WHERE DID <random> GO WRONG?

Martin Hořeňovský

PEX

WHERE DID `<random>` GO WRONG?

Martin Hořeňovský

> *`<random>` is really elegantly designed.*
>
> *I love this header.*

—Stephen T. Lavavej

> *I think it's the best random number library design of all, by a mile.*
>
> — Andrei Alexandrescu

That was some high praise ...

That was some high praise …

… but in practice nobody uses `<random>`

That was some high praise …

… but in practice nobody uses `<random>`

Why?

Some people don't like the complexity

To generate 1D6 throw, you need

To generate 1D6 throw, you need

- an entropy source

To generate 1D6 throw, you need

- an entropy source
- a random number engine

To generate 1D6 throw, you need

- an entropy source
- a random number engine
- a uniform integer distribution

```cpp
#include <random>

int rollD6() {
    std::random_device rd;
    std::mt19937_64 rnd(rd());
    std::uniform_int_distribution<int> dist(1, 6);
    return dist(rnd);
}
```

```cpp
#include <random>

int rollD6() {
    std::random_device rd;
    std::mt19937_64 rnd(rd());
    std::uniform_int_distribution<int> dist(1, 6);
    return dist(rnd);
}
```

```cpp
#include <random>

int rollD6() {
    std::random_device rd;
    std::mt19937_64 rnd(rd());
    std::uniform_int_distribution<int> dist(1, 6);
    return dist(rnd);
}
```

Is it complex?

Is it complex?

Yes

Is it complex?

Yes

Is it big problem?

NO

In the context of C++, it is fine.

In the context of C++, it is fine.

But there is no value in the complexity of random.

Let's refactor the function to be reproducible

Let's refactor the function to be reproducible

```
template <typename RNG>
int rollD6(RNG& rng) {
    std::uniform_int_distribution<int> dist(1, 6);
    return dist(rng);
}
```

Let's refactor the function to be reproducible

```
template <typename RNG>
int rollD6(RNG& rng) {
    std::uniform_int_distribution<int> dist(1, 6);
    return dist(rng);
}
```

But why work with `int`s, when `uint8_t` is sufficient?

But why work with `int`s, when `uint8_t` is sufficient?

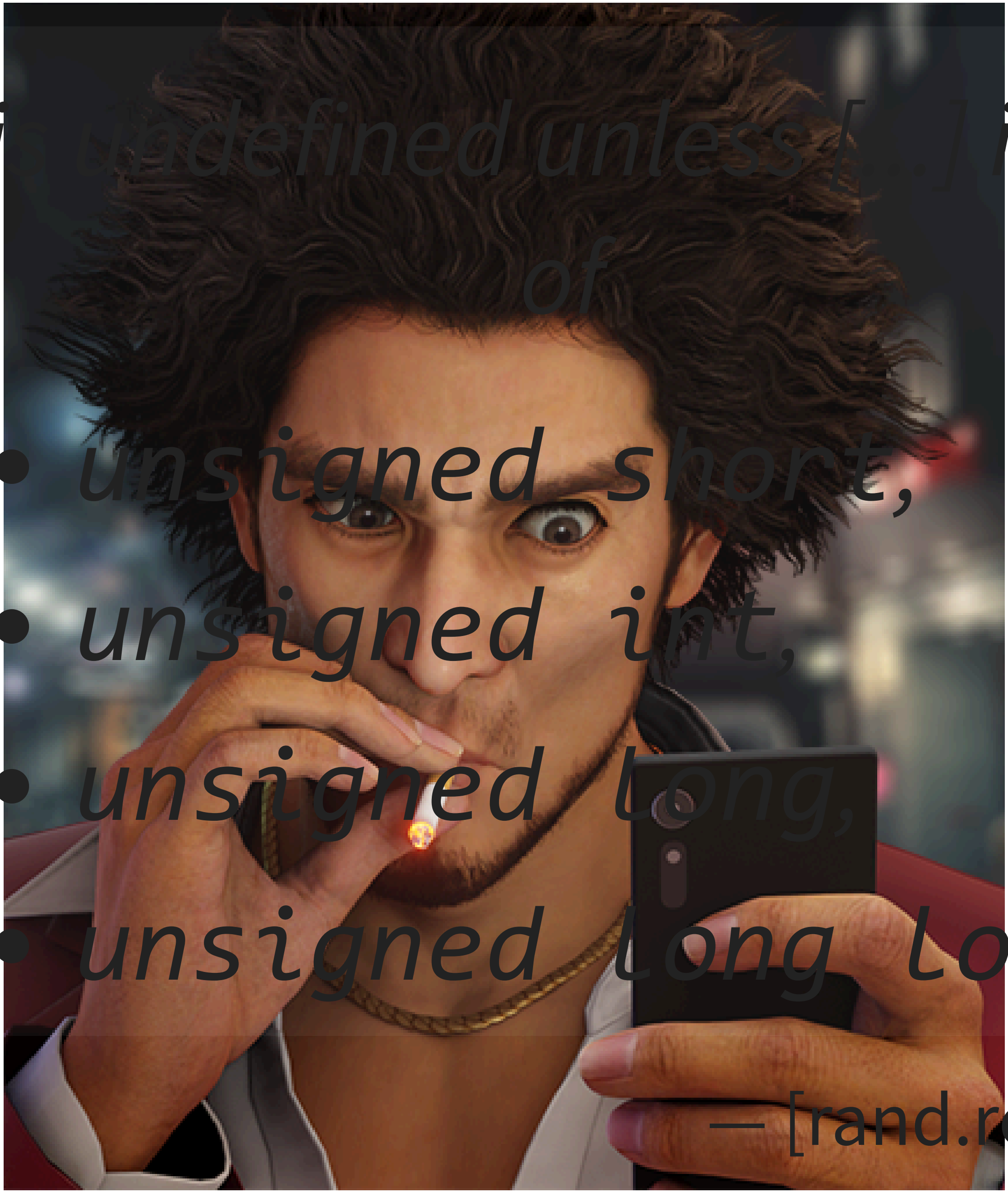
```
template <typename RNG>
uint8_t rollD6(RNG& rng) {
    std::uniform_int_distribution<uint8_t> dist(1, 6);
    return dist(rng);
}
```

But why work with `int`s, when `uint8_t` is sufficient?

```
template <typename RNG>
uint8_t rollD6(RNG& rng) {
    std::uniform_int_distribution<uint8_t> dist(1, 6);
    return dist(rng);
}
```

> *Throughout this subclause [rand], the effect of instantiating a template:*
>
> *[...]*
>
> *that has a template type parameter named `UIntType`*

> *… is undefined unless […] is one of*
>
> - *`unsigned short`,*
> - *`unsigned int`,*
> - *`unsigned long`,*
> - *`unsigned long long`.*
>
> — [rand.req.genl]-1.6



… is undefined unless […] is one of

- *`unsigned short`,*
- *`unsigned int`,*
- *`unsigned long`,*
- *`unsigned long long`.*

— [rand.req.genl]-1.6

STL has opened library issue 2326 about this in 2013.

STL has opened library issue 2326 about this in 2013.

It was closed as NAD in 2017.

STL has opened library issue 2326 about this in 2013.

It was closed as NAD in 2017.

There is a new one (4109) opened few months back.

Let's go back to our previous design

Let's go back to our previous design

```
template <typename RNG>
int rollD6(RNG& rng) {
    std::uniform_int_distribution<int> dist(1, 6);
    return dist(rng);
}
```

Let's go back to our previous design

```
template <typename RNG>
int rollD6(RNG& rng) {
    std::uniform_int_distribution<int> dist(1, 6);
    return dist(rng);
}
```

Different stdlibs can return different results.

Different stdlibs can return different results.

So can different versions of the same stdlib.

The real failing of <random> is that it serves nobody.

Some people want simplicity

Some people want simplicity

Other people want powerful and correct primitives

Some people want simplicity

Other people want powerful and correct primitives

Most people want reproducibility

And <random> provides none of this.

CONTENTS OF THE TALK

- The 1000-feet view of <random>
- The issues with using <random> in practice
- Basic outline of how to fix it all

WHAT'S IN <RANDOM>?

- Uniform bit generators (random engines)

- Uniform bit generators (random engines)
- Statistical distributions

- Uniform bit generators (random engines)
- Statistical distributions
- Helpers

- Uniform bit generators (random engines)
- Statistical distributions
- Helpers

And most (all?) of them are subtly broken.

RANDOM NUMBER ENGINES

<random> provides 7 predefined URBGs

<random> provides 7 predefined URBGs

- `minstd_rand0`

<random> provides 7 predefined URBGs

- `minstd_rand0`
- `minstd_rand`

<random> provides 7 predefined URBGs

- `minstd_rand0`
- `minstd_rand`
- `mt19937`

<random> provides 7 predefined URBGs

- minstd_rand0
- minstd_rand
- mt19937
- mt19937_64

<random> provides 7 predefined URBGs

- `minstd_rand0`
- `minstd_rand`
- `mt19937`
- `mt19937_64`
- `ranlux24`

<random> provides 7 predefined URBGs

- `minstd_rand0`
- `minstd_rand`
- `mt19937`
- `mt19937_64`
- `ranlux24`
- `ranlux48`

<random> provides 7 predefined URBGs

- `minstd_rand0`
- `minstd_rand`
- `mt19937`
- `mt19937_64`
- `ranlux24`
- `ranlux48`
- `knuth_b`

Some random number engine adapters,

Some random number engine adapters,

and generic engine templates.

DISTRIBUTIONS

<random> provides 20 distributions in 5 families

<random> provides 20 distributions in 5 families

- Uniform distributions

<random> provides 20 distributions in 5 families

- Uniform distributions
- Bernoulli distributions

<random> provides 20 distributions in 5 families

- Uniform distributions
- Bernoulli distributions
- Poisson distributions

<random> provides 20 distributions in 5 families

- Uniform distributions
- Bernoulli distributions
- Poisson distributions
- Normal distributions

<random> provides 20 distributions in 5 families

- Uniform distributions
- Bernoulli distributions
- Poisson distributions
- Normal distributions
- Sampling distributions

All distributions are objects.

All distributions are objects.

The standard doesn't* specify the implementation, only statistical properties of outputs.

UTILITIES/HELPERS

- `std::generate_canonical` - Generates single floating point number in $[0, 1)$

- `std::generate_canonical` - Generates single floating point number in $[0, 1)$
- `std::random_device` - Generates random bits

- `std::generate_canonical` - Generates single floating point number in $[0, 1)$
- `std::random_device` - Generates random bits
- `std::seed_seq` - Allows you to seed an engine with multiple values

WHAT'S WRONG WITH <RANDOM>?

RANDOM NUMBER ENGINES

- `minstd_rand0` (1969)
- `minstd_rand` (1993)
- `mt19937` (1998)
- `mt19937_64` (2000)
- `ranlux24` (1994)
- `ranlux48` (1994)
- `knuth_b` (1981)

All the engines are old.

All the engines are old.

All are slow(ish).

All the engines are old.

All are slow(ish).

MT is the best one, but `sizeof(mt19937) == 5000`!

They are impossible to seed.

```
std::mt19937 my_rng(std::random_device{}());
```

```
std::mt19937 my_rng(std::random_device{}());
```

mt19937 has 624 32-bit integers as internal state

```
std::mt19937 my_rng(std::random_device{}());
```

mt19937 has 624 32-bit integers as internal state

We provided 1 `uint` (might not even be 32 bits)

```
std::mt19937 my_rng(std::random_device{}());
```

mt19937 has 624 32-bit integers as internal state

We provided 1 `uint` (might not even be 32 bits)

That's 0.16% of possible seed states

Let's use `std::seed_seq` instead

Let's use `std::seed_seq` instead

```
constexpr size_t seed_data_size = 624;
std::vector<unsigned int> data(seed_data_size);
std::generate(data.begin(), data.end(), std::random_device{});
std::seed_seq seq(data);
std::mt19937 mt(seq);
```

Let's use `std::seed_seq` instead

```
constexpr size_t seed_data_size = 624;
std::vector<unsigned int> data(seed_data_size);
std::generate(data.begin(), data.end(), std::random_device{});
std::seed_seq seq(data);
std::mt19937 mt(seq);
```

Still broken (we will see why later)

How to determine `seed_data_size`?

How to determine `seed_data_size`?

```
constexpr size_t seed_data_size = std::mt19937::state_size;

std::vector<unsigned int> data(seed_data_size);
...
```

How to determine `seed_data_size`?

```
constexpr size_t seed_data_size = std::mt19937::state_size;

std::vector<unsigned int> data(seed_data_size);
...
```

That was easy.

```
constexpr size_t seed_data_size = std::mt19937_64::state_size;

std::vector<unsigned int> data(seed_data_size);
...
```

```
constexpr size_t seed_data_size = std::mt19937_64::state_size;

std::vector<unsigned int> data(seed_data_size);
...
```

This compiles...

```
constexpr size_t seed_data_size = std::mt19937_64::state_size;

std::vector<unsigned int> data(seed_data_size);
...
```

This compiles...

... but is wrong.

```
constexpr size_t seed_data_size = std::mt19937_64::state_size;

std::vector<unsigned int> data(seed_data_size);
...
```

This compiles...

... but is wrong.

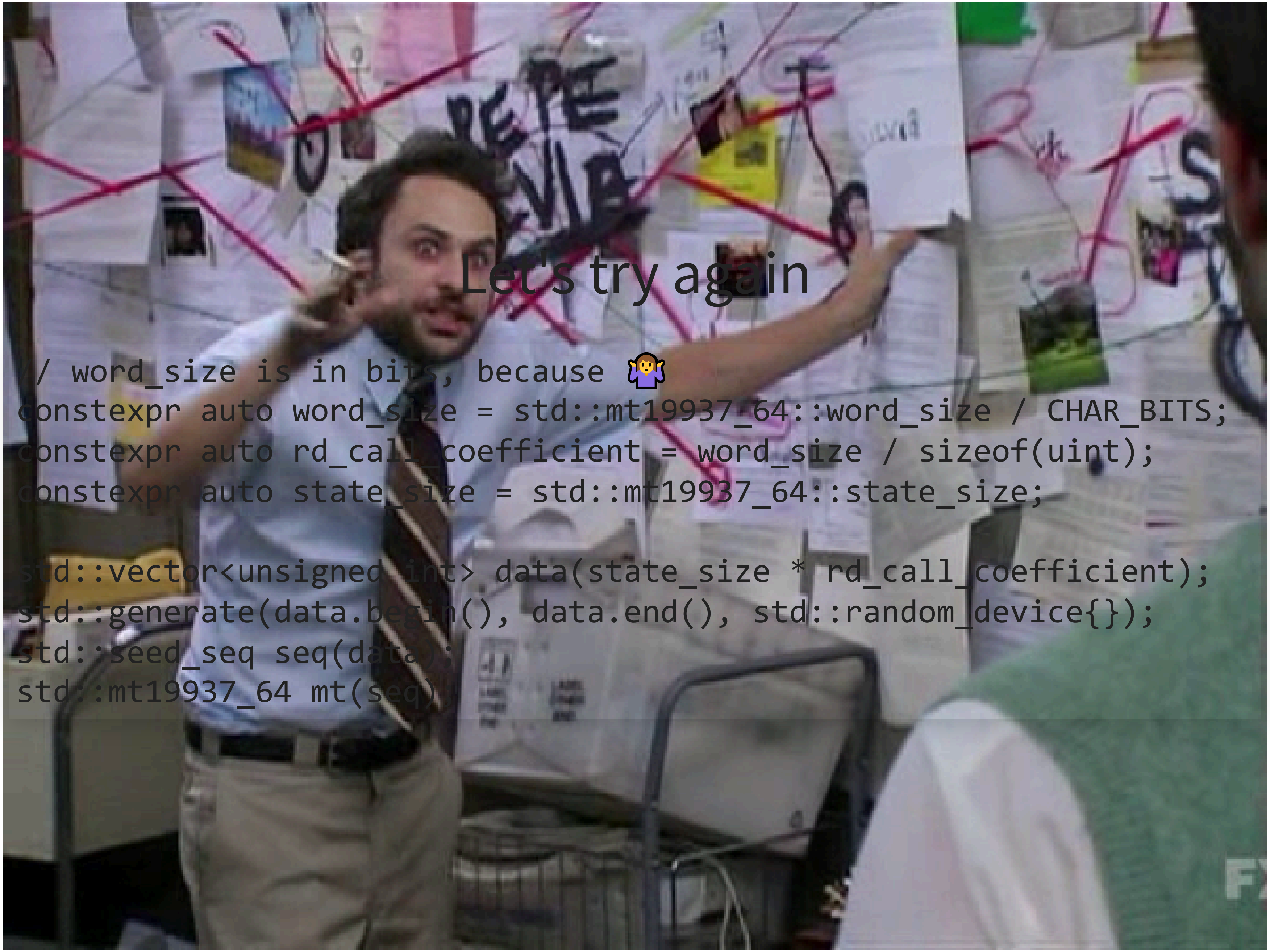
`std::mt19937_64` uses 64 bit types for state.

Let's try again

Let's try again

```
// word_size is in bits, because 🤷‍♀️
constexpr auto word_size = std::mt19937_64::word_size / CHAR_BITS;
constexpr auto rd_call_coefficient = word_size / sizeof(uint);
constexpr auto state_size = std::mt19937_64::state_size;

std::vector<unsigned int> data(state_size * rd_call_coefficient);
std::generate(data.begin(), data.end(), std::random_device{});
std::seed_seq seq(data);
std::mt19937_64 mt(seq);
```



Let's try again

```
// word_size is in bits, because 🤷
constexpr auto word_size = std::mt19937_64::word_size / CHAR_BITS;
constexpr auto rd_call_coefficient = word_size / sizeof(uint);
constexpr auto state_size = std::mt19937_64::state_size;

std::vector<unsigned int> data(state_size * rd_call_coefficient);
std::generate(data.begin(), data.end(), std::random_device{});
std::seed_seq seq(data);
std::mt19937_64 mt(seq);
```

The code is now logically correct ...

The code is now logically correct …

… but practically wrong (details later)

Let's try generic seeding

Let's try generic seeding

```
template <typename RNG>
void seed_rng(RNG& rng) {
    constexpr auto word_size = ????;
    constexpr auto state_size = ????;
}
```

Let's try generic seeding

```
template <typename RNG>
void seed_rng(RNG& rng) {
    constexpr auto word_size = ????;
    constexpr auto state_size = ????;
}
```

We can't query arbitrary engine for state size.

Let's try generic seeding

```
template <typename RNG>
void seed_rng(RNG& rng) {
    constexpr auto word_size = ????;
    constexpr auto state_size = ????;
}
```

We can't query arbitrary engine for state size.

`mt19937` is a standardization accident.

DISTRIBUTIONS

> *Random numbers should not be generated with a method chosen at random.*
>
> — Donald Knuth

Distributions in the standard are

Distributions in the standard are

- irreproducible

Distributions in the standard are

- irreproducible
- opaque

Distributions in the standard are

- irreproducible
- opaque
- **!!buggy!!**

Implementation of the distribution has implications

Implementation of the distribution has implications

Box-Muller transform cannot return values further than 6.66 standard deviation from mean.

Does it matter?

Does it matter?

¯_(ツ)_/¯

6.66 σ means $2.738 * 10^{-11}$ chance of an event.

6.66 σ means $2.738 * 10^{-11}$ chance of an event.

That's roughly $\frac{1}{2^{35}}$.

6.66 σ means $2.738 * 10^{-11}$ chance of an event.

That's roughly $\frac{1}{2^{35}}$.

That's so unlikely, that ...

… if your machine was generating normally distributed numbers

… if your machine was generating normally distributed numbers

It would've generated about dozen during this talk.

LET'S TALK ABOUT BUGS IN DISTRIBUTIONS

`std::uniform_real_distribution(a, b)`

returns values in [a, b)

`std::uniform_real_distribution(a, b)`

returns values in [a, b)

```
std::uniform_real_distribution<> dist(0., 1.);
assert(dist(rng) < 1.); // with primed rng,
                        // fails on some platforms
```

Why?

Why?

There is an *understanding*, that the distribution is transformation on top of `generate_canonical`.

Let's say this isn't a problem for us, because we can rejection-sample the bug away

Let's say this isn't a problem for us, because we can rejection-sample the bug away

`uniform_real_distribution` is not actually uniform

The standard assumes that

The standard assumes that

$$a + x * (b - a)$$

The standard assumes that

$$a + x * (b - a)$$

gives you a number in $[a, b)$ given

$$x \in [0, 1)$$

The standard assumes that

$$a + x * (b - a)$$

gives you a number in $[a, b)$ given

$$x \in [0, 1)$$

For real numbers, this is true.

For floats, it is not.

Floats are **quantized** reals. Rounding will make some floats overrepresented.

There are few more specification bugs in distributions.

There are few more specification bugs in distributions.

Let's not get into them.

HELPERS

Let's start with `generate_canonical`

The standard specifies that it returns numbers in [0, 1).

The standard specifies that it returns numbers in [0, 1).

The standard specifies the algorithm it uses.

The standard specifies that it returns numbers in [0, 1).

The standard specifies the algorithm it uses.

The algorithm is *mathematically* correct.

But only when applied to real numbers.

But only when applied to real numbers.

In floats it will return `1` when fed specific inputs.

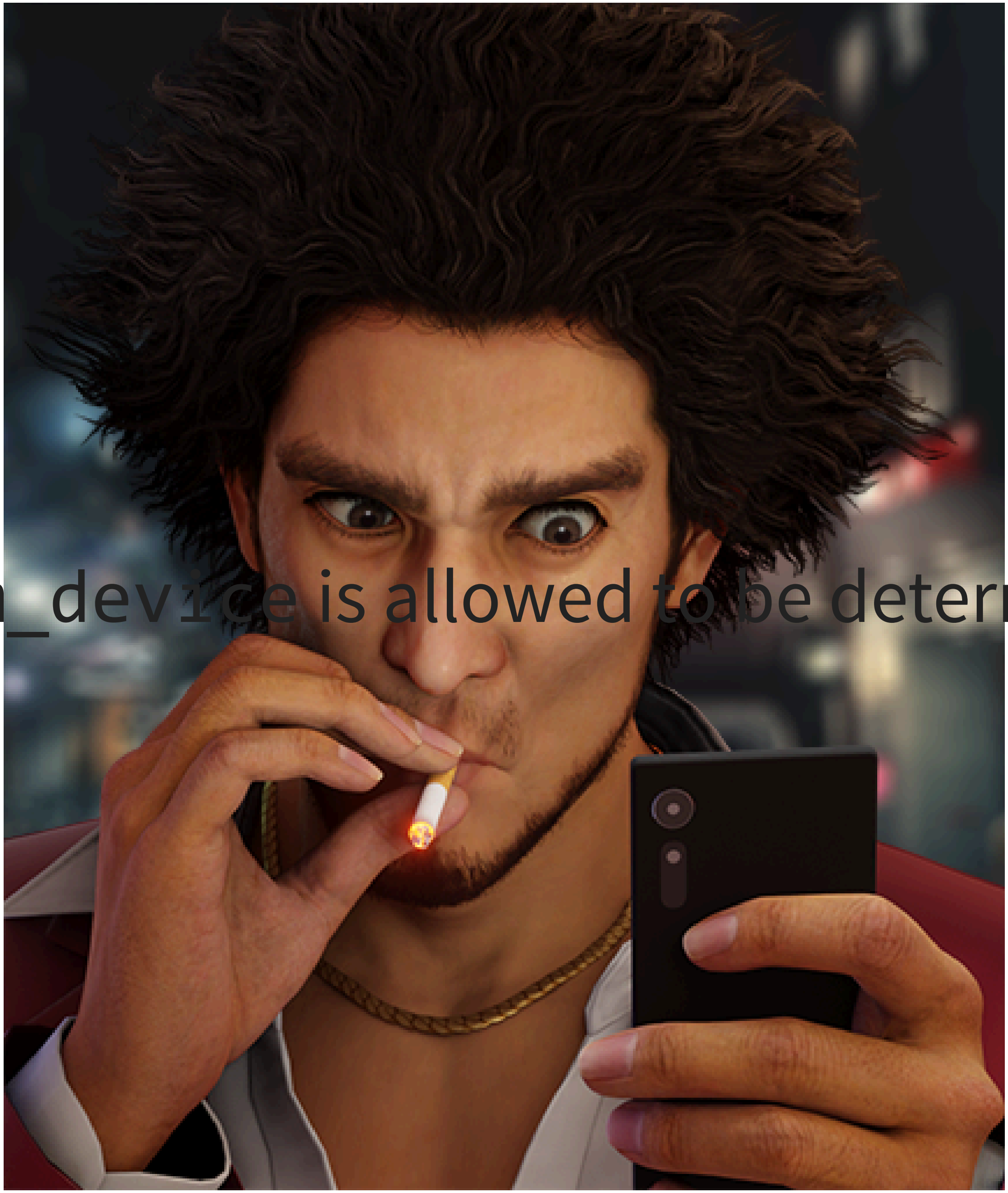
This is a LWG issue that was tentatively fixed for C++23

This is a LWG issue that was tentatively fixed for C++23

But the latest version of libcxx and MS-STL will return 1

Let's talk `std::random_device`

`random_device` is allowed to be deterministic



random_device is allowed to be deterministic

You can check the source code of your library, You can check the source code of your library,

but there is no programmatic way of checking

THIS IS A REAL ISSUE

THIS IS A REAL ISSUE

MinGW used to use `mt19937` as the `random_device`

Due to ABI compatibility, libstdc++ is stuck with this

Due to ABI compatibility, libstdc++ is stuck with this

```
static_assert(sizeof(std::random_device) == 5000);
```

Finally, `std::seed_seq`

`std::seed_seq` has a very simple issue

`std::seed_seq` has a very simple issue

The standard mandates the algorithm it uses to stretch the input bits across arbitrary output size.

THIS ALGORITHM LOSSES ENTROPY

```
int main() {
    std::seed_seq seq1({0xf5e5b5c0, 0xdcb8e4b1}),
                  seq2({0xd34295df, 0xba15c4d0});

    std::array<uint32_t, 2> arr1, arr2;
    seq1.generate(arr1.begin(), arr1.end());
    seq2.generate(arr2.begin(), arr2.end());

    assert(arr1 == arr2);
}
```

But wait! There is more

But wait! There is more

`seed_seq::generate` writes out values mod 2^{32}

But wait! There is more

`seed_seq::generate` writes out values mod 2^{32}

Does your Engine use 32-bit types internally?

Remember this example?

Remember this example?

```
constexpr auto word_size = std::mt19937_64::word_size / CHAR_BITS;
constexpr auto rd_call_coefficient = word_size / sizeof(uint);
constexpr auto state_size = std::mt19937_64::state_size;

std::vector<unsigned int> data(state_size * rd_call_coefficient);
...
```

Remember this example?

```
constexpr auto word_size = std::mt19937_64::word_size / CHAR_BITS;
constexpr auto rd_call_coefficient = word_size / sizeof(uint);
constexpr auto state_size = std::mt19937_64::state_size;

std::vector<unsigned int> data(state_size * rd_call_coefficient);
...
```

All pointless, `mt19937_64` uses 64 bit types for state.

BETTER <RANDOM>

The core design of <random> is good.

The core design of <random> is good.

Splitting generators and distributions was a great idea.

The core design of <random> is good.

Splitting generators and distributions was a great idea.

So how do we fix the actual implementation?

ENGINES

Fixing Engines is as easy as requiring a `state_size_bytes` member in Engines.

Fixing Engines is as easy as requiring a `state_size_bytes` member in Engines.

Oh and we should add some state-of-the-art PRNGs.

DISTRIBUTIONS

Three separate things to fix:

Three separate things to fix:

- Pointless UB

Three separate things to fix:

- Pointless UB
- Reproducibility

Three separate things to fix:

- Pointless UB
- Reproducibility
- uniform_real_distribution is just wrong

Let's talk reproducibility

Let's talk reproducibility

One approach is to keep the names, but standardize the implementation of each distribution.

DOUBLING DOWN ON EXPERT-FRIENDLINESS

Standardize algorithms under their own name.

Standardize algorithms under their own name.

- `box_muller_transform`,

Standardize algorithms under their own name.

- `box_muller_transform`,
- `marsaglia_polar_method`,

Standardize algorithms under their own name.

- `box_muller_transform`,
- `marsaglia_polar_method`,
- ...

Standardize algorithms under their own name.

- `box_muller_transform`,
- `marsaglia_polar_method`,
- ...

This provides full control to the user, so they can pick the best algorithm according to their needs.

Fixing `uniform_real_distribution` is both very easy, and very, very hard.

Fixing `uniform_real_distribution` is both very easy, and very, very hard.

What do we even mean by generating uniformly distributed floats in some range?

Do we mean generating

Do we mean generating

1. uniformly distributed **real** numbers in $[a, b)$, and converting them to corresponding floats?

Do we mean generating

1. uniformly distributed **real** numbers in $[a, b)$, and converting them to corresponding floats?
2. floats in $[a, b)$ with uniform chance of each?

Do we mean generating

1. uniformly distributed **real** numbers in $[a, b)$, and converting them to corresponding floats?
2. floats in $[a, b)$ with uniform chance of each?
3. floats uniformly distributed in $[a, b)$?

Do we mean generating

1. uniformly distributed **real** numbers in $[a, b)$, and converting them to corresponding floats?
2. floats in $[a, b)$ with uniform chance of each?
3. floats uniformly distributed in $[a, b)$?

Each of these 3 is useful to *someone*.

Do we mean generating

1. uniformly distributed **real** numbers in $[a, b)$, and converting them to corresponding floats?
2. floats in $[a, b)$ with uniform chance of each?
3. floats uniformly distributed in $[a, b)$?

Each of these 3 is useful to *someone*.

`uniform_real_distribution` does none of these.

By standardizing algorithms under their name, we can have all of these.

By standardizing algorithms under their name, we can have all of these.

If someone figures out algorithm for the first one 🙃

HELPERS

`generate_canonical` has recently fixed wording

`generate_canonical` has recently fixed wording

Out of the 3 options, it does #3.

`generate_canonical` has recently fixed wording

Out of the 3 options, it does #3.

So what's there to fix?

There are ~1 billion representable `floats` in [0, 1).

There are ~1 billion representable `floats` in [0, 1).

generate_canonical can generate ~17M different floats (1.7%)

There are ~1 billion representable `floats` in [0, 1).

generate_canonical can generate ~17M different floats (1.7%)

For `doubles`, this ratio goes down to 0.2%

Does it matter?

Does it matter?

¯_(ツ)_/¯

Depends on the use case

Depends on the use case

Different algorithm can return any float in [0, 1).

`seed_seq` and the related concepts need to be completely thrown away

Replacement seed sequence type has to

Replacement seed sequence type has to

- not reduce entropy (be "as bijective as possible")

Replacement seed sequence type has to

- not reduce entropy (be "as bijective as possible")
- support any integral type for state

Replacement SeedSequence concepts should support random_device as well as serializable seed sequences.

Finally, `random_device`

`random_device` should never be deterministic

`random_device::entropy` is a failed experiment, drop it.

IS THAT ALL?

No

No

But it's enough to make <random> useful.

OTHER MISC IMPROVEMENTS

All* of <random> can be `constexpr`

All* of <random> can be `constexpr`

All bit generators and distributions should have bulk-generation API

All* of <random> can be `constexpr`

All bit generators and distributions should have bulk-generation API

Engines should be seedable with `random_device`

Clean up types to make more sense

Clean up types to make more sense

Fix wording on various distributions

Clean up types to make more sense

Fix wording on various distributions

...

But again, at this point we are talking about nice to haves.

THE END

Time for questions!

Further reading:

- https://wg21.link/P2058
- https://wg21.link/P2059
- https://wg21.link/P2060
- https://codingnest.com/generating-random-numbers-using-c-standard-library-the-problems/
- https://codingnest.com/generating-random-numbers-using-c-standard-library-the-solutions/
- https://codingnest.com/random-distributions-are-not-one-size-fits-all-part-1/
- https://codingnest.com/random-distributions-are-not-one-size-fits-all-part-2/