Paper Number:	P1068R5
Title:	Vector API for random number generation
Authors:	Ilya Burylov <ilya.burylov@intel.com> Pavel Dyakov <pavel.dyakov@intel.com> Ruslan Arutyunyan <ruslan.arutyunyan@intel.com> Andrey Nikolaev <andrey.nikolaev@intel.com>
Audience:	LEWG
Date:	2021-04-13

I. Introduction

C++11 introduced a comprehensive mechanism to manage generation of random numbers in the <random> header file.

We propose to introduce an additional API based on iterators in alignment with algorithms definition. simd-type based interface presented in previous paper revisions will be submitted as a separate paper.

II. Revision history

Key changes for R5 compared with R4 after LEWG mail list review (Feb-March 2021):

- Renamed member function from operator() to generate()
- Replaced legacy iterators with C++20 iterator and ranges concepts.
- Added ranges-based API
- Discussed iterator constrains from performance perspective
- Added reference to __generate() API added in GNU* libstdc++
- Renamed uniform_vector_random_bit_generator to uniform_bulk_random_bit_generator and added uniform_range_random_bit_generator

Key changes for R4 compared with R3 after LEWGI review (Prague):

- Reverted changes in existing concept uniform_random_bit_generator and introduced uniform_vector_random_bit_generator. Updated corresponding wording.
- Ensured std::random_device benefits from vector API.

Key changes for R3 compared with R2 after SG1 and SG6 review (Belfast):

- Removed execution policies from API, based on Cologne meeting decision.
- Removed simd-based API, for separate consideration as a follow up paper, based on corresponding TS results.
- Added formal wording section for iterators-based API.

Key changes for R2 compared with R1 after SG1 review (Cologne):

- Proposed API for switching between Sequentially consistent and Sequentially inconsistent vectorized results.
- Added performance data measured on the prototype to show price for sequentially consistent results.
- Extended description of the role of generate_canonical in distributions implementations.
- Reworked *Possible approaches to address the problem* chapter to focus on two main approaches under consideration.

Key changes for R1 compared with R0 after SG1 review (Rapperswil):

- Extended the list of possible approaches with simd type direct usage.
- Added performance data measured on the prototype.
- Changed the recommendation to a combined approach.

III. Motivation and Scope

The C++11 random-number API is essentially a scalar one. Stateful nature and the scalar definition of underlying algorithms prevent auto-vectorization by compiler.

However, most existing algorithms for generation of pseudo- [and quasi-]random sequences allow algorithmic rework to generate numbers in batches, which allows the implementation to utilize simd-based HW instruction sets.

Internal measurements show significant scaling over simd-size for key baseline Engines yielding a substantial performance difference on the table on modern HW architectures.

Extension and/or modification of the list of supported Engines and/or Distributions is out of the scope of this proposal.

IV. Libraries and other languages

Vector APIs are common for the area of generation random numbers. Examples:

* Intel® oneAPI Math Kernel Library (oneMKL)

 - Statistical Functions component includes Random Number Generators C vector based API

* Java* java.util.Random

 - Has doubles(), ints(), longs() methods to provide a stream of random numbers

* Python* NumPy* library

 - NumPy array has a method to be filled with random numbers

* NVIDIA* cuRAND

 - host API is vector based

* GNU* libstc++ extension

GNU* libstc++ library implementation has an extension, which introduces __generate() member function to all distributions (but not engines). It was added back in 2012 jointly with simd_fast_mersenne_twister_engine implementation in order to be able to benefit from simd instructions.

Intel oneMKL can be an example of the existing vectorized implementation for verity of engines and distributions. Existing API is C [1] (and FORTRAN), but the key property which allows enabling vectorization is vector-based interface.

Another example of implementation can be intrinsics for the Short Vector Random Number Generator Library [2], which provides an API on simd level and can be considered an example of internal implementation for proposed modifications.

V. Problem description

Main flow of random number generation is defined as a 3-level flow.

User creates Engine and Distribution and calls operator() of Distribution object, providing Engine as a parameter:

```
std::array<float, arrayLength> stdArray;

std::mt19937 gen(777);
std::uniform_real_distribution dis(1.f, 2.f);

for(auto& el : stdArray)
{
    el = dis(gen);
}
```

operator() of a Distribution implements scalar algorithm and typically (but not necessarily so) calls generate_canonical(), passing Engine object further down:

```
uniform_real_distribution::operator()(_URNG& __gen)
{
    return (b() - a()) * generate_canonical<_RealType>(__gen) + a();
}
```

It is necessary to note, that C++ standard does not require calling generate_canonical() function inside any distribution implementation and it does not specify the number of Engine numbers per distribution number. Having said that, 3 main standard library implementations share the same schema, described here.

generate_canonical() has a main intention to generate enough entropy for the type used by Distribution, and it calls operator() of an Engine one or more times (number of times is a compile-time constant):

```
_RealType generate_canonical(_URNG& __gen())
{
    …
    _RealType _Sp = __gen() - _URNG::min();
    for (size_t __i = 1; __i < __k; ++__i, __base *= _Rp)
        _Sp += (__gen() - _URNG::min()) * __base;
    return _Sp / _Rp;
}
```

operator() of an Engine is (almost) always stateful, with non-trivial dependencies between iterations, which prevents any auto-vectorization:

```
mersenne_twister_engine<…>::operator()()
{
    const size_t __j = (__i_ + 1) % __n;
…
    const result_type _Yp = (__x_[__i_] & ~__mask) | (__x_[__j] & __mask);
    const size_t __k = (__i_ + __m) % __n;
    __x_[__i_] = __x_[__k] ^ __rshift<1>(_Yp) ^ (__a * (_Yp & 1));
    result_type __z = __x_[__i_] ^ (__rshift<__u>(__x_[__i_]) & __d);
    __i_ = __j;
…
    return __z ^ __rshift<__l>(__z);
}
```

operator() of the most distributions can be implemented in a way, which compiler can inline and auto-vectorize. generate_canonical() adds additional challenge for the compiler due to loop, but it is resolvable. operator() on the engine level is the key showstopper for the auto-vectorization.

VI. New API

a) Iterators-based API

The following API extension is targeting to cover generation of bigger chunks of random numbers, which allows internal optimizations hidden inside implementation.

API of Engines and Distributions is extended with iterators based API.

```
std::array<float, arrayLength> stdArray;

std::mt19937                   gen(777);
std::uniform_real_distribution dis(1.f, 2.f);

dis.generate(stdArray.begin(), stdArray.end(), gen);
```

The output of this function may or may not be equivalent to the scalar calls of the scalar API:

```
for(auto& el : stdArray)
{
    el = dis(gen);
}
```

b) Ranges-based API

Along the iterators-based API, a new ranges-based API is introduced for all corresponding structures.

```
gen.generate(range.begin(), range.end());
```

Is equivalent to:

```
gen.generate(range);
```

c) Concepts

Both iterators-based and range-based APIs are accompanied with corresponding concepts.

- `uniform_bulk_random_bit_generator` – describes generators with iterators-based API
- `uniform_range_random_bit_generator` – describes generators with range-based API
    - includes `uniform_bulk_random_bit_generator` concept as well to minimize opportunities for ambiguity

`uniform_bulk_random_bit_generator` implementation requires types of iterator and its sentinel, which makes the concept usage over-verbose:

```
template<typename T> requires uniform_bulk_random_bit_generator<
    T,
    std::vector<value_type>::iterator,
    std::vector<value_type>::iterator>
void foo(T gen);
```

We are introducing `random_access_iterator_archetype`, which is intended to describe some generic iterator with an intention to be used as a default value for template parameter of `uniform_bulk_random_bit_generator`.

This allows using of the concept in the most laconic form:

```
void foo(std::uniform_bulk_random_bit_generator auto gen);
```

Same applies to range-based concept and corresponding `random_access_range_archetype`.

VII. Wording proposal

26.6.1 Header <random> synopsis [rand.synopsis]

```
namespace std {
// 26.6.2.3, uniform random bit generator requirements
template<class G>
concept uniform_random_bit_generator = see below ;
template<class G, class I, class S>
concept uniform_bulk_random_bit_generator = see below ;
template<class G, class R>
concept uniform_range_random_bit_generator = see below ;
…
```

26.6.2.3 Uniform random bit generator requirements [rand.req.urng]

A *uniform random bit generator* g of type G is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned. [*Note*: The degree to which g's results approximate the ideal is often determined statistically. — *end note*]

```
…
template<class G>
  concept uniform_random_bit_generator =
    invocable<G&> && unsigned_integral<invoke_result_t<G&>> &&
    requires {
      { G::min() } -> same_as<invoke_result_t<G&>>;
      { G::max() } -> same_as<invoke_result_t<G&>>;
    };
```

Let g be an object of type G. G models uniform_random_bit_generator only if

- both G::min() and G::max() are constant expressions (7.7),
- G::min() < G::max(),
- G::min() <= g(),
- g() <= G::max(), and
- g() has amortized constant complexity.

A class G meets the *uniform random bit generator* requirements if G models uniform_random_bit_generator, invoke_result_t<G&> is an unsigned integer type (6.8.1), and G provides a nested *typedef-name* result_type that denotes the same type as invoke_result_t<G&>.

```
template<typename T>
using random_access_iterator_archetype = unspecified;

template<typename T>
using random_access_range_archetype = unspecified;
```

The name random_access_iterator_archetype is an implementation-defined type such that random_access_iterator<random_access_iterator_archetype> is true and indirectly_writable< random_access_iterator_archetype, T> is true.

The name random_access_range_archetype is an implementation-defined type such that ranges::random_access_range<random_access_iterator_archetype> is true and indirectly_writable< iterator_t(ranges::random_access_range_archetype), T> is true.

A program that creates an instance of random_access_iterator_archetype or random_access_range_archetype is ill-formed.

The following concept is based on unspecified trait, which defines type to G::result_type if it is present, or something, which can be used as template parameter for random_access_iterator_archetype and random_access_range_archetype otherwise.

```
// exposition only trait
```

```
template<typename G, typename = void>
struct result-type-trait {
    using type = void*;
};

template<typename G>
struct result-type-trait<G, std::void_t<typename G::result_type>> {
    using type = typename G::result_type;
};

template<class G, class I = random_access_iterator_archetype<
         typename result-type-trait<G>::type>, class S = I>
  concept uniform_bulk_random_bit_generator =
    uniform_random_bit_generator<G> &&
    random_access_iterator<I> &&
    sentinel_for<S, I> &&
    indirectly_writable<I, typename G::result_type> &&
    requires(G& g, I begin, S end) {
      { g.generate(begin, end) } -> same_as<I>;
    };

template<class G, class R = random_access_range_archetype<
         typename result-type-trait<G>::type>>
  concept uniform_range_random_bit_generator =
    uniform_bulk_random_bit_generator<G, ranges::iterator_t<R>,
                                      ranges::sentinel_t<R>> &&
    ranges::random_access_range<R> &&
    requires(G& g, R&& r) {
      { g.generate(std::forward<R>(r)) } ->
          same_as<std::ranges::borrowed_iterator_t<R>>;
    };
```

Let g be an object of type G, rb be an object of type I, re be an object of type S. Types G, I, S model uniform_bulk_random_bit_generator only if

- G models uniform_random_bit_generator,
- I models random_access_iterator,
- S and I model sentinel_for,
- I and G::result_type model indirectly_writable, and
- G has a member function generate overload, which accepts g, rb, and re as its arguments, and returns an object of type I

Let g be an object of type G, r be an object of type R. Types G and R model uniform_range_random_bit_generator only if

- G, ranges::iterator_t<R>, ranges::sentinel_t<R>, and Proj model uniform_bulk_random_bit_generator,
- R models random_access_range, and
- G has a member function generate overload, which accepts g and r as its arguments, and returns an object of type ranges::borrowed_iterator_t<R>

A class G meets the *uniform range random bit generator* requirements if G models uniform_range_random_bit_generator.

26.6.2.4 Random number engine requirements [rand.req.eng]

A *random number engine* (commonly shortened to *engine*) e of type E is a uniform *range* random bit generator that additionally meets the requirements (e.g., for seeding and for input/output) specified in this subclause.

…

The type I for parameters named first shall model random_access_iterator. The types I and S for parameters named first and last shall model sentinel_for. The type R for parameters named range shall model ranges::random_access_range.

Table 92: Random number engine requirements [tab:rand.req.eng]

Expression	Return type	Pre/post-condition	Complexity
…			
e()	T	Advances e's state e_i to e_{i+1} = TA(e_i) and returns GA(e_i)	per 26.6.2.3
e.generate(first, last)	I	With N = last – first, assigns the result of evaluations of e() through each iterator in the range [first, first + N).	$O(N)$
e.generate(range)	ranges::borrowed_iterator_t<R>	With N = ranges::size(range), assigns the result of evaluations of e() through element in the range.	$O(N)$
…			

26.6.2.5 Random number engine adaptor requirements [rand.req.adapt]

No changes

26.6.2.6 Random number distribution requirements [rand.req.dist]

The type I for parameters named first shall model random_access_iterator. The types I and S for parameters named first and last shall model sentinel_for. The type R for parameters named range shall model ranges::random_access_range.

Table 93: Random number distribution requirements [tab:rand.req.dist]

Expression	Return type	Pre/post-condition	Complexity
…			
d(g)	T	With p = d.param(), the sequence of numbers returned by successive invocations with the same object g is randomly distributed according to the associated p(z | {p}) or P(zi | {p}) function.	amortized constant number of invocations of g
d(g,p)	T	The sequence of numbers returned by successive invocations with the same objects g and p is randomly	amortized constant number of invocations of g

		distributed according to the associated p(z | {p}) or P(zi | {p}) function.	
d.generate(first, last, g)	I	With N = last – first and p = d.param(), the sequence of numbers assigned through each iterator in [first, first + N) is randomly distributed according to the associated p(z | {p}) or P(zi | {p}) function.	$O(N)$
d.generate(first, last, g, p)	I	With N = last – first, the sequence of numbers assigned through each iterator in [first, first + N) is randomly distributed according to the associated p(z | {p}) or P(zi | {p}) function.	$O(N)$
d.generate(range, g)	ranges::borrowed_iterator_t<R>	With N = ranges::size(range), and p = d.param(), the sequence of numbers assigned through each iterator in [first, first + N) is randomly distributed according to the associated p(z | {p}) or P(zi | {p}) function.	$O(N)$
d.generate(range, g, p)	ranges::borrowed_iterator_t<R>	With N = ranges::size(range), the sequence of numbers assigned through each iterator in [first, first + N) is randomly distributed according to the associated p(z | {p}) or P(zi | {p}) function.	$O(N)$
…			

26.6.6 Class random_device [rand.device]

A random_device uniform range random bit generator produces nondeterministic random numbers.

…

VIII. Design considerations

a) Naming

Previous revisions of the paper proposed operator(begin, end) as the interface for vector generation.

We changed it to member function generate(begin, end) starting revision R5.

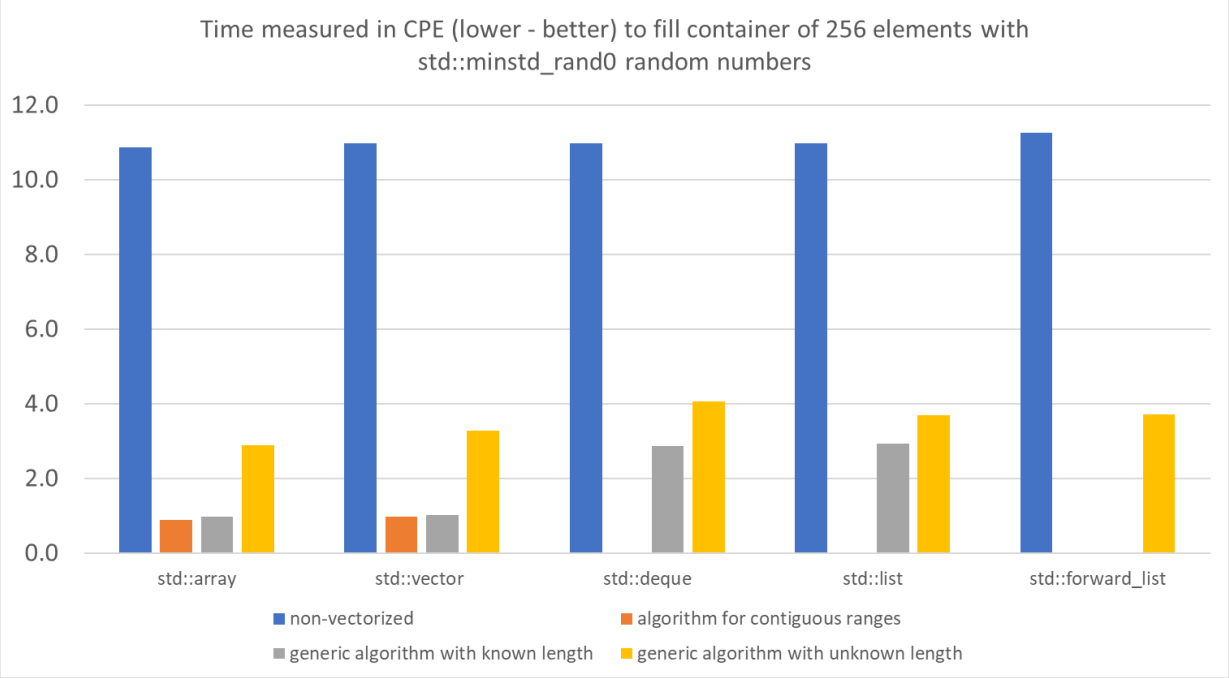
This may allow using engines and random_device (but not distributions) as seed sequences, if other modifications proposed by P0205R0 [5] be accepted.

This may be useful not only for random_device, since there are examples of usage of simpler engines to initialize a big initial state for more sophisticated engines. See, for example, *subtract_with_carry_engine* description, which is initialized by *linear_congruential_engine* by default.

b) Constraining iterators and ranges

Additional performance results were collected to support discussion regarding additional constrains for iterators and ranges. One would likely need 3 different internal implementations to get optimal performance on existing sequence container:

- Algorithm optimal for contiguous ranges
- Generic algorithm for ranges on known length
- Generic algorithm for ranges on unknown length



CPE – cycles per element of container filled with number. Measured on Intel® Xeon® Gold 6152 Processor.

- Contiguous-aware implementation is simpler than generic implementation, but performance benefit can be marginal.
- Knowledge of length allows one making decisions on vectorization of small chunks of numbers and we get a measurable overhead if length is unavailable.

Our experience show, that contiguous use case is the most important for this API. Random access use case is possible in unfortunate case of AOS data layout. Other use cases are exotic for random numbers generation and are not expected to be the target for vectorization.

We choose to limit iterators and ranges to random access type, with our understanding of use cases and guarantee of knowledge of length.

c) Concepts for distributions

Formal concept for engine was defined by this paper, while the concept for distribution was omitted.

There are no existing concepts defined for distributions and we decided that formal definition will worth a separate paper.

d) Bit-to-bit Reproducibility

Current proposal introduces new API for engines, engine adaptors, distributions and random_device.

Engines and *engine adaptors* will guarantee bit-to-bit equivalence of those two use cases:

```
gen.generate(range.begin(), range.end());
```

and

```
for(auto& el : range)
{
    el = gen();
}
```

This guarantee is required to keep statistical properties of random numbers sequence, which was justified for each specific engine by its authors in corresponding papers.

random_device sequencies are not guaranteed to be anyhow reproducible by fundamental design of this feature.

Bit-to-bit reproducibility of *distributions* sequencies is not guaranteed by current wording of the standard even for scalar API. One of the main reasons for that is to allow different distribution algorithms be implemented by different standard libraries. The main guarantee provided by the standard in this case is statistical property of the sequence of the numbers.

Same applies to new API – those two code snippets will result in values which have the same statistical properties, but not necessarily bit-to-bit exact values:

```
dis.generate(range.begin(), range.end(), gen);
```

and

```
for(auto& el : range)
{
    el = dis(gen);
}
```

IX. Performance results

Implementation approaches were prototyped in part of Distribution API (and Engine API, where required for the use case). See P1068R2 for performance results.

X. Impact on the standard

This is a library-only extension. It adds new member functions to some classes. This change is ABI compatible with existing random numbers generation functionality.

XI. References

1. Intel oneMKL documentation: https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/statistical-functions/random-number-generators/basic-generators.html
2. Intrinsics for the Short Vector Random Number Generator Library https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-the-short-vector-random-number-generator-library.html
3. Box-Muller method https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform
4. Inverse transform sampling https://en.wikipedia.org/wiki/Inverse_transform_sampling
5. Allow Seeding Random Number Engines with std::random_device http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0205r0.html

Legal Disclaimer & Optimization Notice

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804