

2011 Dr. Dobbs
Excellence Awards



NOMINATE YOURSELF TODAY!
YOU CAN'T WIN AN AWARD UNLESS YOU ENTER!

Dr. Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Quick and Portable Random Number Generators

By Jerry Dwyer

June 01, 1995

URL: <http://drdobbs.com/184403024>

Jerry Dwyer is a Professor of Economics at Clemson University with a Ph.D. from the University of Chicago. He primarily writes programs for statistical analyses and for real-time experiments. He has been programming for 20 years, and he has been writing in C for 10 years. He can be reached at dwyerg@clemson.edu.

Introduction

A computer-generated random number is a contradiction in terms. Random numbers are unpredictable and computers are perfectly predictable. Computers must do the same thing whenever they are in the same state. Typical random-number generators iterate a function to get a sequence of numbers. No matter how seemingly arbitrary, a sequence of numbers produced by a computer is perfectly predictable; it is not random. Hence the common term "pseudo-random numbers" arises for what I for brevity will call "random numbers."

In addition to not being random, random-number generators on computers are not trivial to implement correctly. In the process of doing some simulations, I spent quite a bit of time reading fairly technical material. In this article, I will explain the most common random-number generators. You will see some 32-bit random number generators that you can easily and comfortably include in your own code for any ANSI C compiler. The routines only assume that the maximum signed integer is at least $2^{31}-1$, which is required by ANSI C. Along the way, you will see how to use approximate factoring to avoid overflow in modular arithmetic.

Lehmer Generators

The most commonly used random-number generator is what I will call the "Lehmer generator." Suggested by D.H. Lehmer in 1951, the "multiplicative congruential random-number generator" is based on the recursion:

$$x(t+1) = (a * x(t)) \bmod m$$

where $x(t)$ is the value at iteration t used to generate the next value $x(t+1)$, a is a multiplier, and \bmod is the modulo function. If $a * x$ is positive or zero and m is positive, the modulo function generates the same value as the remainder operator (%) in C. If m is positive, the modulo function is the residue

$$a * x - \text{RES}(a * x / m) * m$$

where $\text{RES}(a * x / m)$ is notation representing the largest integer less than or equal to $a * x / m$.

For evaluating random-number generators, the relationship between consecutive values is sometimes informative. [Figure 1](#) shows graphs of $x(t+1)$ against $x(t)$ for integers, where $x(t)$ can be any integer from one to six. [Figure 1\(a\)](#) shows the graph for a series of random numbers. It makes sense to think of a series of true random numbers as being able to take on any possible set of values. In other words, true random numbers fill up the space. In addition, it makes sense to require that the values occur with equal probability. In other words, true random numbers have a uniform distribution.

The Lehmer generator produces a nonrandom structure in two or more dimensions. For example, suppose the generator is:

$$x(t+1) = (5 * x(t)) \bmod 7$$

The full set of values in order is 1, 5, 4, 6, 2, 3, 1, ... This sequence generates all of the values from 1 to 6. This sequence does not, however, begin to fill up a two-dimensional space. [Figure 1\(b\)](#) shows the values of $x(t)$ and $x(t+1)$ that occur. Only a few combinations actually occur, and they fall on a couple of lines. When triplets of consecutive values are plotted instead of just pairs as in [Figure 1](#), the combinations of values fall on planes. In higher dimensions, the values continue to fall on higher-dimensional planes.

This particular structure is an inevitable consequence of the Lehmer generator. The structure does not mean that the Lehmer generator is worthless — but it is suspect for an application sensitive to correlations between consecutive pseudo-random numbers. Other random-number generators have different, but discoverable, structures. For this random-number generator, the values of the multiplier and the modulus determine the number and location of the planes. The issue becomes: what values of the multiplier and the modulus generate reasonably good random numbers for a given purpose?

In addition to this particular structure of Lehmer generators, they (and any others on a computer) are repetitive. With a multiplier of 5 and a modulus of 7, the Lehmer generator has a period of 6: the sequence repeats after 6 iterations. Repetition is not peculiar to Lehmer generators. No matter what iterative function is used, a computer has a finite number of states. Sooner or later a sequence must repeat because the function iterates into a state that it has seen before.

It is fairly common to soup up the Lehmer generator by adding an increment to the product $a*x(t)$. This increased complexity has one advantage. The Lehmer generator cannot produce a random number equal to zero. If it did, every succeeding number also would equal zero. Adding an increment makes it possible to get a random number equal to zero, thereby increasing the maximum possible period by one. This is not a compelling reason for the additional complexity, however.

Good Lehmer Generators

Even though random-number generators eventually repeat and have some discoverable structure of sequential values, there are better and worse generators. All other things being equal, longer periods are better than shorter ones. As I mentioned above, the generator:

$$x(t+1) = (5*x(t)) \bmod 7$$

has only 6 possible values of the random numbers, and a period of 6, which is too short for most purposes. Periods on the order of 2^{15} or 32,768, are too short for marginally serious computations, and at least some compiler vendors seem to agree. For example, Borland C++3.1 and Microsoft Visual C 1.0 both have 32-bit random-number generators with longer periods. (However, I still consider these generators unappealing, as their periods are shorter than those in this article. Besides, these compilers do not use the best known multipliers and they rely on non-portable integer overflow.)

The period before repetition is not the only consideration. In any application, you use only a small fraction of the total period. As a result, the seeming randomness of partial sequences of the random numbers is important. You know that the "random numbers" aren't random! It is important, though, that the numbers appear to be random from the viewpoint of what you're doing with them. For example, the predictability of new values of the random numbers given part of the sequence but not the multiplier or the modulus is important for cryptography. Lehmer generators are not good for this purpose because they are predictable by algorithms that know the numbers are produced by a Lehmer generator.

In order to get long periods, I consider only generators of 32-bit integers. There are many possible values for the multiplier and modulus. What are some good values? The modulus has a big impact on the calculations, so it is a natural place to start limiting the search.

One natural value for the modulus is 2^{32} . Why? Assume that multiplication is defined for 32-bit signed integers. Consider the line of pseudocode

```
x <- a * x
```

The maximum possible value of x is $2^{31} - 1$. If signed integer overflow ignores the high bits (a standard fixup) and the sign bit is masked, this code can implement a Lehmer generator with a modulus of 2^{32} . Signed integer overflow need not have this standard fixup, however. In ANSI C, integer overflow is an exception and the behavior of the program is undefined. [You can use unsigned integer arithmetic, but often at a notable cost in performance. — pjp] Furthermore, the effect of this code depends on a machine's representation of integers. You can lose portability across machines, languages, possibly compilers, and even versions of a compiler. It is easy to spend more time cleaning up such incompatibilities than ever is saved by the faster function.

Another common value of the modulus is $2^{31} - 1$, or 2147483647. Why this seemingly odd value? It is prime, which affects the maximum period. In addition, it is the maximum value of signed integers on many machines in languages with only signed integers, such as BASIC, FORTRAN, and Pascal.

At first glance surprisingly, the generator with the larger modulus has about half the period. For the modulus 2^{32} , if the initial value is odd, then the period can be as long as 2^{30} . For the modulus $2^{31} - 1$, the period can be as long as $2^{31} - 2$. All of the multipliers in this article for this modulus have the maximum period, and the program that combines generators has a period not less than about 2^{61} . No need to settle for a shorter period when there are lots of long-period Lehmer generators.

What are some good values for multipliers? It might seem desirable to check on your own for good multipliers, but there is no point if you are simply writing a program that uses a random-number generator. Others have exhaustively studied both of these multipliers and others have worked on ways to write portable 32-bit generators on machines with 32-bit signed integers.

Portable Generators

It is easy to write a 32-bit routine with a modulus of $2^{31} - 1$ that does not generate integer overflow and is portable. The basic problem is integer overflow: the product of the multiplier and the last value of the random number can be larger than a 32-bit integer. As the [sidebar](#) on approximate factoring indicates, Linus Schrage has worked out a fast way to take the modulus of a product by approximately factoring the modulus.

George Fishman and Louis Moore [1] have thoroughly studied Lehmer generators with a modulus of $2^{31} - 1$. All of their best multipliers are too large for

approximate factoring. Nonetheless, several multipliers are small enough and are almost as good by the same criteria. [Table 1](#) shows, in rough order of preference, four good multipliers for this modulus. Among the values is the multiplier 16,807, used by many people over the last 25 years.

How can you check whether you have written the algorithm correctly? Testing the apparent randomness of the numbers from the viewpoint of your application is important for being sure that the generator is suitable. This does not check your program though. The most straightforward way to check your program is to see whether you get the correct values. In [Table 1](#), I provide the values that I get in Mathematica and Borland C in DOS and OS/2 for the generators on the 10,000th draw starting from an initial value of one.

If there is an error in your program, the probability that the same seed will generate the correct value on the 10,000th draw is virtually zero. Because of the nonlinear relationship between consecutive values, as illustrated in [Figure 1](#), by the 10,000th iteration of your incorrect function the likelihood that you will get the correct value probably is on the order of one over the modulus. [Listing 1](#) shows the file *rand.ma*, which is a program in Mathematica that generates the 10,000th value. For me, this is a convenient check on the results from C code at the bottom of [Listing 2](#).

[Listing 2](#), the file *rand_por.c*, is an implementation in C of this portable routine for a particular multiplier. The initialization code may overflow in some compilers, but the rest of the code is portable. Whenever there is a choice between readability and speed in this code, I opt for readability. The most important consequence of emphasizing readability is the implementation of the generator as a function: *genr_rand_port*.

The routines have four basic external functions. Two functions, *init_rand* and *rand_port*, are similar to the Standard C functions *srand* and *rand*. They behave as you would expect. The additional routines are *get_init_rand* and *rand_rect_port*. You can call *get_init_rand* to get a seed to pass to *init_rand*.

If you call *init_rand* with an incorrect value, say 0 or -1 (accidentally or on purpose), *init_rand_port* calls *get_init_rand_port* to get a seed. The return from *init_rand_port* provides the user with the actual seed used, important information for rerunning the generator and getting the same sequence of random numbers. Because a common correct seed is the value one, the routine throws away some initial values to get a more arbitrary first random number than the result of the first few calls after this seed: the multiplier and powers of it. Throwing away exactly 16 values is arbitrary.

The routine *rand_rect_port* in [Listing 2](#) correctly generates a value between zero and one that has a uniform (also known as rectangular) distribution between zero and one. Because many uses of uniformly distributed values assume that the value is not exactly zero, the routine does not generate either zero or one.

Often it is desirable to generate non-overlapping sequences of random numbers, for example in Monte Carlo studies. [Listing 2](#) includes the routine *skip_ahead* to calculate the random number that is *skip* values ahead of the value *init_rand*. From an initial random number of say 3, you can calculate the value of the random number that is generated by *rand_port* 4,000 or 1,000,000 iterations later.

This routine is quite a bit faster than computing the intermediate values. Doing this computation without overflow requires care because the desired output is:

$$(a^{\text{skip}} * \text{init_rand}) \bmod m$$

for all values of *skip* less than the modulus *m*. The routine *skip_ahead* and the called routine *mult_mod* use the Russian-peasant algorithm [4] and approximate factoring to avoid overflow.

Economists have a standard saying, "There's no such thing as a free lunch," and that's true here. The benefits of portability and a longer period have a cost in speed. From my point of view, the cost is not huge. The routine *rand_port* is not exactly sluggish. It takes about 0.23 seconds to get 100,000 random numbers on my 66 MHz 486 under OS/2, which can be compared to 0.09 seconds using Borland's overflow routine. The performance hit is harder in 16-bit DOS: the portable routine takes 1.7 seconds versus 0.22 seconds for Borland's routine. Even at that price, I prefer the portable routine, which has twice the period and lends itself to skipping ahead.

Combining Generators and Shuffling

While a period of $2^{31}-2$, which is about 2.15 billion, is not exactly short for many purposes, it is not necessarily long enough for extensive computations. Furthermore, a Lehmer generator can produce a particular value only once in a complete cycle, and the planes illustrated in [Figure 1](#) can be a problem for some uses of the random numbers. Two techniques for mitigating these problems and lengthening the period are combining numbers from Lehmer generators and shuffling the numbers from a generator.

Obvious combinations of the numbers from Lehmer generators are the sum and the difference. L'Écuyer [2, 3] suggests a difference between two numbers that effectively obliterates planes such as those in [Figure 1](#). The difference between numbers from Lehmer generators in [Listing 3](#) has a period of about 2.31×10^{18} , or 2.31 billion billion.

Shuffling the output of a random-number generator is another way to mitigate problems and lengthen the period. The basic idea is simple. Rather than take the numbers in sequence from the Lehmer generator, several values are stored and one of them is picked at "random" using the most recent random number. For a particular multiplier and modulus, the actual increase in the period from shuffling the generator is difficult to determine analytically, but this technique cannot worsen the generator. (Oddly enough, using a second Lehmer generator to pick the next value can worsen a generator, one of many pitfalls of random numbers.)

The sequence of numbers is more apparently random because it is not the set of consecutive values. Shuffling is especially useful if you are going to use the random numbers in another routine that can have unfortunate interactions with a Lehmer generator, which some transformations of consecutive values can.

[Listing 3](#), the file `rand_com.c`, is a set of routines that combines the results of two generators and shuffles the output. [Table 1](#) gives you the values of the 10,000th draws for the two underlying generators and for the combined generator.

There are no routines for skipping ahead in [Listing 3](#). If the results of a random-number generator are shuffled, there is no known way to skip ahead without computing intermediate values. If you are happy with a period of about 2.31 billion billion, though, you can easily modify [Listing 3](#) to use the routines for skipping ahead in [Listing 2](#). Just delete shuffling from the routines in [Listing 3](#) and use the routines for skipping ahead in [Listing 2](#) for the two underlying Lehmer generators in [Listing 3](#).

In summary, if you want a well-behaved and reasonably quick routine to calculate random numbers and simply want values with equal probabilities, I recommend `rand_port`. If you want a routine with a longer period and more apparently random numbers even though it is slower, I recommend `rand_comb` with or without shuffling.

References

- [1] George S. Fishman and Louis R. Moore III. "An Exhaustive Analysis of Multiplicative Congruential Random-Number Generators with Modulus $2^{31}-1$," *SIAM Journal on Scientific and Statistical Computing* January: 1986.
- [2] Pierre L'Écuyer. "Efficient and Portable Combined Random-Number Generators," *Communications of the ACM* June: 1988.
- [3] Pierre L'Écuyer. "Random Numbers for Simulation," *Communications of the ACM* October: 1990.
- [4] Donald Knuth. *The Art of Computer Programming*, Vol. 2, "Seminumerical Algorithms," Second Edition (Reading MA: Addison Wesley, 1981), pp 441-443.

Further Reading

If you want to read more about random-number generators, you will want to begin with Chapter 3 of Donald Knuth's classic (listed above). But "classic" is a double-edged sword. Some of his discussion is dated. Bratley, Fox, and Schrage provide a nice overview of the issues. James provides a nice overview of alternative generators. Some references in a rough order of accessibility, are:

Bratley, Paul, Bennett L. Fox and Linus E. Schrage. *A Guide to Simulation*, Second Edition. New York NY: Springer Verlag, 1987.

James, F. "A Review of Pseudorandom-Number Generators," *Computer Physics Communications*. October 1990.

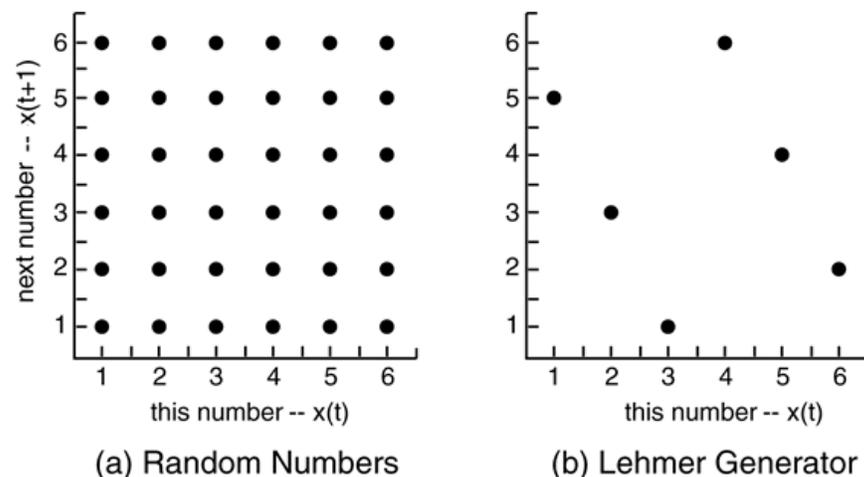
Press, William H., et al. *Numerical Recipes in C*, Second Edition. Cambridge: Cambridge University Press, 1992.

Park, Stephen K., and Keith W. Miller. "Random-Number Generators: Good Ones Are Hard to Find," *Communications of the ACM*. October 1988.

L'Écuyer, Pierre, and Serge Ct. "Implementing a Random-Number Package with Splitting Facilities," *ACM Transactions on Mathematical Software*. March 1991.

Fishman, George S. "Multiplicative Congruential Random-Number Generators with Modulus 2^b : An Exhaustive Analysis for $b=32$ and a Partial Analysis for $b=48$," *Mathematics of Computation*. January 1990.

Figure 1 Random numbers and the Lehmer Generator



Listing 1 *A couple of lines of Mathematica code that execute a Lehmer generator. Print the 10,000th value after initializing at one*

```
Rand[x_] := Mod[41358*x, 2147483647]
x = 1; Do[ x = Rand[x], {i, 10000}]; Print[x];
```

Listing 2 *A portable and reasonably fast multiplicative random number generator*

```
/* Listing 2
   rand_por[t].c
   see
   L'Ecuyer - Comm. of the ACM, Oct. 1990, vol. 33.
   Numerical Recipes in C, 2nd edition, pp. 278-86
   L'Ecuyer and Cote, ACM Transactions on Mathematical
   Software, March 1991
   Russian peasant algorithm -- Knuth, vol. II, pp. 442-43
   Copyright (c) 1994, 1995 by Gerald P. Dwyer, Jr. */

#include <time.h>
#include <stdlib.h>
#include <limits.h>
#include <assert.h>

#define TESTING

#define TRUE (-1)
#define FALSE 0

long init_rand_port(long seed) ;
long get_init_rand_port(void);
long genr_rand_port(long init_rand) ;
long rand_port(void) ;
double rand_rect_port(void) ;
long skip_ahead(long a, long init_rand, long modulus, long skip) ;
long mult_mod(long a, long x, long modulus) ;

#define MOD 2147483647L /* modulus for generator */
#define MULT 41358L /* multiplier */
/* modulus = mult*quotient +
remainder */
#define Q 51924L /* int(modulus / multiplier) */
#define R 10855L /* remainder */
#define MAX_VALUE (MOD-1)

#define EXP_VAL 1285562981L /* value for 10,000th draw */

#define IMPOSSIBLE_RAND (-1)
#define STARTUP_RANDS 16 /* throw away this number of
initial random numbers */

static long rand_num = IMPOSSIBLE_RAND ;

/* initialize random number generator with seed */
long init_rand_port(long seed)
{
    extern long rand_num ;
    int i ;

    if (seed < 1 || seed > MAX_VALUE) /* if seed out of range */
        seed = get_init_rand_port() ; /* get seed */

    rand_num = seed ;
    for (i = 0; i < STARTUP_RANDS; i++) /* and throw away */
        rand_num = genr_rand_port(rand_num) ; /* some initial
ones */

    return seed ;
}

/* get a long initial seed for generator
assumes that rand returns a short integer */
long get_init_rand_port(void)
{
    long seed ;

    srand((unsigned int)time(NULL)); /* initialize system generator */
    do {
        seed = ((long)rand()*rand()) ;
    } while (seed < 1 || seed > MAX_VALUE);
}
```

```

    seed += ((long)rand()*rand() ;
} while (seed > MAX_VALUE) ;

assert (seed > 0) ;

return seed ;
}

/* generate the next value in sequence from generator
uses approximate factoring
residue = (a * x) mod modulus
         = a*x - [(a*x)/modulus]*modulus
where
[(a*x)/modulus] = integer less than or equal to (a*x)/modulus
approximate factoring avoids overflow associated with a*x and
uses equivalence of above with
residue = a * (x - q * k) - r* k + (k-k1) * modulus
where
modulus = a * q + r
q = [modulus/a]
k = [x/q] (= [ax/aq])
k1 = [a*x/modulus]
assumes
a, m > 0
0 < init_rand < modulus
a * a <= modulus
[a*x/a*q]-[a*x/modulus] <= 1
(for only one addition of modulus below) */
long genr_rand_port(long init_rand)
{
    long k, residue ;

    k = init_rand / Q ;
    residue = MULT * (init_rand - Q * k) - R * k ;
    if (residue < 0)
        residue += MOD ;

    assert(residue >= 1 && residue <= MAX_VALUE) ;
    return residue;
}

/* get a random number */
long rand_port(void)
{
    extern long rand_num;

    /* if not initialized, do it now */
    if (rand_num == IMPOSSIBLE_RAND) {
        rand_num = 1 ;
        init_rand_port(rand_num) ;
    }

    rand_num = genr_rand_port(rand_num) ;

    return rand_num;
}

/* generates a value on (0,1) with mean of .5
range of values is [1/(MAX_VALUE+1), MAX_VALUE/(MAX_VALUE+1)]
to get [0,1], use (double)(rand_port()-1)/(double)(MAX_VALUE-1) */
double rand_rect_port(void)
{
    return (double)rand_port()/(double)(MAX_VALUE+1) ;
}

/* skip ahead in recursion
residue = (a^skip * init) mod modulus
Use Russian peasant algorithm */
long skip_ahead(long a, long init_rand, long modulus, long skip)
{
    long residue = 1 ;

    if (init_rand < 1 || init_rand > modulus-1 || skip < 0)
        return -1 ;
    while (skip > 0) {

```

```

    if (skip % 2)
        residue = mult_mod(a, residue, modulus) ;
    a = mult_mod(a, a, modulus) ;
    skip >>= 1 ;
}
residue = mult_mod(residue, init_rand, modulus) ;

assert(residue >= 1 && residue <= modulus-1) ;

return residue ;

}

/* calculate residue = (a * x) mod modulus for arbitrary a and x
without overflow assume 0 < a < modulus and 0 < x < modulus
use Russian peasant algorithm followed by approximate factoring */
long mult_mod(long a, long x, long modulus)
{
    long q, r, k, residue;

    residue = -modulus ;          /* to avoid overflow on addition */

    while (a > SHRT_MAX) { /* use Russian Peasant to reduce a */
        if (a % 2) {
            residue += x;
            if (residue > 0)
                residue -= modulus ;
        }
        x += (x - modulus) ;
        if (x < 0)
            x += modulus ;
        a >>=1;
    }

        /* now apply approximate factoring to a
and compute (a * x) mod modulus */
    q = modulus / a ;
    r = modulus - a * q ;
    k = x / q ;
    x = a * (x - q * k) - r * k ;
    while (x < 0)
        x += modulus ;
        /* add result to residue and take mod */
    residue += x ;
    if (residue < 0) /* undo initial subtraction if necessary */
        residue += modulus ;

    assert(residue >= 1 && residue <= modulus-1) ;

    return residue ;

}

#if defined(TESTING)
/* Test the generator */
#include <stdio.h>
void main(void)
{
    long seed ;
    int i ;
    seed = init_rand_port(1);
    printf("Seed for random number generator is %ld\n", seed) ;
    i = STARTUP_RANDS ; /* threw away STARTUP_RANDS */
    do {
        rand_port() ;
        i++;
    } while (i < 9999) ;

    printf("On draw 10000, random number should be %ld\n",
        EXP_VAL) ;
    printf("On draw %d, random number is %ld\n", i+1,
        rand_port()) ;
}
#endif /* TESTING */
/* End of File */

```

Listing 3 Combination multiplicative random number generator

```

/* rand_com[b].c
   subtract two random numbers modulo moduli-1 and shuffle
   see
   L'Ecuyer, Comm. of the ACM 1988 vol. 31
   Numerical Recipes in C, 2nd edition, pp. 278-86
   shuffling -- Knuth, vol. II
   Copyright (c) 1994, 1995 by Gerald P. Dwyer, Jr.
*/

#include <time.h>
#include <stdlib.h>
#include <float.h>
#include <assert.h>

#define TESTING

#define TRUE (-1)
#define FALSE 0

void init_rand_comb(long *seed1, long *seed2) ;
long get_init_rand(int) ;
long rand_comb(void);
long genr_rand_diff(void);
long genr_rand(long a, long x, long modulus, long q, long r) ;

/* first generator */
#define MOD1 2147483563L /* modulus */
#define MULT1 40014L /* multiplier */
/* modulus=multiplier*quotient+remainder */
#define Q1 53668L /* quotient =[modulus/multiplier] */
#define R1 12211L /* remainder */

/* second generator */
#define MOD2 2147483399L
#define MULT2 40692L
#define Q2 52774L
#define R2 3791L

#define MOD_COMB (MOD1-1)

#define MIN_VALUE1 1
#define MAX_VALUE1 (MOD1-1)
#define MIN_VALUE2 1
#define MAX_VALUE2 (MOD2-1)
#define MAX_VALUE ((MOD1<MOD2) ? MAX_VALUE1 : MAX_VALUE2)
#define EXP_VAL 804307721L

#define GENR1(init_rand) genr_rand(MULT1, init_rand, MOD1, Q1, R1)
#define GENR2(init_rand) genr_rand(MULT2, init_rand, MOD2, Q2, R2)

#define IMPOSSIBLE_RAND (-1)
#define STARTUP_RANDS 16 /* throw away initial draws */
#define DIM_RAND 150 /* size of array shuffled */

static long rand1, rand2 ;
static long rand_num = IMPOSSIBLE_RAND ;
static long rands[DIM_RAND];

/* initialize generators with seeds
   fill rands[] with initial values */
void init_rand_comb(long *seed1, long *seed2)
{
    extern long rand1, rand2 ;
    extern long rand_num;
    extern long rands[] ;
    int i ;

    if (*seed1 <= 0 || *seed1 > MAX_VALUE1)
        *seed1 = get_init_rand(MAX_VALUE1);
    if (*seed2 <= 0 || *seed2 > MAX_VALUE2)
        *seed2 = get_init_rand(MAX_VALUE2);

    /* seed the routine */
    rand1 = *seed1;
    rand2 = *seed2;
    genr_rand_diff() ;

    for (i = 1; i < STARTUP_RANDS; i++) /* throw some away */

```

```

    genr_rand_diff() ;
        /* fill the array of random number values */
for (i = 0; i < DIM_RAND; i++)
    rands[i] = genr_rand_diff() ;
        /* initialize rand_num for shuffling */
rand_num = rands[DIM_RAND-1] ;
}

/* get a long initial seed for generator
assumes that rand returns a short integer */
long get_init_rand(int max_value)
{
    long seed;

    srand((unsigned int)time(NULL)) ; /* initialize system generator */
do {
    seed = ((long)rand())*rand() ;
    seed += ((long)rand())*rand() ;
} while (seed > max_value);

    assert(seed > 0) ;
    return seed ;
}

/* generate the difference between random numbers
assumes 0 < rand1 < MOD1
         0 < rand2 < MOD2
output  1 <= rand_num <= MOD_COMB */
long genr_rand_diff(void)
{
    extern long rand1, rand2;
    long rand_new ;

    rand1 = GENR1(rand1) ;
    rand2 = GENR2(rand2) ;
    rand_new = rand1 - rand2 ;
    if (rand_new <= 0)
        rand_new += MOD_COMB ;

    assert(rand_new >= 1 && rand_new <= MOD_COMB) ;

    return rand_new ;
}

/* generate the next value in sequence from generator
uses approximate factoring
residue = (a * x) mod modulus
         = a*x - [(a*x)/modulus]*modulus

where
[(a*x)/modulus] = integer less than or equal to (a*x)/modulus
approximate factoring avoids overflow associated with a*x and
uses equivalence of above with
residue = a * (x - q * k) - r * k + (k-k1) * modulus
where
    modulus = a * q + r
    q = [modulus/a]
    k = [x/q]                (=[ax/aq])
    k1 = [a*x/m]
assumes
a, m > 0
0 < init_rand < modulus
a * a <= modulus
[a*x/a*q]-[a*x/modulus] <= 1
(for only one addition of modulus below) */
long genr_rand(long a, long x, long modulus, long q, long r)
{
    long k, residue ;

    k = x / q ;
    residue = a * (x - q * k) - r * k ;
    if (residue < 0)
        residue += modulus ;

    assert(residue >= 1 && residue <= modulus-1);
}

```

```

    return residue ;
}

/* get a random number from rands[] and replace it*/
long rand_comb(void)
{
    extern long rand_num ;
    extern long rands[] ;
    int i ;
        /* if not initialized, do it now */
    if (rand_num == IMPOSSIBLE_RAND) {
        rand_num = 1 ;
        init_rand_comb(&rand_num, &rand_num) ;
    }

    /* rand_num from previous call determines next rand_num from
    rands[] */
    i = (int) (((double)DIM_RAND*rand_num)/(double)(MAX_VALUE)) ;
    rand_num = rands[i] ;

        /* replace random number used */
    rands[i] = genr_rand_diff();

    return rand_num ;
}

#ifdef TESTING
/* Test the generator */
#include <stdio.h>
void main(void)
{
    long seed1=1, seed2=1 ;
    int i ;

    init_rand_comb(&seed1, &seed2);
    printf("Seeds for random number generator are %ld  %ld\n",
        seed1, seed2) ;
    i = STARTUP_RANDS + DIM_RAND ;
    do {
        rand_comb();
        i++;
    } while (i < 9999) ;

    printf("On draw 10000, random number should be %ld\n", EXP_VAL) ;
    printf("On draw %d, random number is  %ld\n", i+1, rand_comb()) ;
}
#endif TESTING
/* End of File */

```

Approximate Factoring

Suppose you want to compute $(a*x) \bmod m$ where a and x are positive and less than the modulus m . If integer overflow is not an issue, you can do this by computing $k1 = a*x/m$ where, $a*x/m$ signifies the largest integer less than or equal to $a*x/m$, and then computing the result $k1*m$, called the residue. Suppose, however, that the intermediate value $a*x$ would overflow. The modulus m can always be approximately factored into $m = a*q + r$ with $q = m/a$. Now calculate $k = ax/aq$. Note that k is x/q , thereby avoiding this overflow of $a*x$. Using the fact that $m = a*q+r$, you can see that the residue is:

$$a*x - k1*m$$

or

$$a*(x - k*q) - k*r + (k-k1)*m$$

If $a*a$ is less than m , $k-k1$ is either zero or one, and x is positive and less than m , the following code fragment using signed integers computes the residue correctly without overflow:

```

k = x / q;
residue = a* (x - k * q) - k * r;
if (residue < 0)
    residue += m;

```

The check against only negative values can be used in computing the residue if zero values are impossible, as they are if m is prime.

Table 1 Good multipliers for a Lehmer Generator using approximate factoring

Modulus	Multiplier	q	r	10,000th Draw
---------	------------	---	---	---------------

Alone

231-1	41,358	51,924	10,855	1,285,562,981
231-1	48,271	44,488	3,399	399,268,537
231-1	69,621	30,845	23,902	190,055,451
231-1	16,807	127,773	2,836	1,043,618,065

In combination generator

2,147,483,563	40,014	53,668	12,211	1,919,456,777
2,147,483,399	40,692	52,774	3,791	2,006,618,587

Combination generator with shuffling 804,307,721

Sources: Park and Miller [5]; L'Écuyer [2, 3].

Copyright © 2010 [United Business Media LLC](#)

