Random Number Generators and related classes

The two classes **RNG** and **Random** are used together to generate a variety of random number distributions. A distinction must be made between *random number generators*, implemented by class **RNG**, and *random number distributions*. A random number generator produces a series of randomly ordered bits. These bits can be used directly, or cast to other representations, such as a floating point value. A random number generator should produce a *uniform* distribution. A random number distribution, on the other hand, uses the randomly generated bits of a generator to produce numbers from a distribution with specific properties. Each instance of **Random** uses an instance of class **RNG** to provide the raw, uniform distribution used to produce the specific distribution. Several instances of **Random** classes can share the same instance of **RNG**, or each instance can use its own copy.

RNG

Random distributions are constructed from members of class **RNG**, the actual random number generators. The **RNG** class contains no data; it only serves to define the interface to random number generators. The **RNG::asLong** member returns an unsigned long (typically 32 bits) of random bits. Applications that require a number of random bits can use this directly. More often, these random bits are transformed to a uniform random number:

```
//
// Return random bits converted to either a float or a double
//
float asFloat();
double asDouble();
```

using either **asFloat** or **asDouble**. It is intended that **asFloat** and **asDouble** return differing precisions; typically, **asDouble** will draw two random longwords and transform them into a legal **double**, while **asFloat** will

draw a single longword and transform it into a legal **float**. These members are used by subclasses of the **Random** class to implement a variety of random number distributions.

ACG

Class **ACG** is a variant of a Linear Congruential Generator (Algorithm M) described in Knuth, *Art of Computer Programming, Vol III*. This result is permuted with a Fibonacci Additive Congruential Generator to get good independence between samples. This is a very high quality random number generator, although it requires a fair amount of memory for each instance of the generator.

The **ACG::ACG** constructor takes two parameters: the seed and the size. The seed is any number to be used as an initial seed. The performance of the generator depends on having a distribution of bits through the seed. If you choose a number in the range of 0 to 31, a seed with more bits is chosen. Other values are deterministically modified to give a better distribution of bits. This provides a good random number generator while still allowing a sequence to be repeated given the same initial seed.

The **size** parameter determines the size of two tables used in the generator. The first table is used in the Additive Generator; see the algorithm in Knuth for more information. In general, this table is **size** longwords long. The default value, used in the algorithm in Knuth, gives a table of 220 bytes. The table size affects the period of the generators; smaller values give shorter periods and larger tables give longer periods. The smallest table size is 7 longwords, and the longest is 98 longwords. The **size** parameter also determines the size of the table used for the Linear Congruential Generator. This value is chosen implicitly based on the size of the Additive Congruential Generator table. It is two powers of two larger than the power of two that is larger than **size**. For example, if **size** is 7, the ACG table is 7 longwords and the LCG table is 128 longwords. Thus, the default size (55) requires 55 + 256 longwords, or 1244 bytes. The largest table requires 2440 bytes and the smallest table requires 100 bytes. Applications that require a large number of generators or applications that aren't so fussy about the quality of the generator may elect to use the **MLCG** generator.

MLCG

The **MLCG** class implements a *Multiplicative Linear Congruential Generator*. In particular, it is an implementation of the double MLCG described in ``Efficient and Portable Combined Random Number Generators'' by Pierre L'Ecuyer, appearing in *Communications of the ACM, Vol. 31. No. 6*. This generator has a fairly long period, and has been statistically analyzed to show that it gives good inter-sample independence.

The **MLCG::MLCG** constructor has two parameters, both of which are seeds for the generator. As in the **MLCG** generator, both seeds are modified to give a ``better'' distribution of seed digits. Thus, you can safely use values such as `0' or `1' for the seeds. The **MLCG** generator used much less state than the **ACG** generator; only two longwords (8 bytes) are needed for each generator.

Random

A random number generator may be declared by first declaring a **RNG** and then a **Random**. For example, **ACG gen(10, 20); NegativeExpntl rnd (1.0, &gen);** declares an additive congruential generator with seed 10 and table size 20, that is used to generate exponentially distributed values with mean of 1.0.

The virtual member **Random::operator()** is the common way of extracting a random number from a particular distribution. The base class, **Random** does not implement **operator()**. This is performed by each of the subclasses. Thus, given the above declaration of **rnd**, new random values may be obtained via, for example, **double next_exp_rand = rnd();** Currently, the following subclasses are provided.

Binomial

The binomial distribution models successfully drawing items from a pool. The first parameter to the constructor, **n**, is the number of items in the pool, and the second parameter, **u**, is the probability of each item being successfully drawn. The member **asDouble** returns the number of samples drawn from the pool. Although it is not checked, it is assumed that **n>0** and **0 <= u <= 1**. The remaining members allow you to read and set the parameters.

Erlang

The **Erlang** class implements an Erlang distribution with mean **mean** and variance **variance**.

Geometric

The **Geometric** class implements a discrete geometric distribution. The first parameter to the constructor, **mean**, is the mean of the distribution. Although it is not checked, it is assumed that **0 <= mean <= 1**. **Geometric()** returns the number of uniform random samples that were drawn before the sample was larger than **mean**. This quantity is always greater than zero.

HyperGeometric

The **HyperGeometric** class implements the hypergeometric distribution. The first parameter to the constructor, **mean**, is the mean and the second, **variance**, is the variance. The remaining members allow you to inspect

and change the mean and variance.

NegativeExpntl

The **NegativeExpntl** class implements the negative exponential distribution. The first parameter to the constructor is the mean. The remaining members allow you to inspect and change the mean.

Normal

The **Normal** class implements the normal distribution. The first parameter to the constructor, **mean**, is the mean and the second, **variance**, is the variance. The remaining members allow you to inspect and change the mean and variance. The **LogNormal** class is a subclass of **Normal**.

LogNormal

The **LogNormal**class implements the logarithmic normal distribution. The first parameter to the constructor, **mean**, is the mean and the second, **variance**, is the variance. The remaining members allow you to inspect and change the mean and variance. The **LogNormal** class is a subclass of **Normal**.

Poisson

The **Poisson** class implements the poisson distribution. The first parameter to the constructor is the mean. The remaining members allow you to inspect and change the mean.

DiscreteUniform

The **DiscreteUniform** class implements a uniform random variable over the closed interval ranging from **[low..high]**. The first parameter to the constructor is **low**, and the second is **high**, although the order of these may be reversed. The remaining members allow you to inspect and change **low** and **high**.

Uniform

The **Uniform** class implements a uniform random variable over the open interval ranging from **[low..high]**. The first parameter to the constructor is **low**, and the second is **high**, although the order of these may be reversed. The remaining members allow you to inspect and change **low** and **high**.

Weibull

The **Weibull** class implements a weibull distribution with parameters **alpha** and **beta**. The first parameter to the class constructor is **alpha**, and the second parameter is **beta**. The remaining members allow you to inspect and change **alpha** and **beta**.

RandomInteger

The **RandomInteger** class is *not* a subclass of Random, but a stand-alone integer-oriented class that is dependent on the RNG classes. RandomInteger returns random integers uniformly from the closed interval **[low..high]**. The first parameter to the constructor is **low**, and the second is **high**, although both are optional. The last argument is always a generator. Additional members allow you to inspect and change **low** and **high**. Random integers are generated using **asInt()** or **asLong()**. Operator syntax (**()**) is also available as a shorthand for **asLong()**. Because **RandomInteger** is often used in simulations for which uniform random integers are desired over a variety of ranges, **asLong()** and **asInt** have **high** as an optional argument. Using this optional argument produces a single value from the new range, but does not change the default range.