Real-time programming with the C++ standard library

Timur Doumler

@timur_audio

CppCon
29 October 2021

Pulsar PSR B1509−58
Image credit: NASA/CXC/CfA/P

“Real-time” programming with the C++ standard library

Timur Doumler

@timur_audio

CppCon
29 October 2021

Pulsar PSR B1509−58
Image credit: NASA/CXC/CfA/P

what do we mean by “real-time”?

what do we mean by “real-time”?

what does it have to do with “low latency”?


latency
bandwidth

“real-time”

In order to be considered correct,
not only does the program have to
produce the correct result,
but it also has to produce it
within a certain amount of time.

use cases

- high-frequency trading
- embedded devices
- video games
- audio processing

audio processing

audio callback

```
process(buffer& b)
{
   // write your data
   // into buffer!
}
```

audio processing

audio callback

```
process(buffer& b)
{
    // write your data
    // into buffer!
}
```

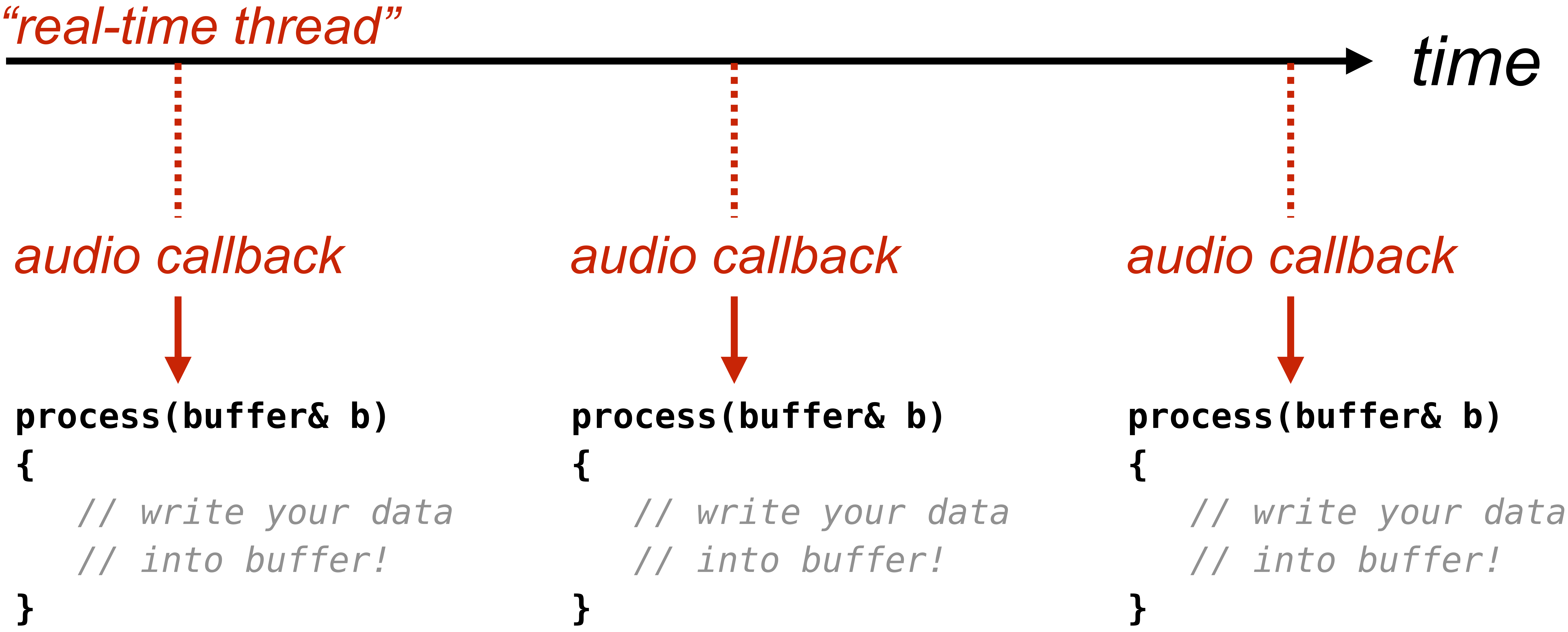
audio callback

```
process(buffer& b)
{
    // write your data
    // into buffer!
}
```

audio callback

```
process(buffer& b)
{
    // write your data
    // into buffer!
}
```

audio processing



"real-time thread"

time

audio callback

```
process(buffer& b)
{
    // write your data
    // into buffer!
}
```

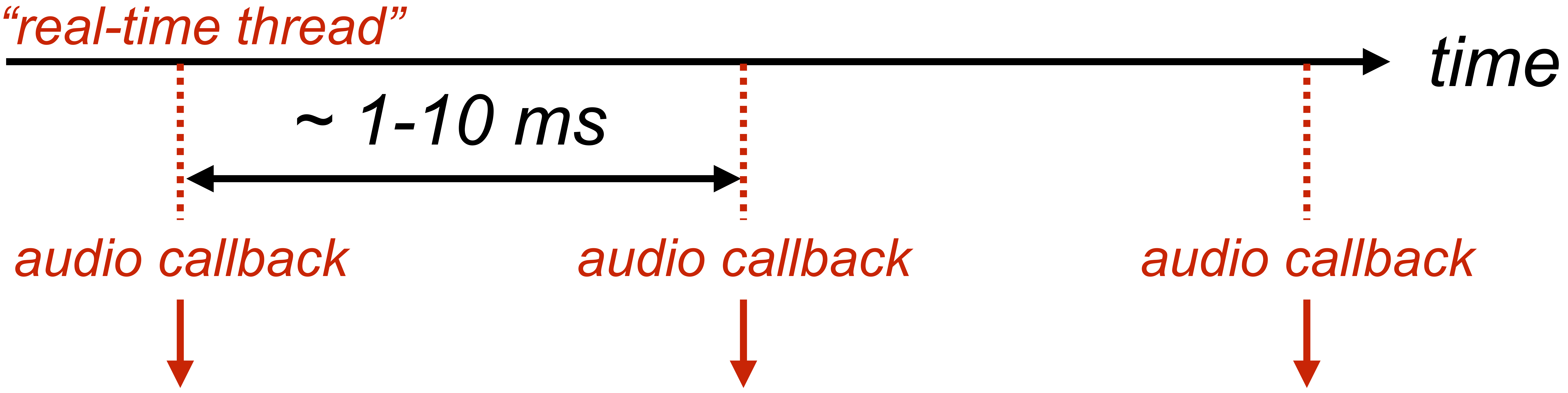
audio callback

```
process(buffer& b)
{
    // write your data
    // into buffer!
}
```

audio callback

```
process(buffer& b)
{
    // write your data
    // into buffer!
}
```

audio processing



```
process(buffer& b)
{
    // write your data
    // into buffer!
}
```

```
process(buffer& b)
{
    // write your data
    // into buffer!
}
```

```
process(buffer& b)
{
    // write your data
    // into buffer!
}
```

“real-time” programming

- on a normal, non-realtime OS kernel *(Windows, macOS, iOS, Linux, Android)*
- cross-platform (portability!)
- on a normal consumer machine
- using a normal C++ implementation *(msvc, clang, gcc)*
- only parts of the program are subject to “real-time” constraints, others (e.g. GUI) are not

audio callback

```
process(buffer& b)
{
   fillWithAudioSamples(buffer);  // are these functions “real-time safe”?
   applyGain(buffer, g);
}
```

“real-time safe code”

- The worst-case execution time is
    - deterministic,
    - known in advance,
    - independent of application data,
    - shorter than the given deadline.
- The code does not fail.

“real-time safe code”

- The worst-case execution time is
  - deterministic,
  - known* in advance, **in principle*
  - independent of application data,
  - shorter than the given deadline.
- The code does not fail.

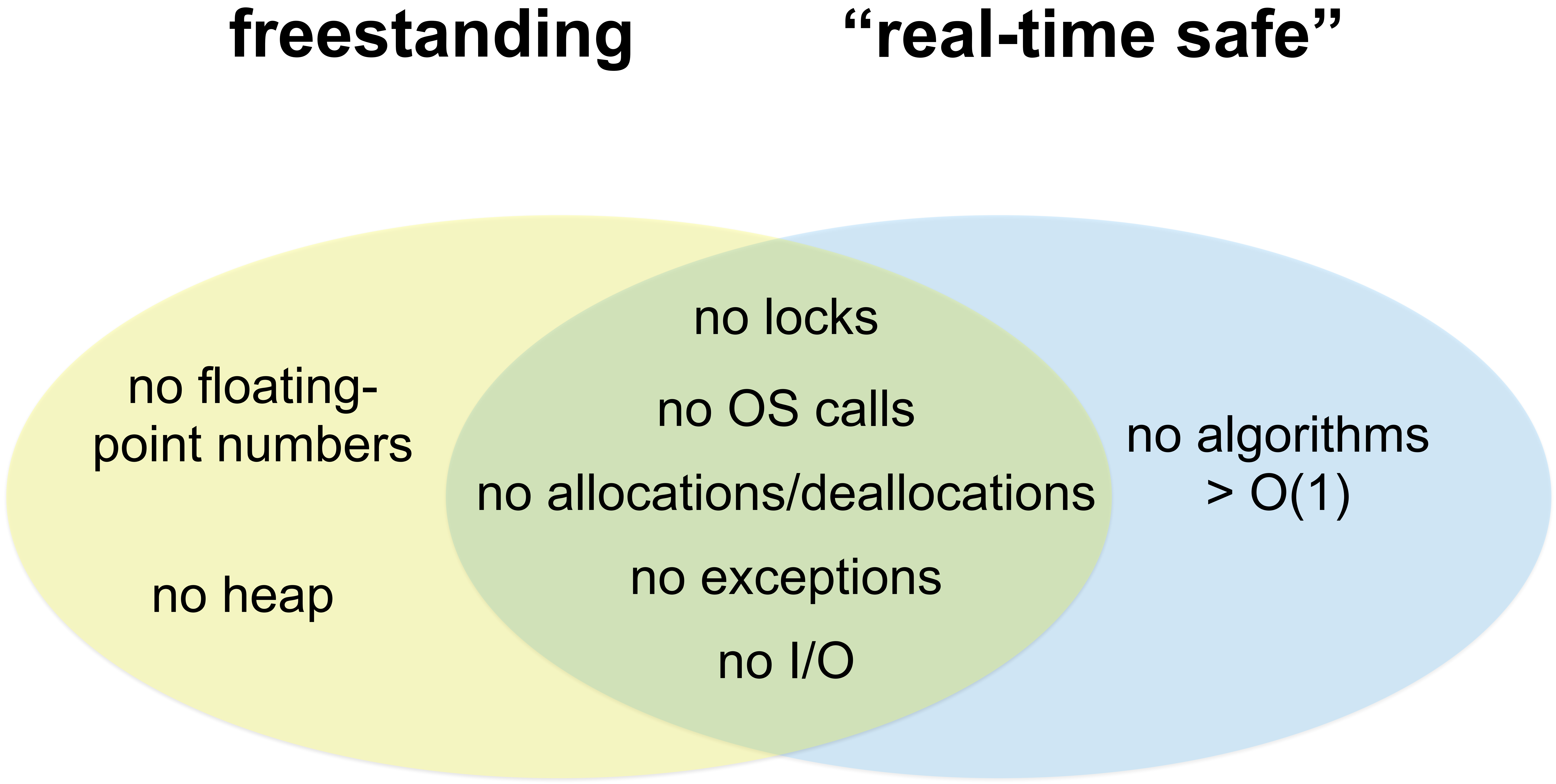
- don’t call anything that might block

  (non-deterministic execution time + priority inversion!)

  - don’t try to acquire a mutex
  - don’t allocate / deallocate memory
  - don’t do any I/O
  - don’t interact with the thread scheduler
  - don’t do any other system calls
- don’t call any 3rdparty code if you don’t know what it’s doing
- don’t use algorithms with > *O(1)* complexity
- don’t use algorithms with *amortised O(1)* complexity

is this the same as “freestanding C++”?

- Proposals by Ben Craig:
  - P0829: *Freestanding Proposal*
  - P1642: *Freestanding Library: Easy [utilities], [ranges], and [iterators]*
  - P2013 *Freestanding Language: Optional ::operator new*
  - P2198 *Freestanding Feature-Test Macros and Implementatio Defined Extensions*
  - P2268 *Freestanding Roadmap*
  - ... more upcoming

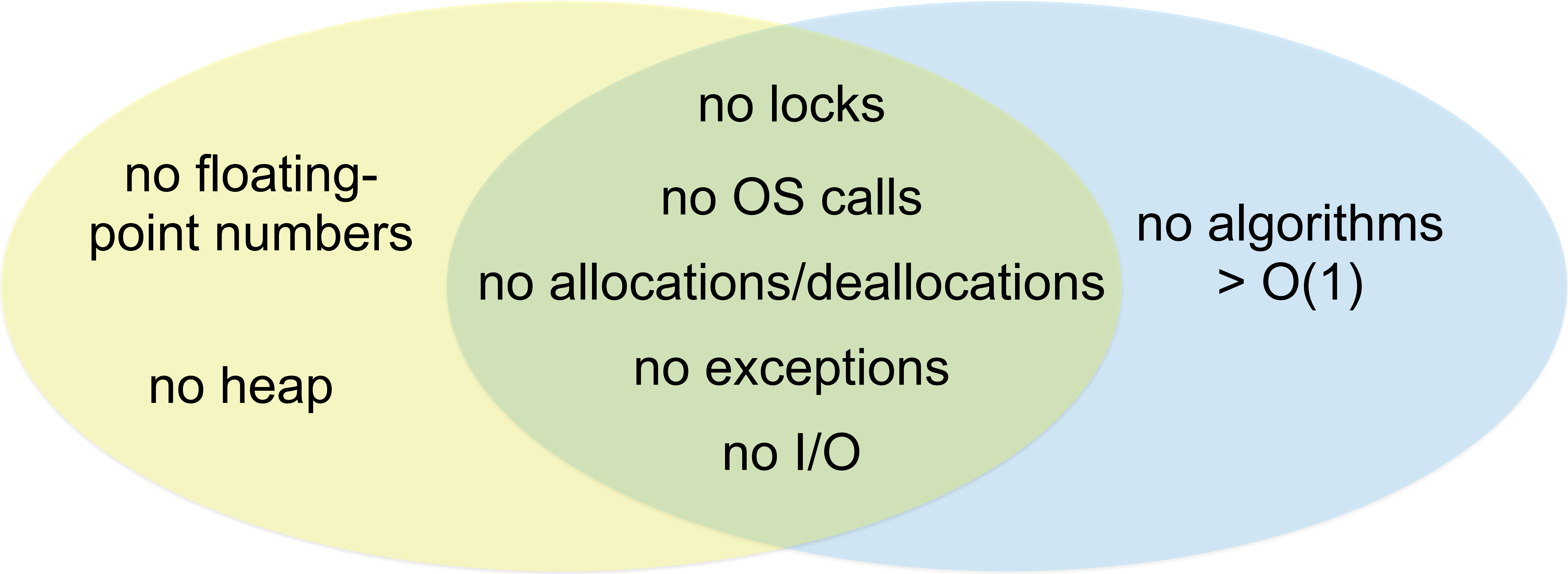

freestanding
“real-time safe”
no floating-point numbers
no heap
no locks
no OS calls
no allocations/deallocations
no exceptions
no I/O
no algorithms > O(1)

freestanding:

“these things do not exist”

“real-time safe”:

“don’t use these things on the real-time thread”



no floating-point numbers

no heap

no locks

no OS calls

no allocations/deallocations

no exceptions

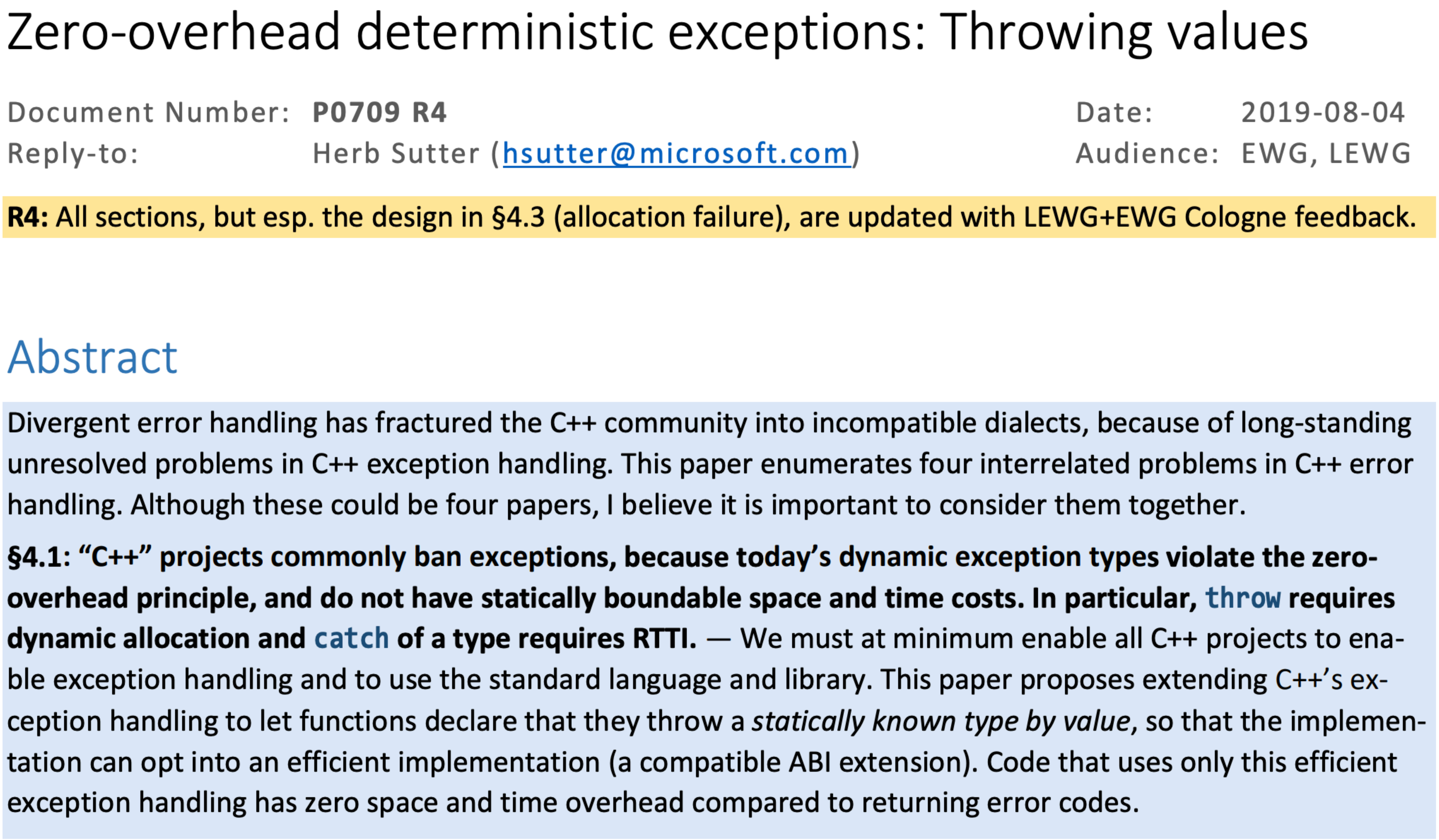
no I/O

no algorithms > O(1)

Which parts of the ***C++ standard library***
are “real-time safe”?

- The C++ standard says ***nothing*** about execution time.
- The C++ standard doesn’t say “f does not allocate memory”
  - Infer from specification that allocations are not needed
  - Sometimes, there are useful sentences like *“f might invalidate iterators”* or *“If there is enough memory, f does X, otherwise…”*
- The C++ standard doesn’t say “X doesn’t use locks”
  - It says *“X may not be accessed from multiple threads simultaneously”*
  - Otherwise, it might say *“X may not introduce data races”*

Exceptions are not “real-time safe”.



Zero-overhead deterministic exceptions: Throwing values

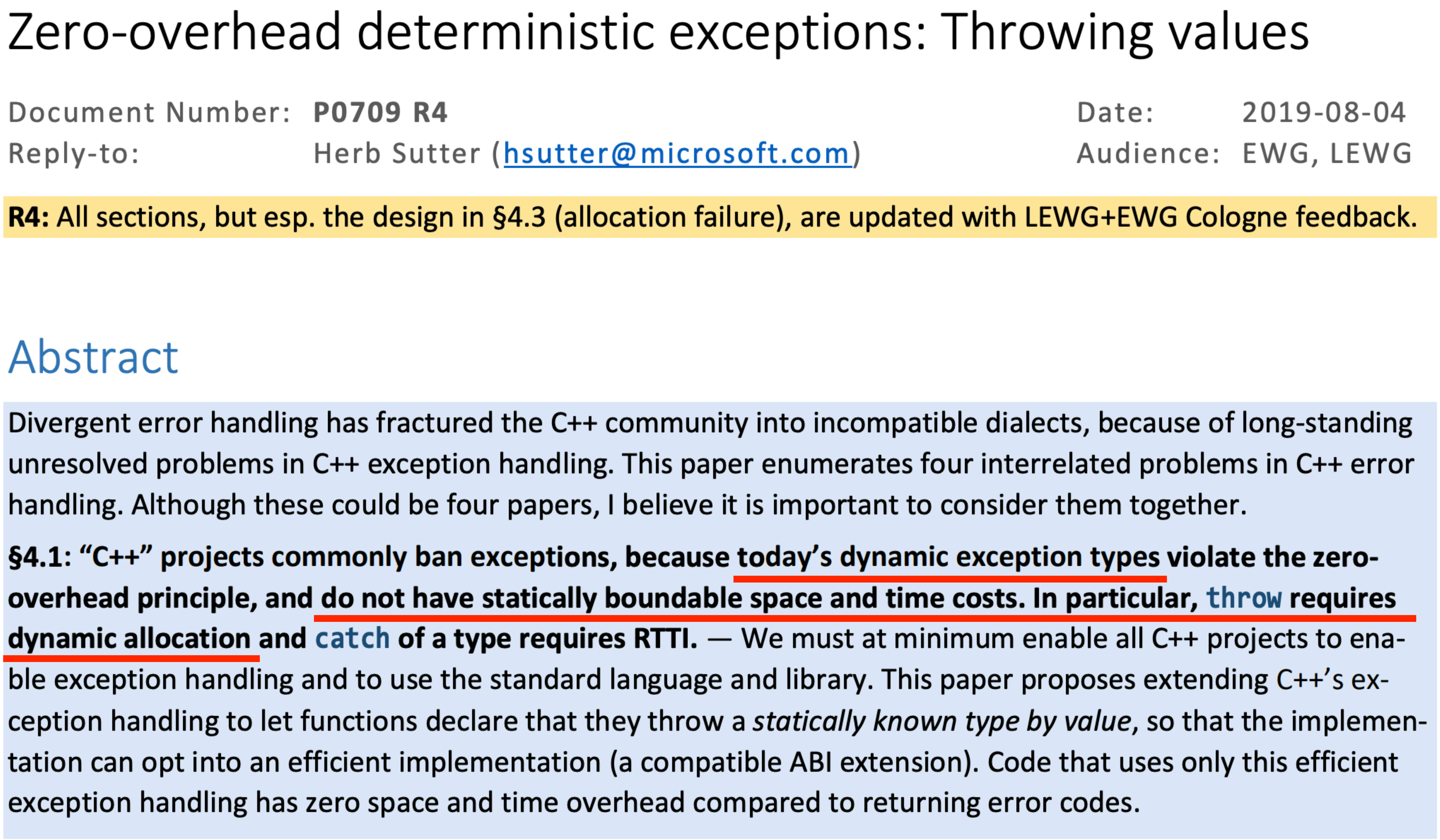
Document Number: **P0709 R4**
Reply-to: Herb Sutter (hsutter@microsoft.com)
Date: 2019-08-04
Audience: EWG, LEWG

R4: All sections, but esp. the design in §4.3 (allocation failure), are updated with LEWG+EWG Cologne feedback.

Abstract

Divergent error handling has fractured the C++ community into incompatible dialects, because of long-standing unresolved problems in C++ exception handling. This paper enumerates four interrelated problems in C++ error handling. Although these could be four papers, I believe it is important to consider them together.

§4.1: "C++" projects commonly ban exceptions, because today's dynamic exception types violate the zero-overhead principle, and do not have statically boundable space and time costs. In particular, `throw` requires dynamic allocation and `catch` of a type requires RTTI. — We must at minimum enable all C++ projects to enable exception handling and to use the standard language and library. This paper proposes extending C++'s exception handling to let functions declare that they throw a *statically known type by value*, so that the implementation can opt into an efficient implementation (a compatible ABI extension). Code that uses only this efficient exception handling has zero space and time overhead compared to returning error codes.



Zero-overhead deterministic exceptions: Throwing values

Document Number: **P0709 R4** Date: 2019-08-04
Reply-to: Herb Sutter (hsutter@microsoft.com) Audience: EWG, LEWG

R4: All sections, but esp. the design in §4.3 (allocation failure), are updated with LEWG+EWG Cologne feedback.

Abstract

Divergent error handling has fractured the C++ community into incompatible dialects, because of long-standing unresolved problems in C++ exception handling. This paper enumerates four interrelated problems in C++ error handling. Although these could be four papers, I believe it is important to consider them together.

§4.1: "C++" projects commonly ban exceptions, because today's dynamic exception types violate the zero-overhead principle, and do not have statically boundable space and time costs. In particular, `throw` requires dynamic allocation and `catch` of a type requires RTTI. — We must at minimum enable all C++ projects to enable exception handling and to use the standard language and library. This paper proposes extending C++'s exception handling to let functions declare that they throw a *statically known type by value*, so that the implementation can opt into an efficient implementation (a compatible ABI extension). Code that uses only this efficient exception handling has zero space and time overhead compared to returning error codes.

Is it “real-time safe” to enter & leave a `try` block if you don’t throw any exceptions?

Exception implementation depends on ABI

Exception implementation depends on ABI

- Unwind info in static tables, aka “zero cost exception model”
  - Itanium ABI (gcc/clang)
  - MSVC 64-bit
  - ARM ABI (32-bit and 64-bit)
- Unwind info generated at runtime
  - MSVC 32-bit

Is it “real-time safe” to enter & leave a `try` block if you don’t throw any exceptions?

Yes.

→ *Ben Craig: “P1886 Error Speed Benchmarking”*

What STL algorithms are “real-time safe”?

What STL algorithms are “real-time safe”?

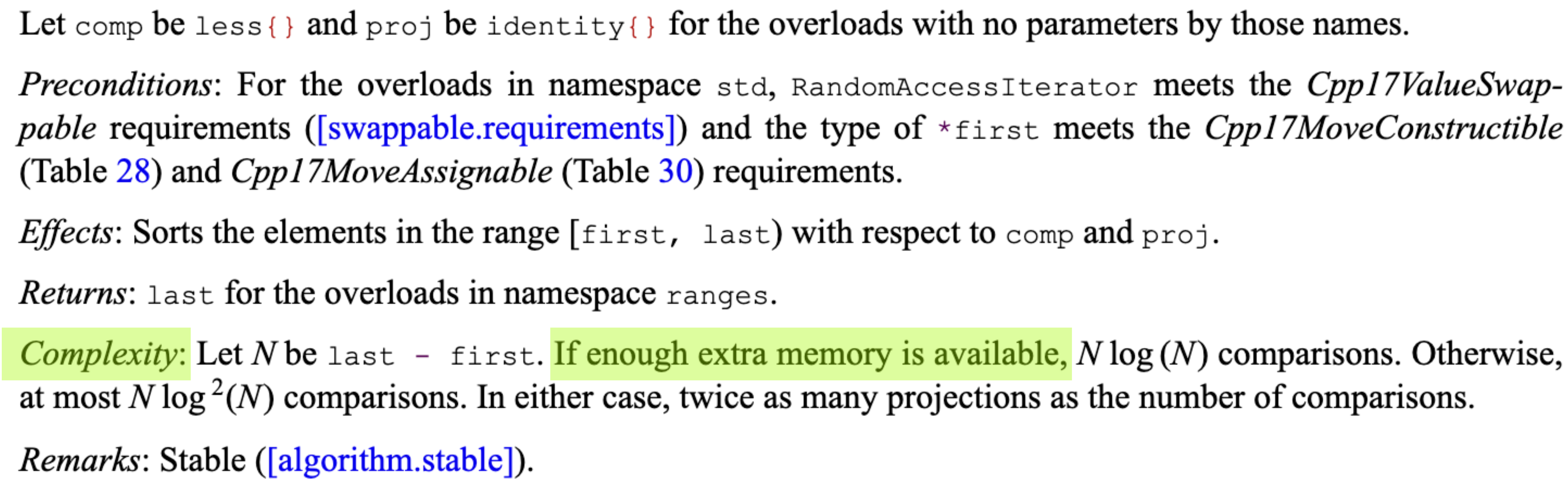
(assuming the element type / iterator type are “real-time safe”)

What STL algorithms are “real-time safe”?

(assuming the element type / iterator type are “real-time safe”)

The standard doesn’t say.

But for ***almost*** all of them, an optimal implementation of the spec doesn’t require additional allocations.



Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

Preconditions: For the overloads in namespace `std`, `RandomAccessIterator` meets the *Cpp17ValueSwappable* requirements ([swappable.requirements]) and the type of `*first` meets the *Cpp17MoveConstructible* (Table 28) and *Cpp17MoveAssignable* (Table 30) requirements.

Effects: Sorts the elements in the range `[first, last)` with respect to `comp` and `proj`.

Returns: `last` for the overloads in namespace `ranges`.

Complexity: Let N be `last - first`. If enough extra memory is available, $N \log(N)$ comparisons. Otherwise, at most $N \log^2(N)$ comparisons. In either case, twice as many projections as the number of comparisons.

Remarks: Stable ([algorithm.stable]).

```
// not "real-time safe":

std::stable_sort
std::stable_partition
std::inplace_merge
```

```
// not "real-time safe":

std::stable_sort
std::stable_partition
std::inplace_merge

std::execution::parallel_*
```

what about STL containers?

- `std::array` is on the stack

  → “realtime-safe”

  (except `at()` which can throw)

- all others use dynamic memory

  → not “realtime-safe”

OK, but what if you need a dynamically-sized container on the real-time thread?

```
void process(buffer& b)
{
    float vla[b.size()]; // variable-length array (VLA)
}
```

```
void process(buffer& b)
{
    float vla[b.size()]; // MSVC: error C2131:
}                        // expression did not evaluate to a constant
```

STL containers with custom allocators?

STL containers with custom allocators?

- general-purpose allocators (tcmalloc, rpmalloc…) are not “real-time safe”

STL containers with custom allocators?

- general-purpose allocators (tcmalloc, rpmalloc…) are not “real-time safe”:
  - minimising average cost, not worst case
  - not constant time
  - multithreaded (locks)
  - eventually go to OS to request dynamic memory

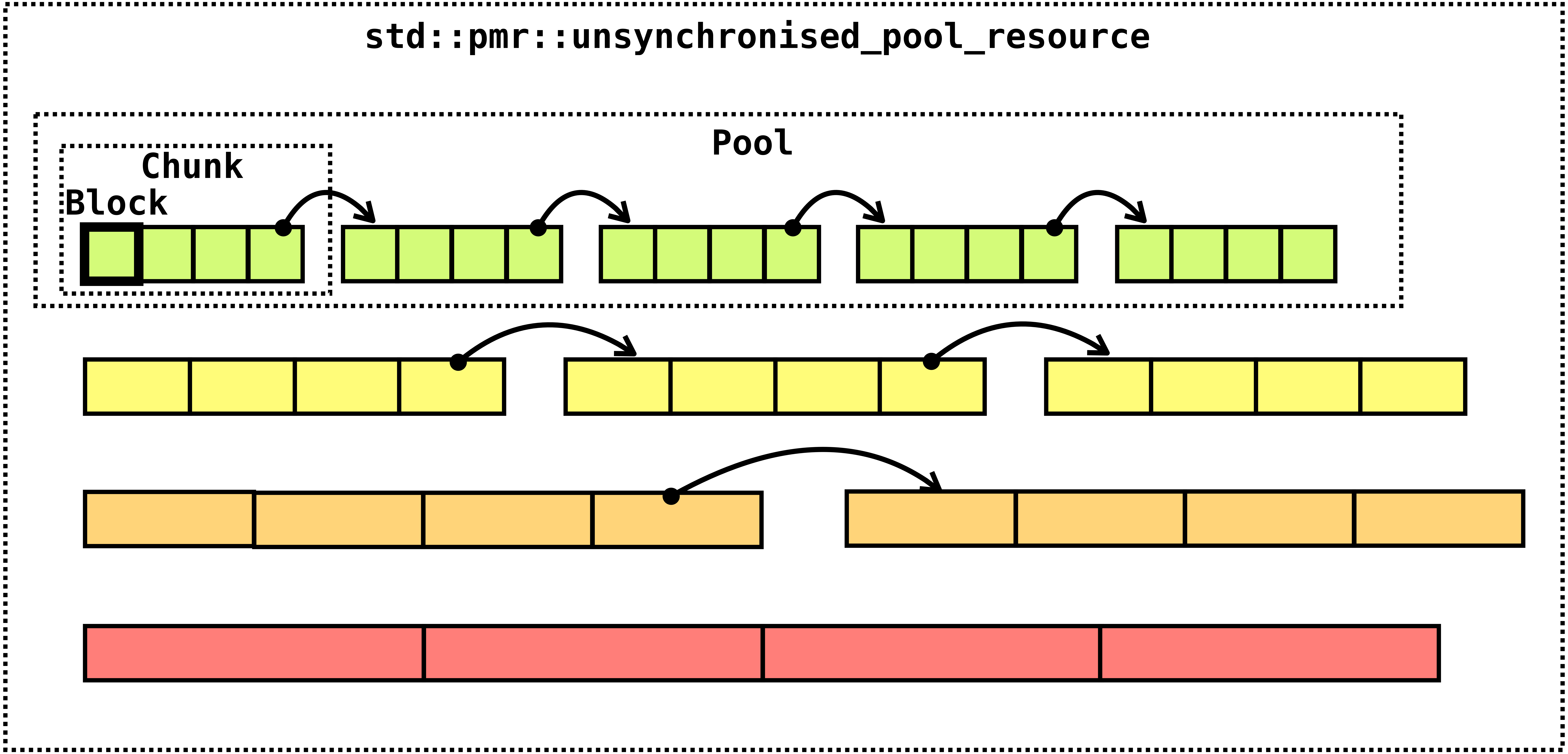
STL containers with custom allocators?

- “real-time safe” allocator:
  - constant time
  - single-threaded
  - only use memory allocated upfront

```
std::array<float, 1024> stack_memory;

void process(buffer& b)
{
    std::pmr::monotonic_buffer_resource monotonic_buffer(
        stack_memory.data(),
        stack_memory.size(),
        std::pmr::null_memory_resource());

    using allocator_t = std::pmr::polymorphic_allocator<float>;
    allocator_t allocator(&monotonic_buffer);

    std::pmr::vector<float> my_vector(b.size(), 0.0f, allocator);
}
```

```
std::array<float, 1024> stack_memory;

void process(buffer& b)
{
    std::pmr::monotonic_buffer_resource monotonic_buffer(
        stack_memory.data(),
        stack_memory.size(),
        std::pmr::null_memory_resource());

    using allocator_t = std::pmr::polymorphic_allocator<float>;
    allocator_t allocator(&monotonic_buffer);

    std::pmr::vector<float> my_vector(b.size(), 0.0f, allocator);
}
```

```
std::array<float, 1024> stack_memory;

void process(buffer& b)
{
    std::pmr::monotonic_buffer_resource monotonic_buffer(
        stack_memory.data(),
        stack_memory.size(),
        std::pmr::null_memory_resource());

    using allocator_t = std::pmr::polymorphic_allocator<float>;
    allocator_t allocator(&monotonic_buffer);

    std::pmr::vector<float> my_vector(b.size(), 0.0f, allocator);
}
```


std::pmr::unsynchronised_pool_resource
Pool
Chunk
Block

```
std::pmr::monotonic_buffer_resource monotonic_buffer(
    safe_memory.data(),
    safe_memory.size(),
    std::pmr::null_memory_resource());

std::pmr::unsynchronized_pool_resource pool(
    std::pmr::pool_options(...),
    &monotonic_buffer);
```

```
std::pmr::monotonic_buffer_resource monotonic_buffer(
    safe_memory.data(),
    safe_memory.size(),
    std::pmr::null_memory_resource());

std::pmr::unsynchronized_pool_resource pool(
    std::pmr::pool_options(...),
    &monotonic_buffer);
```

- Better: `static_vector`
    - smaller
    - faster (no indirection)
    - no need to construct allocator object outside vector

→ *David Stone: “Implementing `static_vector`”*

→ *P0843*

what about utilities?

what about utilities?

- `std::pair/std::tuple` are on the stack

  → “realtime-safe”

what about utilities?

- `std::pair/std::tuple` are on the stack

  → “realtime-safe”

- `std::optional` is just a value + a bool on the stack

  → “realtime-safe”

what about utilities?

- `std::pair/std::tuple` are on the stack
  → “realtime-safe”
- `std::optional` is just a value + a bool on the stack
  → “realtime-safe”
- `std::variant` is just a union on the stack
  → “realtime-safe”

what about utilities?

- `std::pair`/`std::tuple` are on the stack

  → “realtime-safe”
- `std::optional` is just a value + a bool on the stack

  → “realtime-safe”
- `std::variant` is just a union on the stack

  → “realtime-safe” (but `boost::variant` is not!)

```cpp
struct ThrowsOnConstruction
{
    ThrowsOnConstruction() { throw std::exception(); }
};

void variantTest()
{
    std::variant<int, ThrowsOnConstruction> var = 42; // var now holds an int

    try
    {
        var.emplace<ThrowsOnConstruction>(); // ctor fails!
    }
    catch (std::exception&)
    {
        // what does var hold now?
    }
}
```

```
struct ThrowsOnConstruction
{
    ThrowsOnConstruction() { throw std::exception(); }
};

void variantTest()
{
    std::variant<int, ThrowsOnConstruction> var = 42; // var now holds an int

    try
    {
        var.emplace<ThrowsOnConstruction>(); // ctor fails!
    }
    catch (std::exception&)
    {
        // what does var hold now?
    }
}
```

```
struct ThrowsOnConstruction
{
    ThrowsOnConstruction() { throw std::exception(); }
};

void variantTest()
{
    std::variant<int, ThrowsOnConstruction> var = 42; // var now holds an int

    try
    {
        var.emplace<ThrowsOnConstruction>(); // ctor fails!
    }
    catch (std::exception&)
    {
        // what does var hold now?
    }
}
```

```
struct ThrowsOnConstruction
{
    ThrowsOnConstruction() { throw std::exception(); }
};

void variantTest()
{
    std::variant<int, ThrowsOnConstruction> var = 42; // var now holds an int

    try
    {                                       // Boost.Variant:
        var.emplace<ThrowsOnConstruction>(); // temporary heap backup :(
    }
    catch (std::exception&)
    {
        // Boost.Variant: still holds an int.
    }
}
```

```
struct ThrowsOnConstruction
{
    ThrowsOnConstruction() { throw std::exception(); }
};

void variantTest()
{
    std::variant<int, ThrowsOnConstruction> var = 42; // var now holds an int

    try
    {                                       // std::variant:
        var.emplace<ThrowsOnConstruction>(); // no heap allocation :)
    }
    catch (std::exception&)
    {
        // var.valueless_by_exception() == true
    }
}
```

Everything using type erasure is not “real-time safe”

- `std::any`
- `std::function`

Lambdas

```
void process(buffer& b)
{
    std::array a = {1, 1, 2, 3, 5, 8, 13};

    // “real-time safe”
    auto f = [=] {
        return std::accumulate (a.begin(), a.end(), 0);
    };
}
```

```
void process(buffer& b)
{
    std::array a = {1, 1, 2, 3, 5, 8, 13};

    // “real-time safe”
    auto f = [=] {
        return std::accumulate (a.begin(), a.end(), 0);
    };

    do_something(b, f);
}
```

Coroutines

```
// generates the sequence 0, 1, 2, ...
generator<int> f()
{
    int i = 0;
    while (true)
        co_yield I++;
}
```

```
// generates the sequence 0, 1, 2, ...
generator<int> f()
{
    int i = 0;
    while (true)
        co_yield I++;
}

void process(buffer& b)
{
    auto gen = f();
    do_something(b, gen);
}
```

```
// generates the sequence 0, 1, 2, ...
generator<int> f()
{
    int i = 0;
    while (true)
        co_yield I++;
}

void process(buffer& b)
{
    auto gen = f();   // may perform dynamic allocation :(
    do_something(b, gen);
}
```

Options

- rely on the optimiser?
  → *Eyal Zedaka: "Using Coroutines to Implement C++ Exceptions for Freestanding Environments"*
- create and suspend coroutine frame upfront
- write your own promise type, defining its own custom `operator new` and `operator delete`
- Don't use coroutines on the real-time thread

`mutex`
`timed_mutex`
`recursive_mutex`
`recursive_timed_mutex`
`shared_mutex`
`shared_timed_mutex`
`scoped_lock`
`unique_lock`
`shared_lock`

`condition_variable`
`condition_variable_any`
`counting_semaphore`
`binary_semaphore`
`latch`
`barrier`

Nothing in the C++20 thread support library is portably “real-time safe” :(

- ~~`mutex`~~
- ~~`timed_mutex`~~
- ~~`recursive_mutex`~~
- ~~`recursive_timed_mutex`~~
- ~~`shared_mutex`~~
- ~~`shared_timed_mutex`~~
- ~~`scoped_lock`~~
- ~~`unique_lock`~~
- ~~`shared_lock`~~

- ~~`condition_variable`~~
- ~~`condition_variable_any`~~
- ~~`counting_semaphore`~~
- ~~`binary_semaphore`~~
- ~~`latch`~~
- ~~`barrier`~~

```cpp
std::mutex mtx;
shared_object obj;

void process(buffer& b)
{
    if (std::unique_lock lock(mtx, std::try_to_lock); lock.owns_lock())
    {
        do_some_processing(b, obj);
    }
    else
    {
        std::ranges::fill(b, 0.0f);  // fallback strategy: output silence
    }
}
```

```
std::mutex mtx;
shared_object obj;

void process(buffer& b)
{
    if (std::unique_lock lock(mtx, std::try_to_lock); lock.owns_lock())
    {
        do_some_processing(b, obj);
    }
    else
    {
        std::ranges::fill(b, 0.0f);  // fallback strategy: output silence
    }
}
```

```
std::mutex mtx;
shared_object obj;

void process(buffer& b)
{
    if (std::unique_lock lock(mtx, std::try_to_lock); lock.owns_lock())
    {
        do_some_processing(b, obj);
    } // might wake up another thread -> not “real-time safe”
    else
    {
        std::ranges::fill(b, 0.0f);  // fallback strategy: output silence
    }
}
```

C++ has exactly one “real-time safe” thread synchronisation mechanism:

C++ has exactly one “real-time safe” thread synchronisation mechanism:

`std::atomic`

`std::atomic`

- use on its own (for single values shared with real-time thread)
- lock-free queues

  → *not recommended to implement your own ;)*
- spinlocks

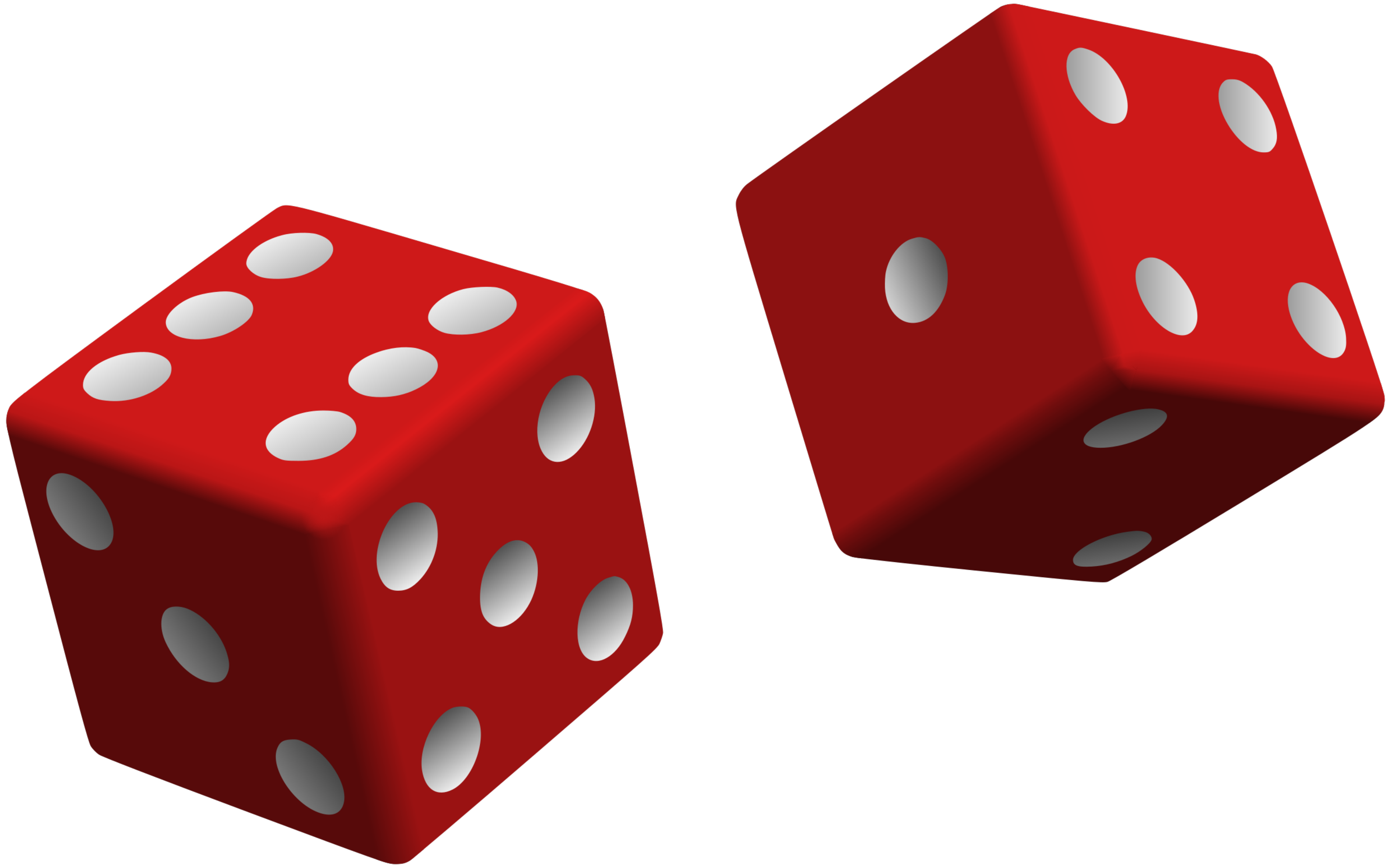
  → *efficient implementation: see my talk at ADC'20*

  "Using locks in real-time audio processing, safely"
- make sure it's lock-free!

  → *"atomic" doesn't mean lock-free, it means "no data race"*

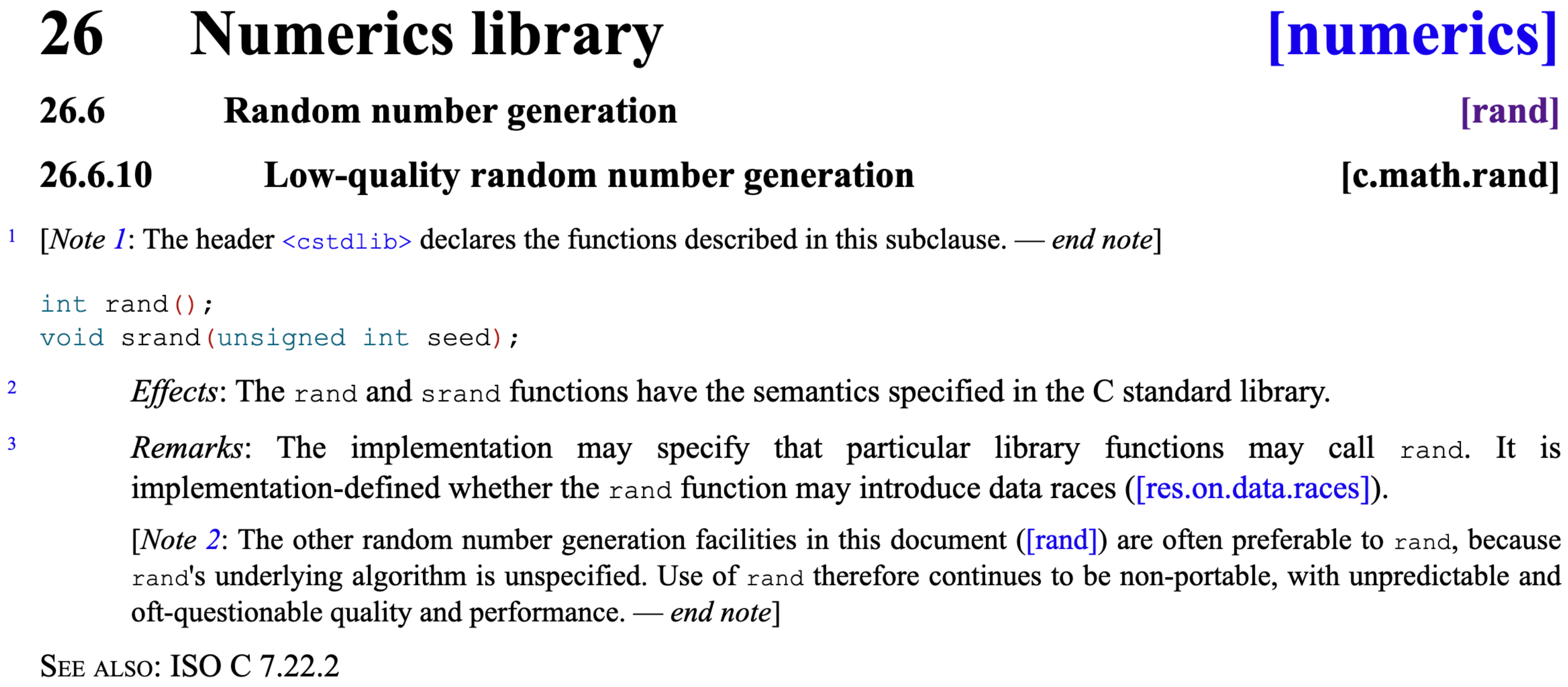
```
using T = std::complex<double>;

// if this fails, your compiler will add locks -> not “real-time safe”
static_assert(std::atomic<T>::is_always_lock_free);
```

What about random number generators?



```
// returns a random float in the interval [0, 1)
float get_random_sample()
{
    return float(std::rand()) / float(INT_MAX);
}

void process(buffer& b)
{
   // fill buffer with random white noise:
   std::ranges::fill(b, get_random_sample);
}
```

```
// returns a random float in the interval [0, 1)
float get_random_sample()
{
    return float(std::rand()) / float(INT_MAX);
}

void process(buffer& b)
{
   // fill buffer with random white noise:
   std::ranges::fill(b, get_random_sample);
}
```



26 Numerics library [numerics]

26.6 Random number generation [rand]

26.6.10 Low-quality random number generation [c.math.rand]

1 [*Note 1*: The header `<cstdlib>` declares the functions described in this subclause. — *end note*]

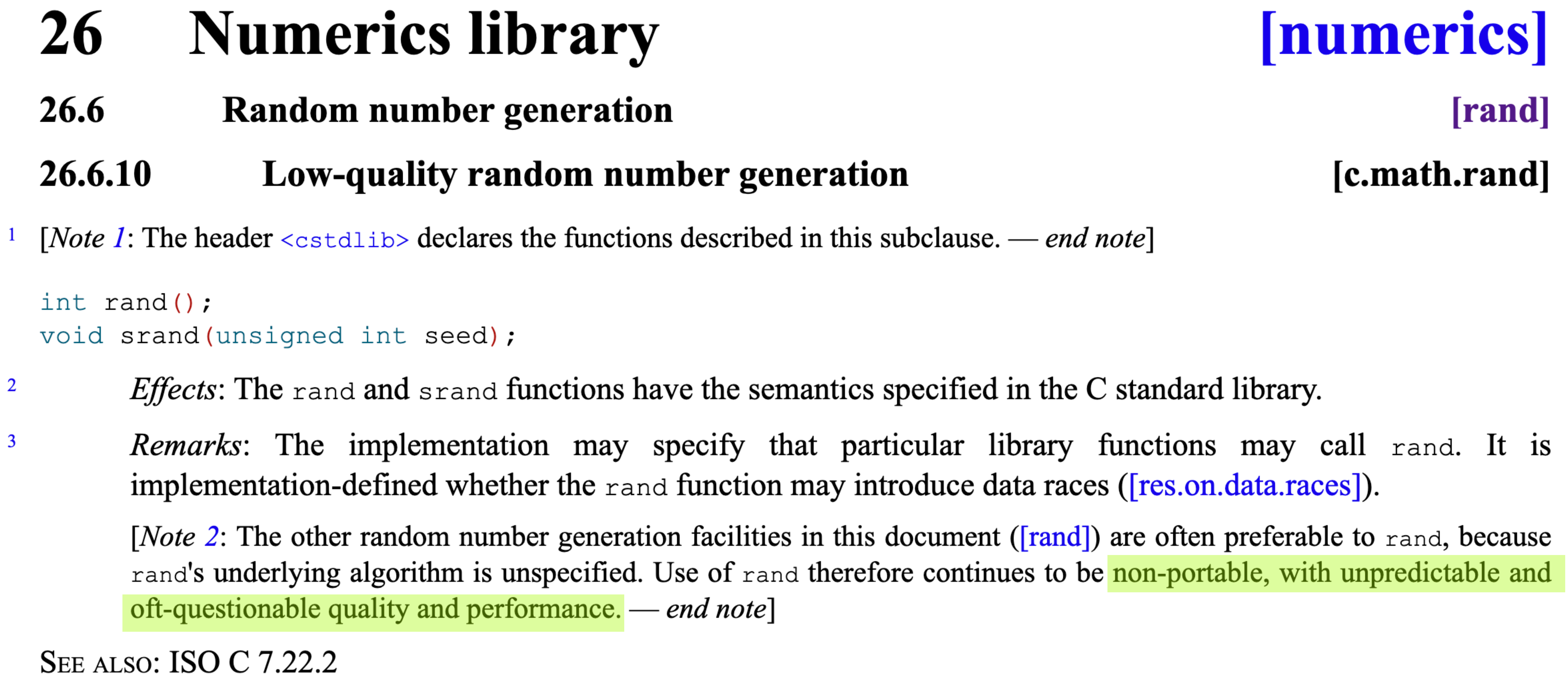
```
int rand();
void srand(unsigned int seed);
```

2 *Effects*: The `rand` and `srand` functions have the semantics specified in the C standard library.

3 *Remarks*: The implementation may specify that particular library functions may call `rand`. It is implementation-defined whether the `rand` function may introduce data races ([res.on.data.races]).

[*Note 2*: The other random number generation facilities in this document ([rand]) are often preferable to `rand`, because `rand`'s underlying algorithm is unspecified. Use of `rand` therefore continues to be non-portable, with unpredictable and oft-questionable quality and performance. — *end note*]

SEE ALSO: ISO C 7.22.2



26 Numerics library [numerics]

26.6 Random number generation [rand]

26.6.10 Low-quality random number generation [c.math.rand]

1 [*Note 1*: The header `<cstdlib>` declares the functions described in this subclause. — *end note*]

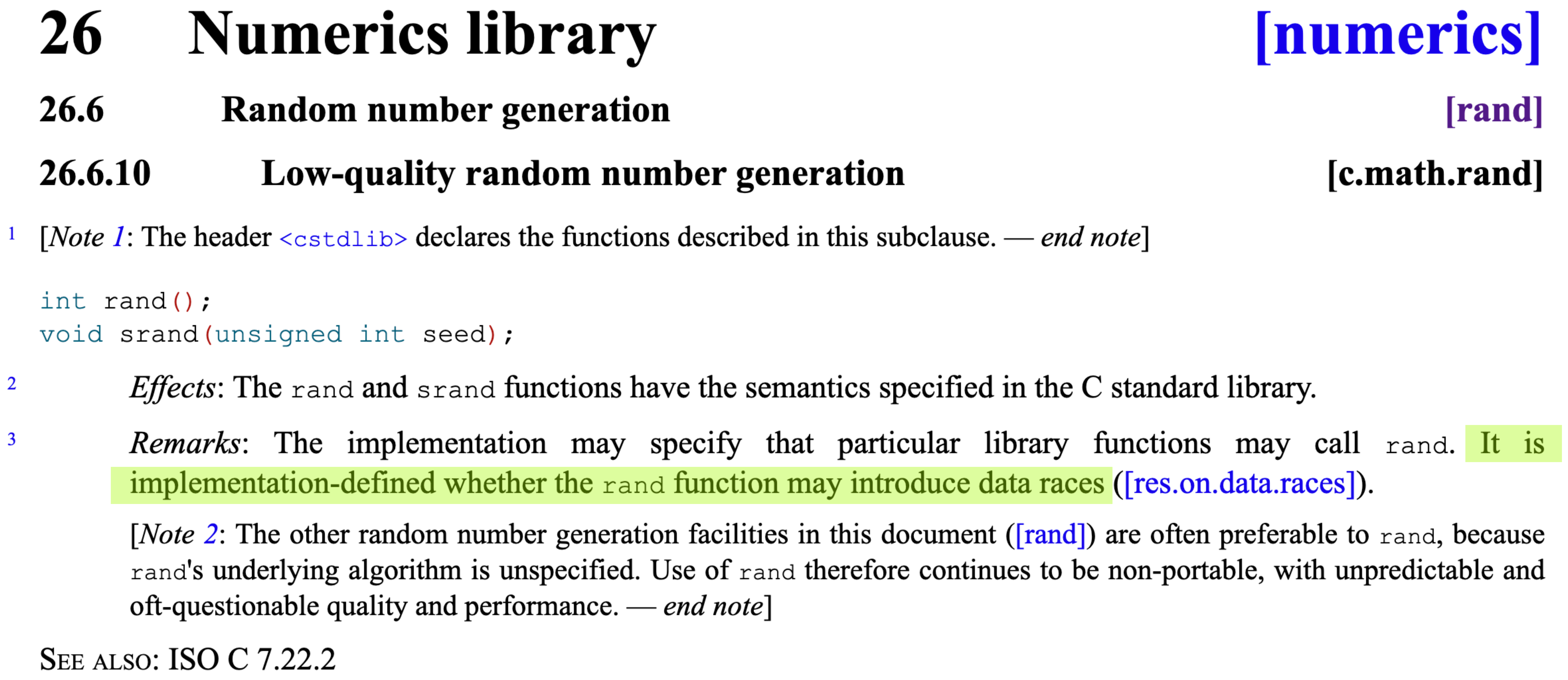
```
int rand();
void srand(unsigned int seed);
```

2 *Effects*: The `rand` and `srand` functions have the semantics specified in the C standard library.

3 *Remarks*: The implementation may specify that particular library functions may call `rand`. It is implementation-defined whether the `rand` function may introduce data races ([res.on.data.races]).

[*Note 2*: The other random number generation facilities in this document ([rand]) are often preferable to `rand`, because `rand`'s underlying algorithm is unspecified. Use of `rand` therefore continues to be non-portable, with unpredictable and oft-questionable quality and performance. — *end note*]

SEE ALSO: ISO C 7.22.2



26 Numerics library [numerics]

26.6 Random number generation [rand]

26.6.10 Low-quality random number generation [c.math.rand]

1 [*Note 1*: The header `<cstdlib>` declares the functions described in this subclause. — *end note*]

```
int rand();
void srand(unsigned int seed);
```

2 *Effects*: The `rand` and `srand` functions have the semantics specified in the C standard library.

3 *Remarks*: The implementation may specify that particular library functions may call `rand`. It is implementation-defined whether the `rand` function may introduce data races ([res.on.data.races]).

[*Note 2*: The other random number generation facilities in this document ([rand]) are often preferable to `rand`, because `rand`'s underlying algorithm is unspecified. Use of `rand` therefore continues to be non-portable, with unpredictable and oft-questionable quality and performance. — *end note*]

SEE ALSO: ISO C 7.22.2

```
// C++ random number generators:

mersenne_twister_engine
linear_congruential_engine
subtract_with_carry_engine
```

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

1 A *uniform random bit generator* `g` of type `G` is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned.

[*Note 1*: The degree to which `g`'s results approximate the ideal is often determined statistically. — *end note*]

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

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

(2.1) — `G::min() <= g()`,

(2.2) — `g() <= G::max()`, and

(2.3) — `g()` has amortized constant complexity.

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

Timur Doumler 🇫🇮 @timur_audio · Jan 7, 2020

Question: In C++, how do I generate random numbers in a context that requires the code to finish in a deterministic amount of time (realtime audio callback)?

All the random number engines in std:: are only *amortised* constant time according to the standard, so they're out 😕

Corentin @Cor3ntin · Jan 7, 2020

I am not entirely sure but I think it's still deterministic if not constant. mersen twister has an internal state of fixed known size N and every N numbers that state is recomputed, so it's O(N) then O(1) N-1 times, then O(1) again. I believe N is rather small (32?).

Peter Bindels 💉⏳ @dascandy42 · Jan 7, 2020

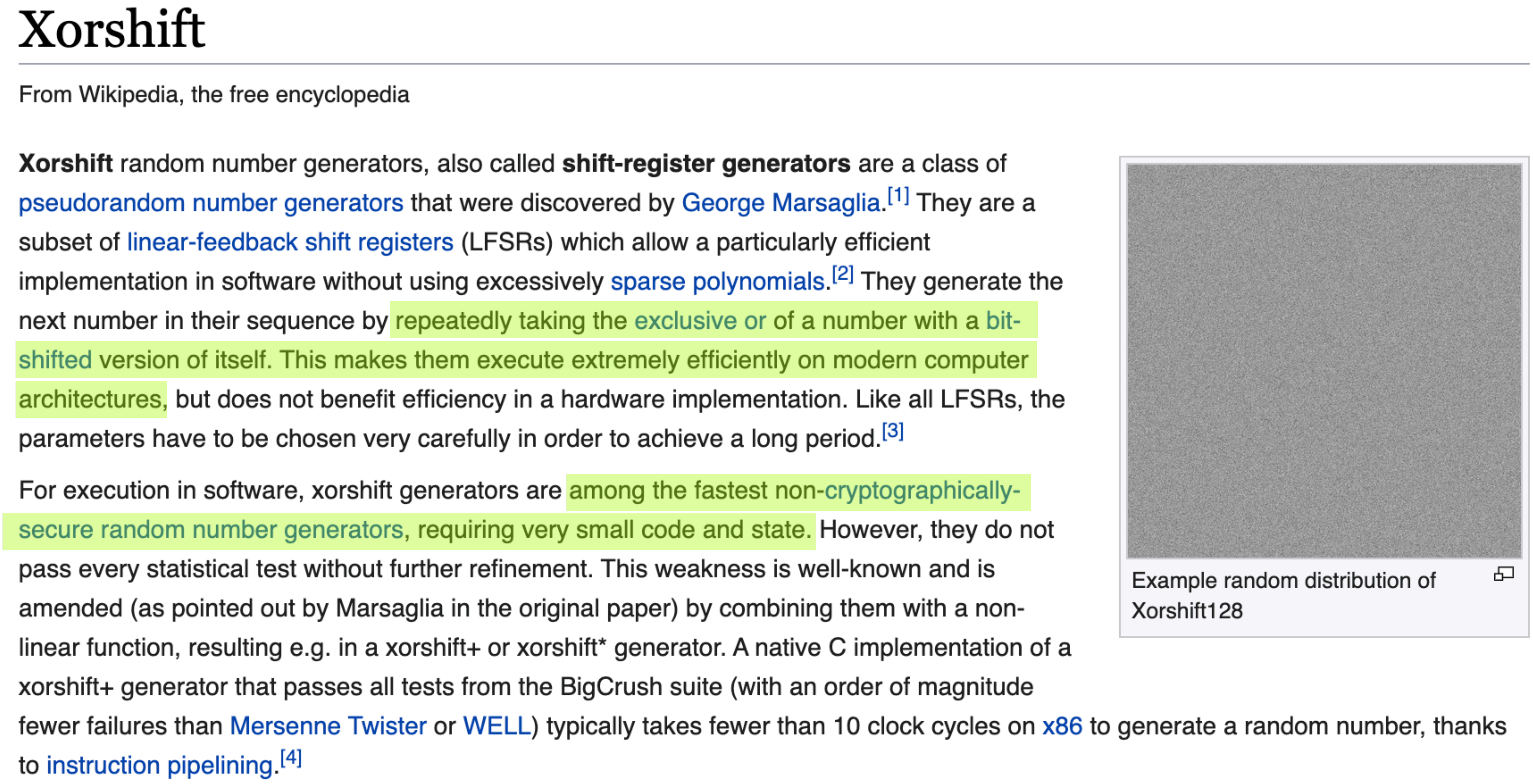
N is 624.

Corentin @Cor3ntin · Jan 7, 2020

I can see that being a problem on a toaster.

26.6.4.2 Class template `linear_congruential_engine` [rand.eng.lcong]

1 A `linear_congruential_engine` random number engine produces unsigned integer random numbers. The state $\mathtt{x}_i$ of a `linear_congruential_engine` object `x` is of size 1 and consists of a single integer. The transition algorithm is a modular linear function of the form $\mathsf{TA}(\mathtt{x}_i) = (a \cdot \mathtt{x}_i + c) \bmod m$; the generation algorithm is $\mathsf{GA}(\mathtt{x}_i) = \mathtt{x}_{i+1}$.



Xorshift

From Wikipedia, the free encyclopedia

Xorshift random number generators, also called **shift-register generators** are a class of pseudorandom number generators that were discovered by George Marsaglia.[1] They are a subset of linear-feedback shift registers (LFSRs) which allow a particularly efficient implementation in software without using excessively sparse polynomials.[2] They generate the next number in their sequence by repeatedly taking the exclusive or of a number with a bit-shifted version of itself. This makes them execute extremely efficiently on modern computer architectures, but does not benefit efficiency in a hardware implementation. Like all LFSRs, the parameters have to be chosen very carefully in order to achieve a long period.[3]

For execution in software, xorshift generators are among the fastest non-cryptographically-secure random number generators, requiring very small code and state. However, they do not pass every statistical test without further refinement. This weakness is well-known and is amended (as pointed out by Marsaglia in the original paper) by combining them with a non-linear function, resulting e.g. in a xorshift+ or xorshift* generator. A native C implementation of a xorshift+ generator that passes all tests from the BigCrush suite (with an order of magnitude fewer failures than Mersenne Twister or WELL) typically takes fewer than 10 clock cycles on x86 to generate a random number, thanks to instruction pipelining.[4]

Example random distribution of Xorshift128

```
struct random_sample_gen
{
    // returns a random float in the interval [0, 1)
    float operator()() { return distr(rng); }

private:
    xorshift_rand rng { std::random_device{}() };
    std::uniform_real_distribution<float> distr { 0, 1.0f };
};

void process(buffer& b)
{
    std::ranges::fill(b, random_sample_gen{});
}
```

```
struct random_sample_gen
{
    // returns a random float in the interval [0, 1)
    float operator()() { return distr(rng); }

private:
    xorshift_rand rng { std::random_device{}() };
    std::uniform_real_distribution<float> distr { 0, 1.0f };
};

void process(buffer& b)
{
    std::ranges::fill(b, random_sample_gen{});
}
```

26 Numerics library [numerics]

26.6 Random number generation [rand]

26.6.9 Random number distribution class templates [rand.dist]

26.6.9.1 In general [rand.dist.general]

1 Each type instantiated from a class template specified in this subclause [rand.dist] meets the requirements of a random number distribution type.

2 Descriptions are provided in this subclause [rand.dist] only for distribution operations that are not described in [rand.req.dist] or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

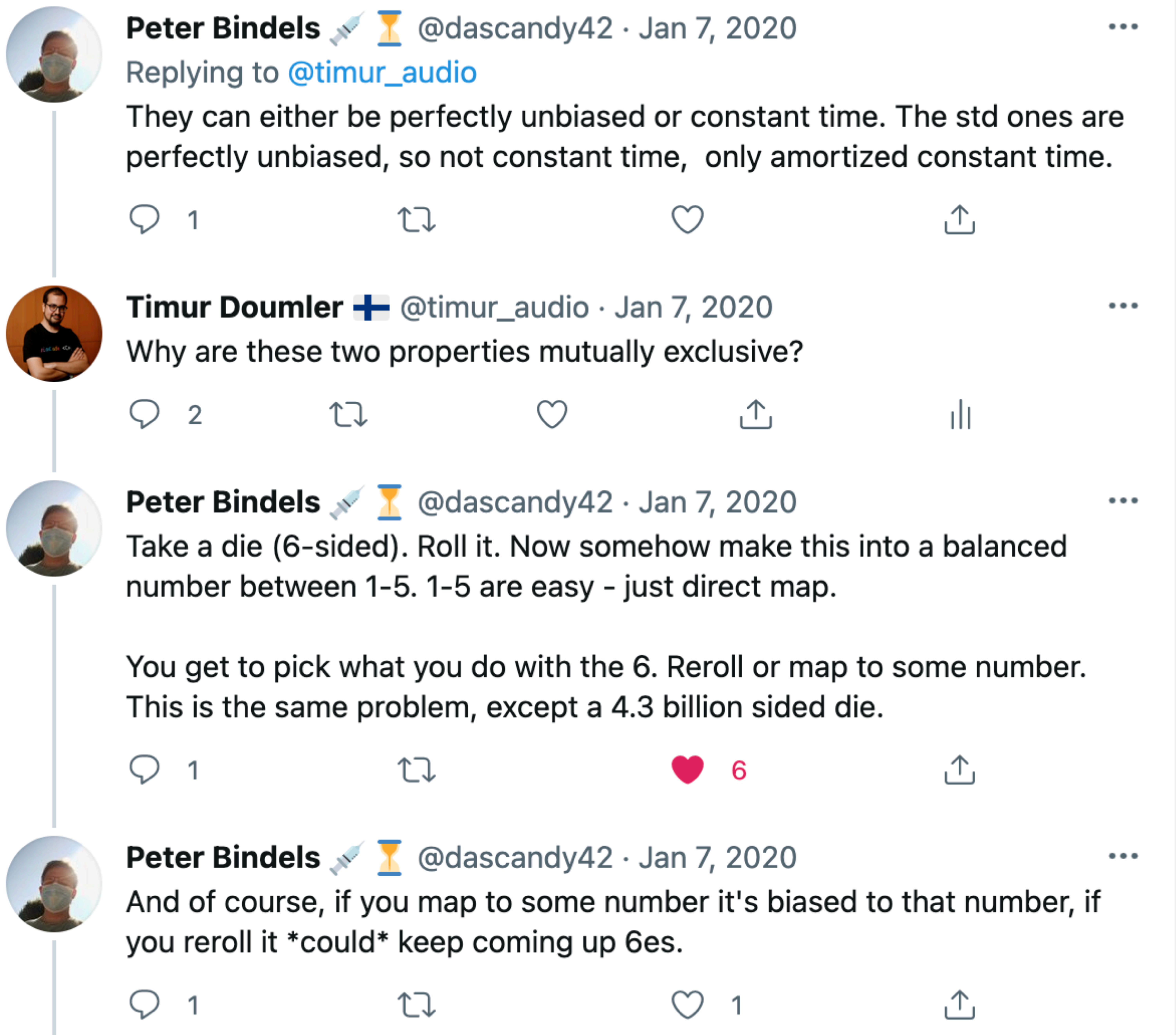
3 The algorithms for producing each of the specified distributions are implementation-defined.

4 The value of each probability density function $p(z)$ and of each discrete probability function $P(z_i)$ specified in this subclause is 0 everywhere outside its stated domain.

The `std::uniform_*_distributions` have

amortised O(1) complexity.

They can discard the generated random number and generate another one.



Peter Bindels 💉⌛ @dascandy42 · Jan 7, 2020

Replying to @timur_audio

They can either be perfectly unbiased or constant time. The std ones are perfectly unbiased, so not constant time, only amortized constant time.

Timur Doumler 🇫🇮 @timur_audio · Jan 7, 2020

Why are these two properties mutually exclusive?

Peter Bindels 💉⌛ @dascandy42 · Jan 7, 2020

Take a die (6-sided). Roll it. Now somehow make this into a balanced number between 1-5. 1-5 are easy - just direct map.

You get to pick what you do with the 6. Reroll or map to some number. This is the same problem, except a 4.3 billion sided die.

Peter Bindels 💉⌛ @dascandy42 · Jan 7, 2020

And of course, if you map to some number it's biased to that number, if you reroll it *could* keep coming up 6es.

```
struct random_sample_gen
{
    // returns a random float in the interval [0, 1)
    float operator()()
    {
        auto x = float (rng() - rng.min()) / float (rng.max() + 1);
        return x;
    }

private:
    xorshift_rand rng { std::random_device{}() };
};

void process(buffer& b)
{
    std::ranges::fill(b, random_sample_gen{});
}
```

```
struct random_sample_gen
{
    // returns a random float in the interval [0, 1)
    float operator()()
    {
        auto x = float (rng() - rng.min()) / float (rng.max() + 1);
        if (x == 1.0f) x -= std::numeric_limits<float>::epsilon();
        return x;
    }

private:
    xorshift_rand rng { std::random_device{}() };
};

void process(buffer& b)
{
    std::ranges::fill(b, random_sample_gen{});
}
```

`[[realtime_safe]]`

Special thanks to:

Fabian Renn-Giles

Peter Bindels

Mattias Jansson

David Stone

Pablo Halpern

Thank you!

Timur Doumler

🐦 @timur_audio

Pulsar PSR B1509−58
Image credit: NASA/CXC/CfA/P