NEWS · DECEMBER 29, 2025

Random Numbers in Delphi: When the RTL is Good Enough (And When It Isn't)

By Dr. Holger Flick

I recently came across several Delphi implementations in which developers used their own random number generators, apparently unaware of—or unconvinced by—the built-in `Randomize` and `Random` functions that have been part of the RTL since Delphi's earliest days. Some developers seem to believe they can do better than the runtime library, while others might simply not know what's already available.

Let's take a fair look at Delphi's built-in random number generation, understand when it's perfectly adequate, and identify the rare cases where you might actually need something more sophisticated.

The Basics: Randomize and Random

Delphi provides two fundamental functions for random number generation:

```
procedure Randomize;
function Random: Extended; overload;
function Random(Range: Integer): Integer; overload;
```

The `Randomize` procedure seeds the random number generator using the system clock. Without calling it, you'll get the same sequence of "random" numbers every time your application runs—useful for debugging, but not what you want in production.

The `Random` function comes in two flavors: parameterless (returning a floating-point value between 0 and 1) and with an integer parameter (returning an integer from 0 to Range-1).

Here's the typical usage pattern:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;  // Seed once at application startup
end;

procedure TForm1.GenerateRandomData;
var
  RandomPercent: Double;
  RandomIndex: Integer;
begin
  RandomPercent := Random;  // 0.0 to 0.999...
  RandomIndex := Random(100);  // 0 to 99
end;
```

What's Under the Hood?

Delphi's `Random` implementation uses a Linear Congruential Generator (LCG), which is a well-understood, fast algorithm. The quality of LCGs depends on their parameters, and Delphi uses constants that have been proven to work reasonably well for general-purpose applications.

The algorithm is simple: each new random number is calculated from the previous one using multiplication, addition, and modulo operations. This makes it:

- **Fast**: Generating millions of random numbers per second is no problem
- **Deterministic**: Given the same seed, you get the same sequence (important for testing and debugging)
- **Predictable**: Not cryptographically secure, but that's rarely needed

When Delphi's Random is Perfectly Fine

For the vast majority of applications, `Randomize` and `Random` are more than adequate. Here are common scenarios where you should absolutely use the built-in functions:

Game Development: Shuffling cards, rolling dice, generating random enemy spawn points, random loot drops, procedural terrain generation for casual games—all of these work perfectly fine with Delphi's Random.

UI and UX: Randomly selecting tips to display, varying animation timings slightly to feel more natural, choosing random colors for data visualization.

Testing and Simulation: Generating test data, simulating user behavior in load tests, creating random delays in stress tests.

Business Applications: Random sampling from datasets for quality control, generating unique-looking invoice numbers (combined with other data), selecting random records for audit purposes.

Data Visualization: Scatter plot jitter to prevent overlapping points, random color selection for chart elements.

I've used Delphi's Random in production applications for decades across all these scenarios, and it has never been the source of a problem. The statistical properties are good enough that patterns won't emerge in ways that affect your application's behavior or user experience.

The Statistical Reality

Let me address the elephant in the room: yes, Delphi's LCG-based Random has some statistical weaknesses if you subject it to rigorous randomness tests like the Diehard tests or TestU01. You might find correlations between consecutive values under certain conditions, and the period (before the sequence repeats) is limited.

But here's the crucial question: **Does this matter for your application?**

If you're shuffling a deck of 52 cards, the fact that Delphi's Random might fail some esoteric statistical test is completely irrelevant. If you're generating random coordinates for 1,000 game objects on a 1920×1080 screen, the statistical imperfections won't create visible patterns.

The people who need to worry about these statistical properties are researchers running Monte Carlo simulations with millions of iterations, or developers working on applications where money or security depends on randomness. That's not most of us.

When You Should Consider Alternatives

There are legitimate cases where Delphi's built-in Random isn't appropriate:

Cryptography and Security: Never, ever use `Random` for generating passwords, encryption keys, session tokens, or anything security-related. For these cases, use Windows Crypto API functions like `CryptGenRandom` or, in modern Delphi, the `System.Hash` unit's cryptographic functions.

```
// Don't do this for security purposes!
function GenerateBadPassword: string;
var
  i: Integer;
begin
  Result := '';
  for i := 1 to 16 do
    Result := Result + Char(Random(26) + Ord('A'));
end;

// Instead, use cryptographically secure random bytes
// and the System.Hash unit or Windows Crypto API
```

Scientific Computing: If you're running serious statistical simulations or Monte Carlo analysis where the quality of randomness directly affects your results' validity, consider specialized libraries like the Mersenne Twister implementation or other modern PRNGs designed for scientific work.

High-Stakes Gaming: Casino applications, lottery systems, or any application where money is directly tied to random outcomes should use certified random number generators with proven statistical properties and proper entropy sources.

Specific Statistical Requirements: If your application needs to pass specific statistical tests or has documented randomness requirements (say, for regulatory compliance), you'll need to verify that Delphi's Random meets those requirements or use an alternative that does.

The Custom Implementation Trap

I've seen developers implement their own random number generators using various schemes: system time milliseconds, mouse coordinates, combination of CPU tick counts, and other creative approaches. Here's the uncomfortable truth: unless you've studied randomness theory and tested your implementation extensively, your custom RNG is almost certainly worse than Delphi's built-in one.

Common mistakes in custom implementations include:

- Using only low-resolution time sources, creating predictable patterns
- Not understanding modulo bias when converting to ranges
- Creating correlations between consecutive values worse than any LCG
- Insufficient seeding leading to repeated sequences
- Platform-specific behavior that breaks cross-platform code

The Delphi RTL team didn't pick their RNG algorithm randomly (pun intended). It's based on decades of research and proven implementations. Unless you have specific, measurable requirements that it doesn't meet, trust the RTL.

Best Practices

Here are my recommendations after 30 years of Delphi development:

Always call Randomize once at application startup, typically in your main form's `OnCreate` event or in your application initialization code.

Use Random directly for all your general-purpose randomness needs—games, testing, UI variety, data generation.

Don't overthink it unless you have specific, documented requirements that Delphi's Random doesn't meet.

Know when to upgrade: If you're doing cryptography, use proper cryptographic functions. If you need specific statistical properties, document why and choose an appropriate library.

Test your assumptions: If you think you need better randomness, first prove that Delphi's Random is actually causing problems in your specific use case.

Conclusion

The Delphi RTL's `Randomize` and `Random` functions are well-designed, fast, and appropriate for the vast majority of applications. They're not perfect by academic standards, but they're far better than most custom implementations and perfectly adequate for typical software development needs.

Before you write your own random number generator, ask yourself: "Do I really need this, or am I just reinventing a wheel that already rolls smoothly?" In most cases, you'll find that the RTL has you covered.

Save your creativity for the actual problems your application solves, not for reimplementing functionality that's been working reliably for decades. And if you genuinely need something beyond what Random provides, make sure you understand exactly what requirements you're trying to meet and choose a proven library rather than rolling your own.

The smart developer isn't the one who writes the most code—it's the one who recognizes when the existing code is already good enough.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —