Tightening the constraints on `std::function`

Aaryaman Sagar (aary@instagram.com)

February 7, 2018
Document #: p0932
Library Working Group

1. Introduction

```
#include <iostream>
#include <functional>

const int& foo(std::function<const int&()> bar) {
    return bar();
}

int main() {
    std::cout << foo([] { return 1; }) << std::endl;
}
```

The above code should not compile, however with the current restraints on `std::function` it does, and exhibits undefined behavior.

2. The current rules

The offending part of the current standard comes first from section `[func.wrap.func]p2`

> A callable type (23.14.2) 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 (Clause 8), is well formed (23.14.3).

And then the corresponding definition of `INVOKE<R>` in section `[func.require]p2`

> Define `INVOKE<R>(f, t1, t2, ..., tN)` as `static_cast<void>(INVOKE(f, t1, t2, ..., tN))` if R is cv void, otherwise `INVOKE(f, t1, t2, ..., tN)` implicitly converted to R.

The problem here is that an invocable with return type `R` and compatible arguments is convertible to a function object of type `std::function<Return(Args...)>` either via construction, assignment and combination of thereof if given that `R` is implicitly convertible to `Return`. This requirement needs to be tightened to accomodate for the common case of implicit conversion from prvalue to a reference type

3. Prevalence

Dangling references are a common problem in C++ and have caused severe problems in critical codebases in the past. In particular this bug has come up in Facebook's codebase in the past and has caused SEVs. The issue of returned dangling references has since been patched in the offending spots in our code - we protect against this by detecting the binding of references to prvalue expressions. The code for this can be found here and here.

A prvalue to reference binding in a function's return type should never be allowed at runtime. And is even prohibited by the current C++17 standard `[class.temporary]p6.10`

> The lifetime of a temporary bound to the returned value in a function return statement (9.6.3) is not extended; the temporary is destroyed at the end of the full-expression in the return statement.

This invalidates conversions from rvalues to references in the return statement of a function. And in practice, compilers like gcc 7.2.0 and clang 5.0.0 warn against this.

4. Backwards Compatibility

This will be a breaking change to the standard. Previous code that might compile will no longer compile. In practice this is a good limitation to impose. There is no code that will run without undefined behavior if this problem manifests at runtime

4.1. Backwards compatible fix

To illustrate that this problem is immediately fixable, there is a route that does not impact backwards compatibility.

The problem is detectable at compile time. So the library can warn the user when this problem manifests at runtime with either an `std::logic_error` exception when `std::function` is invoked from an unsafe wrapped callable or abort the program with a call to `std::abort`. This would carry no runtime overhead in the correct use cases as the incorrect path of execution would be handled through the regular virtual dispatch mechanism within `std::function` implementations

5. Changes to the current C++17 standard

5.1. Section 23.14.3 ([func.require]) paragraph 2

Define `INVOKE<R>(f, t1, t2, ..., tN)` as `static_cast<void>(INVOKE(f, t1, t2, ..., tN))` if R is cv void, otherwise `INVOKE(f, t1, t2, ..., tN)` implicitly converted to R. For exposition-only, define `A` as the type `decltype(INVOKE(f, t1, t2, ..., tN))`; if `std::is_reference_v<R> !std::is_reference_v<A>` is true, then `INVOKE` does not participate in overload resolution (as the resulting expression would exhibit undefined behavior)