# Round & Round

## *How random are your numbers?*

***by Julian Bucknall***

In March 1997, I was pleased to see that a certain competing publication had an article on random numbers. 'At last,' I said (well, at least to myself, speaking out loud to oneself in a cinema queue is somewhat frowned upon), 'a chunky algorithm article showing Delphi programmers that programming is not just dropping components on forms and linking things up. Some tender algorithmic meat for that RAD gravy.'

Unfortunately, as I read on, my heart started to sink: the first part of the article introduced a random number generator that was, well, hopeless, even though based on an algorithm that should have produced a good one. To be fair, the author noticed that the generator was bad and said so, but he had no idea why, or even how to show that it was. Luckily he discarded it for the second part of the article where he used the standard Delphi `Random` function instead to talk about different types of random number distribution (and this part was worth reading). So, what *was* wrong with the generator he proposed? In this article we'll have a look at random numbers, at what makes them random, write a random number generator, and point out the flaw in this author's routine (from now on we'll be calling it *Algorithm K*). In doing so, we'll lay the foundation for an appreciation of the difficulties of random numbers and their generation.

## Confusion

The best place to start with any discussion on random numbers is at the beginning. What is a random number? To quote Don Knuth: 'In a sense, there is no such thing as a random number; for example, is 2 a random number?' (*The Art of Computer Programming*, Volume 2, 3rd Edition, see the boxout). Apart from the joke, he raises a serious point: when we talk about random numbers, we generally don't mean individual ones, we mean a sequence of numbers that 'look' random. Each number in the sequence seems to bear no relation to either its preceding or its succeeding neighbor. Of course, one man's randomness may be another man's deterministic sequence. Is {2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 6, 4, 3, 3, 8, 3, 2, 7, 9} a random sequence of the digits 0-9? It looks pretty random to me, presented in such an in-your-face fashion, but is it? Someone who'd memorized the first 30 digits of π might not necessarily agree, after all these are the digits in its decimal expansion that appear immediately after 3.14159, but does that in turn mean that they are any the less random?

In fact, if you asked someone to reel off a sequence of random digits, you'd probably get something that was anything but random. We, as humans, are conditioned to think of certain digits as 'religious,' 'lucky' or 'magic' numbers (examples being 3 or 7) and we tend to unconsciously favor them. So if you ask someone to think of a random digit, say one hundred times in a row, they'd tend to respond with some digits more often than others. Maybe they'd be prone to linking two digits together as well, eg, a 2 might tend to be followed by a 1 for a subconscious 21. Also, the average person would shy away from saying the same digit twice in a row (which happens in real life about 1 time in 10).

Taking the other viewpoint, we, as humans, are extraordinarily bad at determining whether a random sequence is random or not. If a sequence of random digits had three 4s in a row, one after the other, would you reject it? What about four of them, five of them, 42 of them? Thinking with our probability hats on, the next digit in a random sequence could be a 0 with probability 1/10, a 1 with the same probability, and so on. Hence, the probability of two 4s in a row would be 0.01, three of them 0.001, and so on. Of course, *any* other sequence of length N would have the same probability of appearing as N successive 4s. It's just that N 4s clumped together in a random sequence looks less than random to our eyes.

So how *can* we decide if a random number sequence is, in fact, random? We have to fall back onto statistics. After all, I've just raised the head of the probability monster, so why not call in the statistics monster as well?

The first test we might apply to see if a sequence of digits were random is to count all the 0s, the 1s, the 2s and so on. We'd expect the count of each digit should be about a tenth of the total. So, if we had a 100-digit sequence, we'd expect to see roughly ten of each digit.

Expect. About. Roughly.

Is there some principle more rigorous we can apply here? What if the counts were exactly ten each? That sounds a little fishy to me for a purely random sequence. One would expect that, in a given sequence of 100 digits, some digits would be better represented than others just by chance. In other sequences, other digits might be better represented. But how much of a spread from exactly 10 can we allow before it starts getting a little fishy again?

## Thieves Like Us

Imagine we had a pair of coins that we think someone has tampered with. How could we prove that they were biased? Of course, one crook might be completely dumb and have weighted them to always show heads, but he'd have been caught long ago, leaving our putative master con artist full rein.

| | Two Heads (1) | One Of Each (2) | Two Tails (3) |
|---|---|---|---|
| Our tests (100 tosses) | 28 | 51 | 21 |
| Probability | ¼ | ½ | ¼ |
| Expected number for 100 tosses | 25 | 50 | 25 |

➤ *Table 1: The results of tossing a pair of 'biased' coins 100 times*

Let's throw the coins 100 times, say, and plot the number of times we toss individual scores in a table. Our table might look like Table 1.

I've added the probability of each event to Table 1, and also the expected number of tosses for each event.

Well, just looking at it we could argue the toss (pun intended!) and say that the coins are biased to heads, but is the difference that great? Let's look at the spread (the difference) of our results from the expected values. We'll square these differences to accentuate them and to get rid of any negative values. The sum of these squared differences would be a measure of how biased these coins are. Calculating this sum, I get 26 (= $3^2 + 1^2 + 4^2$). But, hang on, we ought to incorporate the probability of each event somehow. We should get a bigger squared difference for one of each than for two heads, just because the former is more likely to happen. To put it another way, the difference of 3 for two heads seems more significant than the difference of 1 for one of each. So let us divide each squared difference by the expected result of that event. The new sum we calculate is

$$X = \sum_{1}^{3} \frac{(C_i - 100p_i)^2}{100p}$$

where the $C_i$ are our observed counts and the $p_i$ are the probabilities of each event $i$. I get a value for $X$ of 1.02. What we've just calculated is the chi-squared value for our tests. We can look up this value in a standard table of the chi-squared distribution (Table 2).

The table looks slightly daunting, but is quite easy to understand, once explained. The values shown are selected values from the chi-square distribution with ν degrees of freedom. Huh, wazzat funny ν then? Without being rigorous, in our discussion the number of degrees of freedom is one less than the number of 'buckets' we are counting things or events into. In our case, we have three buckets: one for two heads, one for one of each, and one for two tails, and so the number of degrees of freedom for our experiment is 2. Look along the ν = 2 line and there are four values in the four columns. If we look in the 1% column the value there (0.02010) should be read as: 'the value X we calculated should be less than 0.02010 1% of the time.' In other words if we repeated our experiment 100 times then about 1 of them would have an X value of less than 0.02010. If we found that a lot of these experiments had a value less than 0.02010 then it would give a very strong indication that flipping the coins is not a random event, and that they are biased. A similar interpretation can be made for the 5% column. Moving to the 95% column, the value there should be read as: our value X should be less than 5.99121 95% of the time, or it should be greater than this only 5% of the time. Similarly for the 99% column.

We see that our X value falls in between the 5% value and the 95% value and so we don't have a

➤ *Table 2: Percentage points of the Chi-Square distribution*

| | 1% | 5% | 95% | 99% |
|---|---|---|---|---|
| ν = 1 | 0.00016 | 0.00393 | 3.84155 | 6.635 |
| ν = 2 | 0.02010 | 0.10260 | 5.99121 | 9.210 |
| ν = 3 | 0.1148 | 0.35184 | 7.81494 | 11.34 |
| ν = 4 | 0.2971 | 0.71069 | 9.48730 | 13.28 |
| ν = 5 | 0.5543 | 1.14548 | 11.07025 | 15.09 |
| ν = 6 | 0.8721 | 1.63550 | 12.59125 | 16.81 |

strong conclusion either way: we have to assume that the coins are true. If, on the other hand, our X value was 10, we see that this result should only occur in less than 1% of our trials (10 > 9.210, which is the 99% value). And this is therefore a strong indication that the coins were biased. Of course, we should perform more experiments and see how our spread of X values fit into the chi-squared distribution: from an extended set we'll get a better feel for the bias of the coins. We don't want to be caught out with a rogue result, one which probability theory tells us should happen, albeit infrequently.

## Way Of Life

Generally, we take the same boundaries at either end of the range of the chi-squared distribution, say 5% and 95%, and then say that our experiment is significant at the 5% level if it falls outside these boundaries, or is not significant at the 5% level if it falls in between them.

One thing I haven't mentioned so far is this: how many individual events should we generate? In our coin test we did 100 flips. Is this enough, can we get away with less, or should it be more? The answer is unclear. Knuth states that a common rule of thumb is to make sure that the expected number of events for each bucket should be 5 or more (our expected numbers are 25, 50 and 25 so we're all right there), and that the more events to the bucket, the merrier.

Back to our hypothetical random number sequence, and let's apply what we've just learnt. We calculate the count of each digit in our sequence and then calculate the X value and then check it against the chi-square distribution with 9 degrees of freedom (here we have 10 buckets, one for each digit, and thus the number of degrees of freedom is one less than this). We would have to have at least 50 digits to make the expected number for each bucket at least 5.

If we take our sequence as a series of pairs of digits from 00 to 99, then we can bucket the sequence again (counting each pair). There will be 100 buckets this time (99 degrees of freedom), each having a probability of 1/100. We would have to have at least 500 pairs of digits (1000 digits).

We could go further (eg triplets of digits), but the space requirements grow tremendously quickly and there are other tests we could do. Before we look at them, let's have a look at how to generate random number sequences. Once we have a few random number sequence generators under our belt (including Algorithm K) we can test their output against the tests I've shown so far and against the tests to come.

## Every Little Counts

The first thing to realize is that a deterministic algorithm can never generate random number sequences in the same 'way' that throwing dice does, or counting beta particles from a radioactive source. The whole point about a deterministic algorithm is that it generates the same result from the same starting point. If I told you that Generator X using a particular algorithm generates the new random number 65584256 from a starting value (or 'seed') of 12345678, then you'd know 5 months from now that X would calculate exactly the same next value from this same seed. There is absolutely no randomness present in the calculation of the random number sequence, rather it is the sequence of numbers so generated that can be shown (by statistical tests) to be random. Some people prefer to call sequences from a deterministic algorithm pseudo-random, compared with 'true' random numbers from a quantum source such as a decaying radioactive isotope. Personally, I think this distinction is just splitting hairs, especially when it is impossible to distinguish between a sequence of random numbers and a sequence of pseudo-random numbers. Besides which it gets really tedious writing 'psuedo-random number generators' all the time, it's bad enough without the 'pseudo.' Do note, however, that if you know or suspect that a sequence of random digits comes from a pseudo-random number generator of a particular type, you can make some good guesses at the parameters of the generator: this is why cryptographic random number generators (ie those used to encode messages) are extremely complex algorithms to try and hide some subtle correlations between successive random numbers.

The history of random number sequence generators starts off with one of the most illustrious names in computing: John von Neumann. He put forward the following scheme in about 1946: take an N-digit number, square it and from the result (expressed as a 2N-digit number, padded on the left with zeros if required) take the middle N-digits as the next number in the sequence. If we take N as 4, for example, and have 1234 as our starting point, the next few numbers in the random number sequence are 5227, 3215, 3362, 3030, 1809 and so on. This method is known as the middle-square method. It will be helpful to point out a couple of big problems with this algorithm. Using our 4-digit example again, suppose we hit upon a value in the sequence that is less than 10. Calculating the square we get a number less than 100, meaning that the next value in the sequence is 0 (we would take the middle four digits from 000000xx). This again is less than 10, and so the next and all subsequent random numbers in the sequence would also be 0. Hardly random! Also, if you start off with a number like 4100, you'll end up with the sequence 8100, 6100, 2100, 4100... *ad infinitum*. There are other pathological sequences like this and it's quite easy to hit them but difficult to do anything about it. I tried to write a routine, honestly, based on the middle-square method in order to check it with the tests described later, but it was hopeless. Within about 50 or 60 random numbers, it had settled down into generating a series of zeros, or the minimal cycle.

The next big step forward in random number generators came from D H Lehmer in 1949 (putting several nails into the coffin of the middle-square method, if not all of them). What he proposed is known as the *Linear Congruential Method* for generating random number sequences. Choose three numbers, m, a, and c, and a starting seed $X_0$ and use the following formula to generate a sequence of numbers $X_i$:

$$X_{n+1} = (aX_n + c) \bmod m$$

The operation mod m is calculated as the remainder after dividing by m, for example 24 mod 10 is 4.

If we choose our numbers well, the sequence generated will be random. For example, the standard System random number generator in Delphi 1, 2 and 3 uses a = 134775813 ($8088405), c = 1, and m = $2^{32}$, and it is up to us, the programmers, to set the starting seed $X_0$ (it's the RandSeed global variable: we can set it directly or use the Randomize procedure).

It must be noted that if we get a particular value X in the generated sequence at two different points, then the sequence in fact repeats at those two points: the algorithm is deterministic, remember. Because of the modulus operator, no value in the sequence can be greater or equal to m, and all values are between 0 and m-1. Hence, the sequence will repeat itself after at most m values. It may (if we are inept at choosing a, c and m) repeat much sooner, a simple example is a = 0: the sequence boils down to {c, c, c...}, repeating itself after only one term.

So what are good values for these magic numbers a, c and m? Much has been conjectured, posited and proved in the literature. Generally, we choose m to be as large as possible so that our repeat cycle is as large as possible as well, and it's usually chosen as the integer size of the machine (eg, $2^{32}$ for Win32 systems). This also makes the modulus operation that much faster, the modulus is the lower 32-bits of the result of ($aX_n + c$). For c, we generally choose 1. And a is usually chosen to be relatively prime to m (ie, the greatest common divisor of a and m is 1).

```
type
  PRandArray = ^TRandArray;
  TRandArray = array [0..54] of longint;
const
  Inx1  : integer = 0;
  Inx2  : integer = 0;
  RArray: PRandArray = nil;
begin
  if (RArray = nil) then begin
    New(RArray);
    ..fill elements from another generator..
    Inx1 := 0;
    Inx2 := 33;
  end;
  RArray^[Inx1] := RArray^[Inx1] + RArray^[Inx2];
  Result := RArray^[Inx1] shr 1;
  inc(Inx1);
  if (Inx1 = 55) then
    Inx1 := 0;
  inc(Inx2);
  if (Inx2 = 55) then
    Inx2 := 0;
end;
```

➤ *Listing 1*

## State Of The Nation

I'll now introduce Algorithm K. This has a = 31415927, c = 0, and m = 27182819. Seems all right on the surface. Let's investigate a little further. The greatest common divisor of a and m is 1, so we could be on the way to generating a sequence of random numbers that will only repeat after at most 27182819 values. Having c = 0 is a pain in the neck: whenever we get a zero in the sequence, we have to fake it to another number so that we don't start to repeat getting zeros (Algorithm K has a special test to force a zero to a one before the calculation), a better value would've been c = 1 and this problem would have been avoided. Here's the important bit of the routine:

```
var
  iRandSeed : LongInt;
...
iRandSeed :=
   (iRandSeed*31415927) mod 27182819;
```

Can you spot the problem here? It's overflow. The multiplication should produce a 64-bit answer, however, the machine code generated by the compiler will discard the top 32 bits. So only the lower 32 bits of the result of the multiplication is used by the mod operator (there is also a problem with negative values, but we won't go into that here). The calculation, as written and compiled, is throwing away the most significant part of the multiplication. And, believe it or not, that is why Algorithm K fails. If the generator had been written in assembly language (taking care of the 64-bit intermediary result) it would have been perfect. If the 'mod 27182819' had been left off the calculation it would have been pretty good (taking the lower 32 bits of a multiplication is equivalent to mod $2^{32}$, remember). The problem lies in taking the explicit mod 27182819 after doing the implicit mod $2^{32}$: 27182819 divides $2^{32}$ 19.78 times. In other words, the random number sequence would be biased towards the first ¾ of the random number space.

Anyway, onwards with the theory. There is a minor problem in using m = $2^{32}$ which isn't readily obvious: the lower bits of the random number values are much less random than the higher bits. For a discussion of why this is, please see Knuth. However, the sheer ease of taking the modulus generally overwhelms this minor problem and there are other things we can do to make the random number sequence more random. One simple algorithm is to generate the next 32-bit random number in the sequence from two separate calculations and taking the upper 16 bits from each of the sub-calculations. This does have the negative effect of reducing the cycle length.

Another problem with using m = $2^{32}$ stems from the fact that Delphi does not have a true DWORD (ie, a type that can hold a number from 0 to $2^{32}$-1, an unsigned long integer). If we code these random number generators in Delphi we continually have to make sure that we don't have an implicit conversion to a `LongInt` and the production of a negative number from that. Assembly coding removes this problem.

## Bizarre Love Triangle

The Linear Congruential Generator is pretty good, but is there something better? In 1958, Mitchell and Moore suggested the following generator:

$$X_{n+1} = (X_{n-24} + X_{n-55}) \bmod m, \text{ with } n \geq 55$$

and $X_0$, ..., $X_{54}$ are generated from some other generator, like a Linear Congruential algorithm. This is known as an additive generator. To code this in Delphi we use $m = 2^{32}$ (note that again the lower bits of our random numbers will be less 'random' than the upper bits), and we make use of an array of 55 values and two index variables. These variables have to be static (ie, their values have to remain set in between invocations of the random number routine). We also cheat a little, by making sure that the generator only produces numbers between 0 and `MaxLongint`, see Listing 1.

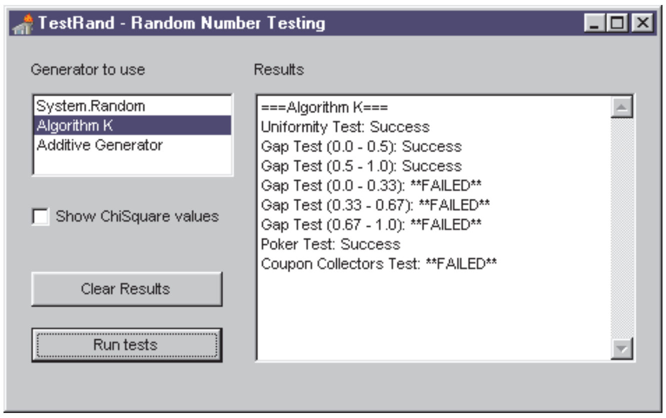
There are obvious improvements we can make to this routine (eg coding as a class, so we can destroy the array at the end of the program, and using assembler). The cycle length of this generator has been shown to be $2^{55}$-1, about 8 million times as long as the standard Delphi one. To put it another way: suppose you have a program that uses 1 million random numbers per second. You'd recycle all the random numbers from the Delphi `Random` function 20 times in a day. At the same rate, this generator would be able to go for just over 1,000 years before repeating.

Well, we now have three random number generators under our belt: the standard `Random` function in the `System` unit (a good linear congruential generator), Algorithm K, and a more complex additive one. Now let's consider how to test them.

## Age Of Consent

The tests all follow the same logic. We'll use random numbers between 0.0 (inclusive) and 1.0 (exclusive). We count various events derived from these random numbers into buckets, calculate the probability associated with each bucket, from which we can work out the chi-square value and apply the chi-square test with the number of degrees of freedom being one less than the number of buckets. A little abstract, but you'll see the idea in a moment.

The first test is the simplest: the uniformity test. This is the one we were discussing earlier. Basically, the random numbers are going to be checked to see that they 'uniformly' cover the range 0.0 to 1.0. We create 100 buckets, generate 10,000 random numbers, and slot them into each bucket. Bucket 0 gets all the random numbers from 0.0 to 0.01, bucket 1 gets then from 0.01 to 0.02, and so on. The probability of a random number falling into a particular bucket is obviously 0.01. We calculate the



```
===Algorithm K===
Uniformity Test: Success
Gap Test (0.0 - 0.5): Success
Gap Test (0.5 - 1.0): Success
Gap Test (0.0 - 0.33): **FAILED**
Gap Test (0.33 - 0.67): **FAILED**
Gap Test (0.67 - 1.0): **FAILED**
Poker Test: Success
Coupon Collectors Test: **FAILED**
```

➤ *Figure 1*

chi-square value for our test and check that against the standard table, using the 99 degrees of freedom line.

The second test is a little more interesting: the Gap test. No, this isn't a test to see whether you can walk past a Gap store without buying anything! The Gap test ensures that you don't get runs of values in one particular range followed by runs in another, flip-flopping between the two, even though as a whole the random numbers are evenly spread out. Define a subrange of the range 0.0 to 1.0, let's say the first half: 0.0 to 0.5. Generate the random numbers. For each random number, test to see whether it hits our subrange or whether it misses. You'll get a sequence of hits and misses. Look at the runs of one or more misses (these are called the gaps between the hits, hence the Gap test). You'll get some runs of just one miss, of two misses, and so on. Bucket these lengths. Let's say the probability of a hit is p (it'll be the width of the subrange expressed as a decimal) and so the probability of a miss is (1-p). We can now calculate the probability of a run of one miss: (1-p).p; of two misses: $(1-p)^2.p$; of n misses: $(1-p)^n.p$, and hence calculate the expected numbers for each run length. From then it's a short step to the chi-squared test. We shall use 10 buckets, hence there are 9 degrees of freedom. Generally, we repeat the Gap test for the first and second halves of the range and for the first, second and third thirds.

The third test is known as the Poker test. The random numbers are grouped into sets or 'hands' of five and the numbers are converted into 'cards', each 'card' actually being a digit from 0 to 9. The number of different cards in each hand is then counted (it'll be from 1 to 5) and this result is bucketed. Because the probability of only one digit repeated 5 times is so low, it is generally grouped into the '2 different-digits' category. Apply the chi-squared test to the four buckets: there will be 3 degrees of freedom. The probability for each bucket is difficult to calculate (it involves some combinatorial values called Stirling numbers) so I won't present it here.

The fourth test is the Coupon Collector's test (the names of these tests are nothing short of bizarre!). The random numbers are read one by one and converted into a 'coupon' or a number from 0 to 4. The length of the sequence required to get a complete set of the coupons (ie, the digits 0 to 4) is counted: this will obviously vary from five onwards. Once a full set is obtained, we start over. We bucket the lengths of these sequences and then apply the chi-squared test to the buckets. We'll use buckets for the sequence lengths from 5 to 19, and then have a composite bucket for every length after that. So, 16 buckets and hence 15 degrees of freedom. Again, like the Poker test, the

## Books

In any programming environment, be it Delphi or something else, you need a good set of algorithm books. Books to teach you about algorithms, discuss efficiency, provide eye openers as to what is actually available, to set your imagination alight. For a very long time (20 years), *the* definitive set of algorithm books has been the three volumes of *The Art of Computer Programming* by Donald Knuth (published by Addison Wesley). Originally planned as a series of seven volumes, Knuth completed three and then moved onto computer typesetting and other research. The names of the three volumes are *Fundamental Algorithms*, *Seminumerical Algorithms*, and *Sorting and Searching*. I well remember consulting them in the library at Kings College, London when I had some thorny problem with one of my programming courses for my degree. Although tough going in places, they are still as valid today as in the late 70s. Suddenly, in 1996, he decided to bring all three volumes up to date, before embarking on Volumes 4 and 5. I'm happy to say that the first two new editions are now available and the Addison Wesley website was promising the third new edition in April 1998. Of particular relevance here is that Knuth rewrote the entire random number section in *Seminumerical Algorithms*, to bring it completely up to date with the furious amount of research going on in random numbers. Recommended, if you are really serious about learning more about algorithms.

As I have your attention, I'm pleased to note that Robert Sedgewick is rewriting his tome called *Algorithms*, another of my favorite well-thumbed algorithm books. The previous editions appeared in versions for C, C++ and Modula 2 as well as Pascal, and Sedgewick has now decided (wisely) just to concentrate on C, leaving it up to the reader to cast the code into the language of his or her choice. Anyway, the first volume has appeared: *Algorithms in C (Third Edition) Parts 1-4*, again by Addison Wesley. The second volume, parts 5-8, are promised in the not too distant future. Also recommended.

calculation of the probability for each bucket is again complex, so I won't present it here.

## Vanishing Point

The complete set of four tests presented above are coded in the Test Rand program on this month's disk. The program presents you with a selection of random number generators and a button to press to perform all the tests. The results of the tests are shown in a memo control. Figure 1 shows the results of using a faulty generator.

I must inject a note of caution here. The TestRan program outputs these big 'FAILED' stickers against each test that fails (to put it another way: the result is significant at the 5% level). Please note that the most perfect random number generator in the world (one based on quantum events) will fail each test about one time in twenty. This does not mean that the generator is flawed (or indeed the test), it's just a natural result of all the probability floating around. On the other hand, if a generator consistently fails a particular test, it certainly indicates that the test is showing that the generator is faulty.

If you are interested in further random number tests, then by all means read the relevant section in Knuth. George Marsaglia, a researcher in random number generators at Florida State University, has written an amazingly complete suite of random number tests as a program called DIEHARD. You provide the program with about 10Mb of random bytes and it'll go through applying the entire suite to your random numbers. You can download DIEHARD from http://stat.fsu.edu/~geo/diehard.html

The TESTRAND program also contains the additive random number generator we've been talking about: an excellent source of random numbers. You can extract the relevant code from the unit `TstRndU2.PAS` and use it in your own programs.

Also on this month's disk is a fairly complete random number generator class (based on an additive generator) which provides a better replacement for `Random`. It's part of the next version of my freeware EZDSL (Easy Data Structures Library for Delphi) library, the current version of which can be downloaded from any friendly ftp site near you, or from *The Delphi Magazine Collection 97* CD-ROM *[Along with tons of other goodies! Ed]*. I'm working on completing EZDSL version 3.00 as I write (in fact you can regard this generator class as being in beta), and by the time you read this it should almost be ready. See

```
home.turbopower.com/
  ~julianb/ezdsl.htm
```

for the latest news.

Until next time: be capricious!

Julian Bucknall programs (duh!), acts, writes articles and helps his girlfriend Donna organize their wedding in September (a New Order begins). Sometimes due to sheer pressure one of these activities has to give way, and so he'd like to apologize to her in writing.