Random Numbers

Random numbers are used to develop video and casino games, business applications, and scientific simulation models. They are also used throughout the book to generate data in programs that test the performance of containers and algorithms. Computers represent chance events by generating integer or real random numbers that fall within a specified range. Integer random numbers lie in a range 0 to n-1, where n can be specified by the user or the software. Real random numbers lie in the interval $0 \le x < 1$. While mathematics specifies a number of conditions for numbers to be random, a good intuitive description implies that numbers are random when values in their range have an equal chance or probability of occurring. For instance, integer numbers in the range 0 to 9 are random when each value has a 1/10 probability of occurring. Real numbers are random when ½ are likely to be greater than 0.5, ¼ are likely to be less than 0.25, and so forth.

A Random Number Sequence A computer random number generator is designed to create a very long sequence of random values. The generator begins with an initial data value, called the *seed*, and uses arithmetic calculations to produce the random number. The algorithm updates the value of the seed and uses the new value in the next calculation of the random number. The process is *deterministic*, since it takes an initial value and executes a fixed set of instructions. Since the entire process is dependent on the initial value of the seed, we say that a computer produces *pseudo-random numbers***.**

Mathematics provides a variety of algorithms to produce a uniform sequence of random numbers. In the implementation of the randomNumber class, we use a simple algorithm called the *linear multiplicative congruential generator*. The algorithm uses constants M and A, where M is a prime and $2 \le A < M$. Recall that M is a prime number provided M >= 2 and M has exactly two divisors , 1 and M. Starting with an initial seed $x_0 > 0$, the algorithm produces the sequence x_1, x_2, x_3, . . . satisfying the equation

$$x_i = (A*x_{i-1}) \% M, i \ge 1$$

The seed x_0 cannot be 0, since all subsequent values x_i will be 0. The remainder operation produces a result that is in the range 0 through M-1. With these facts, we can deduce that all values x_i are not 0. To see this, note that x_i is the remainder after division of $(A*x_{i-1})$ by M. It must be the case that

$$Ax_{i-1} = kM + x_i,$$

where k is the quotient $k = (Ax_{i-1})/M$. Assume that x_i is 0 at some first index i. Our relationship says that

$$Ax_{i-1} = kM$$

and so M divides Ax_{i-1}. Since $2 \le A < M$ and M is a prime, it follows that M must be a prime factor of x_{i-1}. However, $0 = x_{i-1} < M$, so this is impossible. It follows that $x_i > 0$ for any seed $x_0 > 0$.

The overall effectiveness of the generator depends on the choice of the constants A and M. A good choice for M is the largest positive 32-bit integer, $2^{31}-1 = 2147483647$, which happens to be a prime number. With the value A = 48271, the random number generator produces values x_i in the range 1 to $2^{31}-2$ and is a *full period* random number generator. This means that every number from 1 through $2^{31}-2$ is produced before a number repeats.

As the generator is used, the product $A*x_{i-1}$ may eventually become large enough to overflow and become negative. The computation of $(A*x_{i-1}) \% M$ must be done in such a way that overflow is avoided. An application of some mathematics verifies the following formula, which can be computed with no overflow.

$$tmpSeed = A * (x_{i-1} \% Q) - R * (x_{i-1} / Q)$$
$$(A*x_{i-1}) \% M = tmpSeed + \delta M,$$

where

$$d = \begin{cases} 0, & tmpSeed \ge 0 \\ 1, & tmpSeed < 0 \end{cases}$$

and

$Q = M / A,\ R = M \% A$

The implementation of the randomNumb er class includes constants that define the values for A and M along with the two additional constant values Q and M. The values are declared *static* so all randomNumber objects will share one copy of the constant values.

Declaration of the randomNumber Class

The randomNumber class has a constructor that is passed a non-negative integer value to determine the seed for the generator. If the default value 0 is used, the constructor uses the number of seconds elapsed since midnight, January 1, 1970, called *coordinated universal time*, to determine the seed. If the program is run at different times, each run will produce a different random sequence. This type of seed is appropriate for game simulation, such as simulation of a card game. A constructor argument other than 0 is assigned as the seed and allows the user to control the random sequence. Use this approach for applications that require the same random values for different runs. For instance, a flight simulator uses the same sequence of random numbers to compare the effectiveness of two pilots responding to an in-flight situation. Each pilot is subjected to the same sequence of events.

The member functions in the class produce integer and real random numbers. The function frandom() returns a random real number in the range $0 \le x < 1$. For integers, the programmer may use two versions of the function random(). The first version does not have an argument and returns a 32-bit random integer in the range $1 \le x < 2^{31}-2$. The second version takes an argument, n, and returns a random integer in the range $0 \le x \le n-1$. The following is the declaration of the randomNumber class.

CLASS randomNumber   Declaration   "d_random.h"

```
// generate random numbers
class randomNumber
{
   public:
      // initialize the random number generator
      randomNumber(long s = 0);

      // return a 32-bit random integer m, 1 <= m <= 2^31-2
      long random();

      // return a 32-bit random integer m, 0 <= m <= n-1,
      // where n <= 2^31-1
      long random(long n);

      // return a real number x, 0 <= x < 1
      double frandom();

   private:
      static const long A;
      static const long M;
      static const long Q;
      static const long R;
      long seed;
};

const long randomNumber::A = 48271;
const long randomNumber::M = 2147483647;
const long randomNumber::Q = M / A;
const long randomNumber::R = M % A;
```

EXAMPLE

1. The random number object rndA is seeded by the system clock, while the generator rndB is provided a user-defined seed of 100.

```
randomNumber rndA, rndB(100);
```

2. The loop generates 5 integer random numb ers in the range 0 to 40 and 5 real random numbers

```
int item, i;
double x;

for (i = 0; i < 5; i++)
{
   item = rndA.random(40);    // 0 <= item < 40
   x = rndB.frandom();        // 0 <= x < 1

   cout << item << "  " << x << endl;
}

Output:
   20  0.00224779
   38  0.503245
   23  0.135261
   5  0.161128
   11  0.79557
```

3. (a) The random number object rndA is used to produce a random integer representing the value of a single die. The random integer is in the range 0 to 5. By adding 1, the range is shifted to 1 to 6 which are the die values.

```
dieValue = rndA.random(6) + 1;         // 1 <= dieValue <= 6
```

   (b) A building site saves all left-over scraps of 2 x 4 lumber that is at least 1.5 feet. A real random number simulates picking a scrap board in the range 1.5 to 4.5 feet.

```
// multiply by 3 to get a random length; add 1.5 to shift the range
boardLength = rndA.frandom() * 3 + 1.5.
```

PROGRAM ILLUSTRATING RANDOM NUMBERS

The program illustrates the "randomness" of values produced by the member functions random() and frandom(). A loop generates one million random integers in the range 0 to 4 and computes the number of times each possible outcome occurs. The count is stored in array intCount where intCount[0] is the number of times a 0 occurs, intCount[1] is the number of times a 1 occurs, and so forth. Each iteration of the loop also generates a real number and maintains separate totals, under25Count and over60Count, that record the number of times the value is less than 0.25 or greater than 0.60 respectively. For output, the program lists the individual counts and their percentage as a fraction of the total number of values (one-million). In the run, note that each of the five different integer outcomes occurs approximately 20% of the time. A real number is less than 0.25 approximately 25% of the time and greater than 0.60 approximately 40% of the time.

```
#include <iostream>
#include <iomanip>         // for manipulator setw()

#include "d_random.h"      // for randomNumber

using namespace std;

int main()
{
   randomNumber rnd;

   // declare counters and set to 0
   long intCount [5] = {0, 0, 0, 0, 0};
   long under25Count = 0, over60Count = 0;
   int i;

   // floating point values for percentages
   double pctInt, pctUnder25, pctOver60;

   // object that holds a random number
   double x;

   // generate 1,000,000 integer and 1,000,000 real numbers
   for (i = 0; i < 1000000; i++)
   {
      // value in range 0 to 4 is used as index
      intCount [rnd.random(5)]++;

      // generate real number and test location in an interval
      x = rnd.frandom();
      if (x < 0.25)
         under25Count ++;
      else if (x >= 0.6)
         over60Count++;
   }

   // header for the table
   cout << "Number        Count         Pct" << endl;

   // output each array element as a count and as a percentage
   for (i = 0; i < 5; i++)
   {
      // compute as percentage of 1,000,000
      pctInt = (intCount [i]/1000000.00) * 100;
      cout << setw(3) << i << setw(16) << intCount [i]
           << setw(13) << pctInt << "%" << endl;
   }
   cout << endl;
   // output counts for real numbers along with percentage
   pctUnder25 = (under25Count/1000000.0) * 100;
   cout << under25Count << " real random numbers <  0.25 "
        << setw(7) << " ("
        << pctUnder25 << "%)" << endl;

   pctOver60 = (over60Count/1000000.0) * 100;
   cout << over60Count << " real random numbers >= 0.60 "
        << setw(7) << " ("
        << pctOver60 << "%)" << endl << endl;

   return 0;
}
```

```
Run:

Number        Count         Pct
  0          199683      19.9683%
  1          199932      19.9932%
  2          200211      20.0211%
  3          200409      20.0409%
  4          199765      19.9765%

249810 real random numbers <  0.25       (24.981%)
399780 real random numbers >= 0.60       (39.978%)
```