Make `std::random_device` Less Inscrutable

Document #:	D2058R0
Date:	2020-01-13
Project:	Programming Language C++
Audience:	LEWG

Reply-to: Martin Hořeňovský <martin.horenovsky@gmail.com>

Introduction

The C++ standard library provides a set of utilities to generate pseudo-random numbers with different properties. It also provides `std::random_device` for users who desire nondeterministic randomness.

However, implementations are also allowed to employ a deterministic pseudo-random number generator if providing nondeterministic randomness is impossible. This allows implementations to provide `std::random_device` even on any possible platform, but there is a problem with this: the current interface does not provide a good way of checking whether the implementation provides non-deterministic random numbers.

Motivation

The standard already provides a way, specifically the `std::random_device::entropy()` member method, that the user should be able to use to find out whether `random_device` provides nondeterministic random number. In practice, this approach has two large flaws:

- This information is only available at runtime. In practice, an implementation already knows whether it provides nondeterministic `random_device` during compilation, and we should expose this information to the users.
- Implementations do not always implement this function usefully. For example, older versions of libstdc++ used to always return 0. Current versions use Linux-specific syscall to query the available entropy, and on Windows they return 0 even if they use OS-provided secure random facilities. Similarly, libc++ only ever returns 0, even though it only uses platform-provided secure random facilities.

The second flaw is likely exacerbated by the fact that if an implementation is using a platform-provided random facilities, it likely cannot provide a real and meaningful entropy estimate.

Non-goals

There are several other problems with `<random>`, and even `std::random_device` as it is currently standardized. This paper intentionally does not try to fix any of them. This is because there are other papers that target those issues, and we should not let fixing an issue get bogged down because of a lack of consensus on other issues.

Proposed solution

To address these issues, I propose to deprecate the `entropy` member function and instead add `static constexpr bool is_predictable()`[1] function to `random_device`. This function returns `true` if `random_device` is implemented in a way that makes its output predictable, e.g. by using a PRNG and `false` when its output is not predictable, e.g. when it is implemented in terms of OS-provided cryptographically secure random number generation facilities.

The reason to deprecate `entropy` is that only a small niche in the Linux community thinks of entropy in the OS providded CSRNG facilities as depletable. Combined with the fact that 2 out of the big 3 implementations do not return meaningful results on all platforms, and that the users are more interested in simple boolean query of "will the output be unpredictable", `entropy` is not useful in any way.

Even for the niche case where a user on the Linux platform cares about the estimated entropy in `/dev/urandom`, the interface has a TOCTOU problem.

Proposed wording changes

```
class random_device {
    public:
    // types
    using result_type = unsigned int;

    // generator characteristics
    static constexpr result_type min() { return numeric_limits<result_type>::min(); }
    static constexpr result_type max() { return numeric_limits<result_type>::max(); }

    // constructors
    random_device() : random_device(implementation-defined) {}
    explicit random_device(const string& token);

    // generating functions
    result_type operator()();

    // property functions
    double entropy() const noexcept;
```

[1] Possible alternative names that could be used instead: `is_nonpredictable`, `is_deterministic`, or `is_nondeterministic`

```
        static constexpr bool is_predictable();

        // no copy functions
        random_device(const random_device&) = delete;
        void operator=(const random_device&) = delete;
    };
```

`static constexpr bool is_predictable();`

> *Returns:* `true` if default-constructed `random_device` uses a predictable random generator, such as Mersenne Twister. `false` otherwise.

Alternative approach

An alternative approach is to avoid deprecating `entropy()` and instead clarifying that `entropy()` should only return 0 for predictable implementations, such as those that rely on a random number engine as defined in [rand.req.eng], and implementations that use unpredictable implementations, such as using `rand_s` provided by Windows, should return non-zero estimate.

Proposed wording changes

[rand.device]:

> If implementation limitations prevent generating nondeterministic random numbers, the implementation may employ a random number engine as defined in [rand.req.eng].

```
double entropy() const noexcept;
```

> *Returns:* If the implementation employs a random number engine, returns 0.0. Otherwise, returns a non-zero entropy estimate for the random numbers returned by operator(), in the range min() to log2(max()+1).

Goals

To sum up, this paper has two goals,

1. Provide users of `random_device` with a way to query whether the output is predictable or not.
2. Start a discussion on what should happen to `random_device::entropy`