`std::function` move constructor should be `noexcept`

Document number: **P0771R1**
Date: 2018-10-07
Project: Programming Language C++, Library Working Group
Reply-to: Nevin ":)" Liber, nevin@cplusplusguy.com
Pablo Halpern, phalpern@halpernwightsoftware.com

Table of Contents

Introduction

The move constructor for `std::function` should be `noexcept`.

Changes from revision R0 to R1

The `noexcept` move-assignment operator was removed from this proposal because it is inconsistent with `std::experimental::function` as described in the Library Fundamentals TS (N4617), both as-is and as modified by P0987. Specifically, if allocators are added back to the `std::function` interface, the move-assignment operator behaves as a copy assignment under certain conditions, making the `noexcept` guarantee impossible.

In general, algorithmic efficiency can be improved if the move constructor and `swap` functions are `noexcept`, as they would be with this proposal. A much smaller set of algorithms benefit from a `noexcept` move-assignment operator. It would be possible to

make the move-assignment operator conditionally `noexcept`, but it is not clear that it is worth complicating the interface at this time.

Motivation and Scope

It is highly desirable to have `noexcept` move operations, especially when it does not impose an undue burden on implementers or a high cost for users.

The other type-erased standard libraries `any` and `shared_ptr` already require this. `function` is very similar to `any` in that both encourage the small object optimization.

It appears that `function` is required to use the small object optimization, at least to hold a `reference_wrapper` object or function pointer [func.wrap.func.con#4], and this proposal is compatible with that.

Both libstdc++ and libc++ already implement this.

Impact on the Standard

Impact on the standard is minor. The declarations for the move constructor for `function` have to have `noexcept` added, and the throws clause for the move constructor has to be deleted.

Design Decisions

A possible implementation technique: if the object either is too big to fit inside the small object optimization space inside `function` or the object has a `noexcept(false)` move constructor, then store it in the heap; otherwise, store it in the small object optimization space.

Because default construction and `swap` are already `noexcept`, it is very likely that a currently conforming implementation of `function` already does something like this under the covers, even if they don't declare their move constructor as `noexcept`.

Technical Specifications

Changes relative to N4762:

[**func.wrap.func**]

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.con**]

function(function&& f) noexcept;

> *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.
> ~~*Throws*: Shall not throw exceptions 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 or move constructor of the stored callable object.~~
> *Note*: 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.

Acknowledgements

Special thanks to Ion Gaztañaga, Gabriel Dos Reis, Pete Becker, Bjarne Stroustrup, Jonathan Wakely, and Stephan T. Lavavej for the discussion on this way back when; Howard Hinnant for that as well as answering a theoretical design question on `function`, Billy O'Neal for pointing out on an LEWG thread that the small object optimization is required (as well as Stephan and Billy informing me how their version of `function` is implemented), and Geoffrey Romer for recently implicitly reminding me that no one had actually submitted a paper on this yet. Thank them or blame me for the content of this paper.

References

N4762 - Working Draft, Standard for Programming Language C++, Richard Smith, Editor 2018-07-07
N4617: Programming Languages - C++ Extensions for Library Fundamentals, Version 2, 2016-11-28.
std_function.h, libstdc++ (gcc)
functional – libc++ (clang)