The Rust Rand Book

This is the extended documentation for Rust's **Rand**om number library.

This book contains:

1. Quick start
2. An overview of crates, features and portability
3. The Users' Guide
4. Updating guides
5. Contributor's guide

Outside this book, you may want:

- API reference for the latest release
- API reference for the master branch
- The Rand repository
- The Book source

Quick start

Below we list a short example. For more, please refer to the API documentation or the guide.

Lets kick things off with an example (playground link):

```rust
// import commonly used items from the prelude:
use rand::prelude::*;

fn main() {
    // We can use random() immediately. It can produce values of many common
types:
    let x: u8 = rand::random();
    println!("{}", x);

    if rand::random() { // generates a boolean
        println!("Heads!");
    }

    // If we want to be a bit more explicit (and a little more efficient) we
can
    // make a handle to the thread-local generator:
    let mut rng = rand::rng();
    if rng.random() { // random bool
        let x: f64 = rng.random(); // random number in range [0, 1)
        let y = rng.random_range(-10.0..10.0);
        println!("x is: {}", x);
        println!("y is: {}", y);
    }

    println!("Dice roll: {}", rng.random_range(1..=6));
    println!("Number from 0 to 9: {}", rng.random_range(0..10));

    // Sometimes it's useful to use distributions directly:
    let distr = rand::distr::Uniform::new_inclusive(1, 100).unwrap();
    let mut nums = [0i32; 3];
    for x in &mut nums {
        *x = rng.sample(distr);
    }
    println!("Some numbers: {:?}", nums);

    // We can also interact with iterators and slices:
    let arrows_iter = "➡⬈⬆⬉⬅⬋⬇⬊".chars();
    println!("Lets go in this direction: {}", arrows_iter.choose(&mut
rng).unwrap());
    let mut nums = [1, 2, 3, 4, 5];
    nums.shuffle(&mut rng);
    println!("I shuffled my {:?}", nums);
}
```

The first thing you may have noticed is that we imported everything from the prelude. This is the lazy way to `use` rand, and like the standard library's prelude, only imports the most common items. If you don't wish to use the prelude, remember to import the `Rng` trait!

The Rand library automatically initialises a secure, thread-local generator on demand. This can be accessed via the `rng()` and `random` functions. For more on this topic, see Random generators.

While the `random` function can only sample values in a `StandardUniform` (type-dependent) manner, `rng()` gives you a handle to a generator. All generators implement the `Rng` trait, which provides the `random`, `random_range` and `sample` methods used above.

Rand provides functionality on iterators and slices via two more traits, `IteratorRandom` and `SliceRandom`.

Fixed seed RNGs

You may have noticed the use of `rand::rng()` above and wondered how to specify a fixed seed. To do so, you need to specify an RNG then use a method like `seed_from_u64` or `from_seed`.

Note that `seed_from_u64` is **not suitable for cryptographic uses** since a single `u64` cannot provide sufficient entropy to securely seed an RNG. All cryptographic RNGs accept a more appropriate seed via `from_seed`.

We use `ChaCha8Rng` below because it is fast and portable with good quality. See the RNGs section for more RNGs, but avoid `SmallRng` and `StdRng` if you care about reproducible results.

```
use rand::{Rng, SeedableRng};

fn main() {
    let mut rng = rand_chacha::ChaCha8Rng::seed_from_u64(10);
    println!("Random f32: {}", rng.random::<f32>());
}
```

The crate family

```
                                             ┌ statrs
getrandom ┐                                  ├ rand_distr
          └ rand_core ┬─────────────┬ rand ┘
                      ├ rand_chacha ┘
                      ├ rand_pcg
                      └ [other RNG crates]
```

Interfaces

`rand_core` defines `RngCore` and other core traits, as well as several helpers for implementing RNGs.

The `getrandom` crate provides a low-level API around platform-specific random-number sources.

Pseudo-random generators

The following crates implement pseudo-random number generators (see Our RNGs):

- `rand_chacha` provides generators using the ChaCha cipher
- `rand_hc` implements a generator using the HC-128 cipher
- `rand_isaac` implements the ISAAC generators
- `rand_pcg` implements a small selection of PCG generators
- `rand_xoshiro` implements the SplitMix and Xoshiro generators
- `rand_xorshift` implements the basic Xorshift generator

Exceptionally, `SmallRng` is implemented directly in `rand`.

rand (main crate)

The `rand` crate is designed for easy usage of common random-number functionality. This has several aspects:

- the `rngs` module provides a few convenient generators

- the `distr` module concerns sampling of random values
- the `seq` module concerns sampling from and shuffling sequences
- the `Rng` trait provides a few convenience methods for generating random values
- the `random` function provides convenient generation in a single call

Distributions

The `rand` crate only implements sampling from the most common random number distributions: uniform and weighted sampling. For everything else,

- `rand_distr` provides fast sampling from a variety of other distributions, including Normal (Gauss), Binomial, Poisson, UnitCircle, and many more
- `statrs` is a port of the C# Math.NET library, implementing many of the same distributions (plus/minus a few), along with PDF and CDF functions, the *error*, *beta*, *gamma* and *logistic* special functions, plus a few utilities. (For clarity, `statrs` is not part of the Rand library.)

Crate features

It is recommended to check the crate's `Cargo.toml` or `README.md` for features. Since `rand v0.9`, `rust-random` crates only use explicit features (i.e. all features are listed under `[features]`).

Release versions of `Cargo.toml` can be viewed on `docs.rs`:

- https://docs.rs/crate/rand/latest/source/Cargo.toml.orig
- https://docs.rs/crate/rand_core/latest/source/Cargo.toml.orig
- https://docs.rs/crate/rand_distr/latest/source/Cargo.toml.orig
- https://docs.rs/crate/rand_chacha/latest/source/Cargo.toml.orig
- https://docs.rs/crate/rand_xoshiro/latest/source/Cargo.toml.orig
- https://docs.rs/crate/rand_pcg/latest/source/Cargo.toml.orig

Common features

The following features are common to `rand_core`, `rand`, `rand_distr` and potentially some RNG crates:

- `std`: opt into functionality dependent on the `std` lib. This is default-enabled except in `rand_core`; for `no_std` usage, use `default-features = false`.
- `alloc`: enables functionality requiring an allocator (for usage with `no_std`). This is implied by `std`.
- `serde`: enables serialization via `serde`, version 1.0.

rand_distr features

The floating point functions from `num_traits` and `libm` are used to support `no_std` environments and ensure reproducibility. If the floating point functions from `std` are preferred, which may provide better accuracy and performance but may produce different random values, the `std_math` feature can be enabled. (Note that any other crate depending on `num-traits`'s `std` feature (default-enabled) will have the same effect.)

Platform support

Thanks to many community contributions, Rand crates support a wide variety of platforms.

no_std

With `default-features = false`, both `rand` and `rand_distr` support `no_std` builds. See Common features.

getrandom

The `getrandom` crate provides a low-level API around platform-specific random-number sources, and is an important building block of `rand` and `rand_core` as well as a number of cryptography libraries. It is not intended for usage outside of low-level libraries.

WebAssembly

The `wasm32-unknown-unknown` target does not make any assumptions about which JavaScript interface is available, thus the `getrandom` crate requires configuration. See WebAssembly support.

Note that the `wasm32-wasi` and `wasm32-unknown-emscripten` targets do not have this limitation.

Reproducibility

The `rust-random` libraries make limited commitments to reproducibility of seedable PRNGs and stochastic algorithms.

This chapter concerns value-stability of deterministic processes using the `rust-random` libraries.

API-breaking, value-breaking and SemVer

A change (to a library) is considered **API-breaking** if it may cause a compilation failure of code which was compatible with a prior version of the API, or is otherwise an incompatible change.

We aim to follow SemVer rules regarding API-breaking changes and `MAJOR.MINOR.PATCH` versions. That is, post 1.0, new minor versions should not introduce API-breaking changes.

A change is considered **value-breaking** if it is not API-breaking yet would result in changed output values of a deterministic stochastic process using only unchanged parts of the `rust-random` API.

Value-breaking changes are permitted in minor versions.

Non-portable deterministic items

An item in a `rust-random` API (such as a struct or function) may be declared to be **non-portable**, meaning that it opts out of all reproducibility guarantees. Non-portable items may be deterministic, yet yield different results on different platforms and library versions (they may make value-breaking changes in any release).

This is a change in policy affecting `rand` from version `0.10` or `1.0` (whichever release is next); up to version `0.9` non-portable items were not permitted to make value-breaking changes in patch releases.

This non-portable declaration must be clearly mentioned in documentation. The following items make such a declaration:

- `rand::rngs::SmallRng`

- `rand::rngs::StdRng`

Portable items

Some items are clearly non-deterministic (e.g. `rand::rng`). Some items are deterministic but non-portable (above). All other parts of the public API of `rust-random` crates (including PRNGs, distributions and other stochastic algorithms) are expected to be portable:

- Results should be reproducible across platforms
- Results should be reproducible across patch releases
- Minor releases, including after 1.0, may make value-breaking changes to portable items. Such changes must be well motivated and should be clearly mentioned in the CHANGELOG.

Testing

We expect all portable stochastic algorithms to test the value-stability of their output with some form of test vector.

- PRNGs should test against a reference vector where available (example)
- Other algorithms should include their own test vectors within a `value_stability` test or similar (example)

Support for prior versions

We aim to support users of `rust-random` crates using a prior `MAJOR.MINOR` version for the purposes of reproducibility by:

- Providing security fixes as patch versions where appropriate
- Facilitating the back-porting of compatible additions from future crate versions *on request*
- Other fixes may be considered for back-porting, but are often not possible without API-breaking or value-breaking changes

Limitations

Portability of usize

There is unfortunately one non-portable item baked into the heart of the Rust language: `usize` (and `isize`). For example, the size of an empty `Vec` will differ on 32-bit and 64-bit targets. For most purposes this is not an issue, but when it comes to generating random numbers in a portable manner it does matter.

A simple rule follows: if portability is required, *never* sample a `usize` or `isize` value directly.

From `rand v0.9`, `isize` and `usize` types are no longer supported in many parts of the public API, including `StandardUniform`. `usize` is supported by `SampleUniform` and thus `Rng::random_range`, using `u32` sampling whenever possible to maximise portability.

Portability of floats

The results of floating point arithmetic depend on rounding modes and implementation details. In particular, the results of transcendental functions vary from platform to platform. Due to this, results of distributions in `rand_distr` using `f32` or `f64` may not be portable.

To alleviate (or further complicate) this concern, we prefer to use `libm` over `std` implementations of these transcendental functions. See rand_distr features.

Guide

This section attempts to explain some of the concepts used in this library.

1. Getting started with a new crate
2. What is random data and what is randomness anyway?
3. What kind of random generators are there?
4. What random number generators does Rand provide?
5. Seeding PRNGs and reproducibility
6. Parallel RNGs
7. Turning random data into useful values
8. Distributions: more control over random values
9. Random processes: sampling without replacement
10. Sequences
11. Error handling
12. Testing functions which use RNGs

Importing items (prelude)

The most convenient way to import items from Rand is to use the `prelude`. This includes the most important parts of Rand, but only those unlikely to cause name conflicts.

Note that Rand 0.5 has significantly changed the module organization and contents relative to previous versions. Where possible old names have been kept (but are hidden in the documentation), however these will be removed in the future. We therefore recommend migrating to use the prelude or the new module organization in your imports.

Further examples

For some inspiration, see the example applications:

- Monte Carlo estimation of π
- Monty Hall Problem

Getting started

If you haven't already, install Rust.

Next, lets make a new crate and add rand as a dependency:

```
cargo new randomly
cd randomly
cargo add rand --features small_rng
```

Now, paste the following into `src/main.rs`:

```
use rand::prelude::*;

fn main() {
    let mut rng = rand::rng();

    println!("Random die roll: {}", rng.random_range(1..=6));
    println!("Random UUID: 0x{:X}", rng.random::<u128>());

    if rng.random() {
        println!("You got lucky!");
    }
}
```

Now lets go!

```
$ cargo run
   Compiling [..]
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.99s
     Running `target/debug/randomly`
Random die roll: 4
Random UUID: 0xEC3936A465339F8295EE11AB853CCDBF
You got lucky!
```

Other crates

Some other crates are used by this guide. When needed, you can either edit the `[dependencies]` section of your `Cargo.toml` or use `cargo add`:

```
$ cargo add rand_distr
    Updating crates.io index
      Adding rand_distr v0.4.3 to dependencies
             Features:
             + alloc
             + std
             - serde
             - serde1
             - std_math
    Updating crates.io index
```

Random data

```
// get some random data:
let mut data = [0u8; 8];
rand::rng().fill_bytes(&mut data);
println!("{:?}", data)
```

What is randomness?

What does **random** mean? Colloquially the word can mean simply *unexpected* or *unknown*, but we need to be a bit more precise than that. Wikipedia gives us a more specific definition:

> **Randomness** is the lack of pattern or predictability in events.

We can take this further: *lack of pattern* implies there is no *bias*; in other words, all possible values are equally likely.

To understand what a *random value* is, we still need a context: what pool of numbers can our random value come from?

- To give a simple example, consider dice: they have values 1, 2, 3, 4, 5 and 6, and an unbiased (fair) die will make each number equally likely, with probability ⅙th.
- Now lets take a silly example: the natural numbers (1, 2, 3, etc.). These numbers have no limit. So if you were to ask for an unbiased random natural number, 1, 5, 1000, 1 million, 1 trillion — all would be equally likely. In fact, for *any* natural number `k`, the numbers `1, 2, ..., k` are an infinitely small fraction of all the natural numbers, which means the chance of picking a unbiased number from this range is effectively `1/∞ = 0`. Put another way: for *any* natural number, we expect an unbiased random value to be bigger. This is impossible, so there cannot be any such thing as an unbiased random natural number.
- Another example: real numbers between 0 and 1. Real numbers include all the fractions, irrational numbers like π and √2, and all multiples of those... there are infinitely many possibilities, even in a small range like `(0, 1)`, so simply saying "all possibilities are equally likely" is not enough. Instead we interpret *lack of pattern* in a different way: every interval of equal size is equally likely; for example we could subdivide the interval `0,1` into `0,½` and `½,1` and toss a coin to decide which interval our random sample comes from. Say we pick `½,1` we can then toss another coin to

decide between `½,¾` and `¾,1`, restricting our random value to an interval of size `¼`. We can repeat this as many times as necessary to pick a random value between `0` and `1` with as much precision as we want — although we should realise that we are not choosing an *exact* value but rather just a small interval.

What we have defined (or failed to define) above are uniform random number distributions, or simply **uniform distributions**. There are also non-uniform distributions, as we shall see later. It's also worth noting here that a uniform distribution does not imply that its samples will be *evenly* spread (try rolling six dice: you probably won't get 1, 2, 3, 4, 5, 6).

To bring us back to computing, we can now define what a uniformly distributed random value (an unbiased random value) is in several contexts:

- `u32`: a random number between 0 and `u32::MAX` where each value is equally likely
- `BigInt`: since this type has no upper bound, we cannot produce an unbiased random value (it would be infinitely large, and use infinite amounts of memory)
- `f64`: we treat this as an approximation of the real numbers, and, *by convention*, restrict to the range 0 to 1 (if not otherwise specified). We will come back to the conversions used later; for now note that these produce 52-53 bits of precision (depending on which conversion is used, output will be in steps of `ε` or `ε/2`, where `1+ε` is the smallest representable value greater than `1`).

Random data

As seen above, the term "random number" is meaningless without context. "Random data" typically means a sequence of random *bytes*, where for each byte, each of the 256 possible values are equally likely.

`RngCore::fill_bytes` produces exactly this: a sequence of random bytes.

If a sequence of unbiased random bytes of the correct length is instead interpreted as an integer — say a `u32` or `u64` — the result is an unbiased integer. Since this conversion is trivial, `RngCore::next_u32` and `RngCore::next_u64` are part of the same trait. (In fact the conversion is often the other way around — algorithmic generators usually work with integers internally, which are then converted to whichever form of random data is required.)

Types of generators

The previous section introduced `RngCore`, the trait which all *random data sources* must implement. But what exactly is a random data source?

This section concerns theory; see also the chapter on random number generators.

```
use rand::{Rng, SeedableRng};

// prepare a non-deterministic random number generator:
let mut rng = rand::rng();
println!("{}", rng.random::<i32>()); // prints an unknown value

// prepare a deterministic generator:
let mut rng = rand_chacha::ChaCha8Rng::seed_from_u64(123);
println!("{}", rng.random::<i32>()); // prints -416273517
```

True random number generators

A **true** random number generator (TRNG) is something which produces random numbers by observing some natural process, such as atomic decay or thermal noise. (Whether or not these things are *truly* random or are in fact deterministic — for example if the universe itself is a simulation — is besides the point here. For our purposes, it is sufficient that they are not distinguishable from true randomness.)

Note that these processes are often biased, thus some type of *debiasing* must be used to yield the unbiased random data we desire.

Pseudo-random number generators

CPUs are of course supposed to compute deterministically, yet it turns out they can do a pretty good job of emulating random processes. Most pseudo-random number generators are deterministic and can be defined by just:

- some initial *state*
- a function to compute a random value from the state
- a function to advance to the next state
- (optionally) a function to derive the initial state from a *seed* or *key*

The fact that these are deterministic can sometimes be very useful: it allows a simulation, randomised art work or game to be repeated exactly, producing a result which is a function of the seed. For more on this see the reproducibility chapter (note that determinism alone isn't enough to guarantee reproducibility).

The other big attraction of PRNGs is their speed: some of these algorithms require only a few CPU operations per random value, and thus can produce random data on demand much more quickly than most TRNGs.

Note however that PRNGs have several limitations:

- They are no stronger than their seed: if the seed is known or guessable, and the algorithm is known (or guessed), then only a small number of output sequences are likely.
- Since the state size is usually fixed, only a finite number of output values are possible before the generator loops and repeats itself.
- Several algorithms are easily predictable after seeing a few values, and with many other algorithms it is not clear whether they could be "cracked".

Cryptographically secure pseudo-random number generator

Cryptographically secure pseudo-random number generators (CSPRNGs) are the subset of PRNGs which are considered secure. That is:

- their state is sufficiently large that a brute-force approach simply trying all initial values is not a feasible method of finding the initial state used to produce an observed sequence of output values,
- and there is no other algorithm which is sufficiently better than the brute-force method which would make it feasible to predict the next output value.

Achieving secure generation requires not only a secure algorithm (CSPRNG), but also a secure and sufficiently large seed value (typically 256 bits), and protection against side-channel attacks (i.e. preventing attackers from reading the internal state).

Some CSPRNGs additionally satisfy a third property:

- a CSPRNG is backtracking resistant if it is impossible for an attacker to calculate prior output values of the PRNG despite having discovered the value of the current internal state (implying that all future output is compromised).

Hardware random number generator

A **hardware** random number generator (HRNG) is theoretically an adaptor from some TRNG to digital information. In practice, these may use a PRNG to debias the TRNG. Even though an HRNG has some underlying TRNG, it is not guaranteed to be secure: the TRNG itself may produce insufficient entropy (i.e. be too predictable), or the signal amplification and debiasing process may be flawed.

An HRNG may be used to provide the seed for a PRNG, although usually this is not the only way to obtain a secure seed (see the next section). An HRNG might replace a PRNG altogether, although since we now have very fast and very strong software PRNGs, and since software implementations are easier to verify than hardware ones, this is often not the preferred solution.

Since a PRNG needs a random seed value to be secure, an HRNG may be used to provide that seed, or even replace the need for a PRNG. However, since the goal is usually "only" to produce unpredictable random values, there are acceptable alternatives to *true* random number generators (see next section).

Entropy

As noted above, for a CSPRNG to be secure, its seed value must also be secure. The word *entropy* can be used in two ways:

- as a measure of the amount of unknown information in some piece of data
- as a piece of unknown data

Ideally, a random boolean or a coin flip has 1 bit of entropy, although if the value is biased, there will be less. Shannon Entropy attempts to measure this.

For example, a Unix time-stamp (seconds since the start of 1970) contains both high- and low-resolution data. This is typically a 32-bit number, but the amount of *entropy* will depend on how precisely a hypothetical attacker can guess the number. If an attacker can guess the number to the nearest minute, this may be approximately 6 bits (2^6 = 64); if an attacker can guess this to the second, this is 0 bits. `JitterRng` uses this concept to scavenge entropy without an HRNG (but using nanosecond resolution timers and conservatively assuming only a couple of bits entropy is available per time-stamp, after running several tests on the timer's quality).

Our RNGs

There are many kinds of RNGs, with different trade-offs. Rand provides some convenient generators in the `rngs` module. Often you can just use `rand::rng`, a function which automatically initializes an RNG in thread-local memory and returns a reference to it. It is fast, good quality, and (to the best of our knowledge) cryptographically secure.

Contents of this documentation:

The generators

Basic pseudo-random number generators (PRNGs)

The goal of "standard" non-cryptographic PRNGs is usually to find a good balance between simplicity, quality, memory usage and performance. Non-cryptographic generators pre-date cryptographic ones and are in some ways obsoleted by them, however non-cryptographic generators do have some advantages: a small state size, fast initialisation, simplicity, lower energy usage for embedded CPUs. (However, not all non-crypto PRNGs provide these benefits, e.g. the Mersenne Twister has a very large state despite being easy to predict).

These algorithms are very important to Monte Carlo simulations, and also suitable for several other problems such as randomized algorithms and games, where predictability is not an issue. (Note however that for gambling games predictability may be an issue and a cryptographic PRNG is recommended.)

The Rand project provides several non-cryptographic PRNGs. A sub-set of these are summarised below. You may wish to refer to the pcg-random and xoshiro websites.

name	full name	performance	memory	quality
`SmallRng`	(unspecified)	7 GB/s	16 bytes	★★★☆☆

name	full name	performance	memory	quality
`Pcg32`	PCG XSH RR 64/32 (LCG)	3 GB/s	16 bytes	★★★☆☆
`Pcg64`	PCG XSL 128/64 (LCG)	4 GB/s	32 bytes	★★★☆☆
`Pcg64Mcg`	PCG XSL 128/64 (MCG)	7 GB/s	16 bytes	★★★☆☆
`XorShiftRng`	Xorshift 32/128	5 GB/s	16 bytes	★☆☆☆☆
`Xoshiro256PlusPlus`	Xoshiro256++	7 GB/s	32 bytes	★★★☆☆
`Xoshiro256Plus`	Xoshiro256+	8 GB/s	32 bytes	★★☆☆☆
`SplitMix64`	splitmix64	8 GB/s	8 bytes	★☆☆☆☆
`StepRng`	counter	51 GB/s	16 bytes	☆☆☆☆☆

Here, performance is measured roughly for `u64` outputs on a 3.4GHz Haswell CPU (note that this will vary significantly by application; in general cryptographic RNGs do better with byte sequence output). Quality ratings are based on theory and observable defects, roughly as follows:

- ★☆☆☆☆ = suitable for simple applications but with significant flaws
- ★★☆☆☆ = no major issues in qualitative testing
- ★★★☆☆ = good theory, no major issues in qualitative testing
- ★★★★★ = cryptographic quality

Cryptographically secure pseudo-random number generators (CSPRNGs)

CSPRNGs have much higher requirements than basic PRNGs. The primary consideration is security. Performance and simplicity are also important, but in general CSPRNGs are more complex and slower than regular PRNGs. Quality is no longer a concern, as it is a requirement for a CSPRNG that the output is basically indistinguishable from true randomness since any bias or correlation makes the output more predictable.

There is a close relationship between CSPRNGs and cryptographic ciphers. Any block cipher can be turned into a CSPRNG by encrypting a counter. Stream ciphers are basically a CSPRNG and a combining operation, usually XOR. This means that we can easily use any stream cipher as a CSPRNG.

This library provides the following CSPRNGs. We can make no guarantees of any security claims. This table omits the "quality" column from the previous table since CSPRNGs may not have observable defects.

name	full name	performance	initialization	memory
`StdRng`	(unspecified)	1.5 GB/s	fast	136 bytes
`ChaCha20Rng`	ChaCha20	1.8 GB/s	fast	136 bytes
`ChaCha8Rng`	ChaCha8	2.2 GB/s	fast	136 bytes
`Hc128Rng`	HC-128	2.1 GB/s	slow	4176 bytes
`IsaacRng`	ISAAC	1.1 GB/s	slow	2072 bytes
`Isaac64Rng`	ISAAC-64	2.2 GB/s	slow	4136 bytes

It should be noted that the ISAAC generators are only included for historical reasons: they have been with the Rust language since the very beginning. They have good quality output and no attacks are known, but have received little attention from cryptography experts.

Notes on generators

Performance

First it has to be said most PRNGs are very fast, and will rarely be a performance bottleneck.

Performance of basic PRNGs is a bit of a subtle thing. It depends a lot on the CPU architecture (32 vs. 64 bits), inlining, and also on the number of available registers. This often causes the performance to be affected by surrounding code due to inlining and other usage of registers.

When choosing a PRNG for performance it is important to benchmark your own application due to interactions between PRNGs and surrounding code and dependence on the CPU architecture as well as the impact of the size of data requested. Because of all this, we do not include performance numbers here but merely a qualitative rating.

CSPRNGs are a little different in that they typically generate a block of output in a cache, and pull outputs from the cache. This allows them to have good amortised performance, and reduces or completely removes the influence of surrounding code on the CSPRNG performance.

Worst-case performance

Simple PRNGs typically produce each random value on demand. In contrast, CSPRNGs usually produce a whole block at once, then read from this cache until it is exhausted, giving them much less consistent performance when drawing small quantities of random data.

Memory usage

Simple PRNGs often use very little memory, commonly only a few words, where a *word* is usually either `u32` or `u64`. This is not true for all non-cryptographic PRNGs however, for example the historically popular Mersenne Twister MT19937 algorithm requires 2.5 kB of state.

CSPRNGs typically require more memory; since the seed size is recommended to be at least 192 bits and some more may be required for the algorithm, 256 bits would be approximately the minimum secure size. In practice, CSPRNGs tend to use quite a bit more, `ChaChaRng` is relatively small with 136 bytes of state.

Initialization time

The time required to initialize new generators varies significantly. Many simple PRNGs and even some cryptographic ones (including `ChaChaRng`) only need to copy the seed value and some constants into their state, and thus can be constructed very quickly. In contrast, CSPRNGs with large state require an expensive key-expansion.

Quality

Many basic PRNGs are not much more than a couple of bitwise and arithmetic operations. Their simplicity gives good performance, but also means there are small regularities hidden in the generated random number stream.

How much do those hidden regularities matter? That is hard to say, and depends on how the RNG gets used. If there happen to be correlations between the random numbers and the algorithm they are used in, the results can be wrong or misleading.

A random number generator can be considered good if it gives the correct results in as many applications as possible. The quality of PRNG algorithms can be evaluated to some extent analytically, to determine the cycle length and to rule out some correlations. Then there are empirical test suites designed to test how well a PRNG performs on a wide range of possible uses, the latest and most complete of which are TestU01 and PractRand.

CSPRNGs tend to be more complex, and have an explicit requirement to be unpredictable. This implies there must be no obvious correlations between output values.

Quality stars:

PRNGs with 3 stars or more should be good enough for most non-crypto applications. 1 or 2 stars may be good enough for typical apps and games, but do not work well with all algorithms.

Period

The *period* or *cycle length* of a PRNG is the number of values that can be generated after which it starts repeating the same random number stream. Many PRNGs have a fixed-size period, while for others ("chaotic RNGs") the cycle length may depend on the seed and short cycles may exist.

Note that a long period does not imply high quality (e.g. a counter through `u128` values provides a decently long period). Conversely, a short period may be a problem, especially when multiple RNGs are used simultaneously. In general, we recommend a period of at least 2^{128}. (Alternatively, a PRNG with shorter period of at least 2^{64} and support for multiple streams may be sufficient. Note however that in the case of PCG, its streams are closely correlated.)

Avoid reusing values! On today's hardware, a fast RNG with a cycle length of *only* 2^{64} can be used sequentially for centuries before cycling. However, when multiple RNGs are used in parallel (each with a unique seed), there is a significant chance of overlap between the sequences generated. For a generator with a *large* period `P`, `n` independent generators, and a sequence of length `L` generated by each generator, the chance of any overlap between sequences can be approximated by `Ln² / P` when `nL / P` is close to zero. For more on this topic, please see these remarks by the Xoshiro authors.

Collisions and the birthday paradox! For a generator with outputs of equal size to its state, it is recommended not to use more than `√P` outputs. A generalisation for `kw`-bit state and `w`-bit generators is to ensure `kL² < P`. This requirement stems from the *generalised birthday problem*, asking how many unbiased samples from a set of size `d = 2^w` can be taken before the probability of a repeat is at least half. Note that for `kL² > P` a generator with `kw`-dimensional equidistribution *cannot* generate the expected number of repeated samples, however generators without this property are *also* not guaranteed to generate the expected number of repeats.

Security

Predictability

From the context of any PRNG, one can ask the question *given some previous output from the PRNG, is it possible to predict the next output value?* This is an important property in any situation where there might be an adversary.

Regular PRNGs tend to be predictable, although with varying difficulty. In some cases prediction is trivial, for example plain Xorshift outputs part of its state without mutation, and prediction is as simple as seeding a new Xorshift generator from four `u32` outputs. Other generators, like PCG and truncated Xorshift* are harder to predict, but not outside the realm of common mathematics and a desktop PC.

The basic security that CSPRNGs must provide is the infeasibility to predict output. This requirement is formalized as the next-bit test; this is roughly stated as: given the first *k* bits of a random sequence, the sequence satisfies the next-bit test if there is no algorithm able to predict the next bit using reasonable computing power.

A further security that *some* CSPRNGs provide is forward secrecy: in the event that the CSPRNGs state is revealed at some point, it must be infeasible to reconstruct previous states or output. Note that many CSPRNGs *do not* have forward secrecy in their usual formulations.

Verifying security claims of an algorithm is a *hard problem*, and we are not able to provide any guarantees of the security of algorithms used or recommended by this project. We refer you to the NIST institute and ECRYPT network for recommendations.

State and seeding

It is worth noting that a CSPRNG's security relies absolutely on being seeded with a secure random key. Should the key be known or guessable, all output of the CSPRNG is easy to guess. This implies that the seed should come from a trusted source; usually either the OS or another CSPRNG. For this purpose, we recommend using the `getrandom` crate which interfaces the OS's secure random interface. `SeedableRng::from_os_rng` is a wrapper around `getrandom` for convenience. Alternatively, using a user-space CSPRNG such as `ThreadRng` for seeding should be sufficient.

Further, it should be obvious that the internal state of a CSPRNG must be kept secret. With that in mind, our implementations do not provide direct access to most of their internal state, and `Debug` implementations do not print any internal state. This does not fully protect CSPRNG state; code within the same process may read this memory (and we allow cloning and serialisation of CSPRNGs for convenience). Further, a running process may be forked by the operating system, which may leave both processes with a copy of the same generator.

Not a cryptography library

Cryptographic processes such as encryption and authentication are complex and must be implemented very carefully to avoid flaws and resist known attacks. It is therefore recommended to use specialized libraries where possible, for example openssl, ring and the RustCrypto libraries.

The Rand crates attempt to provide unpredictable data sources, with limitations. First, the software is provided "as is", without any form of guarantee. Second, it is generally assumed that program memory is private; if there are concerns in this regard it may be preferred to use an external generator such as `getrandom` instead. Note that even privacy of freed memory is important, and that while we may integrate some mitigations such as zeroize in the future, such measures are incomplete. Note that Rand does not protect against process forks (past versions of Rand up to 0.8.x have a limited mitigation but not full protection). Finally, note that there are many possible ways that the security of unpredictability could be broken, from complex hardware bugs like Spectre to stupid mistakes like printing generator state in log messages.

Extra features

Some PRNGs may provide extra features, like:

- Support for multiple streams, which can help with parallel tasks.
- The ability to jump or seek around in the random number stream; with a large period this can be used as an alternative to streams.

Further reading

There is quite a lot that can be said about PRNGs. The PCG paper is very approachable and explains more concepts.

Another good paper about RNG quality is "Good random number generators are (not so) easy to find" by P. Hellekalek.

Seeding RNGs

As we have seen, the output of pseudo-random number generators (PRNGs) is determined by their initial state.

Some PRNG definitions specify how the initial state should be generated from a key, usually specified as a byte-sequence for cryptographic generators or, for small PRNGs, often just a word. We formalise this for all our generators with the `SeedableRng` trait.

Note: seeding does not imply reproducibility of results. For that you need to use a named RNG with a fixed algorithm (e.g. `ChaCha12Rng` not `StdRng`). See also Reproducibility.

The Seed type

We require all seedable RNGs to define a `Seed` type satisfying `AsMut<[u8]> + Default + Sized` (usually `[u8; N]` for a fixed `N`). We recommend using `[u8; 12]` or larger for non-cryptographic PRNGs and `[u8; 32]` for cryptographic PRNGs.

PRNGs may be seeded directly from such a value with `SeedableRng::from_seed`.

Seeding from ...

Fresh entropy

Using a fresh seed (direct from the OS) is easy using `SeedableRng::from_os_rng`:

```
use rand::prelude::*;
use rand_chacha::ChaCha20Rng;

fn main() {
    let mut rng = ChaCha20Rng::from_os_rng();
    println!("{}", rng.random_range(0..100));
}
```

Note that this requires `rand_core` has the feature `getrandom` enabled.

Another RNG

Quite obviously, another RNG may be used to fill a seed. We provide a convenience method for this:

```
use rand::prelude::*;

fn main() {
    let mut rng = SmallRng::from_rng(&mut rand::rng());
    println!("{}", rng.random_range(0..100));
}
```

But, say you want to save a key and use it later. For that you need to be a little bit more explicit:

```
use rand::prelude::*;
use rand_chacha::ChaCha8Rng;

fn main() {
    let mut seed: <ChaCha8Rng as SeedableRng>::Seed = Default::default();
    rand::rng().fill(&mut seed);
    let mut rng = ChaCha8Rng::from_seed(seed);
    println!("{}", rng.random_range(0..100));
}
```

Obligatory warning: a few simple PRNGs, notably `XorShiftRng`, behave badly when seeded from the same type of generator (in this case, Xorshift generates a clone). For cryptographic PRNGs this is not a problem; for others it is recommended to seed from a different type of generator. `ChaCha8Rng` is an excellent choice for a deterministic master generator (but for cryptographic uses, prefer the 12-round variant or higher).

A simple number

For some applications, especially simulations, all you want are a sequence of distinct, fixed random number seeds, e.g. 1, 2, 3, etc.

`SeedableRng::seed_from_u64` is designed exactly for this use-case. Internally, it uses a simple PRNG to fill the bits of the seed from the input number while providing good bit-avalanche (so that two similar numbers such as 0 and 1 translate to very different seeds and independent RNG sequences).

```
use rand::prelude::*;
use rand_chacha::ChaCha8Rng;

fn main() {
    let mut rng = ChaCha8Rng::seed_from_u64(2);
    println!("{}", rng.random_range(0..100));
}
```

Note that a number with 64-bits or less **cannot be secure**, so this should not be used for applications such as cryptography or gambling games.

A string, or any hashable data

Say you let users enter a string to seed the random number generator. Ideally, all parts of the string should influence the generator and making only a small change to the string should result in a fully independent generator sequence.

This can be achieved via use of a hash function to compress all input data down to a hash result, then using that result to seed a generator. The `rand_seeder` crate is designed for just this purpose.

```
use rand::prelude::*;
use rand_seeder::{Seeder, SipHasher};
use rand_pcg::Pcg64;

fn main() {
    // In one line:
    let mut rng: Pcg64 = Seeder::from("stripy zebra").into_rng();
    println!("{}", rng.random::<char>());

    // If we want to be more explicit, first we create a SipRng:
    let hasher = SipHasher::from("a sailboat");
    let mut hasher_rng = hasher.into_rng();
    // (Note: hasher_rng is a full RNG and can be used directly.)

    // Now, we use hasher_rng to create a seed:
    let mut seed: <Pcg64 as SeedableRng>::Seed = Default::default();
    hasher_rng.fill(&mut seed);

    // And create our RNG from that seed:
    let mut rng = Pcg64::from_seed(seed);
    println!("{}", rng.random::<char>());
}
```

Note that `rand_seeder` is **not suitable** for cryptographic usage. It is **not a password hasher**, for such applications a key-derivation function such as Argon2 must be used.

Parallel RNGs

Theory: multiple RNGs

If you want to use random generators in multiple worker threads simultaneously, then you will want to use multiple RNGs. A few suggested approaches:

1. Use `rng()` in each worker thread. This is seeded automatically (lazily and uniquely) on each thread where it is used.
2. Use `rng()` (or another master RNG) to seed a custom RNG on each worker thread. The main advantage here is flexibility over the RNG used.
3. Use a custom RNG per *work unit*, not per *worker thread*. If these RNGs are seeded in a deterministic fashion, then deterministic results are possible. Unfortunately, seeding a new RNG for each work unit from a master generator cannot be done in parallel, thus may be slow.
4. Use a single master seed. For each work unit, seed an RNG using the master seed and set the RNG's stream to the work unit number. This is potentially a faster than (3) while still deterministic, but not supported by all RNGs.

Note: do not simply clone RNGs for worker threads/units. Clones return the same sequence of output as the original. You may however use clones if you then set a unique stream on each.

Streams

Which RNG families support multiple streams?

- ChaCha: the ChaCha RNGs support 256-bit seed, 64-bit stream and 64-bit counter (per 16-word block), thus supporting 2^{64} streams of 2^{68} words each.
- Hc128 is a cryptographic RNG supporting a 256-bit seed; one could construct this seed from (e.g.) a smaller 192-bit key plus a 64-bit stream.

Note that the above approach of constructing the seed from a smaller key plus a stream counter can only be recommended with cryptographic PRNGs since simpler RNGs often have correlations in the RNG's output using two similar keys, and may also require "random looking" seeds to produce high quality output.

Non-cryptographic PRNGs may still support multiple streams, but likely with significant limitations (especially noting that a common recommendation with such PRNGs is not to consume more than the square root of the generator's period).

- Xoshiro: the Xoshiro family of RNGs support `jump` and `long_jump` methods which may effectively be used to divide the output of a single RNG into multiple streams. In practice this is only useful with a small number of streams, since `jump` must be called `n` times to select the nth "stream".
- Pcg: these RNGs support construction with `state` and `stream` parameters. Note, however, that the RNGs have been critiqued in that multiple streams using the same key are often strongly correlated. See the author's own comments.

  The PCG RNGs *also* support an `fn advance(delta)` method, which might be used to divide a single stream into multiple sub-streams as with Xoshiro's `jump` (but better since the offset can be specified).

Practice: non-deterministic multi-threaded

We use Rayon's parallel iterators, using `map_init` to initialize an RNG in each worker thread. Note: this RNG may be re-used across multiple work units, which may be split between worker threads in non-deterministic fashion.

```rust
use rand::distr::{Distribution, Uniform};
use rayon::prelude::*;

static SAMPLES: u64 = 1_000_000;

fn main() {
    let range = Uniform::new(-1.0f64, 1.0).unwrap();

    let in_circle = (0..SAMPLES)
        .into_par_iter()
        .map_init(|| rand::rng(), |rng, _| {
            let a = range.sample(rng);
            let b = range.sample(rng);
            if a * a + b * b <= 1.0 {
                1
            } else {
                0
            }
        })
        .reduce(|| 0usize, |a, b| a + b);

    // prints something close to 3.14159...
    println!(
        "π is approximately {}",
        4. * (in_circle as f64) / (SAMPLES as f64)
    );
}
```

Practice: deterministic multi-threaded

We use approach (4) above to achieve a deterministic result: initialize all RNGs from a single seed, but using multiple streams. We use `ChaCha8Rng::set_stream` to achieve this.

Note further that we manually batch multiple work-units according to `BATCH_SIZE`. This is important since the cost of initializing an RNG is large compared to the cost of our work unit (generating two random samples plus some trivial calculations). Manual batching could improve performance of the above non-deterministic simulation too.

(Note: this example is https://github.com/rust-random/rand/blob/master/examples/rayon-monte-carlo.rs.)

```
use rand::distr::{Distribution, Uniform};
use rand_chacha::{rand_core::SeedableRng, ChaCha8Rng};
use rayon::prelude::*;

static SEED: u64 = 0;
static BATCH_SIZE: u64 = 10_000;
static BATCHES: u64 = 1000;

fn main() {
    let range = Uniform::new(-1.0f64, 1.0).unwrap();

    let in_circle = (0..BATCHES)
        .into_par_iter()
        .map(|i| {
            let mut rng = ChaCha8Rng::seed_from_u64(SEED);
            rng.set_stream(i);
            let mut count = 0;
            for _ in 0..BATCH_SIZE {
                let a = range.sample(&mut rng);
                let b = range.sample(&mut rng);
                if a * a + b * b <= 1.0 {
                    count += 1;
                }
            }
            count
        })
        .reduce(|| 0usize, |a, b| a + b);

    // prints 3.1409052 (determinstic and reproducible result)
    println!(
        "π is approximately {}",
        4. * (in_circle as f64) / ((BATCH_SIZE * BATCHES) as f64)
    );
}
```

Random values

Now that we have a way of producing random data, how can we convert it to the type of value we want?

This is a trick question: we need to know both the *range* we want and the type of *distribution* of this value (which is what the `next` section is all about).

The `Rng` trait

For convenience, all generators automatically implement the `Rng` trait, which provides short-cuts to a few ways of generating values. This has several convenience functions for producing uniformly distributed values:

- `Rng::random` generates an unbiased (uniform) random value from a range appropriate for the type. For integers this is normally the full representable range (e.g. from `0u32` to `std::u32::MAX`), for floats this is between 0 and 1, and some other types are supported, including arrays and tuples.

  This method is a convenience wrapper around the `StandardUniform` distribution, as documented in the next section.

- `Rng::random_range` generates an unbiased random value in the given range

- `Rng::fill` and `Rng::try_fill` are optimised functions for filling any byte or integer slice with random values

It also has convenience functions for producing non-uniform boolean values:

- `Rng::random_bool` generates a boolean with the given probability
- `Rng::random_ratio` also generates a boolean, where the probability is defined via a fraction

Finally, it has a function to sample from arbitrary distributions:

- `Rng::sample` samples directly from some distribution

Examples:

```rust
use rand::Rng;
let mut rng = rand::rng();

// an unbiased integer over the entire range:
let i: i32 = rng.random();
println!("i = {i}");

// a uniformly distributed value between 0 and 1:
let x: f64 = rng.random();
println!("x = {x}");

// simulate rolling a die:
println!("roll = {}", rng.random_range(1..=6));
```

Additionally, the `random` function is a short-cut to `Rng::random` on the `rng()` :

```rust
println!("Tossing a coin...");
if rand::random() {
    println!("We got lucky!");
}
```

Custom random types

Notice from the above that `rng.random()` yields a different distribution of values depending on the type:

- `i32` values are sampled from `i32::MIN ..= i32::MAX` uniformly
- `f32` values are sampled from `0.0 .. 1.0` uniformly

This is the `StandardUniform` distribution. `Distribution` s are the topic of the next chapter, but given the importance of the `StandardUniform` distribution we introduce it here. As usual, standards are somewhat arbitrary, but chosen according to reasonable logic:

- Values are sampled uniformly: given any two sub-ranges of equal size, each has an equal chance of containing the next sampled value
- Usually, the whole range of the target type is used
- For `f32` and `f64` the range `0.0 .. 1.0` is used (exclusive of `1.0`), for two reasons: (a) this is common practice for random-number generators and (b) because for many purposes having a uniform distribution of samples (along the Real number line) is important, and this is only possible for floating-point representations by restricting the range.

Given that, we can implement the `StandardUniform` distribution for our own types:

```rust
use rand::Rng;
use rand::distr::{Distribution, StandardUniform, Uniform};
use std::f64::consts::TAU; // = 2π

/// Represents an angle, in radians
#[derive(Debug)]
pub struct Angle(f64);
impl Angle {
    pub fn from_degrees(degrees: f64) -> Self {
        Angle(degrees * (std::f64::consts::TAU / 360.0))
    }
}

impl Distribution<Angle> for StandardUniform {
    fn sample<R: Rng + ?Sized>(&self, rng: &mut R) -> Angle {
        // It would be correct to write:
        // Angle(rng.random::<f64>() * TAU)

        // However, the following is preferred:
        Angle(Uniform::new(0.0, TAU).unwrap().sample(rng))
    }
}

fn main() {
    let angle: Angle = rand::rng().random();
    println!("Random angle: {angle:?}");
}
```

Random distributions

For maximum flexibility when producing random values, we define the `Distribution` trait:

```
// a producer of data of type T:
pub trait Distribution<T> {
    // the key function:
    fn sample<R: Rng + ?Sized>(&self, rng: &mut R) -> T;

    // a convenience function defined using sample:
    fn sample_iter<R>(self, rng: R) -> rand::distr::Iter<Self, R, T>
    where
        Self: Sized,
        R: Rng,
    {
        // [has a default implementation]
    }
}
```

Implementations of `Distribution` are *probability distribution*: mappings from events to probabilities (e.g. for a die roll `P(x = i) = ⅙` or for a Normal distribution with mean `μ=0`, `P(x > 0) = ½`).

Note that although probability distributions all have properties such as a mean, a Probability Density Function, and can be sampled by inverting the Cumulative Density Function, here we only concern ourselves with *sampling random values*. If you require use of such properties you may prefer to use the `statrs` crate.

Rand provides implementations of many different distributions; we cover the most common of these here, but for full details refer to the `distr` module and the `rand_distr` crate.

Uniform distributions

The most obvious type of distribution is the one we already discussed: one where each equally-sized sub-range has equal chance of containing the next sample. This is known as *uniform*.

Rand actually has several variants of this, representing different ranges:

- `StandardUniform` requires no parameters and samples values uniformly according to the type. `Rng::random` provides a short-cut to this distribution.
- `Uniform` is parametrised by `Uniform::new(low, high)` (including `low`, excluding `high`) or `Uniform::new_inclusive(low, high)` (including both), and samples values

uniformly within this range. `Rng::random_range` is a convenience method defined over `Uniform::sample_single`, optimised for single-sample usage.
- `Alphanumeric` is uniform over the `char` values `0-9A-Za-z`.
- `Open01` and `OpenClosed01` are provide alternate sampling ranges for floating-point types (see below).

Uniform sampling by type

Lets go over the distributions by type:

- For `bool`, `StandardUniform` samples each value with probability 50%.
- For `Option<T>`, the `StandardUniform` distribution samples `None` with probability 50%, otherwise `Some(value)` is sampled, according to its type.
- For integers (`u8` through to `u128`, `usize`, and `i*` variants), `StandardUniform` samples from all possible values while `Uniform` samples from the parameterised range.
- For `NonZeroU8` and other "non-zero" types, `StandardUniform` samples uniformly from all non-zero values (rejection method).
- `Wrapping<T>` integer types are sampled as for the corresponding integer type by the `StandardUniform` distribution.
- For floats (`f32`, `f64`),
  - `StandardUniform` samples from the half-open range `[0, 1)` with 24 or 53 bits of precision (for `f32` and `f64` respectively)
  - `OpenClosed01` samples from the half-open range `(0, 1]` with 24 or 53 bits of precision
  - `Open01` samples from the open range `(0, 1)` with 23 or 52 bits of precision
  - `Uniform` samples from a given range with 23 or 52 bits of precision
- For the `char` type, the `StandardUniform` distribution samples from all available Unicode code points, uniformly; many of these values may not be printable (depending on font support). The `Alphanumeric` samples from only a-z, A-Z and 0-9 uniformly.
- For tuples and arrays, each element is sampled as above, where supported. The `StandardUniform` and `Uniform` distributions each support a selection of these types (up to 12-tuples and 32-element arrays). This includes the empty tuple `()` and array.

When using `rustc` ≥ 1.51, enable the `min_const_gen` feature to support arrays larger than 32 elements.

- For SIMD types, each element is sampled as above, for `StandardUniform` and `Uniform` (for the latter, `low` and `high` parameters are *also* SIMD types, effectively sampling from multiple ranges simultaneously). SIMD support requires using the `simd_support` feature flag and nightly `rustc`.

- For enums, you have to implement uniform sampling yourself. For example, you could use the following approach:

```rust
pub enum Food {
    Burger,
    Pizza,
    Kebab,
}

impl Distribution<Food> for StandardUniform {
    fn sample<R: Rng + ?Sized>(&self, rng: &mut R) -> Food {
        let index: u8 = rng.random_range(0..3);
        match index {
            0 => Food::Burger,
            1 => Food::Pizza,
            2 => Food::Kebab,
            _ => unreachable!(),
        }
    }
}
```

Non-uniform distributions

The `rand` crate provides only two non-uniform distributions:

- The `Bernoulli` distribution simply generates a boolean where the probability of sampling `true` is some constant (`Bernoulli::new(0.5)`) or ratio (`Bernoulli::from_ratio(1, 6)`).
- The `WeightedIndex` distribution may be used to sample from a sequence of weighted values. See the Sequences section.

Many more non-uniform distributions are provided by the `rand_distr` crate.

Integers

The `Binomial` distribution is related to the `Bernoulli` in that it models running `n` independent trials each with probability `p` of success, then counts the number of successes.

Note that for large `n` the `Binomial` distribution's implementation is much faster than sampling `n` trials individually.

The `Poisson` distribution expresses the expected number of events occurring within a fixed interval, given that events occur with fixed rate λ. `Poisson` distribution sampling generates `Float` values because `Float` s are used in the sampling calculations, and we prefer to defer to the user on integer types and the potentially lossy and panicking associated conversions. For example, `u64` values can be attained with `rng.sample(Poisson) as u64`.

Note that out of range float to int conversions with `as` result in undefined behavior for Rust <1.45 and a saturating conversion for Rust >=1.45.

Continuous non-uniform distributions

Continuous distributions model samples drawn from the real number line ℝ, or in some cases a point from a higher dimension ($\mathbb{R}^2$, $\mathbb{R}^3$, etc.). We provide implementations for `f64` and for `f32` output in most cases, although currently the `f32` implementations simply reduce the precision of an `f64` sample.

The exponential distribution, `Exp`, simulates time until decay, assuming a fixed rate of decay (i.e. exponential decay).

The `Normal` distribution (also known as Gaussian) simulates sampling from the Normal distribution ("Bell curve") with the given mean and standard deviation. The `LogNormal` is related: for sample `X` from the log-normal distribution, `log(X)` is normally distributed; this "skews" the normal distribution to avoid negative values and to have a long positive tail.

The `UnitCircle` and `UnitSphere` distributions simulate uniform sampling from the edge of a circle or surface of a sphere.

The `Cauchy` distribution (also known as the Lorentz distribution) is the distribution of the x-intercept of a ray from point `(x0, γ)` with uniformly distributed angle.

The `Beta` distribution is a two-parameter probability distribution, whose output values lie between 0 and 1. The `Dirichlet` distribution is a generalisation to any positive number of parameters.

Random processes

You may have noticed that the `Distribution` trait does not allow mutation of self (no `&mut self` methods). This is by design: a probability distribution is defined as a mapping from events to probabilities.

In contrast, a Stochastic Process concerns a family of variables (or state) which mutate in a random manner.

We do not attempt to define a general API covering random processes or to provide direct support for modelling them. Here we merely discuss some.

Sampling without replacement

Given, for example, a bag of 10 red marbles and 30 green marbles, the initial probability that a marble sampled from the bag is red is `10/(10 + 30) = ¼ = 0.25`. If the first marble *is* red and *is not replaced*, then the probability that the second marble sampled from the bag is red is `9/(9 + 30) = 3/13 ≅ 0.23`.

The `rand` crate does not provide any system supporting step-wise sampling without replacement. What it does provide is support for sampling multiple distinct values from a sequence in a single step: `IteratorRandom::choose_multiple` and `SliceRandom::choose_multiple`.

If you wish to implement step-wise sampling yourself, here are a few ideas:

- Place all elements in a `Vec`. Each step sample and remove one value. Note that if the set of all possible elements is large this is inefficient since `Vec::remove` is `O(n)` and since all elements must be constructed.
- Place all elements in a `Vec` and shuffle. Each step simply take the next element.
- Construct a method of sampling values from the initial distribution plus an empty `HashSet` representing "taken" values. Each step, sample a value; if it is in the `HashSet` then reject the value and sample again, otherwise place a copy in a `HashSet` and return. Note that this method is inefficient unless the number of samples taken is much smaller than the number of available elements.
- Investigate `src/seq/index.rs`: several sampling algorithms are used which may be adjusted to this application.

Sequences

Rand implements a few common random operations on sequences via the `IteratorRandom` and `SliceRandom` traits.

Generating indices

To sample:

- a single index within a given range, use `Rng::random_range`
- multiple distinct indices from `0..length`, use `index::sample`
- multiple distinct indices from `0..length` with weights, use `index::sample_weighted`

Shuffling

To shuffle a slice:

- `SliceRandom::shuffle`: fully shuffle a slice
- `SliceRandom::partial_shuffle`: partial shuffle; useful to extract `amount` random elements in random order

Sampling

The following provide a convenient way of sampling a value from a slice or iterator:

- `SliceRandom::choose`: sample one element from a slice (by ref)
- `SliceRandom::choose_mut`: sample one element from a slice (by ref mut)
- `SliceRandom::choose_multiple`: sample multiple distinct elements from a slice (returns iterator of references to elements)
- `IteratorRandom::choose`: sample one element from an iterator (by value)
- `IteratorRandom::choose_stable`: sample one element from an iterator (by value), where RNG calls are unaffected by the iterator's `size_hint`
- `IteratorRandom::choose_multiple_fill`: sample multiple elements, placing into a buffer

- `IteratorRandom::choose_multiple` : sample multiple elements, returning a `Vec`

Note that operating on an iterator is often less efficient than operating on a slice.

Weighted sampling

For example, weighted sampling could be used to model the colour of a marble sampled from a bucket containing 5 green, 15 red and 80 blue marbles.

With replacement

Sampling *with replacement* implies that any sampled values (marbles) are replaced (thus, the probability of sampling each variant is not affected by the action of sampling).

This is implemented by the following distributions:

- `WeightedIndex` has fast setup and `O(log N)` sampling
- `WeightedAliasIndex` has slow setup and `O(1)` sampling, thus *may* be faster with a large number of samples

For convenience, you may use:

- `SliceRandom::choose_weighted`
- `SliceRandom::choose_weighted_mut`

Without replacement

Sampling *without replacement* implies that the action of sampling modifies the distribution. Since the `Distribution` trait is built around the idea of immutable distributions, we offer the following:

- `SliceRandom::choose_multiple_weighted` : sample `amount` distinct values from a slice with weights
- `index::sample_weighted` : sample `amount` distinct indices from a range with weights
- Implement yourself: see the section in Random processes

Error handling

Error handling in Rand is a compromise between simplicity and necessity. Most RNGs and sampling functions will never produce errors, and making these able to handle errors would add significant overhead (to code complexity and ergonomics of usage at least, and potentially also performance, depending on the approach). However, external RNGs can fail, and being able to handle this is important.

It has therefore been decided that *most* methods should not return a `Result` type, but with a few important exceptions, namely:

- `Rng::try_fill`
- `RngCore::try_fill_bytes`
- `SeedableRng::from_rng`

Most functions consuming random values will not attempt any error handling, and reduce to calls to `RngCore`'s "infallible" methods. Since most RNGs cannot fail anyway this is usually not a problem, but the few generators which can may be forced to fail in this case:

- `OsRng` is a wrapper over `getrandom`. From the latter's documentation: "In general, on supported platforms, failure is highly unlikely, though not impossible." `OsRng` will forward errors through `RngCore::try_fill_bytes` while other methods panic on error.
- `rng` seeds itself via `OsRng` on first use and periodically thereafter, thus can potentially fail, though unlikely. If initial seeding fails, a panic will result. If a failure happens during reseeding (less likely) then the RNG continues without reseeding; a log message (warning) is emitted if logging is enabled.

Testing functions which use RNGs

Occasionally a function that uses random number generators might need to be tested. For functions that need to be tested with test vectors, the following approach might be adapted:

```
use rand::{TryCryptoRng, rngs::OsRng};

pub struct CryptoOperations<R: TryCryptoRng = OsRng> {
    rng: R
}

impl<R: TryCryptoRng> CryptoOperations<R> {
    #[must_use]
    pub fn new(rng: R) -> Self {
        Self {
            rng
        }
    }

    pub fn xor_with_random_bytes(&mut self, secret: &mut [u8; 8]) -> [u8; 8] {
        let mut mask = [0u8; 8];
        self.rng.try_fill_bytes(&mut mask).unwrap();

        for (byte, mask_byte) in secret.iter_mut().zip(mask.iter()) {
            *byte ^= mask_byte;
        }

        mask
    }
}

fn main() {
    let rng = OsRng;
    let mut crypto_ops = <CryptoOperations>::new(rng);

    let mut secret: [u8; 8] = *b"\x00\x01\x02\x03\x04\x05\x06\x07";
    let mask = crypto_ops.xor_with_random_bytes(&mut secret);

    println!("Modified Secret (XORed): {:?}", secret);
    println!("Mask: {:?}", mask);
}
```

To test this, we can create a `MockCryptoRng` implementing `TryRngCore` and `TryCryptoRng` in our testing module. Note that `MockCryptoRng` is private and `#[cfg(test)] mod tests` is cfg-gated to our test environment, thus ensuring that `MockCryptoRng` cannot accidentally be used in production.

```
#[cfg(test)]
mod tests {
    use super::*;

    #[derive(Clone, Copy, Debug)]
    struct MockCryptoRng {
        data: [u8; 8],
        index: usize,
    }

    impl MockCryptoRng {
        fn new(data: [u8; 8]) -> MockCryptoRng {
            MockCryptoRng {
                data,
                index: 0,
            }
        }
    }

    impl CryptoRng for MockCryptoRng {}

    impl RngCore for MockCryptoRng {
        fn next_u32(&mut self) -> u32 {
            unimplemented!()
        }

        fn next_u64(&mut self) -> u64 {
            unimplemented!()
        }

        fn fill_bytes(&mut self, dest: &mut [u8]) {
            for byte in dest.iter_mut() {
                *byte = self.data[self.index];
                self.index = (self.index + 1) % self.data.len();
            }
        }

        fn try_fill_bytes(&mut self, dest: &mut [u8]) -> Result<(),
rand::Error> {
            unimplemented!()
        }
    }

    #[test]
    fn test_xor_with_mock_rng() {
        let mock_crypto_rng =
MockCryptoRng::new(*b"\x57\x88\x1e\xed\x1c\x72\x01\xd8");
        let mut crypto_ops = CryptoOperations::new(mock_crypto_rng);

        let mut secret: [u8; 8] = *b"\x00\x01\x02\x03\x04\x05\x06\x07";
        let mask = crypto_ops.xor_with_random_bytes(&mut secret);
        let expected_mask = *b"\x57\x88\x1e\xed\x1c\x72\x01\xd8";
```

```
        let expected_xored_secret = *b"\x57\x89\x1c\xee\x18\x77\x07\xdf";

        assert_eq!(secret, expected_xored_secret);
        assert_eq!(mask, expected_mask);
    }
}
```

Updating

This guide is intended to facilitate upgrading to the next minor or major version of Rand. Note that updating to the next patch version (e.g. 0.5.1 to 0.5.2) should never require code changes.

This guide gives a few more details than the changelog, in particular giving guidance on how to use new features and migrate away from old ones.

Updating to 0.5

The 0.5 release has quite significant changes over the 0.4 release; as such, it may be worth reading through the following coverage of breaking changes. This release also contains many optimisations, which are not detailed below.

Crates

We have a new crate: `rand_core`! This crate houses some important traits, `RngCore`, `BlockRngCore`, `SeedableRng` and `CryptoRng`, the error types, as well as two modules with helpers for implementations: `le` and `impls`. It is recommended that implementations of generators use the `rand_core` crate while other users use only the `rand` crate, which re-exports most parts of `rand_core`.

The `rand_derive` crate has been deprecated due to very low usage and deprecation of `Rand`.

Features

Several new Cargo feature flags have been added:

- `alloc`, used without `std`, allows use of `Box` and `Vec`
- `serde1` adds serialization support to some PRNGs
- `log` adds logging in a few places (primarily to `OsRng` and `JitterRng`)

`Rng` and friends (core traits)

`Rng` trait has been split into two traits, a "back end" `RngCore` (implemented by generators) and a "front end" `Rng` implementing all the convenient extension methods.

Implementations of generators must `impl RngCore` instead. Usage of `rand_core` for implementations is encouraged; the `rand_core::{le, impls}` modules may prove useful.

Users of `Rng` *who don't need to implement it* won't need to make so many changes; often users can forget about `RngCore` and only import `Rng`. Instead of `RngCore::next_u32()` /

`next_u64()` users should prefer `Rng::gen()`, and instead of `RngCore::fill_bytes(dest)`, `Rng::fill(dest)` can be used.

`Rng` / `RngCore` methods

To allow error handling from fallible sources (e.g. `OsRng`), a new `RngCore::try_fill_bytes` method has been added; for example `EntropyRng` uses this mechanism to fall back to `JitterRng` if `OsRng` fails, and various handlers produce better error messages. As before, the other methods will panic on failure, but since these are usually used with algorithmic generators which are usually infallible, this is considered an appropriate compromise.

A few methods from the old `Rng` have been removed or deprecated:

- `next_f32` and `next_f64`; these are no longer implementable by generators; use `gen` instead
- `gen_iter`; users may instead use standard iterators with closures: `::std::iter::repeat(()).map(|()| rng.gen())`
- `gen_ascii_chars`; use `repeat` as above and `rng.sample(Alphanumeric)`
- `gen_weighted_bool(n)`; use `gen_bool(1.0 / n)` instead

`Rng` has a few new methods:

- `sample(distr)` is a shortcut for `distr.sample(rng)` for any `Distribution`
- `gen_bool(p)` generates a boolean with probability `p` of being true
- `fill` and `try_fill`, corresponding to `fill_bytes` and `try_fill_bytes` respectively (i.e. the only difference is error handling); these can fill and integer slice / array directly, and provide better performance than `gen()`

Constructing PRNGs

New randomly-initialised PRNGs

A new trait has been added: `FromEntropy`. This is automatically implemented for any type supporting `SeedableRng`, and provides construction from fresh, strong entropy:

```
use rand_0_5::{ChaChaRng, FromEntropy};

let mut rng = ChaChaRng::from_entropy();
```

Seeding PRNGs

The `SeedableRng` trait has been modified to include the seed type via an associated type (`SeedableRng::Seed`) instead of a template parameter (`SeedableRng<Seed>`). Additionally, all PRNGs now seed from a byte-array (`[u8; N]` for some fixed N). This allows generic handling of PRNG seeding which was not previously possible.

PRNGs are no longer constructed from other PRNGs via `Rand` support / `gen()` , but through `SeedableRng::from_rng` , which allows error handling and is intentionally explicit.

`SeedableRng::reseed` has been removed since it has no utility over `from_seed` and its performance advantage is questionable.

Implementations of `SeedableRng` may need to change their `Seed` type to a byte-array; this restriction has been made to ensure portable handling of Endianness. Helper functions are available in `rand_core::le` to read `u32` and `u64` values from byte arrays.

Block-based PRNGs

rand_core has a new helper trait, `BlockRngCore` , and implementation, `BlockRng` . These are for use by generators which generate a block of random data at a time instead of word-sized values. Using this trait and implementation has two advantages: optimised `RngCore` methods are provided, and the PRNG can be used with `ReseedingRng` with very low overhead.

Cryptographic RNGs

A new trait has been added: `CryptoRng` . This is purely a marker trait to indicate which generators should be suitable for cryptography, e.g. `fn foo<R: Rng + CryptoRng>(rng: &mut R)` . *Suitability for cryptographic use cannot be guaranteed.*

Error handling

A new `Error` type has been added, designed explicitly for no-std compatibility, simplicity, and enough flexibility for our uses (carrying a `cause` when possible):

```
pub struct Error {
    pub kind: ErrorKind,
    pub msg: &'static str,
    // some fields omitted
}
```

The associated `ErrorKind` allows broad classification of errors into permanent, unexpected, transient and not-yet-ready kinds.

The following use the new error type:

- `RngCore::try_fill_bytes`
- `Rng::try_fill`
- `OsRng::new`
- `JitterRng::new`

External generators

We have a new generator, `EntropyRng`, which wraps `OsRng` and `JitterRng` (preferring to use the former, but falling back to the latter if necessary). This allows easy construction with fallback via `SeedableRng::from_rng`, e.g. `IsaacRng::from_rng(EntropyRng::new())?`. This is equivalent to using `FromEntropy` except for error handling.

It is recommended to use `EntropyRng` over `OsRng` to avoid errors on platforms with broken system generator, but it should be noted that the `JitterRng` fallback is very slow.

PRNGs

Pseudo-Random Number Generators (i.e. deterministic algorithmic generators) have had a few changes since 0.4, and are now housed in the `prng` module (old names remain temporarily available for compatibility; eventually these generators will likely be housed outside the `rand` crate).

All PRNGs now do not implement `Copy` to prevent accidental copying of the generator's state (and thus repetitions of generated values). Explicit cloning via `Clone` is still available. All PRNGs now have a custom implementation of `Debug` which does not print any internal state; this helps avoid accidentally leaking cryptographic generator state in log files. External PRNG implementations are advised to follow this pattern (see also doc on `RngCore`).

`SmallRng` has been added as a wrapper, currently around `XorShiftRng` (but likely another algorithm soon). This is for uses where small state and fast initialisation are important but cryptographic strength is not required. (Actual performance of generation varies by benchmark; depending on usage this may or may not be the fastest algorithm, but will always be fast.)

`ReseedingRng`

The `ReseedingRng` wrapper has been significantly altered to reduce overhead. Unfortunately the new `ReseedingRng` is not compatible with all RNGs, but only those using `BlockRngCore`.

ChaCha

The method `ChaChaRng::set_counter` has been replaced by two new methods, `set_word_pos` and `set_stream`. Where necessary, the behaviour of the old method may be emulated as follows:

```
let lower = 88293;
let higher = 9300932;

// previously:
// let mut rng = rand::ChaChaRng::new_unseeded();
// rng.set_counter(lower, higher);

// now:
let mut rng = ChaChaRng::from_seed([0u8; 32]);
rng.set_word_pos(lower << 4);
rng.set_stream(higher);

assert_eq!(4060232610, rng.next_u32());
assert_eq!(2786236710, rng.next_u32());
```

ISAAC PRNGs

The `IsaacRng` and `Isaac64Rng` PRNGs now have an additional construction method: `new_from_u64(seed)`. 64 bits of state is insufficient for cryptography but may be of use in simulations and games. This will likely be superseded by a method to construct any PRNG from any hashable object in the future.

HC-128

This is a new cryptographic generator, selected as one of the "stream ciphers suitable for widespread adoption" by eSTREAM. This is now the default cryptographic generator, used by `StdRng` and `thread_rng()` .

Helper functions/traits

The `Rand` trait has been deprecated. Instead, users are encouraged to use `Standard` which is a real distribution and supports the same sampling as `Rand` . `Rng::gen()` now uses `Standard` and should work exactly as before. See the documentation of the `distributions` module on how to implement `Distribution<T>` for `Standard` for user types `T`

`weak_rng()` has been deprecated; use `SmallRng::from_entropy()` instead.

Distributions

The `Sample` and `IndependentSample` traits have been replaced by a single trait, `Distribution` . This is largely equivalent to `IndependentSample` , but with `ind_sample` replaced by just `sample` . Support for mutable distributions has been dropped; although it appears there may be a few genuine uses, these are not used widely enough to justify the existence of two independent traits or of having to provide mutable access to a distribution object. Both `Sample` and `IndependentSample` are still available, but deprecated; they will be removed in a future release.

`Distribution::sample` (as well as several other functions) can now be called directly on type-erased (unsized) RNGs.

`RandSample` has been removed (see `Rand` deprecation and new `Standard` distribution).

The `Closed01` wrapper has been removed, but `OpenClosed01` has been added.

Uniform distributions

Two new distributions are available:

- `Standard` produces uniformly-distributed samples for many different types, and acts

as a replacement for `Rand`
- `Alphanumeric` samples `char` s from the ranges `a-z A-Z 0-9`

Ranges

The `Range` distribution has been heavily adapted, and renamed to `Uniform` :

- `Uniform::new(low, high)` remains (half open `[low, high)`)
- `Uniform::new_inclusive(low, high)` has been added, including `high` in the sample range
- `Uniform::sample_single(low, high, rng)` is a faster variant for single usage sampling from `[low, high)`

`Uniform` can now be implemented for user-defined types; see the `uniform` module.

Non-uniform distributions

Two distributions have been added:

- Poisson, modeling the number of events expected from a constant-rate source within a fixed time interval (e.g. nuclear decay)
- Binomial, modeling the outcome of a fixed number of yes-no trials

The sampling methods are based on those in "Numerical Recipes in C".

Exponential and Normal distributions

The main `Exp` and `Normal` distributions are unchanged, however the "standard" versions, `Exp1` and `StandardNormal` are no longer wrapper types, but full distributions. Instead of writing `let Exp1(x) = rng.gen();` you now write `let x = rng.sample(Exp1);` .

Updating to 0.6

During the 0.6 cycle, Rand found a new home under the rust-random project. We already feel at home, but if you'd like to help us decorate, a new logo would be appreciated!

We also found a new home for user-centric documentation — this book!

PRNGs

All PRNGs in our old PRNG module have been moved to new crates. We also added an additional crate with the PCG algorithms, and an external crate with Xoshiro / Xoroshiro algorithms:

- `rand_chacha`
- `rand_hc`
- `rand_isaac`
- `rand_xorshift`
- `rand_pcg`
- `xoshiro`

SmallRng

This update, we switched the algorithm behind `SmallRng` from Xorshift to a PCG algorithm (either `Pcg64Mcg` aka XSL 128/64 MCG, or `Pcg32` aka XSH RR 64/32 LCG aka the standard PCG algorithm).

Sequences

The `seq` module has been completely re-written, and the `choose` and `shuffle` methods have been removed from the `Rng` trait. Most functionality can now be found in the `IteratorRandom` and `SliceRandom` traits.

Weighted choices

The `WeightedChoice` distribution has now been replaced with `WeightedIndex`, solving a few issues by making the functionality more generic.

For convenience, the `SliceRandom::choose_weighted` method (and `_mut` variant) allow a `WeightedIndex` sample to be applied directly to a slice.

Other features

SIMD types

Rand now has rudimentary support for generating SIMD types, gated behind the `simd_support` feature flag.

`i128` / `u128` types

Since these types are now available on stable compilers, these types are supported automatically (with recent enough Rust version). The `i128_support` feature flag still exists to avoid breakage, but no longer does anything.

Updating to 0.7

Since the 0.6 release, rust-random gained a logo and a new crate: getrandom!

Dependencies

Rand crates now require `rustc` version 1.32.0 or later. This allowed us to remove all `build.rs` files for faster compilation.

The Rand crate now has fewer dependencies overall, though with some new ones.

Getrandom

As mentioned above, we have a new crate: getrandom, delivering a minimal API around platform-independent access to fresh entropy. This replaces the previous implementation in `OsRng`, which is now merely a wrapper.

Core features

The `FromEntropy` trait has now been removed. Fear not though, its `from_entropy` method continues to provide easy initialisation from its new home in the `SeedableRng` trait (this requires that `rand_core` has the `std` or `getrandom` feature enabled):

```
use rand::{SeedableRng, rngs::StdRng};
let mut rng = StdRng::from_entropy();
```

The `SeedableRng::from_rng` method is now considered value-stable: implementations should have portable results.

The `Error` type of `rand_core` and `rand` has seen a major redesign; direct usage of this type is likely to need adjustment.

PRNGs

These have seen less change than in the previous release, but noteworthy is:

- `rand_chacha` has been rewritten for much better performance (via SIMD instructions)
- `StdRng` and `ThreadRng` now use the ChaCha algorithm. This is a value-breaking change for `StdRng`.
- `SmallRng` is now gated behind the `small_rng` feature flag.
- The `xoshiro` crate is now `rand_xoshiro`.
- `rand_pcg` now includes `Pcg64`.

Distributions

For the most widely used distributions (`Standard` and `Uniform`), there have been no significant changes. But for *most* of the rest...

- We added a new crate, `rand_distr`, to house the all distributions (including re-exporting those still within `rand::distributions`). If you previously used `rand::distributions::Normal`, now you use `rand_distr::Normal`.
- Constructors for many distributions changed in order to return a `Result` instead of panicking on error.
- Many distributions are now generic over their parameter type (in most cases supporting `f32` and `f64`). This aids usage with generic code, and allows reduced size of parameterised distributions. Currently the more complex algorithms always use `f64` internally.
- `Standard` can now sample `NonZeroU*` values

We also added several distributions:

- `rand::distributions::weighted::alias_method::WeightedIndex`
- `rand_distr::Pert`
- `rand_distr::Triangular`
- `rand_distr::UnitBall`
- `rand_distr::UnitDisc`
- `rand_distr::UnitSphere` (previously named `rand::distributions::UnitSphereSurface`)

Sequences

To aid portability, all random samples of type `usize` now instead sample a `u32` value when the upper-bound is less than `u32::MAX`. This means that upgrading to 0.7 is a value-breaking change for use of `seq` functionality, but that after upgrading to 0.7 results should be consistent across CPU architectures.

Updating to 0.8

In the following, instructions are provided for porting your code from `rand 0.7` and `rand_distr 0.2` to `rand 0.8` and `rand_distr 0.3`.

Dependencies

Rand crates now require `rustc` version 1.36.0 or later. This allowed us to remove some unsafe code and simplify the internal `cfg` logic.

The dependency on `getrandom` was bumped to version 0.2. While this does not affect Rand's API, you may be affected by some of the breaking changes even if you use `getrandom` only as a dependency:

- You may have to update the `getrandom` features you are using. The following features are now available:
  - `"rdrand"`: Use the RDRAND instruction on `no_std` `x86/x86_64` targets.
  - `"js"`: Use JavaScript calls on `wasm32-unknown-unknown`. This replaces the `stdweb` and `wasm-bindgen` features, which are removed.
  - `"custom"`: Allows you to specify a custom implementation.
- Unsupported targets no longer compile. If you require the previous behavior (panicking at runtime instead of failing to compile), you can use the `custom` feature to provide a panicking implementation.
- Windows XP and stdweb are, as of `getrandom` version 0.2.1, no longer supported. If you require support for either of these platforms you may add a dependency on `getrandom = "=0.2.0"` to pin this version.
- Hermit, L4Re and UEFI are no longer officially supported. You can use the `rdrand` feature on these platforms.
- The minimum supported Linux kernel version is now 2.6.32.

If you are using `getrandom`'s API directly, there are further breaking changes that may affect you. See its changelog.

Serde has been re-added as an optional dependency (use the `serde1` feature flag), supporting many types (where appropriate). `StdRng` and `SmallRng` are deliberately excluded since these types are not portable.

Core features

`ThreadRng`

`ThreadRng` no longer implements `Copy`. This was necessary to fix a possible use-after-free in its thread-local destructor. Any code relying on `ThreadRng` being copied must be updated to use a mutable reference instead. For example,

```
let rng = rand_0_7::thread_rng();
let a: u32 = Standard.sample_iter(rng).next().unwrap();
let b: u32 = Standard.sample_iter(rng).next().unwrap();
```

can be replaced with the following code:

```
let mut rng = thread_rng();
let a: u32 = Standard.sample_iter(&mut rng).next().unwrap();
let b: u32 = Standard.sample_iter(&mut rng).next().unwrap();
```

`gen_range`

`Rng::gen_range` now takes a `Range` instead of two numbers. Thus, replace `gen_range(a, b)` with `gen_range(a..b)`. We suggest using the following regular expression to search-replace in all files:

- replace `gen_range\(([^,]*),\s*([^)]*)\)`
- with `gen_range(\1..\2)`
- or with `gen_range($1..$2)` (if your tool does not support backreferences)

Most IDEs support search-replace-across-files or similar; alternatively an external tool such as Regexxer may be used.

This change has a couple of other implications:

- inclusive ranges are now supported, e.g. `gen_range(1..=6)` or `gen_range('A'..='Z')`
- it may be necessary to explicitly dereference some parameters
- SIMD types are no longer supported (`Uniform` types may still be used directly)

`fill`

The `AsByteSliceMut` trait was replaced with the `Fill` trait. This should only affect code implementing `AsByteSliceMut` on user-defined types, since the `Rng::fill` and

`Rng::try_fill` retain support for previously-supported types.

`Fill` supports some additional slice types which could not be supported with `AsByteSliceMut`: `[bool]`, `[char]`, `[f32]`, `[f64]`.

`adapter`

The entire `rand::rngs::adapter` module is now restricted to the `std` feature. While this is technically a breaking change, it should only affect `no_std` code using `ReseedingRng`, which is unlikely to exist in the wild.

Generators

StdRng has switched from the 20-round ChaCha20 to ChaCha12 for improved performance. This is a reduction in complexity but the 12-round variant is still considered secure: see rand#932. This is a value-breaking change for `StdRng`.

SmallRng now uses the Xoshiro128++ and Xoshiro256++ algorithm on 32-bit and 64-bit platforms respectively. This reduces correlations of random data generated from similar seeds and improves performance. It is a value-breaking change.

We now implement `PartialEq` and `Eq` for `StdRng`, `SmallRng`, and `StepRng`.

Distributions

Several smaller changes occurred to rand distributions:

- The `Uniform` distribution now additionally supports the `char` type, so for example `rng.gen_range('a'..='f')` is now supported.
- `UniformSampler::sample_single_inclusive` was added.
- The `Alphanumeric` distribution now samples bytes instead of chars. This more closely reflects the internally used type, but old code likely has to be adapted to perform the conversion from `u8` to `char`. For example, with Rand 0.7 you could write:

```
let chars: String = std::iter::repeat(())
    .map(|()| rng.sample(Alphanumeric))
    .take(7)
    .collect();
```

With Rand 0.8, this is equivalent to the following:

```
let chars: String = std::iter::repeat(())
    .map(|()| rng.sample(Alphanumeric))
    .map(char::from)
    .take(7)
    .collect();
println!("chars = \"{chars}\"");
```

- The alternative implementation of `WeightedIndex` employing the alias method was moved from `rand` to `rand_distr::weighted_alias::WeightedAliasIndex`. The alias method is faster for large sizes, but it suffers from a slow initialization, making it less generally useful.

In `rand_distr` v0.4, more changes occurred (since v0.2):

- `rand_distr::weighted_alias::WeightedAliasIndex` was added (moved from the `rand` crate)
- `rand_distr::InverseGaussian` and `rand_distr::NormalInverseGaussian` were added
- The `Geometric` and `Hypergeometric` distributions are now supported.
- A different algorithm is used for the `Beta` distribution, improving both performance and accuracy. This is a value-breaking change.
- The `Normal` and `LogNormal` distributions now support a `from_mean_cv` constructor method and `from_zscore` sampler method.
- `rand_distr::Dirichlet` now uses boxed slices internally instead of `Vec`. Therefore, the weights are taken as a slice instead of a `Vec` as input. For example, the following `rand_distr 0.2` code

  ```
  Dirichlet::new(vec![1.0, 2.0, 3.0]).unwrap();
  ```

  can be replaced with the following `rand_distr 0.3` code:

  ```
  Dirichlet::new(&[1.0, 2.0, 3.0]).unwrap();
  ```

- `rand_distr::Poisson` does no longer support sampling `u64` values directly. Old code may have to be updated to perform the conversion from `f64` explicitly.
- The custom `Float` trait in `rand_distr` was replaced with `num_traits::Float`. Any implementations of `Float` for user-defined types have to be migrated. Thanks to the math functions from `num_traits::Float`, `rand_distr` now supports `no_std`.

Additionally, there were some minor improvements:

- The treatment of rounding errors and NaN was improved for the `WeightedIndex` distribution.
- The `rand_distr::Exp` distribution now supports the `lambda = 0` parametrization.

Sequences

Weighted sampling without replacement is now supported, see `rand::seq::index::sample_weighted` and `SliceRandom::choose_multiple_weighted`.

There have been value-breaking changes to `IteratorRandom::choose`, improving accuracy and performance. Furthermore, `IteratorRandom::choose_stable` was added to provide an alternative that sacrifices performance for independence of iterator size hints.

Feature flags

`StdRng` is now gated behind a new feature flag, `std_rng`. This is enabled by default.

The `nightly` feature no longer implies the `simd_support` feature. If you were relying on this for SIMD support, you will have to use `simd_support` feature directly.

Tests

Value-stability tests were added for all distributions (rand#786), helping enforce our rules regarding value-breaking changes (see Reproducibility section).

Updating to 0.9

In the following, instructions are provided for porting your code from `rand 0.8` and `rand_distr 0.4` to `rand 0.9` and `rand_distr 0.5`.

The following is a migration guide focussing on potentially-breaking changes. For a full list of changes, see the relevant changelogs:

- CHANGELOG.md.
- rand_core/CHANGELOG.md.
- rand_distr/CHANGELOG.md.

Renamed functions and methods

In the 2024 edition, `gen` is a reserved keyword. The raw syntax `r#gen()` is awkward, so some methods in `rand::Rng` have been renamed:

- `gen` -> `random`
- `gen_range` -> `random_range`
- `gen_bool` -> `random_bool`
- `gen_ratio` -> `random_ratio`

Additionally, `rand::thread_rng()` has been renamed to the simpler `rng()`.

The previous names still exist but are deprecated.

Security

It was determined in #1514 that "rand is not a crypto library". This change clarifies that:

1. The rand library is a community project without any legally-binding guarantees
2. The rand library provides functionality for generating unpredictable random numbers but does not provide any high-level cryptographic functionality
3. `rand::rngs::OsRng` is a stateless generator, thus has no state to leak or need for (re)seeding
4. `rand::rngs::ThreadRng` is an automatically seeded generator with periodic reseeding using a cryptographically-strong pseudo-random algorithm, but which does not have protection of its in-memory state, in particular it does not automatically zero its

memory when destructed. Further, its design is a compromise: it is designed to be a "fast, reasonably secure generator".

Further, the former very limited fork-protection for `ReseedingRng` and `ThreadRng` were removed in #1379. It is recommended instead that reseeding be the responsibility of the code causing the fork (see `ThreadRng` docs for more details):

```rust
fn do_fork() {
    let pid = unsafe { libc::fork() };
    if pid == 0 {
        // Reseed ThreadRng in child processes:
        rand::rng().reseed();
    }
}
```

Dependencies

Rand crates now require `rustc` version 1.63.0 or later.

The dependency on `getrandom` was bumped to version 0.3. This release includes breaking changes for some platforms (WASM is particularly affected).

Features

Feature flags:

- `serde1` was renamed to `serde`
- `getrandom` was renamed to `os_rng`
- `thread_rng` is a new feature (enabled by default), required by `rng()` (`ThreadRng`)
- `small_rng` is now enabled by default
- `rand_chacha` is no longer an (implicit) feature; use `std_rng` instead

Core traits

In #1424, a new trait, `TryRngCore`, was added to `rand_core`:

```rust
pub trait TryRngCore {
    /// The type returned in the event of a RNG error.
    type Error: fmt::Debug + fmt::Display;

    /// Return the next random `u32`.
    fn try_next_u32(&mut self) -> Result<u32, Self::Error>;
    /// Return the next random `u64`.
    fn try_next_u64(&mut self) -> Result<u64, Self::Error>;
    /// Fill `dest` entirely with random data.
    fn try_fill_bytes(&mut self, dst: &mut [u8]) -> Result<(), Self::Error>;

    // [Provided methods hidden]
}
```

This trait is generic over both fallible and infallible RNGs (the latter use `Error` type `Infallible`), while `RngCore` now only represents infallible RNGs.

The trait `CryptoRng` is now a sub-trait of `RngCore`. A matching trait, `TryCryptoRng`, is available to mark implementors of `TryRngCore` which are cryptographically strong.

Seeding RNGs

The trait `SeedableRng` had a few changes:

- `type Seed` now has additional bounds: `Clone` and `AsRef<[u8]>`
- `fn from_rng` was renamed to `try_from_rng` while an infallible variant was added as the new `from_rng`
- `fn from_entropy` was renamed to `from_os_rng` along with a new fallible variant, `fn try_from_os_rng`

Generators

`ThreadRng` is now accessed via `rng()` (previously `thread_rng()`).

Sequences

The old trait `SliceRandom` has been split into three traits: `IndexedRandom`, `IndexedMutRandom` and `SliceRandom`. This allows `choose` functionality to be made available to `Vec` -like containers with non-contiguous storage, though `shuffle` functionality remains limited to slices.

Distributions

The module `rand::distributions` was renamed to `rand::distr` for brevity and to match `rand_distr` .

Several items in `distr` were also renamed or moved:

- Struct `Standard` -> `StandardUniform`
- Struct `Slice` → `slice::Choose`
- Struct `EmptySlice` → `slice::Empty`
- Trait `DistString` → `SampleString`
- Struct `DistIter` → `Iter`
- Struct `DistMap` → `Map`
- Struct `WeightedIndex` → `weighted::WeightedIndex`
- Enum `WeightedError` → `weighted::Error`

Some additional items were renamed in `rand_distr` :

- Struct `weighted_alias::WeightedAliasIndex` → `weighted::WeightedAliasIndex`
- Trait `weighted_alias::AliasableWeight` → `weighted::AliasableWeight`

The `StandardUniform` distribution no longer supports sampling `Option<T>` types (for any `T`).

`isize` and `usize` types are no longer supported by `Fill` , `WeightedAliasIndex` or `StandardUniform` . `isize` is also no longer supported by `Uniform` . `usize` remains supported by `Uniform` through `UniformUsize` and now has portable results across 32- and 64-bit platforms.

The constructors `fn new` , `fn new_inclusive` for `Uniform` and `UniformSampler` now return a `Result` instead of panicking on invalid inputs. Additionally, `Uniform` now supports `TryFrom` (instead of `From`) for range types.

Nightly features

SIMD

SIMD support now targets `std::simd`.

Reproducibility

See the `CHANGELOG.md` files for details of reproducibility-breaking changes affecting `rand` and `rand_distr`.

Contributing

Thank you for your interest in contributing to Rand!

We are open to all contributors, but please consider that we have limited resources, usually have other on-going work within the project, and that even accepting complete PRs costs us time (review and potentially on-going support), thus we may take considerable time to get back to you.

All contributions

- **Scope:** please consider whether your "issue" falls within the existing scope of the project or is an enhancement. Note that whether something is considered a *defect* may depend on your point of view. We may choose to reject contributions to avoid increasing our workload.

  If you wish to expand the scope of the project (e.g. new platforms or additional CI testing) then please be prepared to provide on-going support.

- **Fixes:** if you can easily fix this yourself, please consider making a PR instead of opening an issue. On the other hand if it's less easy or looks like it may conflict with other work, don't hesitate to open an issue.

Pull Requests

- **Changelog:** unless your change is trivial, please include a note in the changelog (`CHANGELOG.md`) of each crate affected, under the `[Unreleased]` heading at the top (add if necessary). Please include the PR number (this implies the note must be added *after* opening a PR).

- **Commits:** if contributing large changes, consider splitting these over multiple commits, if possible such that each commit at least compiles. Rebasing commits may be appropriate when making significant changes.

- **Documentation:** we require documentation of all public items. Short examples may be included where appropriate.

- **Maintainability:** it is important to us that code is easy to read and understand and not hard to review for correctness.

- **Performance:** we always aim for good performance and sometimes do considerable extra work to get there, however we must also make compromises for the sake of maintainability, and consider whether a minor efficiency gain is worth the extra code complexity. Use benchmarks.
- **Style:** make it neat. *Usually* limit length to 80 chars.
- **Unsafe:** use it where necessary, not if there is a good alternative. Ensure `unsafe` code is easy to review for correctness.
- **License and attribution:** this project is freely licenced under the MIT and Apache Public Licence v2. We assume that all contributions are made under these licence grants. Copyrights are retained by their contributors.

  Our works are attributed to "The Rand Project Developers". This is not a formal entity but merely the collection of all contributors to this project. For more, see the `COPYRIGHT` file.
- **Thank you!**

Documentation

Style

All documentation is in English, but no particular dialect is preferred.

The documentation should be accessible to multiple audiences: both seasoned Rustaceans and relative newcomers, those with experience in statistical modelling or cryptography, as well as those new to the subjects. Since it is often impossible to write appropriate one-size-fits-all documentation, we prefer concise technical documentation with reference to extended articles aimed at more specific audiences.

API documentation

Rand crates

It is recommended to use nightly Rust for correct link handling.

To build all API documentation for all crates in the rust-random/rand repository, run:

```
# Build doc for all modules:
cargo doc --all --no-deps

# And open it:
xdg-open target/doc/rand/index.html
```

On Linux, it is easy to set up automatic rebuilds after any edit:

```
while inotifywait -r -e close_write src/ rand_*/; do cargo doc; done
```

After editing API documentation, we recommend testing examples:

```
cargo test --doc
```

Rand API docs are automatically built and hosted at rust-random.github.io/rand for the latest code in master.

Getrandom crate

The rust-random/getrandom repository contains only a single crate, hence a simple `cargo doc` will suffice.

Cross-crate links

When referring to another crate, we prefer linking to the crate page on crates.io since (a) this includes the README documenting the purpose of the crate and (b) this links directly to both the repository and the API documentation. Example:

```
// Link to the crate page:
//! [`rand_chacha`]: https://crates.io/crates/rand_chacha
```

When referring to an item from within another crate,

1. if that item is accessible via a crate dependency (even if not via the public API), use the Rust item path
2. when linking to another crate within the `rust-random/rand` repository, relative paths within the generated documentation files (under `target/doc`) can be used; these work on rust-random.github.io/rand but not currently on `docs.rs` (see docs#204)
3. if neither of the above are applicable, use an absolute link
4. consider revising documentation, e.g. refer to the crate instead

Examples:

```
//! We depend on rand_core, therefore can use the Rust path:
//! [`BlockRngCore`]: rand_core::block::BlockRngCore

//! rand_chacha is not a dependency, but is within the same repository:
//! [`ChaCha20Rng`]: ../../rand_chacha/struct.ChaCha20Rng.html

//! Link directly to docs.rs, with major & minor but no patch version:
//! [`getrandom`]: https://docs.rs/getrandom/0.1/getrandom/fn.getrandom.html
```

Auxiliary documentation

README files

README files contain a brief introduction to the crate, shield badges, useful links, feature-flag documentation, licence information, and potentially an example.

For the most part these files do not have any continuous testing. Where examples are included (currently only for the `rand_jitter` crate), we enable continuous testing via `doc_comment` (see lib.rs:62 onwards).

CHANGELOG files

Changelog formats are based on the Keep a Changelog format.

All significant changes merged since the last release should be listed under an `[Unreleased]` section at the top of log.

The book

The source to this book is contained in the rust-random/book repository. It is built using mdbook, which makes building and testing easy:

```
cargo install mdbook --version "^0.4"

mdbook build --open
mdbook test

# To automatically rebuild after any changes:
mdbook watch
```

Note that links in the book are relative and designed to work in the published book. If you build the book locally, you might want to set up a symbolic link pointing to your build of the API documentation:

```
ln -s ../rand/target/doc rand
```

Scope

Over time, the scope of the project has grown, and Rand has moved from using a monolithic crate to using a "main" crate plus multiple single-purpose crates. For new functionality, one must consider where, and whether, it fits within the Rand project.

Small, focused crates may be used for a few reasons, but we aim *not* to maximally divide functionality into small crates. Valid reasons for using a separate crate for a feature are therefore:

- to allow a clear dependency hierarchy (`rand_core`)
- to make the feature available in a stand-alone fashion (e.g. `getrandom`)
- to remove little-used features with non-trivial amounts of code from widely used crates (e.g. `rand_jitter` and `rand_distr` both extracted functionality from `rand`)
- to allow choice, without including large amounts of unused code for all users, but also without producing an enormous number of new crates (RNG family crates like `rand_xoshiro` and `rand_isaac`)

Traits, basics and UI

The main user interface to the Rand project remains the central `rand` crate. Goals for this crate are:

- ease of use
- expose commonly used functionality in a single place
- permit usage of additional randomness sources and distribution samplers

To allow better modularity, the core traits have been moved to the `rand_core` crate. Goals of this crate are:

- expose the core traits with minimal dependencies
- provide common tools needed to implement various randomness sources

External random sources

The main (and usually only) external source of randomness is the Operating System, interfaced via the `getrandom` crate. This crate also supports usage of RDRAND on a few `no_std` targets.

Support for other `no_std` targets has been discussed but with little real implementation effort. See getrandom#4.

The `rand_jitter` crate provides an implementation of a CPU Jitter entropy harvester, and is only included in Rand for historical reasons.

Pseudo-random generators

The Rand library includes several pseudo-random number generators, for the following reasons:

- to implement the `StdRng` and `SmallRng` generators
- to provide a few high-quality alternative generators
- historical usage

These are implemented within "family" crates, e.g. `rand_chacha`, `rand_pcg`, `rand_xoshiro`.

We have received several requests to adopt new algorithms into the library; when evaluating such requests we must consider several things:

- purpose for inclusion within Rand
- whether the PRNG is cryptographically secure, and if so, how trustworthy such claims are
- statistical quality of output
- performance and features of the generator
- reception and third-party review of the algorithm

Distributions

The `Distribution` trait is provided by Rand, along with commonly-used distributions (mostly linear ones).

Additional distributions are packaged within the `rand_distr` crate, which depends on `rand` and re-exports all of its distributions.

Testing

Rand has a number of unit tests, though these are not comprehensive or perfect (improvements welcome). We prefer to have tests for all new functionality.

The first line of testing is simply to run `cargo test` from the appropriate directory. Since Rand supports `no_std` (core-only), `core+alloc` and `std` environments, it is important to test all three (depending on which features are applicable to the code in question):

```
# Test using std:
cargo test
# Test using only core:
cargo test --tests --no-default-features
# Test using core + alloc (requires nightly):
cargo +nightly test --tests --no-default-features --features=alloc
```

It may also be worth testing with other feature flags:

```
cargo test --all-features
```

Note that this only tests the current package (i.e. the main Rand lib when run from the repo's top level). To test another lib, `cd` to its directory.

We do not recommend using Cargo's `--package` option due to its surprising interactions with `--feature` options and failure when multiple versions of the same package are in the build tree. The CI instead uses `--manifest-path` to select packages; while developing, using `cd` is easier.

Writing tests

Tests may be unit tests within a `test` sub-module, documentation examples, example applications (`examples` dir), integration tests (`tests` dir), or benchmarks (`benches` dir).

Note that *only* unit tests and integration tests are expected to pass in `no_std` (core only) and `core+alloc` configurations. This is a deliberate choice; example code should only need to target the common case (`std`).

Random Number Generators

Often test code needs some RNG to test with, but does not need any particular RNG. In this case, we prefer use of `::test::rng` which is simple, fast to initialise and deterministic:

```
let mut rng = ::test::rng(528); // just pick some number
```

Various tests concern properties which are *probably* true, but not definitely. We prefer that such tests are deterministic to avoid spurious failures.

Benchmarks

We already have many benchmarks:

```
cargo +nightly bench

# In a few cases, nightly features may use different code paths:
cargo +nightly bench --features=nightly
```

Benchmarks for distributions now live in the `rand_distr` crate; all other benchmarks (including all our RNGs) live in the main `rand` crate (hence the many dev-dependencies).

A lot of code in Rand is performance sensitive, most of it is expected to be used in hot loops in some libraries/applications. If you change code in `rand_core`, in PRNG crates, or in the `rngs` or `distr` modules (especially when an 'obvious cleanup'), make sure the benchmarks do not regress.

Please report before-and-after results for any affected benchmarks. If you are optimising something previously not benchmarked, please add new benchmarks first, then add your changes in a separate commit (to make before-and-after benchmarking easy).