

The `tryLog` function does a best-effort attempt at logging a message. If an exception is thrown, it is silently ignored. This makes `tryLog` usable in critical sections of code. In some circumstances it would be silly for some important transaction to fail just because a log message couldn't be written. Code with transactional semantics relies critically on certain portions never throwing, and `nothrow` is the way you can ensure such facts statically.

The semantic checking of `nothrow` functions ensures an exception may never leak out of the function. That means that any statement in that function is inside a `try` statement that swallows the exception, or can't throw (e.g., is a call to another `nothrow` function). To illustrate the latter case:

```
nothrow void sensitive(Widget w) {
    tryLog("Starting sensitive operation");
    try {
        w.mayThrow();
        tryLog("Sensitive operation succeeded");
    } catch (Exception) {
        tryLog("Sensitive operation failed");
    }
}
```

The first call to `tryLog` needn't be inside a `try` statement because the compiler already knows it can't throw. Similarly, the call inside the `catch` clause does not need to be protected by an extra `try` statement.

What is the relationship between pure and `nothrow`? It might appear that they are entirely orthogonal, but there may be a certain degree of correlation. At least in the standard library, many mathematical functions are both pure and `nothrow`, for example most transcendental functions (`exp`, `sin`, `cos` etc.).

4.12 Compile-Time Evaluation

In keeping with the saying that good things come to those who wait (or read patiently), this last section discusses a very interesting feature of D. The best part is, you don't even need to learn most anything to use this feature gainfully.

Let's use an example that's large enough to be meaningful. Suppose you want to define the ultimate random number generator library. There are many random number generators out there, among which the fast and well-studied linear congruential generators [11, § 3.2.1, pp. 10–26]. Such generators have three integral parameters: the modulus $m > 0$, the multiplier $0 < a < m$, and the increment³ $0 < c < m$. Starting from an arbitrary seed $0 \leq x_0 < m$, the linear congruential generator yields pseudorandom numbers using the following recurrence formula:

³The case $c = 0$ is also allowed, but the associated theory is more difficult so in here we require $c > 0$.

$$x_{n+1} = (ax_n + c) \bmod m$$

Coding such an algorithm is simple—all you need is to keep the state defined by m , a , c , and x_n , and define a `getNext` function that changes x_n into x_{n+1} and returns it.

But there is a rub. Not all combinations of m , a , and c lead to good random number generators. For starters, if $a = 1$ and $c = 1$ the generator gives the sequence $0, 1, \dots, m-1, 0, 1, \dots, m-1, 0, 1, \dots$ which is admittedly quite non-random.

With larger values of a and c such obvious risks are avoided, but a subtler problem appears: periodicity. Due to the modulus operator the generated number is always between 0 and $m-1$, so it's good to choose m as large as possible (usually it's a power of 2 to match the machine word size, in which case the `mod` comes for free). The problem is that the generated sequence may have a period much smaller than m . Say we operate with `uint` and choose $m = 2^{32}$ so we don't even need a modulus operation, then $a = 210$, $c = 123$, and some crazy value for x_0 , such as $1,780,588,661$. Let's run this program:

```
import std.stdio;

void main() {
    enum uint a = 210, c = 123, x0 = 1_780_588_661;
    auto x = x0;
    foreach (i; 0 .. 100) {
        x = a * x + c;
        writeln(x);
    }
}
```

Instead of a colorful random display of digits, we see something rather surprising:

```
1 261464181
2 3367870581
3 2878185589
4 3123552373
5 3110969461
6 468557941
7 3907887221
8 317562997
9 2263720053
10 2934808693
11 2129502325
12 518889589
13 1592631413
```

```
14 3740115061
15 3740115061
16 3740115061
17 ...
```

The generator starts with great aplomb. At least to the untrained eye, it does a good job at generating random numbers. But it doesn't take more than 14 steps to stall in a fixed point: through one of those strange coincidences that only math is capable of, 3740115061 is (and was and will be) exactly equal to $(3740115061 * 210 + 123) \bmod 2^{32}$. That's a period of one, the worst possible!

So we need to make sure that m , a , and c are chosen such that the generator has a large period. Investigating the matter further, it turns out that the conditions for generating a sequence of period m (the largest period possible) are the following:

1. c and m are relatively prime;
2. $a - 1$ is divisible by all prime factors of m ; and
3. if $a - 1$ is multiple of 4, then m is a multiple of 4 too.

The relative primality of c and m can be easily checked by comparing their greatest common divisor against 1. To compute the greatest common divisor, we use Euclid's algorithm:⁴

```
// Implementation of Euclid's algorithm
ulong gcd(ulong a, ulong b) {
    while (b) {
        auto t = b;
        b = a % b;
        a = t;
    }
    return a;
}
```

Euclid expressed his algorithm by using subtraction instead of modulus. The modulus version takes fewer iterations, but on today's machines, '%' can be quite slow, something that Euclid might have had in mind.

The second test is a tad more difficult to implement. We could write a function `factorize` that returns all prime factors of a number with their powers and then use it, but `factorize` is more than the bare necessity. Going with the simplest design that could possibly work, probably a simple choice is to write a function `primeFactorsOnly(n)` that returns the product of n 's prime factors, but without the

⁴Somehow, Euclid's algorithm always manages to make its way into good (ahem) programming books.

powers. Then the requirement boils down to checking $(a - 1) \% \text{primeFactorsOnly}(m) == 0$. So let's implement `primeFactorsOnly`.

There are many ways to go about getting the prime factors of some number n . A simple one would be to generate prime numbers p_1, p_2, p_3, \dots and check in turn whether n divides p_k , in which case p_k is multiplied to an accumulator r . When p_k has become greater than n , the accumulator contains the desired answer: the product of all of n 's prime factors, each taken once.

(I know you are asking yourself what this has to do with compile-time evaluation. It does. Please bear with me.)

A simpler version would be to do away with generating prime numbers and simply evaluate $n \bmod k$ for increasing values of k starting at 2: 2, 3, 5, 7, 9, ... Whenever k divides n , the accumulator is multiplied by k and then n is "depleted" of all powers of k , i.e. n is reassigned n/k for as long as k divides n . That way, we recorded the divisor k and also we reduced n until it became irreducible by k . That seems like a wasteful method, but note that generating prime numbers would probably entail comparable work, at least in a straightforward implementation. An implementation of this idea would look like this:

```
ulong primeFactorsOnly(ulong n) {
    ulong accum = 1;
    ulong iter = 2;
    for (; n >= iter * iter; iter += 2 - (iter == 2)) {
        if (n % iter) continue;
        accum *= iter;
        do n /= iter; while (n % iter == 0);
    }
    return accum * n;
}
```

The update `iter += 2 - (iter == 2)` bumps `iter` by 2 except when `iter` is 2, in which case the update brings `iter` to 3. That way `iter` spans 2, 3, 5, 7, 9 and so on. It would be wasteful to check any even number for example 4 because 2 has been tested already and all powers of 2 have been extracted out of n .

Why does the iteration go on while $n \geq \text{iter} * \text{iter}$ as opposed to $n \geq \text{iter}$? The answer is a bit subtle. If `iter` is greater than \sqrt{n} then we can be sure `iter` can't be a divisor of n : if it were, it would need to have some multiplier k such that $n == k * \text{iter}$, but all divisors smaller than `iter` have been tried already, so k must be greater than `iter` and consequently $k * \text{iter}$ is greater than n , which makes the equality impossible.

Let's unittest the `primeFactorsOnly` function:

```
unittest {
    assert(primeFactorsOnly(100) == 10);
    assert(primeFactorsOnly(11) == 11);
}
```

```

assert(primeFactorsOnly(7 * 7 * 11 * 11 * 15) == 7 * 11 * 15);
assert(primeFactorsOnly(129 * 2) == 129 * 2);
}

```

To conclude, we need a small wrapper that performs the three checks against three candidate linear congruential generator parameters:

```

bool properLinearCongruentialParameters(ulong m, ulong a, ulong c) {
    // Bounds checking
    if (m == 0 || a == 0 || a >= m || c == 0 || c >= m) return false;
    // c and m are relatively prime
    if (gcd(c, m) != 1) return false;
    // a - 1 is divisible by all prime factors of m
    if ((a - 1) % primeFactorsOnly(m)) return false;
    // if a - 1 is multiple of 4, then m is a multiple of 4 too.
    if ((a - 1) % 4 == 0 && m % 4) return false;
    // Passed all tests
    return true;
}

```

Let's unittest a few popular values of m, a, and c:

```

unittest {
    // Our broken example
    assert(!properLinearCongruentialParameters(
        1UL << 32, 210, 123));
    // Numerical Recipes book [18]
    assert(properLinearCongruentialParameters(
        1UL << 32, 1664525, 1013904223));
    // Borland C/C++ compiler
    assert(properLinearCongruentialParameters(
        1UL << 32, 22695477, 1));
    // glibc
    assert(properLinearCongruentialParameters(
        1UL << 32, 1103515245, 12345));
    // ANSI C
    assert(properLinearCongruentialParameters(
        1UL << 32, 134775813, 1));
    // Microsoft Visual C/C++
    assert(properLinearCongruentialParameters(
        1UL << 32, 214013, 2531011));
}

```

It looks like `properLinearCongruentialParameters` works like a charm, so we're done with all the details of testing the soundness of a linear congruential generator. It's about time to stop stalling and fess up. What does all that primality and factorization stuff have to do with compile-time evaluation? Where's the meat? Where are the templates, macros, or whatever they call them? The clever `static ifs`? The mind-blowing code generation and expansion?

Well, here's the truth: you just saw everything there is to be seen. Given any constant numbers `m`, `a`, and `c`, you can evaluate `properLinearCongruentialParameters` *during compilation* without any change in that function or the function it calls. The D compiler embeds an interpreter that evaluates D functions during compilation—with arithmetic, loops, mutation, early returns, even arrays and transcendental functions.

All you need to do is clarify to the compiler that the evaluation must be performed at compile time. There are several ways to do that:

```
unittest {
    enum ulong m = 1UL << 32, a = 1664525, c = 1013904223;
    // Method 1: use static assert
    static assert(properLinearCongruentialParameters(m, a, c));
    // Method 2: assign the result to an enum
    enum proper1 = properLinearCongruentialParameters(m, a, c);
    // Method 3: assign the result to a static value
    static proper2 = properLinearCongruentialParameters(m, a, c);
}
```

We haven't looked closely at structs and classes yet, but just to anticipate a little, the typical way you'd use `properLinearCongruentialParameters` would be inside a struct or a class that defines a linear congruential generator, for example:

```
struct LinearCongruentialEngine(UIntType,
    UIntType a, UIntType c, UIntType m) {
    static assert(properLinearCongruentialParameters(m, a, c),
        "Incorrect instantiation of LinearCongruentialEngine");
    ...
}
```

In fact, the lines above were copied from the eponymous struct found in the standard module `std.random`.

There are two interesting consequences of moving the test from runtime to compile time. First, `LinearCongruentialEngine` could have deferred the test to runtime, for example by placing it in its constructor. As a general rule however, it is better to discover errors sooner rather than later, particularly in a library that has little control over how it is being used. The static test does not make erroneous instantiations of `LinearCongruentialEngine` signal the error; it makes them nonexistent. Second, the

code using compile-time constants has a good chance to be faster than code that uses regular variables for `m`, `a`, and `b`. On most of today's processors, literal constants can be made part of the instruction stream so loading them causes no extra memory access at all. And let's face it—linear congruential engines aren't the most random out there, so the primary reason you'd have to use one is speed.

The interpretation process is slower than generating code by a couple of orders of magnitude, but that is already much faster and more scalable than traditional metaprogramming carried with C++ templates. Besides, within reason, compile-time activity is in a way “free.”

At the time of this writing, the interpreter has certain limitations. Allocating class objects and memory in general are not allowed (though built-in arrays work). Static data, inline assembler, and unsafe features such as unions and certain casts are also disallowed. But the limits of what can be done during interpretation form an envelope under continuous pressure. The plan is to allow everything in the safe subset of D interpretable during compilation. All in all, the ability to interpret code during compilation is recent, and opens very exciting possibilities that deserve further exploration.