Abstract

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

Tony Table

<table>
<thead>
<tr>
<th>Before</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto lambda[<a href="">&amp;</a> /<em>const</em>/ { .. }];</td>
<td>auto lambda[<a href="">&amp;</a> /<em>const</em>/ { .. }]);</td>
</tr>
<tr>
<td>function(void&gt; func(lambda);</td>
<td>copyable_function&lt;void(void)&gt; func0(lambda);</td>
</tr>
<tr>
<td>const auto &amp; ref(func);</td>
<td>const auto &amp; ref(func);</td>
</tr>
<tr>
<td>func();</td>
<td>func0();</td>
</tr>
<tr>
<td>ref();</td>
<td>func0();</td>
</tr>
<tr>
<td></td>
<td>copyable_function&lt;void(void)&gt; const func1(lambda);</td>
</tr>
<tr>
<td></td>
<td>const auto &amp; ref(func1);</td>
</tr>
<tr>
<td></td>
<td>func1();</td>
</tr>
<tr>
<td></td>
<td>ref1(); //operator() is const!</td>
</tr>
<tr>
<td>auto lambda[<a href="">&amp;</a> mutable { .. }];</td>
<td>auto lambda[<a href="">&amp;</a> mutable { .. }]);</td>
</tr>
<tr>
<td>function(void&gt; func(lambda);</td>
<td>copyable_function&lt;void(void)&gt; func(lambda);</td>
</tr>
<tr>
<td>const auto &amp; ref(func);</td>
<td>const auto &amp; ref(func);</td>
</tr>
<tr>
<td>func();</td>
<td>func();</td>
</tr>
<tr>
<td>ref(); //operator() is const!</td>
<td>ref(); //operator() is const!</td>
</tr>
<tr>
<td>//this is the infamous constness-bug</td>
<td>copyable_function&lt;void(void)&gt; const tmp(lambda);</td>
</tr>
</tbody>
</table>
- Removed open question on the deprecation of function.
- Replaced previously proposed conversion operators to move_only_function.
- Added section on conversions between standard library polymorphic function wrappers.
- Added section on potential allocator support.

R4: Updates after LEWG Review on 2022-11-11:

- Removed mandatory optimization for conversion to move_only_function.

R5: Updates after LEWG Review on 2023-03-07:

- Added section on naming of this class.
- Extended wording with recommended practice to avoid double wrapping of type-erased function wrappers.
- Fixed some wording bugs.

Motivation

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

[P0288R9] introduced move_only_function, a move-only type-erased callable wrapper. In addition to dropping the copyable requirement, move_only_function extends the supported signature to R(Args...) const (&|&&) noexcept op and forwards all qualifiers to its call operator, introduces a strong non-empty precondition for invocation instead of throwing bad_function_call and drops the dependency to typeid/RTTI (there is no equivalent to function’s target_type() or target()).

Concurrently, [P0792R10] introduced function_ref, a type-erased non-owning reference to any callable matching a function signature in the form of R(Args...) const noexcept op. Like move_only_function, it forwards the noexcept-qualifier to its call operator. As function_ref acts like a reference, it does not support ref-qualifiers and does not forward the const-qualifier to its call operator.

As a result, function is now the only type-erased function wrapper not supporting any form of qualifiers in its signature. Whilst amending function with support for ref/noexcept-qualifiers would be a straightforward extension, the same is not true for the const-qualifier due to the long-standing constness-bug. Without proper support for the const-qualifier, function would still be inconsistent with its closest relative.

Therefore, this paper proposes to introduce a replacement to function in the form of copyable_function, a class that closely mirrors the design of move_only_function and adds copyability as an additional affordance.

Design space

The main goal of this paper is consistency between the move-only and copyable type-erased function wrappers. Therefore, we follow the design of move_only_function very closely and only introduce three extensions:

1. Adding a copy constructor
2. Adding a copy assignment operator
3. Requiring callables to be copy-constructible
Conversions between function wrappers

Given the proliferation of proposals for polymorphic function wrappers, LEWG requested an evaluation of the “conversion story” of these types. Note that conversions from function_ref always follow reference semantics for obvious reasons.

<table>
<thead>
<tr>
<th>From</th>
<th>function</th>
<th>move_only_function</th>
<th>copyable_function</th>
<th>function_ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>move_only_function</td>
<td></td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>copyable_function</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>function_ref</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

It is recommended that implementors do not perform additional allocations when converting from a copyable_function instantiation to a compatible move_only_function instantiation, but this is left as quality-of-implementation.

Concerning allocator support

After having reviewed R2, LEWG requested a statement about potential allocator support. As this proposal aims for feature parity with move_only_function (apart from the extensions mentioned above) and considering the somewhat recent removal of allocator support from function [P0302], we refrain from adding allocator support to copyable_function. We welcome an independent paper introducing said support to both classes.

Naming discussion

During the review of R4, there were questions raised for the rationale for the name copyable_function, especially as it was perceived inconsistent with move_only_function. Our rationale for the name is as follows: copyable_function is a copyable function call wrapper that requires the target object to be copyable, so the copyable-prefix references both aspects. Furthermore, there isn’t actually an inconsistency with move_only_function, as the move_only-prefix only applies to the wrapper; the wrapper is move-only, but there is no reason to require the target object to be as well.

Impact on the Standard

This proposal is a pure library addition.

Implementation Experience

The proposed design has been implemented at https://github.com/MFHava/P2548.

Proposed Wording

Wording is relative to [N4928]. Additions are presented like this, removals like this, and drafting notes like this.

[version.syn]

```cpp
#define __cpp_lib_copyable_function YYYYMM //also in <functional>
[DRAFTING NOTE: Adjust the placeholder value as needed to denote this proposal’s date of adoption.]
```
22.10.2 Header `<functional>` synopsis

```cpp
namespace std {
    // [func.wrap.move], move only wrapper
template<class... S> class move_only_function;
    // not defined
template<class R, class... ArgTypes>
    class move_only_function<R(ArgTypes...)
        cv ref noexcept(noex)); // see below

    // [func.wrap.copy], copyable wrapper
    template<class... S> class copyable_function;
    // not defined
    template<class R, class... ArgTypes>
    class copyable_function<R(ArgTypes...)
        cv ref noexcept(noex)); // see below

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

    ...
}
```

22.10.17 General

Subclause [func.wrap] describes polymorphic wrapper classes that encapsulate arbitrary callable objects.

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

[DRAFTING NOTE: It’s the intended design that moves can be elided, even if they would be observable when double wrapping:]

```cpp
move_only_function<void(T)>

T f(T);
```

Both are acceptable.

22.10.17.2 Class `bad_function` [func.wrap.badfunc]

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

22.10.17.7 Copyable wrapper [func.wrap.copy]

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

- `cv` is either `const` or empty,
- `ref` is either `&`, `&&`, or empty, and
- `noex` is either `true` or `false`.

For each of the possible combinations of the placeholders mentioned above, there is a placeholder `inv-quals` defined as follows:

- If `ref` is empty, let `inv-quals` be `cv &`,
- otherwise, let `inv-quals` be `cv ref`.

22.10.17.7.2 Class template `copyable_function` [func.wrap.class]

```cpp
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;

    // [func.wrap.copy.ctor], constructors, assignments, and destructors
    copyable_function() noexcept;
    copyable_function(nullptr_t) noexcept;
    copyable_function(const copyable_function&);
    copyable_function(copyable_function&&) noexcept;
    template<class F> copyable_function(F&&);
    template<class T, class U, 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=(const copyable_function&);
    copyable_function& operator=(copyable_function&&);
    template<class F> copyable_function operator=(F&&);
    template<class F> copyable_function operator=(in_place_type_t<F>); 
    
    explicit operator bool() const noexcept;

    R operator()(ArgTypes...)
        cv ref noexcept(noex));
```
Recommended practice: Implementations should avoid the use of dynamically allocated memory for a small contained value. Such small object optimization can only be applied to types that meet the is_nothrow_constructible credef, see below.

Postconditions:

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 Args&&... respectively.

Postconditions: *this has no target object.

Postconditions: *this has no target object if f had no target object.

Postconditions: The target object of *this is a copy of the target object of f.

Postconditions: Any exception thrown by the initialization of the target object may throw bad_alloc.

Postconditions: The target object of *this has no target object.

Postconditions: *this is a copy of the target object of f.

Postconditions: The target object of *this is a copy of the target object of f had before construction, and f is in a valid state with an unspecified value.
Constraints:

— is_constructible_v<VT, initializer_list<U>&, Args...> is true, and
— is_callable_from<VT> is true.

Mandates:

— VT is the same type as `f`, and
— is_copy_constructible_v<VT> is true.

Preconditions: `VT` meets the Cpp17Destructible and Cpp17CopyConstructible requirements.

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

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.

Effects:

- `operator=(const copyable_function& f);`
  - Equivalent to: `copyable_function(f).swap(*this)`.
  - Returns: `*this`.

- `operator=(copyable_function&& f);`
  - Equivalent to: `copyable_function(std::move(f)).swap(*this)`.
  - Returns: `*this`.

- `operator=(nullptr_t) noexcept;`
  - Destroys the target object of `*this`, if any.
  - Returns: `*this`.

- `template<class F> operator=(F&& f);`
  - Equivalent to: `copyable_function(std::forward<F>(f)).swap(*this)`.
  - Returns: `*this`.

Effects:

- `explicit operator bool() const noexcept;`
  - Returns: `true` if `*this` has a target object, otherwise `false`.

- `operator()(ArgTypes... args) cv ref noexcept(noex);`
  - Preconditions: `*this` has a target object.
  - Effects: Equivalent to:
    
    ```cpp
    return INVOKE<R>(static_cast<Finvquals>(f), std::forward<ArgTypes>(args)...);
    ```
  - where `f` is an lvalue designating the target object of `*this` and `F` is the type of `f`.

Effects:

- `swap(copyable_function& other) noexcept;`
  - Exchanges the target objects of `*this` and `other`.

- `friend void swap(copyable_function& f1, copyable_function& f2) noexcept;`
  - Effects: Equivalent to `f1.swap(f2)`.

- `friend bool operator==(const copyable_function& f1, nullptr_t) noexcept;`
  - Returns: `true` if `f1` has no target object, otherwise `false`.

Acknowledgements

Thanks to RISC Software GmbH for supporting this work. Thanks to Peter Kulczycki for proof reading and discussions. Thanks to Matt Calabrese for helping to get conversions to `move_only_function` to work.