P3689R0
Convenience functions for Random number generation

Thomas Mejstrik

May 11, 2025

Contents

1 Motivation

The random-number facilities in `<random>` deliver high-quality, flexible, and portable pseudo-random number generation, but using them correctly requires a non-trivial amount of boilerplate. Everyday tasks like *"give me a random int between 1 and 6"* or *"flip a weighted coin"* take five lines of code and half a dozen type names. Other mainstream languages expose convenience functions that cover the 95% use-cases in a single call.

> This paper proposes to add a small set of free functions that wrap the existing machinery while preserving determinism, reproducibility, and performance.

Convenience functions will help generate better numbers. Beginners never need all the facilities of `<random>`. By giving them short, correct snippets that internally use good engines, we steer them away from global state like `rand()` and toward quality randomness. People who need good random numbers (cryptographers, developers of monte carlo simulations, ...) know what they need and will use what suits them.

1.0.1 Typical mistakes by beginners with `rand` in C

- Not considering that `rand() % n` does not produce equally distributed numbers in general
- Not considering that `rand()` is not threadsafe
- Using `time( 0 )` as a seed inside a loop

1.0.2 Typical mistakes by beginners in C++

- Make a new Rng engine each time a random number is generated, because users do not know that constructing an engine is expensive
- Store the Distribution in some static variable, because user may think constructing a distribution is expensive
- Seed Rng badly
- Seed Rng in a loop
- Starting Mersenne-Twister without seeding
- Start Mersenne-Twister with a seed which has a lot of zeros
- Using `std::random_device` directly

1.1 Tony Table

Before:

```
std::random_device rd;
const auto seed = rd();
std::mt19937 engine( seed );
std::uniform_int_distribution< int > distribution( 1, 6 );
auto num = distribution( engine );
```

After:

```
auto num = std::random( 1, 6 );
```

1.2 Scope

The proposal targets generation of *scalar* random values (integral, floating-point, Boolean) and access to a thread-local default engine. Vectorized distributions, secure RNGs, and sampling algorithms are out of scope. *Note: This does not replace the* `<random>` *APIs.*

1.3 Properties of convenience functions

- Thread safe
- Different threads shall not produce the same sequence of random numbers
- All convenience random-number functions in one thread use the same Rng
- Default Rng-Engine (second template argument) is implementation defined, and may change
- The generated random numbers may change, when the program is compiled anew
- Input type determines output type

```
auto _ = std::random( 1., 2. );  // generates floats in [1 2]
auto _ = std::random( 1,  2 );   // generates ints in [1 2]
auto _ = std::random( 1., 2 );   // compilation error
```

- Open questions:
  - Shall it be allowed that the generated random numbers change in subsequent runs of the same program

1.4 Detailed description

1.4.1 convenience_engine

```
using convenience_random_engine = philox4x64;
```

- A typedef for the default random number engine used for the convenience functions
- The typedef is explicitly allowed to change in a future C++ standard
- Open questions:
  - Shall `default_random_engine` be used instead?

1.4.2 random

```
// (1a)
template< typename T,
          typename Engine = std::convenience_random_engine >
    requires std::floating_point< T > ||
             std::integral< T >
T random( const T & lb, const T & ub );

// (1b)
template< typename T,
          typename Engine = std::convenience_random_engine >
    requires std::floating_point< T > ||
             std::is_same_v< T, bool >
T random();

// (2)
template< typename T,
          typename Engine = convenience_random_engine,
          std::floating_point P >
    requires std::is_same_v< T, bool >
T random( P p = P(0.5) )

// (3a)
template< std::ranges::random_access_range Range,
          typename Engine = std::convenience_random_engine >
std::ranges::range_reference_t< Range > random( Range && range );

// (3b)
template< typename T,
          typename Engine = std::convenience_random_engine >
T random( std::initializer_list< T > il );

// (3c)
template< std::random_access_iterator It,
          typename Engine = std::convenience_random_engine >
std::iter_reference_t< It > random( It first, It last );
```

- `// (1a)` Picks a uniformly distributed number of type `T` in [`lb`, `ub`]. If `T` is integer type, then `ub` is included, otherwise excluded.

  Preconditions: $-\infty <$ `lb` $\leq$ `ub` $< \infty$. If not fulfilled, then UB.

  `// (1b)` Same as `// (1a)` with default values `lb = T(0)`, `ub = T(1)`. `T` must be floating point type or `bool`.

  `// (2)` Generates `bool` with probability `p`

  Preconditions: $0 \leq$ `p` ≤ 1. If not fulfilled, then UB.

  `// (3)` Returns a random element from a random access range, each element with the same probability.

  Preconditions: Range must not be empty. If not fulfilled, then `std::out_of_range` is thrown

- User is allowed to specialize this function w.r.t. `Out` for any user defined type

1.4.3 randn

```
template< std::floating_point T,
          typename Engine = std::convenience_random_engine >
T randn( const T & mu = T(0), const T & sigma = T(1) );
```

- Picks a normally distributed number of type `T` with mean `mu` and standard deviation `sigma`
- Can be specialized for any user defined type
- Preconditions:
  - `mu`, `sigma` must be finite,
  - $\texttt{sigma}^2 \geq 0$
  - `T` must be a floating point type
- Open questions:
  - Maybe it should be $\texttt{sigma}^2 > 0$

1.4.4 seed

```
// (S1)
template< typename Engine = std::convenience_random_engine,
          typename Seed >
void seed( const Seed & s );

// (S2)
template< typename Engine = std::convenience_random_engine,
          typename Seed >
void seed();
```

- (1) Sets this threads Rng state

  (2) Sets this threads Rng state using `std::random_device{}()`

2 Impact on the Standard

- Library only change; no modifications to the core language.
- No ABI impact – All entities are inline functions in the header.
- No existing code breaks – Names are new; the proposal is a pure extension.

2.1 Feature-test macro

Add the macro `#define __cpp_lib_random_convenience 202506L`

3 Proposed implementation

```
// #include <thread>

namespace {
    template< typename Engine >
    auto & convenience_engine() {  // exposition only
//      static thread_local Engine engine(
//          std::hash< std::thread::id >{}(
//              std::this_thread::get_id() ) );
        static thread_local Engine engine{};
        return engine;
    }
}


namespace std {
```

```
using convenience_random_engine = std::philox4x64;


// (1)
template< typename T,
          typename Engine = convenience_random_engine >
    requires std::floating_point< T > ||
             std::integral< T >
T random( const T & lb, const T & ub ) {
    if constexpr( std::floating_point< T > ) {
        auto & engine = convenience_engine< Engine >();
        std::uniform_real_distribution< T > distribution( lb, ub );
        return distribution( engine );
    } else {
        auto & engine = convenience_engine< Engine >();
        std::uniform_int_distribution< T > distribution( lb, ub );
        return distribution( engine );
    }
}

template< typename T,
          typename Engine = convenience_random_engine >
    requires std::floating_point< T > ||
             std::is_same_v< T, bool >
T random() {
    if constexpr( std::floating_point< T > ) {
        auto & engine = convenience_engine< Engine >();
        std::uniform_real_distribution< T > distribution;
        return distribution( engine );
    } else {
        auto & engine = convenience_engine< Engine >();
        std::bernoulli_distribution distribution;
        return distribution( engine );
    }
}

// (2)
template< typename T,
          typename Engine = convenience_random_engine,
          std::floating_point P >
    requires std::is_same_v< T, bool >
T random( P p = P(0.5) ) {
    auto & engine = convenience_engine< Engine >();
    std::bernoulli_distribution distribution( p );
    return distribution( engine );
}


// (3)
template< std::ranges::random_access_range Range,
          typename Engine = std::convenience_random_engine >
std::ranges::range_reference_t< Range > random( Range && range ) {
    auto n = std::ranges::size( range );
    if( n == 0 ) {
        throw std::out_of_range{ "random( Range ): empty range" };
    }
    using diff_t = std::ranges::range_difference_t< Range >;
    auto idx = random< diff_t, Engine >( 0, n - 1 );
    return std::ranges::begin( range )[ idx ];
}

template< typename T,
          typename Engine = std::convenience_random_engine >
T random( std::initializer_list< T > il ) {
    auto n = il.size();
    if( n == 0 ) {
```

```
        throw std::out_of_range{ "random( initializer_list ): empty list" };
    }
    std::uniform_int_distribution<std::size_t> dist(0, n - 1);
    auto idx = random< std::size_t, Engine >( 0, n - 1 );
    return il.begin()[ idx ];
}

template< std::random_access_iterator It,
          typename Engine = std::convenience_random_engine >
std::iter_reference_t< It > random( It first, It last ) {
    auto n = last - first;
    if( n <= 0 ) {
        throw std::out_of_range{ "random( begin, end ): empty range" };
    }
    auto idx = random< decltype(n), Engine >( 0, n - 1 );
    return first[ idx ];
}



template< std::floating_point T,
          typename Engine = convenience_random_engine >
T randn( const T & mu = T(0), const T & sigma = T(1) ) {
    auto & engine = convenience_engine< Engine >();
    std::normal_distribution< T > distribution( mu, sigma );
    return distribution( engine );
}

// (S1)
template< typename Engine = convenience_random_engine,
          typename Seed >
void seed( const Seed & s ) {
    convenience_engine< Engine >().seed( s );
}

// (S2)
template< typename Engine = convenience_random_engine >
void seed() {
    std::random_device rd;
    convenience_engine< Engine >().seed( rd() );
}

}
```