Make Pseudo-random Numbers Portable

Document #:	D2059R0
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 in a range of user's choosing. This is done by applying a type that represents the desired statistical distribution (e.g. `uniform_int_distribution`) with a type conforming to the Uniform Random Bit Generator (URBG) concept (e.g. `mt19937`).

This makes the standard-provided facilities highly customizable, but they are rarely used because the generated numbers are not reproducible across different platforms. The reason for this is that the output of standard-provided distributions[1] is implementation-specified and thus can (and does) differ across implementations.

This paper proposes changes that would allow users to use these facilities for domains that require cross-platform reproducibility, such as testing, scientific simulations, and games[2].

Motivation

A straightforward "Hello world" of random number generation looks something like this:

```
#include <random>
#include <iostream>

int main() {
    std::mt19937 urbg;
    std::uniform_int_distribution<> dist(0, 100);
    std::cout << dist(urbg) << '\n';
}
```

On my machine[3], this code always prints out `53`. On other popular platforms, the code prints `82`[4] and `92`[5].

[1] The output of standard-provided URBGs is portable due to a happy accident during the original `<random>` standardization

[2] See Appendix A: Uses for repeatable distributions for more use cases that require cross-platform reproducibility.

[3] MSVC bundled with VS 16.4.0

[4] libstdc++

[5] libc++

This difference in outputs makes the standard-provided `<random>` facilities unusable in contexts where reproducibility is required, such as procedural generation and physics simulations.

However, cross-platform reproducibility is useful even in cases where it is not required, such as property-based tests. For property-based testing, it is desirable for a fixed seed to generate the same test case across different platforms, so that developers can share relevant seeds between themselves.

Because of this, `<random>` is currently unusable in many domains and applications, and if we want to change that, we need to ensure that a deterministically seeded code, such as the snippet above, generates the same numbers on all platforms.

Non-goals

There are several other problems with `<random>` as it is currently standardized, and 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.

Possible solutions

This section outlines possible solutions to the problem explained above, together with their advantages and disadvantages as seen by the author of this paper.

Standardize implementation for current distributions

One possibility (the one preferred by the author), is to mandate a specific implementation of all distributions in `<random>`. This option has two advantages, in that

- it retroactively fixes broken code
- it does not introduce further maintenance burden on implementations of the standard library

They are also disadvantages at the same time, in that

- it is a (potentially) breaking change
- it forbids implementations from providing the fastest possible code for their platform

The second disadvantage can be fixed by adding `fast_meow_distribution`s to the standard.

Provide `portable_meow_distributions`

Another possibility is to add `portable_meow_distributions` to `<random>`. This option is essentially the inverse solution of the first option, in that the old code is left as it was, and a new set of types is provided for portability.

However, this option is less desirable one as it makes the simpler code do the wrong thing, and is harder to teach.

Provide specific algorithms as distributions

The last option is to keep the status quo for already standardized types, but also provide specific algorithms under their own name, e.g. provide an implementation of Marsaglia polar method as `std::marsaglia_polar_method_distribution`.

This option has some compeling advantages, in that

- it lets experts pick the right algorithm for their domain
- allows for future expansion with better portable algorithms as they are found
- does not break backward compatibility

However, this option is the least teachable option by far, and it is also the least beginner friendly of all options presented in this paper. The common C++ programmer wants to generate some reproducible normally distributed numbers, without having to know the differences between Marsaglia polar method, Box-Muller transform and the Ziggurat algorithm.

It also does not absolve the standard from prescribing at least parts of the implementation of specific algorithms. As an example, the already mentioned Marsaglia polar method generates 2 normally distributed floating point numbers in single run. The order in which these 2 numbers are returned would need to be specified in the standard to avoid differences in behaviour between different implementations[6].

Goals

The goal of this paper is to start discussion and find a consensus on 3 questions:

1. Should the standard provide portable distributions?
2. Which of the three approaches should standard take?
3. How should the standard deal with distributions on floating point numbers?[7]

[6] This difference already exists in the wild, as both MSVC and libstdc++ use this algorithm for their implementation of `std::normal_distribution`, but return the generated numbers in different order.

[7] Remember that the standard does not even specify whether floating point numbers adhere to IEEE-754.

Appendix A: Uses for repeatable distributions

To come up with a more comprehensive list of problems where people need repeatable random number generation, I opened up a Reddit thread.

This is a short list of the answers:

- Fuzzing in general
- Property based testing
- Benchmarking of data-dependent algorithms (e.g. sorting)
- Photoshop filters
- Procedural generation of X in games (usually maps)
- Scientific simulations