // freestanding, not defined
template<class R, class... ArgTypes>
  class function_ref<R(ArgTypes...) cv noexcept(noex)>;                 // freestanding, see below

// ??, searchers
template<class ForwardIterator1, class BinaryPredicate = equal_to<>>
  class default_searcher;                                                           // freestanding

template<class RandomAccessIterator,
         class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
         class BinaryPredicate = equal_to<>>
  class boyer_moore_searcher;

template<class RandomAccessIterator,
         class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
         class BinaryPredicate = equal_to<>>
  class boyer_moore_horspool_searcher;
```

```
  // ??, class template hash
  template<class T>
    struct hash;                                                    // freestanding

  namespace ranges {
    // ?.?.9, concept-constrained comparisons
    struct equal_to;                                                // freestanding
    struct not_equal_to;                                            // freestanding
    struct greater;                                                 // freestanding
    struct less;                                                    // freestanding
    struct greater_equal;                                           // freestanding
    struct less_equal;                                              // freestanding
  }

  template<class Fn, class... Args>
    concept callable =                                              // exposition only
      requires (Fn&& fn, Args&&... args) {
        std::forward<Fn>(fn)(std::forward<Args>(args)...);
      };

  template<class Fn, class... Args>
    concept nothrow-callable =                                      // exposition only
      callable<Fn, Args...> &&
      requires (Fn&& fn, Args&&... args) {
        { std::forward<Fn>(fn)(std::forward<Args>(args)...) } noexcept;
      };

  template<class Fn, class... Args>
    using call-result-t = decltype(declval<Fn>()(declval<Args>()...));  // exposition only

  template<const auto& T>
    using decayed-typeof = decltype(auto(T));                       // exposition only
}
```

1 [*Example 1*: If a C++ program wants to have a by-element addition of two vectors `a` and `b` containing `double` and put the result into `a`, it can do:

```
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
```

— *end example*]

2 [*Example 2*: To negate every element of `a`:

```
transform(a.begin(), a.end(), a.begin(), negate<double>());
```

— *end example*]

1 Subclause [illegible].[illegible].18 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.

[illegible].[illegible].18.2 Class `bad_function_call` [func.wrap.badcall]

1 An exception of type `bad_function_call` is thrown by `function::operator()` when the function wrapper object has no target.

```
namespace std {
  class bad_function_call : public exception {
  public:
    // see ?? 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.

[illegible].[illegible].18.3 Class template `function` [func.wrap.func]

[illegible].[illegible].18.3.1 General [func.wrap.func.general]

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

    // 𝔷.𝔷.18.3.2, 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();

    // 𝔷.𝔷.18.3.3, function modifiers
    void swap(function&) noexcept;

    // 𝔷.𝔷.18.3.4, function capacity
    explicit operator bool() const noexcept;

    // 𝔷.𝔷.18.3.5, function invocation
    R operator()(ArgTypes...) const;

    // 𝔷.𝔷.18.3.6, 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, given a call signature.

2 A callable type `F` is *Lvalue-Callable* for argument types `ArgTypes` and return type `R` if the expression *INVOKE*`<R>(declval<F&>(` `declval<ArgTypes>()...)`, considered as an unevaluated operand, is well-formed.

3 The `function` class template is a call wrapper whose call signature is `R(ArgTypes...)`.

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

𝔷.𝔷.18.3.2 Constructors and destructor [func.wrap.func.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) — `FD` is Lvalue-Callable for argument types `ArgTypes...` and return type `R`.

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* `&`$_{opt}$ `noexcept`$_{opt}$ or of the form `R(*)(G, A...) 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...) 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*: `decay_t<F>` is Lvalue-Callable for argument types `ArgTypes...` and return type `R`.

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`.

𝟐𝟑.𝟏𝟏.3.3 Modifiers [func.wrap.func.mod]

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

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

𝟐𝟑.𝟏𝟏.3.4 Capacity [func.wrap.func.cap]

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

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

𝟐𝟑.𝟏𝟏.3.5 Invocation [func.wrap.func.inv]

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

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

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

𝟐𝟑.𝟏𝟏.3.6 Target access [func.wrap.func.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.

𝟐𝟑.𝟏𝟏.3.7 Null pointer comparison operator functions [func.wrap.func.nullptr]

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

1 *Returns*: `!f`.

𝔉.𝔉.18.3.8 Specialized algorithms [func.wrap.func.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);`

𝔉.𝔉.18.4 Move-only wrapper [func.wrap.move]

𝔉.𝔉.18.4.1 General [func.wrap.move.general]

1 The header provides partial specializations of `move_only_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*.

𝔉.𝔉.18.4.2 Class template `move_only_function` [func.wrap.move.class]

```
namespace std {
  template<class R, class... ArgTypes>
  class move_only_function<R(ArgTypes...) cv ref noexcept(noex)> {
  public:
    using result_type = R;

    // 𝔉.𝔉.18.4.3, constructors, assignment, and destructor
    constexpr move_only_function() noexcept;
    constexpr move_only_function(nullptr_t) noexcept;
    constexpr move_only_function(move_only_function&&) noexcept;
    template<class F> constexpr move_only_function(F&&);
    template<class T, class... Args>
      constexpr explicit move_only_function(in_place_type_t<T>, Args&&...);
    template<class T, class U, class... Args>
      constexpr explicit move_only_function(in_place_type_t<T>, initializer_list<U>, Args&&...);

    constexpr move_only_function& operator=(move_only_function&&);
    constexpr move_only_function& operator=(nullptr_t) noexcept;
    template<class F> constexpr move_only_function& operator=(F&&);

    constexpr ~move_only_function();

    // 𝔉.𝔉.18.4.4, invocation
    constexpr explicit operator bool() const noexcept;
    constexpr R operator()(ArgTypes...) cv ref noexcept(noex);

    // 𝔉.𝔉.18.4.5, utility
    constexpr void swap(move_only_function&) noexcept;
    friend constexpr void swap(move_only_function&, move_only_function&) noexcept;
    friend constexpr bool operator==(const move_only_function&, nullptr_t) noexcept;

  private:
    template<class VT>
      static constexpr bool is-callable-from = see below;       // exposition only
  };
}
```

1 The `move_only_function` class template provides polymorphic wrappers that generalize the notion of a callable object. These wrappers can store, 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_move_constructible_v<T>` is `true`. — *end note*]

𝔉.𝔉.18.4.3 Constructors, assignment, and destructor [func.wrap.move.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...>
```

```
constexpr move_only_function() noexcept;
constexpr move_only_function(nullptr_t) noexcept;
```

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

```
constexpr move_only_function(move_only_function&& f) noexcept;
```

3 *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> constexpr move_only_function(F&& f);
```

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

5 *Constraints*:

(5.1) — `remove_cvref_t<F>` is not the same type as `move_only_function`, and

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

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

6 *Mandates*: `is_constructible_v<VT, F>` is `true`.

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

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

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

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

(8.3) — `remove_cvref_t<F>` is a specialization of the `move_only_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)`.

9 *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>
  constexpr explicit move_only_function(in_place_type_t<T>, Args&&... args);
```

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

11 *Constraints*:

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

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

12 *Mandates*: `VT` is the same type as `T`.

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 a target object of type `VT` direct-non-list-initialized with `std::forward<Args>(args)...`.

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

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

17 *Constraints*:

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

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

18 *Mandates*: `VT` is the same type as `T`.

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 of type `VT` direct-non-list-initialized with `ilist, std::forward<Args>(args)...`.

21 *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`.

```
constexpr move_only_function& operator=(move_only_function&& f);
```

22 *Effects*: Equivalent to: `move_only_function(std::move(f)).swap(*this);`

23 *Returns*: `*this`.

```
constexpr move_only_function& operator=(nullptr_t) noexcept;
```

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

25 *Returns*: `*this`.

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

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

27 *Returns*: `*this`.

```
constexpr ~move_only_function();
```

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

𝟐𝟑.𝟐𝟑.18.4.4 Invocation [func.wrap.move.inv]

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

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

```
constexpr 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`.

𝟐𝟑.𝟐𝟑.18.4.5 Utility [func.wrap.move.util]

```
constexpr void swap(move_only_function& other) noexcept;
```

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

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

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

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

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

𝔉.𝔉.18.5 Copyable wrapper [func.wrap.copy]

𝔉.𝔉.18.5.1 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*.

𝔉.𝔉.18.5.2 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;

    // 𝔉.𝔉.18.5.3, constructors, assignments, and destructors
    constexpr copyable_function() noexcept;
    constexpr copyable_function(nullptr_t) noexcept;
    constexpr copyable_function(const copyable_function&);
    constexpr copyable_function(copyable_function&&) noexcept;
    template<class F> constexpr copyable_function(F&&);
    template<class T, class... Args>
      constexpr explicit copyable_function(in_place_type_t<T>, Args&&...);
    template<class T, class U, class... Args>
      constexpr explicit copyable_function(in_place_type_t<T>, initializer_list<U>, Args&&...);

    constexpr copyable_function& operator=(const copyable_function&);
    constexpr copyable_function& operator=(copyable_function&&);
    constexpr copyable_function& operator=(nullptr_t) noexcept;
    template<class F> constexpr copyable_function& operator=(F&&);

    constexpr ~copyable_function();

    // 𝔉.𝔉.18.5.4, invocation
    constexpr explicit operator bool() const noexcept;
    constexpr R operator()(ArgTypes...) cv ref noexcept(noex);

    // 𝔉.𝔉.18.5.5, utility
    constexpr void swap(copyable_function&) noexcept;
    friend constexpr void swap(copyable_function&, copyable_function&) noexcept;
    friend constexpr 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. 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_move_constructible_v<T>` is `true`. — *end note*]

𝔉.𝔉.18.5.3 Constructors, assignments, and destructors [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...>
```

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

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

```
constexpr 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`.

```
constexpr 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.

```
constexpr 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 type 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 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>
  constexpr 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>
  constexpr 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`.

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

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

25 *Returns*: `*this`.

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

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

27 *Returns*: `*this`.

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

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

29 *Returns*: `*this`.

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

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

31 *Returns*: `*this`.

```
constexpr ~copyable_function();
```

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

𝟐𝟕.𝟏𝟎.18.5.4 Invocation [func.wrap.copy.inv]

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

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

```
constexpr 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`.

𝔉.𝔉.18.5.5 Utility [func.wrap.copy.util]

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

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

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

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

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

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

𝔉.𝔉.18.6 Non-owning wrapper [func.wrap.ref]

𝔉.𝔉.18.6.1 General [func.wrap.ref.general]

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

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

(1.2) — *noex* is either `true` or `false`.

𝔉.𝔉.18.6.2 Class template `function_ref` [func.wrap.ref.class]

```
namespace std {
  template<class R, class... ArgTypes>
  class function_ref<R(ArgTypes...) cv noexcept(noex)> {
  public:
    // 𝔉.𝔉.18.6.3, constructors and assignment operators
    template<class F> function_ref(F*) noexcept;
    template<class F> constexpr function_ref(F&&) noexcept;
    template<auto f> constexpr function_ref(nontype_t<f>) noexcept;
    template<auto f, class U> constexpr function_ref(nontype_t<f>, U&&) noexcept;
    template<auto f, class T> constexpr function_ref(nontype_t<f>, cv T*) noexcept;

    constexpr function_ref(const function_ref&) noexcept = default;
    constexpr function_ref& operator=(const function_ref&) noexcept = default;
    template<class T> function_ref& operator=(T) = delete;

    // 𝔉.𝔉.18.6.4, invocation
    R operator()(ArgTypes...) const noexcept(noex);

  private:
    template<class... T>
      static constexpr bool is-invocable-using = see below;      // exposition only

    R (*thunk-ptr)(BoundEntityType, Args&&...) noexcept(noex);   // exposition only
    BoundEntityType bound-entity;                                // exposition only
  };

  // 𝔉.𝔉.18.6.5, deduction guides
  template<class F>
    function_ref(F*) -> function_ref<F>;
  template<auto f>
    function_ref(nontype_t<f>) -> function_ref<see below>;
```

```
  template<auto f, class T>
    function_ref(nontype_t<f>, T&&) -> function_ref<see below>;
}
```

1 An object of class `function_ref<R(Args...) cv noexcept(noex)>` stores a pointer to function *thunk-ptr* and an object *bound-entity*. *bound-entity* has an unspecified trivially copyable type *BoundEntityType*, that models `copyable` and is capable of storing a pointer to object value or a pointer to function value. The type of *thunk-ptr* is `R(*)(BoundEntityType, Args&&...) noexcept(noex)`.

2 Each specialization of `function_ref` is a trivially copyable type that models `copyable`.

3 Within [func.wrap.ref].18.6, *call-args* is an argument pack with elements such that `decltype((call-args))...` denote `Args&&...` respectively.

[func.wrap.ref].18.6.3 Constructors and assignment operators [func.wrap.ref.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 F> function_ref(F* f) noexcept;
```

2 *Constraints*:

(2.1) — `is_function_v<F>` is `true`, and

(2.2) — *is-invocable-using*`<F>` is `true`.

3 *Preconditions*: `f` is not a null pointer.

4 *Effects*: Initializes *bound-entity* with `f`, and *thunk-ptr* with the address of a function *thunk* such that *thunk*`(`*bound-entity*`, `*call-args*`...)` is expression-equivalent to `invoke_r<R>(f, `*call-args*`...)`.

```
template<class F> constexpr function_ref(F&& f) noexcept;
```

5 Let `T` be `remove_reference_t<F>`.

6 *Constraints*:

(6.1) — `remove_cvref_t<F>` is not the same type as `function_ref`,

(6.2) — `is_member_pointer_v<T>` is `false`, and

(6.3) — *is-invocable-using*`<`*cv* `T&>` is `true`.

7 *Effects*: Initializes *bound-entity* with `addressof(f)`, and *thunk-ptr* with the address of a function *thunk* such that *thunk*`(`*bound-entity*`, `*call-args*`...)` is expression-equivalent to `invoke_r<R>(static_cast<`*cv* `T&>(f), `*call-args*`...)`.

```
template<auto f> constexpr function_ref(nontype_t<f>) noexcept;
```

8 Let `F` be `decltype(f)`.

9 *Constraints*: *is-invocable-using*`<F>` is `true`.

10 *Mandates*: If `is_pointer_v<F> || is_member_pointer_v<F>` is `true`, then `f != nullptr` is `true`.

11 *Effects*: Initializes *bound-entity* with a pointer to an unspecified object or null pointer value, and *thunk-ptr* with the address of a function *thunk* such that *thunk*`(`*bound-entity*`, `*call-args*`...)` is expression-equivalent to `invoke_r<R>(f, `*call-args*`...)`.

```
template<auto f, class U>
  constexpr function_ref(nontype_t<f>, U&& obj) noexcept;
```

12 Let `T` be `remove_reference_t<U>` and `F` be `decltype(f)`.

13 *Constraints*:

(13.1) — `is_rvalue_reference_v<U&&>` is `false`, and

(13.2) — *is-invocable-using*<F, *cv* T&> is true.

14 *Mandates*: If is_pointer_v<F> || is_member_pointer_v<F> is true, then f != nullptr is true.

15 *Effects*: Initializes *bound-entity* with addressof(obj), and *thunk-ptr* with the address of a function *thunk* such that *thunk*(*bound-entity*, *call-args*...) is expression-equivalent to invoke_r<R>(f, static_cast<*cv* T&>(obj), *call-args*...).

```
template<auto f, class T>
  constexpr function_ref(nontype_t<f>, cv T* obj) noexcept;
```

16 Let F be decltype(f).

17 *Constraints*: *is-invocable-using*<F, *cv* T*> is true.

18 *Mandates*: If is_pointer_v<F> || is_member_pointer_v<F> is true, then f != nullptr is true.

19 *Preconditions*: If is_member_pointer_v<F> is true, obj is not a null pointer.

20 *Effects*: Initializes *bound-entity* with obj, and *thunk-ptr* with the address of a function *thunk* such that *thunk*(*bound-entity*, *call-args*...) is expression-equivalent to invoke_r<R>(f, obj, *call-args*...).

```
template<class T> function_ref& operator=(T) = delete;
```

21 *Constraints*:

(21.1) — T is not the same type as function_ref,

(21.2) — is_pointer_v<T> is false, and

(21.3) — T is not a specialization of nontype_t.

𝔉.𝔉.18.6.4 Invocation [func.wrap.ref.inv]

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

1 *Effects*: Equivalent to: return *thunk-ptr*(*bound-entity*, std::forward<ArgTypes>(args)...);

𝔉.𝔉.18.6.5 Deduction guides [func.wrap.ref.deduct]

```
template<class F>
  function_ref(F*) -> function_ref<F>;
```

1 *Constraints*: is_function_v<F> is true.

```
template<auto f>
  function_ref(nontype_t<f>) -> function_ref<see below>;
```

2 Let F be remove_pointer_t<decltype(f)>.

3 *Constraints*: is_function_v<F> is true.

4 *Remarks*: The deduced type is function_ref<F>.

```
template<auto f, class T>
  function_ref(nontype_t<f>, T&&) -> function_ref<see below>;
```

5 Let F be decltype(f).

6 *Constraints*:

(6.1) — F is of the form R(G::*)(A...) *cv* &$_{opt}$ noexcept(E) for a type G, or

(6.2) — F is of the form M G::* for a type G and an object type M, in which case let R be invoke_result_t<F, T&>, A... be an empty pack, and E be false, or

(6.3) — F is of the form R(*)(G, A...) noexcept(E) for a type G.

7 *Remarks*: The deduced type is function_ref<R(A...) noexcept(E)>.

6 Impact on the standard

A pure library extension, affecting no other parts of the library or language.

The proposed changes are relative to the current working draft [N4910].

References

[N4910] Thomas Köppe. N4910: Working draft, standard for programming language c++. `https://wg21.link/n4910`, 3 2022.