PXXXXR0: `std::generate_canonical_centered` — Uniform Floating-Point Generation Centered at Zero

Document #:	D0000R1
Date:	2025-07-03
Project:	Programming Language C++
Audience:	Library Evolution
	Library
Revises:	D0000R0
Reply-to:	Lucas Rey
	<lrey4@bloomberg.com>

Contents

0.1 Abstract

We propose a new utility function `std::generate_canonical_centered`, which generates uniformly distributed floating-point values over the symmetric interval $[0, 0.5] \cup (-0.5, 0)$. This addresses the entropy distribution problem identified in LWG 2524 by leveraging the high precision and subnormal range near zero provided by IEEE 754 floating-point types. It provides a numerically superior alternative to `std::generate_canonical` without altering existing behavior.

0.2 1. Motivation

The standard function `std::generate_canonical<RealType, bits, URNG>` is intended to generate uniformly distributed floating-point values over the interval $[0, 1)$. However, due to the structure of IEEE 754 types, this interval suffers from reduced representable precision as values approach 1, and the granularity of representable numbers near 0 is underutilized.

In LWG 2524, it was observed that certain implementations of std::generate_canonical can produce the value 1.0 due to floating-point rounding, even though the function is specified to generate values in the half-open interval [0, 1). This can cause subtle bugs in simulations and algorithms that assume the upper bound is never reached.

Instead of modifying `std::generate_canonical`, we propose a new function—`std::generate_canonical_centered`—that maintains uniformity but distributes values over a symmetric interval centered at zero. This resolves the quality-of-implementation concerns without breaking compatibility.

0.3 2. Design Overview

IEEE 754 floating-point types allocate representable numbers most densely near zero, particularly in the subnormal range. This implies:

- Intervals centered near zero can encode more representable values within a fixed width than intervals further from zero.
- The interval $[-0.5, 0.5)$ (or the half-open symmetric form we adopt: $[0, 0.5] \cup (-0.5, 0)$) contains the **maximum number of distinct representable floating-point values** of any interval of length 1.

By redefining the output range to $[0, 0.5] \cup (-0.5, 0)$, we:

- Maximize the number of distinct representable floating-point values (minimizing rounding and duplication).
- Preserve uniformity by generating symmetric unsigned and signed samples.
- Avoid the low-resolution tail near 1 present in $[0, 1)$ generation.
- Improve the quality of derived symmetric distributions (e.g., normal distribution) by ensuring that the underlying uniform distribution has symmetric resolution about zero, which helps reduce numerical bias and asymmetry introduced during transformation.
- Allow full entropy usage from wide sources like `__uint128_t` (available in GCC and Clang) without ever reaching the value `1.0` boundary of `[0, 1)`, ensuring correctness even in low-precision types like `std::float64_t`. This ensures compatibility with potential future additions of new types to the standard.

0.4 3. Proposal

Add the following to `<random>`:

```cpp
namespace std {
  template<class RealType = double, int bits, class URNG>
  RealType generate_canonical_centered(URNG& g);
}
```

0.5 4. Exponential distribution

The exponential_distribution could now avoid the issue from LWG 2524 as follows:

```cpp
namespace std {

template <class RealType = double>
class exponential_distribution {
public:
    using result_type = RealType;

    explicit exponential_distribution(RealType lambda = 1.0)
        : lambda_(lambda), lambda_inv_(1 / lambda) {}

    void reset() {}

    template <class URNG>
    result_type operator()(URNG& g) {
        return generate(g, lambda_, lambda_inv_);
    }

private:
    template <class URNG>
    static result_type generate(URNG& g, RealType lambda, RealType lambda_inv) {
        const auto u = std::generate_canonical_centered<result_type,
                           std::numeric_limits<result_type>::digits>(g);

        if (std::copysign(1.0, u) > 0) {
            return -std::log(result_type(1) - u) * lambda_inv;
        } else {
            return -std::log(-u) * lambda_inv;
        }
    }

    result_type lambda_;
    result_type lambda_inv_;
};

}
```

0.6 5. Backward compatibility

If backward compatibility with existing uses of `std::generate_canonical` is a concern, a transitional variant of the centered interval can be considered. Instead of using the symmetric interval $[0, 0.5] \cup (-0.5, 0)$, the range can be expanded to $[0, *a*] \cup (- * a*, 0)$ for some a very close to 1 (e.g., $*a* = 1 - \varepsilon$). This would maintain the benefits of symmetric resolution around zero while keeping the overall range very close to the original $[0, 1)$ interval. As a result, many values that would be generated by an $[0, 1)$ generator would remain unchanged. This approach provides a migration-friendly compromise that retains higher precision near zero without introducing large discrepancies in the generated values. However, this relaxation should only be considered in environments where backward compatibility outweighs the precision and symmetry benefits of the strict [−0.5, 0.5] approach.