Random Number Generation
in Phobos and beyond

Joseph Wakeling

(WebDrake)

“

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin

”

(John von Neumann, 1946)

The (pseudo-random) essentials

Random number generators are at heart defined by a few simple elements:

- a state variable s with initial value s_0
- a (pure) *generation algorithm* mapping state to the corresponding value (*variate*)
- a (pure?) *transition algorithm* S mapping each state to the next

$$s_{k+1} = S(s_k)$$

```
struct MyRNG
{
    private MyState state;

    uint front () { … }

    void popFront () { … }
}
```

A Phobos RNG range

```
struct LinearCongruentialGenerator(UIntType, UIntType a, UIntType b, UIntType m)
{
  private:
    UIntType x;    // the state variable

  public:
    enum bool empty = false;   // an RNG never runs out

    UIntType front() pure @property
    {
        return x;   // in this case, the mapping state => variate is very simple
    }


    void popFront() pure
    {
        x = (a * x + c) % m   // the transition function
    }


    typeof(this) save() @property   // only possible with pseudo-RNGs
    {
        return this;
    }
}
```

D in action

```
import std.random, std.stdio;

void main()
{
    Mt19937 gen;    // uniform random number generator

    gen.seed(unpredictableSeed());  // seed non-deterministically

    // generate uniformly-distributed variates

    foreach (i; 0 .. 10) {
        writeln(uniform!"[]"(1, 6, gen));
    }

    // note that uniform() does not provide a range!
}
```

An alternative PoV: the C++11 standard

- Distinguishes between random *engines*, random *devices*, and random *distributions*
- Random *engines* are sources of uniformly-distributed pseudo-random bits
- Random *devices* are sources of uniformly-distributed non-deterministic random bits
- Random *distributions* map uniformly-distributed random bits to other types (integers, floating-point, …) such that the resulting values follow a specified probability distribution

An alternative PoV: the C++11 standard

- whether engine or device, C++11 RNGs are function objects (i.e. defining `operator()`) returning unsigned integer values
  - i.e. one function both returns the generated value and "pops" the generator
  - contrast with D's separate `front()` and `popFront()`
- C++11 random distributions are function objects whose `operator()` method accepts an RNG parameter passed by ref
  - contrast with D's `uniform` which is simply a function

C++11 in action

```
#include <functional>
#include <iostream>
#include <random>

int main()
{
    std::random_device rd;  // random device used for seeding
    std::mt19937 engine;    // generator of pseudo-random bits

    engine.seed(rd());      // seed from random device

    std::uniform_int_distribution<int> distribution(1, 6);   // random distribution

    // use-case 1: call distribution directly, passing engine by ref

    for (int i = 0; i < 10; ++i) {
        std::cout << distribution(engine) << std::endl;
    }

    // use-case 2: bind distribution and engine together

    auto six_sided_die = std::bind(distribution, engine);

    for (int i = 0; i < 10; ++i) {
        std::cout << six_sided_die() << std::endl;
    }
}
```

C++11 vs D functionality

- C++11 `<random>`
  - wide range of random number engines, implemented as function objects
  - `random_device` class for non-deterministic random bits
  - wide selection of random distributions (uniform, exponential, normal, ...)
  - random adaptors (random number engines that transform the output of other, “base” engines)
- D `std.random`
  - good selection of random number engines (different from C++11), implemented as forward ranges
    - “thread-global” default RNG instance `rndGen`
  - `unpredictableSeed`
  - no random distributions apart from `uniform` and `uniform01`
  - `randomCover`, `randomSample`, `randomShuffle`

RNGs and range dynamics

… this is where it starts to go wrong :-(

Wrapping RNGs causes problems

```
import std.random, std.range, std.stdio;

void main ()
{
    Mt19937 gen;
    gen.seed(unpredictableSeed());

    gen.take(10).writeln;  // these two uses of the RNG
    gen.take(10).writeln;  // both produce the same result!

    iota(10).randomCover(gen).writeln;   // so do these two:
    iota(10).randomCover(gen).writeln;   // every time.

    // but these two produce different results
    iota(10).map!(a => uniform(0.0, 1.0, gen)).writeln;
    iota(10).map!(a => uniform(0.0, 1.0, gen)).writeln;
}
```

Similar inconsistencies in C++11

```
#include <functional>
#include <iostream>
#include <random>

int main ()
{
    std::random_device rd;
    std::mt19937 engine;
    engine.seed(rd());

    std::uniform_real_distribution<double> dist1(0.0, 1.0);
    std::uniform_real_distribution<double> dist2(0.0, 1.0);

    // these two loops produce different results
    for (int i = 0; i < 10; ++i) {
        std::cout << dist1(engine) << "\t" << dist2(engine) << std::endl;
    }

    // the two different bindings produce identical results
    auto gen1 = std::bind(dist1, engine);
    auto gen2 = std::bind(dist2, engine);

    for (int i = 0; i < 10; ++i) {
        std::cout << gen1() << "\t" << gen2() << std::endl;
    }
}
```

C++11 vs D problems

- std::bind results in a copy-by-value …
  - … but because C++11 works with function objects, we can always pass an RNG by reference
- with ranges that take another range input, we are always going to get "bind"-like effects

```
// typical phobos range handling
struct Consumes(Source)
    if (isInputRange!Source)
{
    private Source source_;

    this(Source source)
    {
        this.source_ = source;
    }
}
```

A workaround — always freshly seed?

```
import std.random, std.range, std.stdio;

void main()
{
    // These two calls produce different results
    iota(100).randomSample(10, Random(unpredictableSeed)).writeln;
    iota(100).randomSample(10, Random(unpredictableSeed)).writeln;
}

/* The above solution “works”, and reflects C++11 recommendations
 * when using std::bind, but I don’t like it:
 *
 *     * it relies on programmer virtue
 *
 *     * it’s annoyingly verbose
 *
 *     * interferes with reproducibility of program results
 *       (OK, OK, there are ways round this).
 */
```

`rndGen` to the rescue?

```
/**
 * Thread-global (i.e. global and thread-local) singleton
 * instance of default pseudo-random number generator
 */
@property ref Random rndGen() @safe
{
    import std.algorithm : map;
    import std.range : repeat;

    static Random result;
    static bool initialized;
    if (!initialized) {
        // (missing out one more complex seeding option)
        result = Random(unpredictableSeed);
        initialized = true;
    }
    return result;
}
```

rndGen to the rescue?

```
struct RandomCover (Range, UniformRNG = void) {
    private Range _input;
    private size_t _current, _alreadyChosen = 0;

    static if (is(UniformRNG == void)) {
        this(Range input) {
            _input = input;
            // no RNG copied internally in this case
        }
    } else {
        private UniformRNG _rng;

        this(Range input, ref UniformRNG rng) {
            _input = input;
            _rng = rng;   // if UniformRNG is a struct, copies by value
            // etc.
        }
    }

    // ... to be continued ...
}
```

rndGen to the rescue?

```
struct RandomCover (Range, UniformRNG = void) {
    // ... continuing ...

    void popFront() {
        // ... missing a bunch of details ...
        size_t k = _input.length - _alreadyChosen;
        foreach (e; _input) {
            static if (is(UniformRNG == void)) {
                // uses rndGen
                auto chooseMe = uniform(0, k) == 0;
            } else {
                // uses copied RNG instance
                auto chooseMe = uniform(0, k, _rng) == 0;
            }
        }
        // etc.
    }
}
```

rndGen to the rescue?

```
import std.random, std.range, std.stdio;

void main()
{
    auto rng = Random(unpredictableSeed);

    // these two calls produce the same results
    iota(10).array.randomCover(rng).writeln;
    iota(10).array.randomCover(rng).writeln;

    // these two calls produce different results
    iota(10).array.randomCover.writeln;
    iota(10).array.randomCover.writeln;
}
```

A slightly more generic static RNG

```
struct StaticRNG(UniformRNG)
    if (isUniformRNG!UniformRNG)
{
  private:
    static UniformRNG rng_;

  public:
    enum isUniformRandom = UniformRNG.isUniformRandom;

    auto min() @property { return rng_.min; }

    auto max() @property { return rng_.max; }

    bool empty() @property { return rng_.empty; }

    auto front() @property { return rng_.front; }

    void popFront() { rng_.popFront(); }

    static if (isSeedable!UniformRNG)
    {
        void seed(Seed)(Seed s) { rng_.seed(s); }
    }
}
```

Reference-type RNGs

- Use `RefRange` or `RefCounted` RNG instances
  - *problem:* currently `isUniformRNG` fails for `RefRange!Random`, `RefCounted!Random` etc. *(probaby easy to fix)*
  - relies on user virtue (i.e. knowing to use `RefRange` or `RefCounted`, and why)
- Implement RNGs as (final) classes (`hap.random`)
  - easy reference type semantics
  - also simplifies other RNG-related functionality like `randomCover`, `randomSample`
  - *problem:* by default on the heap; creates potential allocation/GC issues
    - more of an issue for `random{Cover,Sample}` and future random distributions
  - *problem:* un-idiomatic for Phobos?
- Implement as structs, but have reference type internal state
  - annoying to implement
  - still have potential allocation issues
- … but solving the copy-by-value problem is not sufficient :-(

Problematic function assumptions

```
auto doSomething(Range)(auto ref Range r)
    if (isForwardRange!Range)
{
    auto rcopy = r.save;

    // do stuff with rcopy, because hey, it couldn't
    // be bad to not consume the original range, right?
}

// REAL example:
auto rng = new ClassBasedRNG(unpredictableSeed);
cartesianProduct(rng.take(2), iota(2)).writeln;
cartesianProduct(rng.take(2), iota(2)).writeln;
// produces identical output both times

// cartesianProduct used to .save only the first
// of its arguments, now saves both
```

Forward or Input? Or … ?

- Currently all `std.random` pseudo-RNGs are implemented as forward ranges
  - a natural assumption for any deterministic sequence where current state can be saved?
  - trouble is, even a pseudo-RNG is supposed to seem non-deterministic to its callers
  - Phobos functions make strong assumptions about deterministic meaning of forward ranges
- Alternative: `InputRange` with different method (`.dup`?) for explicit copying?
  - in truth, pseudo-RNGs — equivalent to random number engines in C++11 — sit in between Input and Forward ranges

The key issues

- We have a great collection of RNG algorithms, but …
- Value-type and/or forward range RNGs create far too many circumstances where RNGs get copied without meaning to
  - risk of far too many unintentional correlations
- Relying on programmer virtue to know how to work around these issues is not a viable long-term solution
- Already, handling these issues often leads to finnicky workarounds (e.g. special treatment of `rndGen`)
- We need RNG functionality where *the easy and obvious thing to do* is also *the statistically correct thing to do*

Where (I think) we should be going

- Reference-type, input-range RNGs
  - with `.dup` for random engines
  - lots of nice functionality becomes much easier to implement (`randomCover`, `randomSample`, random distributions...)
  - the challenge here is managing allocation and stack vs. heap issues
- Clearer definitions & separations between different aspects of random number generation
  - range-based equivalents to C++11's *engines*, *devices* and *distributions*
  - *distributions* in particular are sorely missed
- For some (incomplete!) sketches in the above directions, take a look at `hap.random`: https://github.com/WebDrake/hap

Thanks for listening!

Questions, observations, ideas?

Oh, and — sociomantic is hiring!

www.sociomantic.com/careers