ThinAir
A Pseudo-Random Number Generator Class Library
Version 0.9b

G. Wade Johnson

February 20, 1997

Contents

1 Introduction

Unfortunately, most of the pseudorandom number sequence generators (PRNGs) available to the average programmer are not very good. Usually it's not easy to learn which generator is provided. This makes finding the weaknesses or problems with a particular generator almost impossible. If you realize the potential problems with a provided generator, you usually don't know how to go about testing and verifying another PRNG.

Even if you have the knowledge and experience to choose a good PRNG, you usually aren't completely satisfied. Once you've chosen a PRNG, changing to another is not easy. This makes verifying a simulation using different sources of random numbers a difficult proposition.

This document describes the ThinAir class library that defines a standard interface for PRNGs. The ThinAir library also provides several generators, each with its own strengths and weaknesses. This standard interface allows you to change generators relatively easily.

In many simulations, it is difficult to determine what are results and what are merely artifacts of the random number sequence supplied to the simulation. Some of the *bad* generators listed in Section 10 can serve as controls for these situations.

The ThinAir library and this documentation supply different things for different people. There are several different ways of using this library, ranging from *I need a generator now!* to *I need to build a completely new kind of generator.* The following chart should help you find what you need.

I need a generator now!	Section 2
I need other generators now.	Section 8
I'm already using *rand()*.	Section 5.2 and the FAQ
I want to learn about random number sequences.	Section 3 and the FAQ
I want to choose a generator.	Section 4
I want to understand the whole library.	Section 6 first
I want to modify a generator.	Section 13.4
I want to build a new generator.	Section 13.5
I need a nonrandom sequence.	Section 10

2 Quick Start

The ThinAir library contains more than 40 random number generators. This section lists a half-dozen good generators and how to call them from your program. If you want to know some of the tradeoffs of different generators or more advanced ways of using them, you'll need to read beyond this section.

The simplest way to use this library is to pick a generator and use the *rand()* interface to access it. You can accomplish this using a few simple steps.

1. Include the ThinAir **rand_rng.h** header file in any file where you use *rand()*, *random()*, *randomize()*, *srand()*, or *MAX_RAND*.

2. At the beginning of your code, call the function *setup_rand()* with a heap-based your chosen generator as the argument.

3. Use the functions *rand()*, *random()*, *randomize()*, and *srand()* as described in the C++ standard library documentation.

Here are six examples of the correct use of this function. Each of these generators take an optional seed argument when they are constructed.

```
setup_rand( new Knuth27RandGen() );                  // found in erndmcnm.h
setup_rand( new BoroshNiederreiterRandGen() );       // found in erndmcon.h
setup_rand( new WatermanRandGen() );                 // found in erndmcon.h
setup_rand( new LavauxJanssensRandGen() );           // found in erndmcon.h
setup_rand( new MarsagliaMCRandGen() );              // found in erndmcon.h
setup_rand( new Knuth28RandGen() );                  // found in erndsecm.h
```

Unlike the standard *rand()* function that returns *short ints*, the ThinAir *rand()* function returns *long ints*. If you use the *random(limit)* function instead of *rand()*, you will not notice any difference. For an example and further information on this interface, see Section 5.2.

3 Characteristics of Random Number Sequences

What do you look for in a random number sequence? The easiest answer to this question is *unpredictability*. But how do you measure unpredictablility? One way to determine the unpredictability of a number sequence is to look for characteristics of a perfectly random sequence of numbers. What *are* the characteristics of a good random sequence? Which characteristics of a random sequence are important?

Once we know what characteristics are desirable in a random sequence, we can evaluate PRNGs in terms of the sequences they generate. Just as important, we can decide which characteristics do not affect our use of the PRNG. If a particular characteristic has no effect on your problem, there is no need to consider that characteristic when evaluating PRNGs.

3.1 Distribution

Probably the most fundamental characteristic of a sequence of numbers is its *distribution*. The PRNGs in the ThinAir library (with a few exceptions) produce *uniformly distributed* number sequences. Other distributions include *normal*, *Poisson*, *geometric*, *binomial*, and *student-t*. Other authors have published algorithms that convert uniformly distributed number sequences into other distributions. (See [4, pages 131-133])

3.2 Range

Another fundamental characteristic of a random number sequence is its *range*. What is the smallest number in the sequence? What is the largest? These two numbers define the range of the whole sequence. The range of a sequence is more important and more subtle than most people realize. If the range of a random number sequence is 0 to 64, it is useless for generating random percentages. One good example of a bad range problem appears in some simple card games. In these examples, a 16-bit random number generator is used to choose a layout of a deck of cards. The problem lies in the fact that the generator can produce at most 65,536 different numbers. A deck of cards can produce $80.66 * 10^{66}$ different layouts. If you are very careful in your use of the generator, this problem is solvable. If not, most of the possible layouts never occur.

3.3 Type

Another useful characteristic of a random sequence is its *type*. What kind of numbers make up this sequence. Are we talking about bits, integers, floating point numbers, strings, or something else entirely. Most of these types can convert into each other relatively easily. In general, PRNGs fall into three categories:

1. PRNGs that generate a sequence of integers.
2. PRNGs that generate a sequence of floating point numbers between 0.0 and 1.0.
3. PRNGs that generate a stream of bits.

3.4 Period

All algorithm-based PRNGs repeat if they are allowed to run for long enough. This is caused by the fact that the generator has a finite amount of internal *state*, and by the fact that an algorithm cannot produce truly random output. Since we know the sequence must repeat, we would obviously like to have it repeat after as long a time as possible. The number of individual numbers generated by the PRNG before it begins to repeat is known as the *period* of the sequence. This sequence of numbers is often called a *cycle*. The amount of internal state retained by a PRNG defines the upper limit on the period. A PRNG with a period equal to the maximum predicted by its internal state is sometimes called a *maximal-period generator*.

Depending on the algorithm used, a PRNG may actually generate more than one cycle. The more individual cycles a generator has, the shorter the period each cycle has. The best generator would, of course, have a single cycle that was longer than we could ever use (effectively infinite). Some multiplicative congruential random number generators have two cycles, one being very long (approximately $2^{32}-1$ numbers in the sequence), the other being a cycle of period 1 at 0.

3.5 Coverage

The term *coverage* describes how well a given sequence covers its output range. For example, a PRNG with an output range of 0 to $2^{32}-1$ should generate every number between 0 and $2^{32}-1$ if allowed to run for long enough. Not all PRNGs cover their entire output range.

If a PRNG has multiple cycles, the coverage issue is even more interesting. The generator may actually be capable of covering the whole range of output values. However, only certain values are covered in each cycle. The multiplicative congruential PRNGs mentioned above have a maximum coverage one less than expected in the main cycle. This is because an output of zero is never part of the main cycle.

Many tests have been devised to study coverage. Two of these are the *equidistribution*, or *frequency*, test and the *gap* test. The equidistribution test tries to verify that all of the numbers in the sequence are equally likely. The gap test studies the probabilities of gaps of different sizes between numbers in a particular range.

3.6 Runs

One interesting characteristic of a sequence of numbers concerns *runs*. When each successive number in a subsequence is larger than the previous number, we have a *run up*. When each successive number in a subsequence is smaller than the previous number, we have a *run down*. In a good random sequence we would expect the number of runs up to be pretty close to the number of runs down. We would also expect that longer runs would be less likely than shorter runs.

3.7 Serial Correlation

A good sequence of random numbers should not have any *correlation* between successive numbers. This means that you should not be able to predict the next number in the sequence by looking at previous numbers. The classic example of serial correlation in a PRNG is the linear congruential generator. When a linear congruential PRNG is used to generate pairs of numbers for use as x, y coordinates on a graph, a pattern often shows up in the graph. The pairs of numbers tend to fall on a set of parallel lines on the graph. This *artifact* is a simple result of the algorithm used.

One way to test for high serial correlation is the *Poker* test. The Poker test looks at the combinations of five successive numbers taken from the sequence. These numbers are analyzed for pairs, three-of-a-kind, full house, etc. The probabilities of these combinations should approach that of a random number stream.

3.8 Permutations

In a truly random number stream, any permutation of a set of numbers is as likely as any other permutation of the same numbers. If one permutation of a set of numbers is more likely than another, a few numbers in the set may supply enough information to help predict following numbers.

3.9 Spectral Characteristics

Even if a sequence of numbers does not exhibit any obvious patterns, it may have higher order artifacts. A truly random sequence would have no particular frequency component more pronounced than any other. Since any PRNGs must repeat after some length of time, there is at least one dominant frequency component. Studying the spectral characteristics tells us if there are any others.

4 Choosing a Random Number Generator

What you need in a PRNG depends on what you will use it for. If you need three unrelated numbers, you would probably not use the same generator as someone building a world-class crypto-system. Likewise, a solitare game has different requirements than a numeric integration package.

If you need a random sequence of numbers for a cryptographic system, almost none of the generators in the ThinAir library are useful. The same applies to a password-generation system. It is possible to use a PRNG in either of these applications, but the PRNG is only a small part of the overall system. The selection of a cryptographically secure random number source and its use in cryptography or password generation is far beyond the scope of this document. See [8] for more info.

4.1 Instant Gratification

In many cases, you don't want to know everything (or maybe anything) about the theory of these PRNGs. You just need a generator, now. This section should help to steer you in the direction of generators you can use right away.

If all you need is a generator to solve a problem right now, see Section 2 for six good, simple generators. All of these generators have good characteristics. They are also relatively fast and require no special knowledge to use. If you need a more powerful generator and don't need blinding speed, try KNUTH28RANDGEN (p. 22) or LINMULTWCARRYRANDGEN (p. 21). These generators are slightly slower than the common linear congruential PRNG, but have somewhat better characteristics.

If you need an extremely long period, try the MITCHELLMOORERANDGEN (p. 23) generator. It has been well studied and has a period on the order of $2^{85}-1$. If you requested one number from this generator every nanosecond, this PRNG would repeat after 1.2 billion years! At a similar sampling rate, most of the PRNGs in this library would repeat in less than 5 seconds. In actuality, you can't use any of the PRNGs in this library at this speed, but it is something to think about.

If you don't mind a slower generator and want practically no correlation between successive numbers, you should consider the *Secure Hash Algorithm* based generators, SHACTRRANDGEN (p. 24) and SHARNGRANDGEN (p. 24). These generators require more seed information than the simpler generators. These generators use the Secure Hash Algorithm (see [9]) to generate the numbers in the sequence. This one-way function should make it practically impossible to determine a number in the sequence, even given a large number of preceeding values.

If you want to test your code with a truly bad generator to see how that would affect your output, look in the file **randnot.h**. (Section 10) All of these generators are really bad. For a torture test of your code, use the CONSTANTRANDGEN (p. 25) or COUNTERRANDGEN (p. 25) classes. These generators are worst-case scenarios for most programs.

4.2 A little bit of work

If you don't mind a little learning curve, many more generators are available with only a minor amount of configuration. One downside of this approach is that you can easily come up with a bad generator if you don't have some background in PRNG theory. Those of you who are familiar with the various issues that may arise in customizing PRNGs might be interested in some of these.

If you are looking for the classics, maybe I could interest you in variations of the linear congruential PRNG. The classes LINCONRANDGEN (p. 14), LINCONMODRANDGEN (p. 14), MULTCONRANDGEN (p. 15), and MULTCONMODRANDGEN (p. 15) supply various versions of this generator. The most configurable of these is the LinConModRandGen class, which allows selection of multiplier, constant term, and modulus. The least configurable is MultConRandGen which only allows you to specify a multiplier.

Choosing the various parameters for a the linear congruential PRNGs is not a trivial exercise. Many sets of parameters have been tested over time and yield good results. You can find lists of some of these in [4] and [8]. In addition, Knuth gives good rules to use in choosing your own parameters in Section 3.2 of [4].

What if you're worried about the linear artifacts produced by the linear congruential PRNG, try some of the nonlinear congruential generators. For instance, SecConRandGen (p. 15), SecConModRandGen (p. 16), InvConRandGen (p. 17), InvConModRandGen (p. 16), and even QuadConRandGen (p. 17) replace the obvious linear artifacts with more subtle effects.

If you would like to use a table of random numbers from another source, look at TextTableRandGen (p. 12), BinaryTableRandGen (p. 18), and their children. One particularly interesting use of these generator types involves the DecimalTableRandGen (p. 18) and the file **rand.rnt**. This is the well-known Rand table of random digits (published in 1955). This combination allows you to supply this standard set of random numbers to your code while still using the ThinAir interfaces.

4.3 Willing to work for it

If you have a particular kind of generator in mind that ThinAir doesn't immediately provide, the library is extensible enough to suit almost anyone. It is often possible to combine a few simple PRNGs to create a better one. The CombineRandGen (p. 13) and its children simplify this task. Using the DifferenceRandGen (p. 19) or ShuffledMRandGen (p. 19) classes, requires more setup than most of the other generators. However, careful choices in their configuration may result in a fast generator with good characteristics. Then again, there's always the possibility of building your own generator based on ThinAir.

People with a method of generating random sequences from hardware might want to take a look at the file-based PRNGs, TextTableRandGen (p. 12), BinaryTableRandGen (p. 18), and their children. You could write the output of a hardware sequence generator to a file and use these classes to feed them to your code. Then again, you might want to look at Section 13.6 to see how to create a new generator that hooks directly to your hardware. Of course, a hardware-based generator may not support tokens since there is no way to replay a sequence.

5 The Class Interfaces

A library is only as good as its interface. A powerful interface often requires a learning curve. A simple interface limits sophisticated usage of the library. ThinAir solves this problem by providing three different interfaces, each with different strengths and weaknesses. The main interface to ThinAir is provided by the base class of the library, RandomGenerator. This interface is powerful and easy to use. Since most C and C++ programmers are familiar with the *rand()* function interface, ThinAir provides this interface. However, the *rand()* interface is not very powerful or flexible. ThinAir also provides an interface based on function objects.

5.1 The RandomGenerator Interface

The interface for all of the PRNG classes in the ThinAir library is defined by the RandomGenerator abstract base class. This class is implemented in the files **randgen.h** and **randgen.cpp**. The constructor for the class has the following prototype:

RandomGenerator(void)
  Does nothing.

The class also provides default implementation for some of the more general member functions. The members of this interface are

size_rand MaxRandom() const;
  Returns the largest number that a given PRNG can ever generate.

BOOL bad(void) const;
: This function returns *true* if the generator is unable to function properly. Most algorithm-based generators will always return *false*. This member function is most useful for generators which require some resource to run. These generators would return *true* from *bad()* if they are unable to acquire the needed resource.

size_rand IntSeed(size_rand seed);
: Modifies a generator in a class-specific way based on a single number. If possible, *IntSeed()* should have the same effect every time a particular *seed* value is used.

size_rand Number();
: Returns a random number between 0 and *MaxRandom()*, inclusive.

double FloatNumber();
: Returns a random *double* number greater than or equal to 0.0 and less than 1.0.

size_rand LimitedNumber(size_rand limit);
: Returns a random number between 0 and *limit*-1, inclusive.

void Reset();
: Resets the generator to its initial conditions.

RandomGenerator *ReCreate() const;
: Clones the generator in its initial state.

RandomGenerator *Clone() const;
: Clones the generator complete with its current state.

When performing simulations, it is often useful to rerun the same sequence of random numbers through two different tests to compare responses. The RANDGENTOKEN class allows you to save the current state of a PRNG object for later restoration. This is an instance of the *Token* design pattern described in [3]. The member functions of RANDOMGENERATOR that support this pattern are

RandGenToken CreateToken() const;
: Creates a RANDGENTOKEN object that stores the current state of the PRNG object.

int SetToken(const RandGenToken &pToken);
: Restores the PRNG object to the state saved in *pToken*.

Additionally, the token classes also have a useful public interface:

string CreateInitString() const;
: Creates an initialization string that describes the initial state of the generator associated with this token.

int Initialize(const string &InitString);
: Takes the initialization string created by *CreateInitString()* and reinitializes the token.

int Initialize(istream &InitStream);
: Same as above except that the function is passed an input stream. This is the more flexible of the two functions, but is often not as convenient.

The RANDGENTOKEN class supports extraction from and insertion into streams. In addition, the token classes have a pair of protected functions that support the stream functions.

istream &operator>>(istream &is, RandGenToken &Token);
: Extracts the token from the istream *is* using *Read()*.

ostream &operator<<(ostream &os, RandGenToken const &Token);
: Inserts the token into the ostream *os* using *Print()*.

void   Print(ostream &os) const;
Writes the token's internal data to the ostream *os*.

void   Read(istream &is);
Reads the token's internal data from the istream *is*.

Each kind of PRNG has a token class derived from RandGenToken. However, the definition of the token classes are private to the PRNG class they support. There should never be any good reason for digging into the token class definitions, except for the creation of new generator types.

5.2 The rand() Interface

Many programs written in C/C++ that need random number sequences use the standard library *rand()* function and its support functions. What you may not have realized is that this function uses an implementation-defined generator that may or may not be any good. So what do you do if your code already uses *rand()*, and you would like to try ThinAir?

This library contains a set of functions to duplicate the *rand()* function interface wrapped around one of the classes from this library. The idea is to provide the interface an average C programmer might find familiar, with a simple method of changing the generator behind it.

By including the ThinAir **rand_rng.h** header file, you can have this interface for any random number class you choose. To use one of these classes with the *rand()* interface, you must call the function *setup_rand()* with a heap-based pointer to your chosen generator as the argument.

Like the standard library version of *rand()*, the PRNG class library provides the following interface, plus one extra function:

RAND_MAX
The largest number that the current generator can return.

size_rand   rand();
Returns a number between 0 and *RAND_MAX*.

size_rand   random(size_rand lim);
Returns a number between 0 and *lim*.

size_rand   srand(size_rand seed);
Applies the class-specific function *IntSeed()* to the current generator object with the argument *seed*. This should change the output sequence in some defined way.

void   randomize(void);
Calls the *srand()* function in a simple fashion to change the current generator. Like the standard library equivalent, *randomize()* uses the current time as a *seed*.

void   setup_rand(RandomGenerator *pRNG);
Plugs a new generator into the *rand()* interface. This function expects a pointer to a heap-based PRNG object.

5.3 The Function Object Interface

With support for the STL being written into the C++ standard (see [5]), function objects are becoming an important idea to consider. The STL supports a *random_shuffle()* algorithm that accepts a random number generator function object as an argument. In order to support this algorithm, the PRNG class library provides a Function Object interface for the classes. This interface also supplies other function objects which you may find useful. This interface is provided by five classes, which are defined in the file **randfunc.h**. All of these objects are used by calling the *operator()* member function.

Much of the time you want to use a PRNG the same way over and over again. These classes simplify this approach to using a PRNG.

Each of the Function Objects expects a pointer to a PRNG object that was allocated from the heap. Don't pass the address of a stack-based PRNG object to these functions. After construction, the Function Object owns the PRNG object, so you do not need to *delete* the PRNG.

5.3.1 RNGFuncAdaptor

This function object class defines an object which returns a random number from a particular PRNG object. The *operator()* member function calls the *Number()* member function of its PRNG. The constructor for this class has the following prototype:

explicit RNGFuncAdaptor(RandomGenerator *pRNG)
: Requires a pointer to a heap-based PRNG object. The RNGFuncAdaptor object owns this PRNG and *delete*s it when the object is destroyed.

5.3.2 RNGFuncAdaptorFloat

This function object class defines an object which returns a random *double* greater than or equal to 0.0 and less than 1.0. The *operator()* member function calls the *FloatNumber()* member function of this PRNG. The constructor for this class has the following prototype:

explicit RNGFuncAdaptorFloat(RandomGenerator *pRNG)
: Requires a pointer to a heap-based PRNG object. The RNGFuncAdaptorFloat object owns this PRNG and *delete*s it when the object is destroyed.

5.3.3 RNGFuncAdaptorLimit

This function object class defines an object which returns a number between 0 and a programmer-supplied limit value. The *operator()* member function calls the *LimitedNumber()* member function of this PRNG passing the selected limit. This class also has a second *operator()* member function that expects a single argument. This member function uses the supplied argument instead of the stored limit when calling *LimitedNumber()*. The constructor for this class has the following prototype:

RNGFuncAdaptorLimit(RandomGenerator *pRNG, size_rand limit)
: Requires a pointer to a heap-based PRNG object. The RNGFuncAdaptorLimit object owns this PRNG and *delete*s it when the object is destroyed. The *limit* argument defines the range of values returned by the *operator()* member function.

5.3.4 RNGFuncAdaptorPercentLikely

This function object class defines an object which returns a non-zero result a programmer-supplied percentage of the time. The *operator()* member function returns a non-zero value *percent* times out of a hundred on average. This class also has a second *operator()* member function that expects a single argument. This member uses the supplied *percent* instead of the stored one. The constructor for this class has the following prototype:

RNGFuncAdaptorPercentLikely(RandomGenerator *pRNG, unsigned percent)
: Requires a pointer to a heap-based PRNG object. The RNGFuncAdaptorPercentLikely object owns this PRNG and *delete*s it when the object is destroyed.

5.3.5 RNGFuncAdaptorRange

This function object class defines an object which returns a number between two programmer-supplied limits. The *operator()* member function returns a value between *lower* and *upper*, inclusive. The constructor for this class has the following prototype:

RNGFuncAdaptorRange(RandomGenerator *pRNG, size_rand lower, size_rand upper)
: Requires a pointer to a heap-based PRNG object. The RNGFuncAdaptorRange object owns this PRNG and *deletes* it when the object is destroyed. The arguments *lower* and *upper* define the range of values that can be returned by the *operator()* member function.

6 Abstract Classes

Many generators are based on similar models. The abstract classes described in this section formalize these similarities. If you are not interested in the construction of new generators or in the architecture of the ThinAir library, you might find Section 7, Concrete Classes more useful. In the abstract classes, the functions *Number()* and *ReCreate()* are always left undefined. None of these classes can be instantiated, they serve as base classes for other parts of the library.

6.1 RandomGenerator

The RandomGenerator abstract base class is the root of the entire hierarchy. This class interface is defined in Section 5.

6.2 CongruentialRandGen

Several types of PRNGs are based on the model of applying a simple function to the last output and taking the modulus of the result.

$$X_n = f(X_{n-1}) \bmod m$$

This result is returned as the new output. The abstract class CongruentialRandGen gathers the common features of these generators. This class is derived directly from RandomGenerator (p. 7) and is implemented in the files **randcong.h** and **randcong.cpp**. The constructor for this class has the following prototype:

explicit CongruentialRandGen(size_rand seed)
: Stores the *seed* and then initializes itself.

In addition to the public interface, the CongruentialRandGen class has a protected interface for use by its derived classes:

size_rand Seed(void) const;
: Returns the initial seed value used when the current generator was created.

size_rand CurrSeed(void) const;
: Returns the current seed value that the generator uses in creating the next random number.

void SetSeed(size_rand seed);
: Set the initial seed value used when the current generator was created.

void SetCurrSeed(size_rand currSeed);
: Set the current seed value that the generator uses to create the next random number.

6.3 TableRandGen

The TableRandGen class defines an interface for generators that read a number sequence from a table on disk. It is derived directly from RandomGenerator (p. 7) and is implemented in the files **randtabl.h** and **randtabl.cpp**. The TableRandGen class makes no assumptions about the table contained in the file. The constructor for this class has the following prototype:

TableRandGen(const char *szFileName, streampos start)
: Opens the file specified by *szFileName* and stores the value of *start*.

In addition to the public interface, the TABLERANDGEN class has a protected interface for use by its derived classes:

ifstream &Table(void);
: Returns the current file input stream used by this generator.

const char *FileName(void) const;
: Returns the name of the file used by this generator.

streampos Start(void) const;
: Returns the initial stream position specified when the current generator was created.

void SetStart(streampos start);
: Sets the initial stream position specified when the current generator was created.

void MakeBad(void);
: Marks the current generator as invalid. This member function is used by derived classes to signal an error that prevents the generator from running.

void ClearBad(void);
: Marks the current generator as valid. This member function is used by derived classes to signal that the generator can continue to run.

6.4 TextTableRandGen

The TEXTTABLERANDGEN class defines an interface for generators that read a number sequence from a text table on disk. It is derived from TABLERANDGEN (p. 11) and is implemented in the files **randtabl.h** and **randtabl.cpp.** The TEXTTABLERANDGEN class makes only two assumptions about the file. The first is that it is basically a text file. The second is that the file may have comments at the beginning of the file. The table will begin after a line containing the phrase `Begin Table`. The constructor for this class has the following prototype:

TextTableRandGen(const char *szFileName, streampos start)
: Calls the constructor for TABLERANDGEN with the arguments *szFileName* and *start*.

In addition to the public interface, the TEXTTABLERANDGEN class has a protected interface for use by its derived classes:

void InitTable(void);
: Finds the beginning of the table in the text file and moves the cursor to an offset of *start* from the start of the table.

6.5 ByteTableRandGen

The BYTETABLERANDGEN class specializes the TEXTTABLERANDGEN (p. 12) class to deal with a table of bytes, as opposed to a table containing a list of long integers. The table is expected to be a list of digits that may be used to construct the requested random number. The meaning of a given byte is not specified in this abstract class. The BYTETABLERANDGEN class is implemented in the files **randbtbl.h** and **randbtbl.cpp**. The constructor for this class has the following prototype:

ByteTableRandGen(const char *szFileName, streampos start)
: Calls the constructor for TEXTTABLERANDGEN with the arguments *szFileName* and *start*.

In addition to the public interface, the BYTETABLERANDGEN class has a protected interface for use by its derived classes:

size_rand getNumber(unsigned digits);
: Reads *digits* bytes from the table and constructs a number from these bytes.

unsigned getNumberDigits(size_rand limit);
: Returns the number of digits needed to produce a number the size of *limit*.

6.6 CombineRandGen

`(Add shrinking generator)` One of the tradeoffs in building PRNGs is speed vs. unpredictability of the sequence. One approach that has been tried to further *randomize* a fast generator is to combine the output of two fast generators in some way. The CombineRandGen abstract class defines a basic interface for a PRNG that combines the output of two other PRNGs. The CombineRandGen class is derived from RandomGenerator (p. 7) and is implemented in the files **randcomb.h** and **randcomb.cpp**. The constructor for this class has the following prototype:

CombineRandGen(RandomGenerator *pGenX, RandomGenerator *pGenY)
: Requires pointers to two heap-based PRNG objects. The CombineRandGen object owns the two PRNGs and *delete*s them when the object is destroyed.

In addition to the public interface, the CombineRandGen class has a protected interface for use by its derived classes:

RandomGenerator *pRandGenX(void) const;
: Returns a pointer to the first of two generators used by the current generator.

RandomGenerator *pRandGenY(void) const;
: Returns a pointer to the second of two generators used by the current generator.

Once the CombineRandGen object has been created, it owns the PRNGs passed to it. Do not try to *delete* these pointers yourself.

6.7 AdaptorRandGen

The AdaptorRandGen abstract class defines a basic interface for a PRNG that modifies the output of some other PRNG. It is derived from RandomGenerator (p. 7) and is implemented in the files **randadpt.h** and **randadpt.cpp**. Some possibilities include a generator that maps a uniformly distributed random number sequence into a normal or Poisson distribution. The constructor for this class has the following prototype:

explicit AdaptorRandGen(RandomGenerator *pGen)
: Requires a pointer to a heap-based PRNG object. The AdaptorRandGen object owns this PRNG and *delete*s it when the object is destroyed.

In addition to the public interface, the AdaptorRandGen class has a protected interface for use by its derived classes:

RandomGenerator *pRandGen(void) const;
: Returns a pointer to the base generator modified by the current generator.

Once the AdaptorRandGen object has been created, it owns the PRNG passed to it. Do not try to *delete* this pointer yourself.

6.8 SHARandGen

The SHARandGen class uses the SHA algorithm (see [9]) to scramble the data in a buffer to generate a digested version of the buffer. The SHARandGen class abstracts a generator that uses a constant string and a variable source of numbers to fill the buffer that is to be digested. The generator then uses the first *sizeof(size_rand)* bytes of the digest as the result of the generator. This class is derived from RandomGenerator (p. 7) and is implemented in the files **randsha.h** and **randsha.cpp**. The constructors for this class have the following prototypes:

SHARandGen(unsigned short size, const char *seedStr)
: Requires a buffer of data that the object uses as part of the message in the SHA algorithm. The length of the supplied buffer is *size* and *seedStr* points to the data. The constructor makes a copy of the data for the object's use.

explict SHARandGen(const char *seedStr)

Simplifies the construction of a SHARandGen object by allowing a null-terminated string as its argument.

With many PRNGs, knowledge of the algorithm and a few consecutive numbers are enough to duplicate the rest of the pseudo-random number sequence. The output of the SHA-based generators should not be analyzable in this fashion.

7 Concrete Classes

The classes described in this section are the main reason you would wish to use the ThinAir library. These classes represent a significant fraction of the PRNGs described in the literature.

7.1 LinConRandGen

The LinConRandGen class implements the linear congruential pseudo-random number generator with a constant term. This kind of generator was introduced by D. H. Lehmer in 1949. It is derived from CongruentialRandGen (p. 11) and is implemented in the files **randlcon.h** and **randlcon.cpp**. The generator is based on the equation

$$X_n = (a * X_{n-1} + c) \bmod 2^{32}.$$

The constructor for this class has the following prototype.

LinConRandGen(unsigned long mult, unsigned long incr, size_rand seed)

The *mult* takes the place of a in the formula and *incr* is the value of c. The *seed* is the value of X_0 and has a default value of 1.

In addition to the public interface, the LinConRandGen class has a protected interface for use by its derived classes:

unsigned long   Multiplier(void) const;

Returns the multiplier the generator was created with.

unsigned long   Increment(void) const;

Returns the increment value the generator was created with.

7.2 LinConModRandGen

Like LinConRandGen, this class implements generators based on the equation

$$X_n = (a * X_{n-1} + c) \bmod m.$$

It is derived from LinConRandGen (p. 14) and is implemented in the files **rndlconm.h** and **rndlconm.cpp**. This kind of generator was introduced by D. H. Lehmer in 1949. Unlike LinConRandGen, LinConModRandGen generators have a modulus other than 2^{32}. The price for this extra flexibility is slower operation. The constructor for this class has the following prototype.

LinConModRandGen(unsigned long mult, unsigned long incr, unsigned long mod, size_rand seed)

The *mult* takes the place of a in the formula and *incr* is the value of c. The value of m in the formula is supplied by *mod*. The *seed* is the value of X_0 and has a default value of 1.

In addition to the public interface, the LinConModRandGen class has a protected interface for use by its derived classes:

unsigned long   Modulus(void) const;

Returns the modulus the generator was created with.

7.3 MultConRandGen

The MultConRandGen class implements the linear congruential without a constant term. The generator is based on the equation

$$X_n = (a * X_{n-1}) \bmod 2^{32}.$$

This kind of generator was introduced by D. H. Lehmer in 1949. It is derived from CongruentialRandGen (p. 11) and is implemented in the files **randmcon.h** and **randmcon.cpp**. The constructor for this class has the following prototype:

MultConRandGen(unsigned long mult, size_rand seed)
: Requires only the value of a (supplied by *mult*) and an initial value for X_0 (supplied by *seed*). If no value is given for *seed*, a 1 is used.

In addition to the public interface, the MultConRandGen class has a protected interface for use by its derived classes:

unsigned long Multiplier(void) const;
: Returns the multiplier this generator was created with.

An equivalent generator can be constructed from LinConRandGen (p. 14) by setting the *incr* term to 0. However, this class provides a small increase in speed by removing the overhead of the increment term.

7.4 MultConModRandGen

Like MultConRandGen, this class implements generators based on the equation

$$X_n = (a * X_{n-1}) \bmod m.$$

This kind of generator was introduced by D. H. Lehmer in 1949. MultConModRandGen is derived from MultConRandGen (p. 15) and is implemented in the files **rndmconm.h** and **rndmconm.cpp**. Unlike MultConRandGen, MultConModRandGen generators have a modulus other than 2^{32}. The price for this extra flexibility is slower operation. The constructor for this class has the following prototype:

MultConModRandGen(unsigned long mult, unsigned long mod, size_rand seed)
: The value of a (supplied by *mult*), the modulus m (supplied by *mod*), and an initial value for X_0 (supplied by *seed*). If no value is given for *seed*, a 1 is used.

In addition to the public interface, the MultConModRandGen class has a protected interface for use by its derived classes:

unsigned long Modulus(void) const;
: Returns the modulus the generator was created with.

An equivalent generator can be constructed from LinConModRandGen (p. 14) by setting the *incr* term to 0. However, this class provides a small increase in speed by removing the overhead of the increment term.

7.5 SecConRandGen

The SecConRandGen class implements a second-order congruential generator. It is derived from CongruentialRandGen (p. 11) and is implemented in the files **randsec.h** and **randsec.cpp**. This generator is based on the equation

$$X_n = (a * X_{n-1} + b * X_{n-2} + c) \bmod 2^{32}.$$

The constructor for this class has the following prototype:

SecConRandGen(long multLast, long multCurr, unsigned long incr, size_rand seed)
: Uses the values of *multCurr* and *multLast* to replace the formula coefficients a and b, respectively. The value of c from the formula is supplied by the argument *incr*. The argument *seed* supplies the initial value of X_0. The value for X_{-1} is always 0. The default value for *seed* is 1.

In addition to the public interface, the SecConRandGen class has a protected interface for use by its derived classes:

size_rand LastSeed(void) const;
Returns the previous seed value.

unsugned long MultLast(void) const;
Returns the last seed multiplier. (The constant b from the formula above.)

unsigned long MultCurr(void) const;
Returns the current seed multiplier. (The constant a from the formula above.)

unsigned long Increment(void) const;
Returns the increment value the generator was created with. (The constant c from the formula above.)

void SetLastSeed(size_rand seed);
Sets the last seed value that the generator uses in creating the next random number.

7.6 SecConModRandGen

Like SecConRandGen, this class implements generators based on the equation

$$X_n = (a * X_{n-1} + b * X_{n-2} + c) \bmod m.$$

It is derived from SecConRandGen (p. 15) and is implemented in the files **randsecm.h** and **randsecm.cpp**. Unlike SecConRandGen, SecConModRandGen generators have a modulus other than 2^{32}. The price for this extra flexibility is slower operation. The constructor for this class has the following prototype:

SecConModRandGen(long multLast, long multCurr, unsigned long incr, unsigned long mod, size_rand seed)
Uses the values of *multCurr* and *multLast* to replace the formula coefficients a and b, respectively. The value of c from the formula is supplied by the argument *incr*. The argument *mod* supplies the modulus, m, for the formula above. The argument *seed* supplies the initial value of X_0. The value for X_{-1} is always 0. The default value for *seed* is 1.

In addition to the public interface, the SecConModRandGen class has a protected interface for use by its derived classes:

unsigned long Modulus(void) const;
Returns the modulus the generator was created with.

7.7 InvConModRandGen

This class implements the Inverse Congruential PRNG described by Andreas Weingartner in his 1994 Masters thesis, [10]. The class implements generators based on the equation

$$X_n = (a * \overline{X}_{n-1} + b) \bmod m.$$

Where $\overline{X}$ is the inverse of X in the modulus field m. It is derived from CongruentialRandGen (p. 11) and is implemented in the files **randicon.h** and **randicon.cpp**. The algorithm works best if this modulus is a prime number. An absolute minimum requirement is that the modulus is not even. The constructor for this class has the following prototype:

InvConModRandGen(unsigned long mult, unsigned long incr, unsigned long mod, size_rand seed)
The *mult* takes the place of a in the formula and *incr* is the value of b. The value of m in the formula is supplied by *mod*. The *seed* is the value of X_0 and has a default value of 1.

In addition to the public interface, the InvConModRandGen class has a protected interface for use by its derived classes:

unsigned long   Multiplier(void) const;
    Returns the multiplier the generator was created with.

unsigned long   Increment(void) const;
    Returns the increment value the generator was created with.

unsigned long   Modulus(void) const;
    Returns the modulus the generator was created with.

unsigned long   InvertSeed(void);
    Returns the result of inverting the current seed in the $2^{32} - 1$ modulus field.

7.8 InvConRandGen

This class implements the Inverse Congruential PRNG described by Andreas Weingartner in his 1994 Masters thesis, [10]. It is derived from InvConRandGen and is implemented in the files **randicon.h** and **randicon.cpp**. This generator is based on the equation

$$X_n = (a * \overline{X}_{n-1} + b) \bmod (2^{32} - 5).$$

Where $\overline{X}$ is the inverse of X in the modulus field $2^{32} - 5$. The modulus chosen is the largest prime number less than 2^{32}, according to [4, page 390]. The constructor for this class has the following prototype:

InvConRandGen(unsigned long mult, unsigned long incr, size_rand seed)
    The *mult* takes the place of a in the formula and *incr* is the value of b. The *seed* is the value of X_0 and has a default value of 1.

7.9 QuadConRandGen

This class implements the Quadratic Congruential PRNG. This generator is really not much better than the LinConRandGen (p. 14) PRNG described above. This PRNG is discussed in [10]. The main reason I included this class was to show how a new generator may be derived from another class in Section 13.5 below. This class is derived from CongruentialRandGen (p. 11) and is implemented in the files **randquad.h** and **randquad.cpp**. This generator is based on the equation

$$X_n = (a * {X_{n-1}}^2 + b * X_{n-1} + c) \bmod 2^{32}.$$

The constructor for this class has the following prototype:

QuadConRandGen(unsigned long quad, unsigned long lin, unsigned long incr, size_rand seed)
    The *quad* replaces the place of a in the formula, *lin* replaces b, and *incr* is the value of c. The *seed* is the value of X_0 and has a default value of 1.

In addition to the public interface, the QuadConRandGen class has a protected interface for use by its derived classes:

unsigned long   QuadMultiplier(void) const;
    Returns the coefficient for the second-order term the generator was created with.

unsigned long   LinMultiplier(void) const;
    Returns the coefficient for the linear term the generator was created with.

unsigned long   Increment(void) const;
    Returns the increment value the generator was created with.

7.10 NumberTableRandGen

The NumberTableRandGen class reads numbers from a file-based table containing whitespace separated numbers in an ASCII-format. Any source of numbers that can write a file filled with numeric strings can build tables for this class. The table begins after a line containing the phrase `Begin Table`. The NumberTableRandGen class is derived from TextTableRandGen (p. 12) and is implemented in the files **randntbl.h** and **randntbl.cpp**. The constructor for this class has the following prototype:

explicit NumberTableRandGen(const char *szFileName, streampos start)
: Calls the constructor for TextTableRandGen with the arguments *szFileName* and *start*. The *start* argument has a default value of 0.

7.11 BinaryTableRandGen

This class defines an interface for generators that read a sequence of numbers from a binary table on disk. It is derived from TableRandGen (p. 11) and is implemented in the files **randtabl.h** and **randtabl.cpp**. The BinaryTableRandGen class makes only one assumption about the file. The file is assumed to be a list of 4-byte binary numbers with no formatting. The constructor for this class has the following prototype:

explicit BinaryTableRandGen(const char *szFileName, streampos start)
: Calls the constructor for TableRandGen with the arguments *szFileName* and *start*. The *start* argument has a default value of 0.

7.12 DecimalTableRandGen

The DecimalTableRandGen class is derived from ByteTableRandGen (p. 12). The file it uses must contain a list of ASCII digits. Only the characters '0' - '9' are considered significant in the file. The table begins after a line containing the phrase `Begin Table`. The DecimalTableRandGen combines the digits into a random number as needed. The DecimalTableRandGen class is implemented in the files **randbtbl.h** and **randbtbl.cpp**. The constructor for this class has the following prototype:

explicit DecimalTableRandGen(const char *szFileName, streampos start)
: Calls the constructor for ByteTableRandGen with the arguments *szFileName* and *start*. The *start* argument has a default value of 0.

In addition to the public interface, the DecimalTableRandGen class has a protected interface for use by its derived classes:

size_rand getNumber(unsigned digits);
: Reads *digits* bytes from the table and constructs a number from these bytes.

unsigned getNumberDigits(size_rand limit);
: Returns the number of digits needed to produce a number the size of *limit*.

7.13 HexTableRandGen

The HexTableRandGen class is derived from ByteTableRandGen (p. 12). The file it uses must contain a list of hexadecimal digits. Only the characters '0' - '9', 'a' - 'f', and 'A' - 'F' are considered significant in the file. The table begins after a line containing the phrase `Begin Table`. The HexTableRandGen combines the hexadecimal digits into a random number as needed. The HexTableRandGen class is implemented in the files **randbtbl.h** and **randbtbl.cpp**. The constructor for this class has the following prototype:

explicit HexTableRandGen(const char *szFileName, streampos start)
: Calls the constructor for ByteTableRandGen with the arguments *szFileName* and *start*. The *start* argument has a default value of 0.

In addition to the public interface, the HexTableRandGen class has a protected interface for use by its derived classes:

size_rand getNumber(unsigned digits);
: Reads *digits* bytes from the table and constructs a number from these bytes.

unsigned getNumberDigits(size_rand limit);
: Returns the number of digits needed to produce a number the size of *limit*.

7.14 DifferenceRandGen

A DifferenceRandGen object is constructed with pointers to two other PRNG objects. The number sequence generated by the DifferenceRandGen object are formed by subtracting the result of one of the generators (*pGenY*) from the result of the other (*pGenX*). The DifferenceRandGen class is derived from CombineRandGen (p. 13) and is implemented in the files **randdiff.h** and **randdiff.cpp**. The constructor for this class has the following prototype:

DifferenceRandGen(RandomGenerator *pGenX, RandomGenerator *pGenY)
: Requires pointers to two heap-based PRNG objects. The DifferenceRandGen object owns these PRNGs and *delete*s them when the object is destroyed.

7.15 ShuffledMRandGen

cmtVerify: probably need to implement algorithm B as an Adaptor. The ShuffledMRandGen class implements the shuffling algorithm described in Algorithm M, [4, Section 3.2.2]. This generator was prposed by M. D. MacLaren and G. Marsaglia in 1968. One generator (*pGenX*) is used to fill an array with numbers. A second generator (*pGenY*) selects which entry is used as output. Then, we use the first generator to reset the array entry to a new value. It is derived from CombineRandGen (p. 13). The ShuffledMRandGen class is implemented in the files **randshuf.h** and **randshuf.cpp**. The constructor for this class has the following prototype:

ShuffledMRandGen(unsigned size, RandomGenerator *pGenX, RandomGenerator *pGenY)
: Requires pointers to two heap-based PRNG objects. The ShuffledMRandGen object owns these PRNGs and *delete*s them when the object is destroyed. The ShuffledMRandGen object also needs a *size* value used to define the internal array.

7.16 A15BitRandGen

The A15BitRandGen class is derived from AdaptorRandGen (p. 13). This generator provides a method of converting the output of a 32-bit generator into a 15-bit generator. PRNGs built for 16-bit architectures often limit their output to 15-bits. Two examples of this kind of generator are the BorlandRandGen (p. 21) and StdCRandGen (p. 21) classes. The A15BitRandGen generator is implemented in **randadpt.h**. The constructor for this class has the following prototype:

A15BitRandGen(RandomGenerator *pGen)
: Requires a pointer to a heap-based PRNG object. The A15BitRandGen object owns this PRNG and *delete*s it when the object is destroyed.

7.17 LaggedFibonacciRandGen

The LaggedFibonacciRandGen implements a class of generators described in [4, page 26] and [6]. These generators implement the equation

$$X_n = (X_{n-u} + X_{n-l}) \bmod 2^{32}.$$

Research has shown that with proper choices of u and l, the period of the least significant bits of the output of this generator can be $2^u - 1$. (for $u > l$) This guarantees a period for the generator of at least this long. The only reported problem involves a birthday spacings test for small values of l and u. The LaggedFibonacciRandGen class is derived from RandomGenerator (p. 7) and is implemented in the files **randlfib.cpp** and **randlfib.h**. The constructor for this class has the following prototype:

LaggedFibonacciRandGen(unsigned lower, unsigned upper, RandomGenerator *pRng)
: Requires a pointer to a heap-based PRNG object. The LaggedFibonacciRandGen object owns this PRNG and *delete*s it when the object is destroyed. The LaggedFibonacciRandGen object also needs a *lower* and *upper* offset into the past data to define which old data contributes to the new value. The *upper* limit also determines how large a buffer to allocate for past state.

In addition to the public interface, the LaggedFibonacciRandGen class has a protected interface for use by its derived classes:

unsigned UpperTap() const;
: Returns the value of the upper tap into the past data buffer.

unsigned LowerTap() const;
: Returns the value of the lower tap into the past data buffer.

size_rand LowValue() const;
: Returns the value of the current low position in the past data buffer.

size_rand HighValue() const;
: Returns the value of the current high position in the past data buffer.

RandomGenerator *Initializer() const;
: Returns a pointer to the PRNG object used to initialize the past data buffer.

void SetNewElement(size_rand val);
: Sets the new element the past data buffer to *val*.

void InitArray(void);
: Initialize the past data buffer using the PRNG supplied at constructor time.

void InitTaps(void);
: Initialize the current taps to their initial values.

void IncrTaps(void);
: Move the current taps to their next location.

7.18 LaggedFibonacciModRandGen

The LaggedFibonacciRandGen implements a class of generators described in [4, page 26] and [6]. These generators implement the equation

$$X_n = (X_{n-u} + X_{n-l}) \bmod m.$$

The only difference between this class and the LaggedFibonacciRandGen class is the modulus m. This modulus makes the generator more flexible, but also slows it down. The LaggedFibonacciModRandGen class is derived from LaggedFibonacciRandGen (p. 19) and is implemented in the files **rndlfibm.cpp** and **rndlfibm.h**. The constructor for this class has the following prototype:

LaggedFibonacciModRandGen(unsigned lower, unsigned upper, unsigned long mod, RandomGenerator *pRng)
: Requires a pointer to a heap-based PRNG object. The LaggedFibonacciModRandGen object owns this PRNG and *delete*s it when the object is destroyed. The LaggedFibonacciModRandGen object also needs a *lower* and *upper* offset into the past data to define which old data contributes to the new value. The *upper* limit also determines how large a buffer to allocate for past state. The *mod* parameter defines the modulus used in the calculations.

In addition to the public interface, the LaggedFibonacciModRandGen class has a protected interface for use by its derived classes:

unsigned long Modulus(void) const;
: Returns the modulus the generator was created with.

7.19 LinMultWCarryRandGen

The LinMultWCarryRandGen class implements a special case of Marsaglia's *multiply with carry* PRNG. The class is derived from CongruentialRandGen (p. 11) and is implemented in the files **randlmwc.h** and **randlmwc.cpp**. The generator is based on the equation

$$X_n = (a * X_{n-1} + c_{n-1}) \bmod 2^{32}.$$

Unlike the linear congruential PRNG, the increment term in this equation changes as the algorithm progresses. This term is generated by the *carry* or quotient resulting from the dividing $a * X_{n-1} + c_{n-1}$ by the modulus. The constructor for this class has the following prototype.

LinMultWCarryRandGen(unsigned long mult, size_rand carry, size_rand seed)
: The *mult* takes the place of a in the formula. The initial value of c in the formula is supplied by *carry* and has a default value of 0. The *seed* is the value of X_0 and has a default value of 1.

In addition to the public interface, the LinMultWCarryRandGen class has a protected interface for use by its derived classes:

size_rand Multiplier(void) const;
: Returns the multiplier the generator was created with.

size_rand Carry(void) const;
: Returns the carry the generator was created with.

size_rand CurrCarry(void) const;
: Returns the carry generated by in the last operation.

void SetCarry(size_rand carry);
: Resets the carry the generator was created with.

void SetCurrCarry(size_rand carry);
: Resets the carry value for the generator.

8 Generator Instantiations

8.1 BorlandRandGen

The BorlandRandGen class implements the generator from the *rand()* function supplied in the Borland C++ v4.5 standard library. This generator is derived from the LinConRandGen (p. 14) class and is implemented in the file **erndlcon.h**. Although the generator itself retains 32 bits of seed, the output range is limited to 0-32767. The constructor for this class has the following prototype:

explicit BorlandRandGen(size_rand seed)
: Requires a *seed* value. This *seed* defaults to 1 if none is supplied.

8.2 StdCRandGen

The StdCRandGen class implements the generator from the *rand()* function described in the ANSI C Standard ([7]). This generator is derived from the LinConRandGen (p. 14) class and is implemented in the file **erndlcon.h**. Although the generator itself retains 32 bits of seed, the output range is limited to 0-32767. The constructor for this class has the following prototype:

explicit StdCRandGen(size_rand seed)
: Requires a *seed* value. This *seed* defaults to 1 if none is supplied.

8.3 BoroshNiederreiterRandGen

The generator implemented by the BoroshNiederreiterRandGen class is described in [4, pages 102-104]. It is derived from MultConRandGen (p. 15) and is implemented in the file **erndmcon.h**. This generator did very well on the spectral test for dimensions 2, 3, and 5, according to Knuth. The constructor for this class has the following prototype:

explicit BoroshNiederreiterRandGen(size_rand seed)
Requires a *seed* value. This *seed* defaults to 1 if none is supplied.

8.4 WatermanRandGen

The generator implemented by the WatermanRandGen class is described in [4, pages 102-104]. It is derived from MultConRandGen (p. 15) and is implemented in the file **erndmcon.h**. This generator did very well on the spectral test for dimensions 2, 3, 4, and 5, according to Knuth. The constructor for this class has the following prototype:

explicit WatermanRandGen(size_rand seed)
Requires a *seed* value. This *seed* defaults to 1 if none is supplied.

8.5 MarsagliaMCRandGen

The MarsagliaMCRandGen class implements a Multiplicative random number generator that was nominated by George Marsaglia as "a candidate for the best of all multipliers" according to [4]. This distinction was based on the spectral test for dimensions 2 through 5, and partly because the multiplier is easy to remember. It is derived from MultConRandGen (p. 15) and is implemented in the file **erndlcon.h**. The constructor for this class has the following prototype:

explicit MarsagliaMCRandGen(size_rand seed)
Requires a *seed* value. This *seed* defaults to 1 if none is supplied.

8.6 LavauxJanssensRandGen

The generator implemented by the LavauxJanssensRandGen class is described in [4, pages 102-104]. It is derived from MultConRandGen (p. 15) and is implemented in the file **erndlcon.h**. This generator did very well on the spectral test for dimensions 2 through 6, according to Knuth. The constructor for this class has the following prototype:

explicit LavauxJanssensRandGen(size_rand seed)
Requires a *seed* value. This *seed* defaults to 1 if none is supplied.

8.7 Knuth28RandGen

The Knuth28RandGen class is derived from SecConModRandGen (p. 16) and is implemented in the file **erndsecm.h**. In [4], this generator is described as having a period length of $(2^{31}-1)^2-1$. The Knuth28RandGen class is based on the equation

$$X_n = (271828183X_{n-1} - 314159269X_{n-2}) \bmod (2^{31}-1).$$

Although it does have a long period, this generator only did very well on dimensions 2 and 3 of the spectral test, according to Knuth. The constructor for this class has the following prototype:

explicit Knuth28RandGen(size_rand seed)
Requires a *seed* value. This *seed* defaults to 1 if none is supplied.

8.8 Knuth22RandGen

The generator implemented by the KNUTH22RANDGEN class is described in [4, pages 102-104]. It is derived from MULTCONMODRANDGEN (p. 15) and is implemented in the file **erndmcnm.h**. This generator did very well on the spectral test for dimension 2, according to Knuth. The constructor for this class has the following prototype:

explicit Knuth22RandGen(size_rand seed)
: Requires a *seed* value. This *seed* defaults to 1 if none is supplied.

8.9 Knuth27RandGen

The generator implemented by the KNUTH27RANDGEN class is described in [4, pages 102-104]. It is derived from MULTCONMODRANDGEN (p. 15) and is implemented in the file **erndmcnm.h**. This generator did very well on the spectral test for dimensions 2 through 6, according to Knuth. The constructor for this class has the following prototype:

explicit Knuth27RandGen(size_rand seed)
: Requires a *seed* value. This *seed* defaults to 1 if none is supplied.

8.10 MitchellMooreRandGen

The generator implemented by the MITCHELLMOORERANDGEN class is described in [4, page 26]. The class is derived from the LAGGEDFIBONACCIMODRANDGEN (p. 20) and is implemented in the file **erndlfib.h**. This generator has a number of interesting properties. The least significant bits of any sequence generated by this generator will have a period of $2^{55} - 1$. The period has been found to be approximately $2^{85} - 1$. The MITCHELLMOORERANDGEN generator is based on the equation

$$X_n = (X_{n-24} + X_{n-55}) \bmod m.$$

Knuth had reported that the main reason for not using this generator was a lack of theory regarding its operation. Since that time, research reported in [6] shows that generators of this type can be quite good. The constructor for this class has the following prototype:

explicit MitchellMooreRandGen(unsigned long mod, RandomGenerator *pRng)
: Requires a pointer to a heap-based PRNG object. The MITCHELLMOORERANDGEN object owns this PRNG and *delete*s it when the object is destroyed. The *mod* parameter defines the modulus used in the calculations.

9 Special Generators

9.1 DwyerRandGen

This class is derived from MULTCONMODRANDGEN (p. 15) and based on an article by Jerry Dwyer in CUJ [2]. The algorithm used in this generator is supposedly immune to undefined effects of overflow in multiplications. Although, the MULTCONMODRANDGEN class is also immune to these effects, its implementation is slower than Dwyer's. However, Dwyer's algorithm limits the size of the multipliers to the square root of the modulus. The constructor for this class has the following prototype:

DwyerRandGen(unsigned long mult, unsigned long mod, size_rand seed)
: The value of a (supplied by *mult*), the modulus m (supplied by *mod*), and an initial value for X_0 (supplied by *seed*). If no value is given for *seed*, a 1 is used.

In addition to the public interface, the DWYERRANDGEN class has a protected interface for use by its derived classes:

unsigned long Quotient(void) const;
: Returns the quotient obtained by dividing the *mod* by the *mult*.

unsigned long Remainder(void) const;
: Returns the remainder obtained by dividing the *mod* by the *mult*.

Dwyer's article listed four specific generators using his algorithm. These have been implemented as DWYER1RANDGEN, DWYER2RANDGEN, DWYER3RANDGEN, and DWYER4RANDGEN. All of these generators and their base class are implemented in the files **rnddwyer.h** and **rnddwyer.cpp**.

9.2 CompilerRandGen

All C/C++ compilers come with a PRNG implemented in their standard library. The COMPILERRANDGEN class allows access to this generator through the interface specified by RANDOMGENERATOR (p. 7). The COMPILERRANDGEN class is derived from RANDOMGENERATOR and is implemented in the files **randcomp.h** and **randcomp.cpp**. The constructor for this class has the following prototype:

explicit CompilerRandGen(size_rand seed)
: Requires a *seed* value. This *seed* defaults to 1 if none is supplied.

In addition to the public interface, the COMPILERRANDGEN class has a protected interface for use by its derived classes.

size_rand Seed(void) const;
: Returns the initial seed value used when the current generator was created.

void SetSeed(size_rand seed);
: Sets the initial seed value used when the current generator was created.

9.3 SHACtrRandGen

The SHACTRRANDGEN generator uses a counter to provide a variable source of numbers for use by the SHARANDGEN algorithm. The SHACTRRANDGEN class is derived from the SHARANDGEN (p. 13) class and is implemented in the files **randsha.h** and **randsha.cpp**. The constructors for this class have the following prototypes:

SHACtrRandGen(unsigned short size, const char *seedStr, size_rand seed)
: Requires a buffer of data that the object uses as part of the message in the SHA algorithm. The length of the supplied buffer is *size* and *seedStr* points to the data. The constructor makes a copy of the data for the object's use. The initial value for the counter is supplied by *seed*. The default value for *seed* is 0.

explicit SHACtrRandGen(const char *seedStr, size_rand seed)
: Simplifies the construction of a SHACTRRANDGEN object by allowing a null-terminated string as its argument. The argument *seedStr* has a default value of "SHAPRNG". The initial value for the counter is supplied by *seed*. The default value for *seed* is 0.

With many PRNGs, knowledge of the algorithm and a few consecutive numbers are enough to duplicate the rest of the pseudo-random number sequence. The output of the SHACTRRANDGEN generator should not be analyzable in this fashion.

9.4 SHARngRandGen

The SHARNGRANDGEN generator uses a PRNG supplied at construction time to provide a variable source of numbers for use by the SHARANDGEN algorithm. The SHARNGRANDGEN class is derived from the SHARANDGEN (p. 13) class and is implemented in the files **randsha.h** and **randsha.cpp**. The constructors for this class have the following prototypes:

SHARngRandGen(unsigned short size, const char *seedStr, RandomGenerator *pRNG)
Requires a buffer of data that the object uses as part of the message in the SHA algorithm. The length of the supplied buffer is *size* and *seedStr* points to the data. The constructor makes a copy of the data for the object's use. The constructor also requires a pointer to a heap-based PRNG object. The SHARNGRANDGEN object owns this PRNG and *delete*s it when the object is destroyed.

SHARngRandGen(const char *seedStr, RandomGenerator *pRNG)
Simplifies the construction of a SHARNGRANDGEN object by allowing a null-terminated string as one of its argument. The constructor also requires a pointer to a heap-based PRNG object. The SHARNGRANDGEN object owns this PRNG and *delete*s it when the object is destroyed.

With many PRNGs, knowledge of the algorithm and a few consecutive numbers are enough to duplicate the rest of the pseudo-random number sequence. The output of the SHARNGRANDGEN generator should not be analyzable in this fashion.

10 Bad Pseudo-Random Number Generators

PRNGs that produce non-random sequencies are useful for testing. Although these generators do not produce pseudo-random number sequences, they have the same interface as the other classes in the ThinAir library. This allows them to be used in worst-case scenarios and in testing random number generator test code. In this section, I also include some notoriously bad generators that have been proposed, and used, in the past.

10.1 ConstantRandGen

The CONSTANTRANDGEN takes a single argument when constructed. Every time the *Number()* member function is called, CONSTANTRANDGEN returns that argument. The *LimitedNumber()* member function returns the modulus of its value with the *limit* argument. This generator is derived from RANDOMGENERATOR (p. 7) and is implemented in **randnot.h** and **randnot.cpp**. The constructor for this class has the following prototype:

explicit ConstantRandGen(size_rand seed)
Uses its *seed* argument as its output value.

10.2 CounterRandGen

The COUNTERRANDGEN takes two arguments when constructed, a starting value and an increment. The behavior of the COUNTERRANDGEN class is defined be the equation

$$X_n = (X_{n-1} + incr) \bmod 2^{32}.$$

This generator is derived from LINCONRANDGEN (p. 14) and is implemented in **randnot.h** and **randnot.cpp**. The constructor for this class has the following prototype:

explicit CounterRandGen(size_rand seed, size_rand incr)
The *seed* argument supplies the initial value of the generator. The *incr* argument is added to the current value when a new value is required. The default value for *seed* is 0. The default value for *incr* is 1.

10.3 SawtoothRandGen

The SAWTOOTHRANDGEN generator counts from 0 to *limit* and then resets to 0. This generator is derived from LINCONMODRANDGEN (p. 14) and is implemented in **randnot.h** and **randnot.cpp**. The constructor for this class has the following prototype:

explicit SawtoothRandGen(size_rand limit)
Takes a single *limit* argument when constructed.

10.4 SquareRandGen

The SquareRandGen class implements a generator that oscillates between two values. When constructing an object of this class, you specify the two numbers and the number of times each is output in a row. It is derived from RandomGenerator (p. 7) and is implemented in **randnot.h** and **randnot.cpp**. The constructor for this class has the following prototype:

SquareRandGen(size_rand first, size_rand second, size_rand dura)
: The *first* argument and the *second* argument is each returned by the generator. The *dura* argument defines how many of each value the generator returns before swapping values.

10.5 RANDURandGen

The RANDURandGen class implements a generator that has been implemented widely. In spite of a long period, you should avoid this generator because it does not generate uncorrelated numbers. The RANDURandGen class is derived from MultConModRandGen (p. 15) and is implemented in **erndmcnm.h**. Its parameters are described in [4, page 104]. The constructor for this class has the following prototype.

explicit RANDURandGen(size_rand seed)
: The *seed* argument defines the initial value for the generator. The default value for *seed* is 1.

This generator is often cited as a bad example simply because it has been so widely used. Although the generator is not a good one, RANDU was the standard PRNG for many math libraries and systems for years.

10.6 FibonacciRandGen

The FibonacciRandGen class implements an extremely bad example of a PRNG. In [4, page 26], this generator is described as having a period greater than the modulus. However, it is fairly obvious that the numbers returned by this generator do not appear very random. The FibonacciRandGen class is derived from SecConModRandGen (p. 16) and is implemented in **randnot.h**. The constructor for this class has the following prototype:

explicit FibonacciRandGen(unsigned long mod, size_rand seed)
: The *mod* argument sets the modulus. The *seed* value sets the initial value of the generator. The default value for *seed* is 1.

11 Testing a Generator

George Marsaglia at Florida State University has collected/developed a set of tests that exercise a sequence of numbers to determine how unpredictable they are. The **randfile.pl** Perl script gives an easy way to build the random number file needed to test a generator using DIEHARD. Execute the perl script **randfile.pl**. The script gives you the opportunity to select a generator and to supply parameters to the constructor for that generator. When you are finished, the script generates the file **randfile.cpp**. Compile **randfile.cpp** and run RandFile. The program asks you for a filename and then creates a file that it fills with approximately 11MB of random numbers. This file may be passed to DIEHARD for testing.

Included in the source for ThinAir is the file **rand.cpp**. I used this file for interface testing and regression testing of the library. If an interface is functioning incorrectly, the output contains the word *error*. Otherwise, the test works through a series of generators, printing useful signposts and information to standard output. When making a low-level change to the library, I would save the output of a run. After making my changes, I would recompile and save the output of a new run. There should be no differences between the two output files. This approach is especially useful during the porting of ThinAir to a new compiler or platform.

12 Utility Functions and Classes

In order to make the ThinAir classes function properly, some utility code is needed. This section describes these utilities.

12.1 The Secure Hash Algorithm

The SHARandGen (p. 13) classes require SHA code to implement. The SHA code I used was written by Peter C. Gutmann and released into the public domain in 1992. These functions are implemented in the files **sha.c** and **sha.h**. Also included with the SHA code is a program shatest.c that verifies the code was compiled correctly.

12.2 The Modulus Arithmetic Functions

In implementing the explicit modulus versions of the various congruential PRNGs, I found problems with any modulus that is relatively prime to 2^{32}. I also found that some combinations of multiplier and modulus would even overflow *double*s. I wrote add and multiply functions that properly dealt with the modulus based on algorithms suggested by Rick Hoselton.

The modulus arithmetic functions are implemented in the files **modarith.h** and **modarith.cpp**. This is not a general library of modulus-based arithmetic. I only implemented what I needed.

unsigned long mod_add(unsigned long a, unsigned long b, unsigned long mod)
: Adds the two numbers *a* and *b* in the *mod* modulus field.

unsigned long mod_mult(unsigned long a, unsigned long b, unsigned long mod)
: Multiplies the two numbers *a* and *b* in the *mod* modulus field.

unsigned long mod_invert(unsigned long num, unsigned long mod)
: Finds the multiplicative inverse of *num* in the *mod* modulus field.

mod_result mod_add_c(unsigned long a, unsigned long b, unsigned long mod)
: Adds the two numbers *a* and *b* in the *mod* modulus field returning both the result and the quotient of the modulus operation.

mod_result mod_mult_c(unsigned long a, unsigned long b, unsigned long mod)
: Multiplies the two numbers *a* and *b* in the *mod* modulus field returning both the result and the quotient of the modulus operation.

mod_result mod_add_c(unsigned long a, unsigned long b)
: Adds the two numbers *a* and *b* in the 2^{32} modulus field returning both the result and the quotient of the modulus operation.

mod_result mod_mult_c(unsigned long a, unsigned long b)
: Multiplies the two numbers *a* and *b* in the 2^{32} modulus field returning both the result and the quotient of the modulus operation.

In order to keep these functions from being too slow, I spent a small amount of time profiling and optimizing the C code. I did not convert these routines to assembly language because I wanted the code to remain as portable as possible. The optimizations used should reduce the inefficiencies in the code in a compiler-independent way. Obviously the actual effects of the optimizations depends on the compiler used, but the current code should always be faster than the obvious implementation. The code also has the advantage that it returns the right answer even if it is slower than the more obvious floating point solution.

12.3 The RingBuffer class

Some of the PRNGs in the ThinAir library must store several old values of the generator. After coding this functionality by hand a couple of times, I realized the a *ring buffer* class was in order. The RingBuffer class is implemented in the files **ringbuff.h** and **ringbuff.cpp**. This class is not as full-featured as I would have normally liked. However, I felt that a small class, with less overhead would be more useful to ThinAir. This class implements a rotating buffer of unsigned long integers.

The constructors for the RingBuffer class have the following prototype:

explicit RingBuffer(unsigned size)
: Creates a rotating buffer containing *size* elements.

RingBuffer(const RingBuffer &rhs);
: Copy constructor for the RingBuffer class.

The public interface for the RingBuffer class supports writing to and reading from the buffer. In the spirit of the STL, this class also can create an iterator object which allows the programmer to retain multiple entry points into the buffer. The public interface for RingBuffer is shown below.

void Init();
: Initialize the ring buffer's insertion cursor.

unsigned max_size() const;
: Returns the size of the ring buffer.

unsigned size() const;
: Returns the current number of entries in the ring buffer.

unsigned empty() const;
: Returns *true* if the ring buffer is empty.

void add(unsigned long val);
: Add a new value at the current insertion point.

RingBuffer &operator=(const RingBuffer &rhs);
: Assignment operator for the RingBuff class.

iterator MakeIterator(unsigned index);
: Return an iterator that points to position *index* in the ring buffer.

iterator begin();
: Return an iterator that points to the beginning of the data in the ring buffer.

iterator end();
: Return an iterator that points after the end of the data in the ring buffer.

void Print(ostream &os) const;
: Write the current ring buffer to *os*.

void Read(istream &is);
: Read the current ring buffer from *is*.

The RingBuffer's iterator has all of the functionality you would expect from an iterator. The iterator's public interface is shown below.

iterator();
: Default constructor.

iterator(const iterator &rhs);
: Copy constructor.

unsigned Index() const;
Returns my index into the ring buffer.

const unsigned long &operator *() const;
Return the value pointed to by this iterator in the ring buffer.

unsigned long &operator *();
Return the lvalue of the position pointed to by this iterator in the ring buffer.

iterator operator++();
Preincrement the iterator.

iterator operator++(int);
Postincrement the iterator.

iterator operator--();
Predecrement the iterator.

iterator operator--(int);
Postdecrement the iterator.

iterator &operator=(const iterator &rhs);
Assign a new iterator to the current iterator.

bool operator==(const iterator &rhs);
Tests two iterators for equality.

bool operator!=(const iterator &rhs);
Tests two iterators for inequality.

13 Defining New Generators

There are several levels at which you can define new generators. The simplest only requires generating a new instantiation of one of the concrete generators. Another relatively easy extension to the ThinAir library is the creation of specialized function objects. The next level of complexity occurs when you modify the output of a standard generator slightly in your derived class.

More complex changes may be made by deriving new generators from the abstract classes. At this level of complexity you may have to create a token class to deal with saving and restoring the generator's state. The most knowledgeable programmers may wish to create entirely new types of PRNGs by creating new abstract classes derived from RandomGenerator (p. 7).

13.1 General Design Considerations

When attempting to add a class to a class hierarchy like the ThinAir library, you must keep in mind two sets of design considerations. The first of these considerations involves what you want your new class to do. The second set of considerations were those made by the library's creator. By understanding and following the creator's decisions, you can make your class fit into the library more seamlessly. In addition, following the creator's designs allows you to benefit from his experience in building the rest of the library.

Many of the design criteria for this library are not immediately obvious. Most of these criteria evolved as the library grew and changed. Some of the design criteria are general:

1. Reduce compiler dependencies (portability).
2. Must be easy to begin using.
3. Must be extensible.
4. Should supply functionality that would be hard for average programmer to provide for him/herself.

5. Must be hard to misuse.

These criteria relate to most class libraries and should not be too difficult to understand. The main instrument to ensure portability in the library code is the **environ.h** file. This file contains macros which allow the rest of the code to be defined without consideration of certain compiler deficiencies. These deficiencies include handling of exceptions, the new C++ casting styles, differing return types on overridden member functions, the *bool* type, and the *string* class. At the time this library was written, not all compilers supported all of these C++ language features. In addition, the **environ.h** file also contains macros to exploit compiler features that may be helpful. The Gnu C++ compiler's support of 64-bit integers is abstracted and available in the **environ.h** file.

Where possible, ThinAir uses the *const* keyword to declare any member function which does not logically modify an object. Any member data which never changes is also declared *const*. I have found that this keyword has reduced certain kinds of errors in my code. In addition, it is easier for another programmer to determine some of my intent by seeing which functions promise not to change the object.

On a slightly different note, the ANSI C++ standard has added a new keyword, *explicit*. The compiler is forbidden to use any constructor marked with this keyword in an implicit conversion from one type to another. This compiler behavior only applies to constructors with just one argument. In the ThinAir library, there are no classes which reasonably should be implicitly converted in this fashion. I have, therefore, been careful to use the macro *EXPLICIT* everywhere the *explicit* keyword would apply. Any compiler which supports this new keyword uses it, in all other cases the macro disappears.

Other design criteria relate to the specific problem space.

1. It must be easy to change generators without extensive recompilation.
2. All generators in the library must have equivalent interfaces.
3. You must be able to reset generator to initial state.
4. You must be able to save/restore state of generator.
5. It would be nice to be able to save/restore generator state from a file.
6. A generator must provide a range of integer values for use.
7. A Generator must be capable of providing doubles in the range $[0.0, 1.0)$.
8. Copying a generator is not important to the design.
9. Assignment of one generator to another makes little or no sense.
10. Cloning a generator may be a useful function.

In order to meet the first two criteria, I have derived every generator in the library, either directly or indirectly, from the RandomGenerator (p. 7) class. A programmer can then assign the address of any generator to a RandomGenerator pointer and none of the code needs to know which particular generator is used. (Except of course the code at the point of creation.) More importantly, if you wish to change generators, only the module containing the code creation needs to be recompiled.

The next criterion is provided by the *Reset()* member function which restores any generator to its initial state. The next two criteria are handled by the RandGenToken (p. 8) classes. These classes allow the current state of the generator to be stored. (See [3] for more information on the Token design pattern.) In spite of the fact that each kind of generator requires saving/restoring different amounts of state, the RandGenToken classes abstract the concept very neatly. A token can only be generated by a generator type which can use it. The token classes contain support for reading and writing themselves on streams. One important thing to remember about Tokens is that any time you create a new generator that has a different amount of state than its parent, you must create a Token class for it as well.

The main purpose of the generator classes is described by criteria 6 and 7. The member functions *Number()* and *LimitedNumber()* provide generator output in the form of integers in some range. The *FloatNumber()* member function provides doubles in the range $[0.0, 1.0)$.

In order to simplify design and debugging, I decided that the generators would not need to be copied or assigned. (Criteria 8 and 9) In order to prevent the compiler from building default versions of these functions, I followed James Coplien's advice from page 45 of [1]. Both of these member functions are declared private and not defined in the classes in the ThinAir library.

As the construction of the generator classes continued, I found a few places where copying a generator could be useful in very limited circumstances. In order to make this copying well-defined, I added the *ReCreate()* member function (which recreates the current generator in its initial state) and the *Clone()* member function (which duplicates the current generator and its state. If you create a new generator which changes the functionality of its base class in any way, you must define a *ReCreate()* function for it. The default *Clone()* member function supplied by RandomGenerator should be sufficient for almost all classes in the ThinAir library.

13.2 Instantiating a new concrete generator

The simplest way to create a new concrete PRNG class is to construct a concrete generator with some or all of the constructor parameters filled in. For example, Section 3.3.4 of [4, pages 102-104], Knuth attributes the generator from line 25 of Table 1 (p. 102) as Marsaglia's candidate for "the best of all multipliers." Checking the table, we find that this particular generator has a multiplier of 69069 and a modulus of 2^{32}.

The text describes this generator as a linear congruential generator without a constant term. This narrows our choices down to the MultConRandGen (p. 15) class and the MultConModRandGen (p. 15) class. Since the required modulus is the default for the MultConRandGen class of generators, we use this class as a base. The derived class only needs a new constructor since all other member functions do exactly what is needed. It seems reasonable to allow the users of the new generator to specify a seed at construction time, so all we need to fill in is the multiplier. So we derive MarsagliaMCRandGen (p. 22) publicly from MultConRandGen. The constructor is defined as having a single argument, *seed*. The constructor must pass our multiplier and the user's seed to the constructor for the base class. The body of the constructor does nothing. The only other thing to remember is to make the constructor for MarsagliaMCRandGen *public* so that this class can actually be constructed.

The actual MarsagliaMCRandGen class is defined in the file **erndmcon.h**.

```
class  MarsagliaMCRandGen : public MultConRandGen
 {
public:
  explicit MarsagliaMCRandGen( size_rand seed=1 ) :
     MultConRandGen( 69069UL, seed )
   { }
 } ;
```

13.3 The RNGFuncAdaptorRange Function Object

The function object concept defines an object which is called as a function. In other words, the *operator()* member function is defined. Unlike a normal function, a function object can retain state. When defining the RNGFuncAdaptorRange (p. 10) class, I wanted a function object which generates random numbers between *lower* and *upper*. When creating an object of this class, you must pass a pointer to a heap-based PRNG object, a lower limit and an upper limit. Every time you call the *operator()* member function, it returns a number between *lower* and *upper*, inclusive.

To build this class, we need data members for storing the limits, and the PRNG object pointer. We also need to define the *operator()* member function, a constructor, and a destructor. The destructor is the easy one. All it needs to do is delete the PRNG object pointer. The constructor must store the parameters for use by the *operator()* member function. The easiest way to define the *operator()* member function uses the *LimitedNumber()* member of the PRNG object. Unfortunately, the upper and lower limits are not as useful for *LimitedNumber()* as a lower limit and a range. After realizing this, I redesigned the class to store the range instead of an upper limit. The resulting class definition is shown below.

```
class RNGFuncAdaptorRange
```

```
  {
   RandomGenerator  *myRNG;
   const size_rand    myLower;
   const size_rand    myRange;

 public:
   RNGFuncAdaptorRange( RandomGenerator *pRNG, size_rand lower,
                        size_rand upper ) :
    myRNG(pRNG), myLower(lower), myRange(upper-lower+1)
    { }
   ~RNGFuncAdaptorRange( void ) { delete myRNG; }

   size_rand operator()( void )
    {
     return myLower + myRNG->LimitedNumber( myRange ));
    }
  } ;
```

This class is easy to use. Let's say you wanted to design a class to emulate a six-sided die. Let's also say we want to base it on the MarsagliaMCRandGen (p. 22) class described in the last section. The following code fragment shows how to construct the die object and *roll* it 3 times.

```
int main( void )
 {
  RNGFuncAdaptorRange  die( new MarsagliaMCRandGen( 5 ), 1, 6 );

  for( int i=1;i<4;i++)
   {
    cout << "Roll #" << i << " = " << die() << endl;
   }

  return 0;
 }
```

13.4 Minor Modifications to a generator

Sometimes the output of a generator needs to be modified just a little in order to get the generator you want. The StdCRandGen (p. 21) class is a linear congruential PRNG with a minor twist. Although the modulus is 2^{32}, this generator always returns values in the range 0 to 32767.

In order to build this class, we start out by using the technique described in Section 13.2, above. This time we publicly derive the StdCRandGen class from LinConRandGen (p. 14). The multiplier (1103515245) and increment (12345) were taken from the C Standard. Because we are limiting the range of output values, it is also necessary to limit the size of the seed. This change is handled as the parameters are passed to the base constructor.

Since the output range is modified in the StdCRandGen class, we must redefine the *MaxRandom()* member function to return the new largest number returned by a generator of this class. Likewise, we must modify the *IntSeed()* member function so that it only uses the lower 15 bits of *seed*. Next, we need to actually implement the difference between this class and its base class. The *Number()* member function needs to be modified to return 15 of the upper 16 bits of the current seed.

The last change needed to this class is a new *ReCreate()* function. This is the change that is easiest to forget. Since we are changing the functionality of the class and not just providing parameters, we need a new *ReCreate()* function. The entire class is shown below:

```
 class  StdCRandGen : public LinConRandGen
  {
```

```
public:
  EXPLICIT StdCRandGen( size_rand seed=1 ) :
    LinConRandGen( 1103515245L, 12345L, (seed&0x7fff) )
    { }
  virtual  void       IntSeed( size_rand seed )
   {
    LinConRandGen::IntSeed( (seed & 0x7fff) );
   }
  virtual size_rand   Number( void )
   {
    //  The example uses ((Number/65536)%32768).
    return ((unsigned)(LinConRandGen::Number() >> 16) & 0x7fff);
   }
  virtual  size_rand   MaxRandom( void ) const { return 0x7fff; }
  virtual  DIFFRET(StdCRandGen)  *ReCreate( void ) const
   {
    return new StdCRandGen( Seed() );
   }
 } ;
```

13.5 Deriving the Quadratic Congruential generator

Let's define the QuadConRandGen (p. 17) class as being derived from the CongruentialRandGen (p. 11) class. This class is much like the LinConRandGen (p. 14) class, so we use that class as a model. First, we lay out the interface in the header file **randquad.h**. Since the CongruentialRandGen class takes care of the seed value, all this class needs to retain are the coefficients and increment value.

In keeping with our need-to-know policy on member data, all of these values are declared *private*. Since the values will never change, they are also declared constant. It is possible that someone may want to derive from this class one day, so we provide the protected accessor functions *QuadMultiplier()*, *LinMultiplier()*, and *Increment()* to allow read-only access to this data.

As part of our design criteria, we do not allow copying or assignment of generator objects. This means we need to declare private prototypes for the copy constructor and assignment operators. These member functions have no implementation.

The *public* interface of the class contains only four functions: the constructor, destructor, *ReCreate()*, and *Number()*. All of the other public functions are provided by either CongruentialRandGen or RandomGenerator (p. 7). The constructor and destructor for this class are both quite simple and are, therefore, defined inline. Likewise *ReCreate()* is a simple function, so it is also declared inline in the header.

The class declaration in **randquad.h** now looks like this:

```
class  QuadConRandGen : public CongruentialRandGen
 {
private:
  const unsigned long   myQuadMultiplier;
  const unsigned long   myLinMultiplier;
  const unsigned long   myIncrement;

private:
  // protect the = operator and copy constructor so they cannot be used.
  QuadConRandGen& operator=( QuadConRandGen const & );
  QuadConRandGen( QuadConRandGen const & );

protected:
  // access methods for private data.
  unsigned long   QuadMultiplier() const { return myQuadMultiplier; }
```

```
    unsigned long   LinMultiplier()  const { return myLinMultiplier; }
    unsigned long   Increment()      const { return myIncrement; }

  public:
    QuadConRandGen( unsigned long quad, unsigned long lin, unsigned long incr,
                    size_rand seed=1 )
     : CongruentialRandGen(seed), myQuadMultiplier(quad), myLinMultiplier(lin),
       myIncrement(incr)
     { }

    virtual   ~QuadConRandGen() { }

    virtual DIFFRET(QuadConRandGen)  *ReCreate( void ) const
     {
      return new QuadConRandGen( QuadMultiplier(), LinMultiplier(),
                                 Increment(), Seed() );
     }

    virtual  size_rand   Number( void );
   } ;
```

Now we need to provide an implementation for the remaining function in the class. In the implementation file **RandQuad.cpp**, we define the *Number()* member function. This function just implements the equation listed in the documentation from Section 7.9. The *Number()* function is shown below.

```
size_rand   QuadConRandGen::Number( void )
 {
  size_rand  RandNum = QuadMultiplier() * CurrSeed() * CurrSeed()
                     + LinMultiplier()  * CurrSeed()
                     + Increment();
  SetCurrSeed( RandNum );

  return RandNum;
 }
```

The member functions *CurrSeed()* and *SetCurrSeed()* are provided by the CongruentialRandGen base class. I am also using this class's own accessor functions for the private data, even though that is not required. I have found that this practice simplifies my task if I need to derive a new class from this one. By accessing the private data through the accessor functions, I have made more certain that I have provided all of the access that a derived class needs. Sometimes, I have found that a derived class needs more access. But I make fewer mistakes, if I do it this way.

13.6 Creating a generator from scratch

The most advanced way of creating a new generator is to build one completely from scratch, deriving only from RandomGenerator. The only portion of the design of a brand-new generator class that is determined for you is the interface that all ThinAir PRNGs share.

In this section, we cover the creation of the CompilerRandGen (p. 24). Unlike the other generators in this library, CompilerRandGen just wraps an interface around the Standard Library's generator. This means that the class cannot take advantage of the code from any other class. The main design criterion is simple: Provide as much of the ThinAir functionality and interface as possible for the Standard Library PRNG.

A quick look at the RandomGenerator interface shows that we need to implement several functions. We must create the following functions:

- *RandomGenerator   *ReCreate() const;*

- *void Reset();*
- *size_rand IntSeed(size_rand seed);*
- *size_rand MaxRandom() const;*
- *size_rand Number();*
- *size_rand LimitedNumber(size_rand limit);*
- *RandGenToken CreateToken() const;*
- *int SetToken(const RandGenToken &pToken);*

Of these, only *LimitedNumber()* is truly optional. But, since the standard implementation already supports a function that does this exactly, we will use that function to make the CompilerRandGen mimic the standard PRNG as far as possible. The other issue we must consider early is the need for a *Token* class. This Token must be capable of storing as much of the state of the generator as we can manage.

Let's start by matching the requirements and interface of the standard PRNG with those of a ThinAir PRNG. This would give us a starting point for the new class. As described in Section 5.2, the stardard PRNG's interface/implementation consists of four functions (or function-like macros) and a constant. This interface is reproduced below.

- *RAND_MAX*
- *size_rand rand();*
- *size_rand random(size_rand lim);*
- *size_rand srand(size_rand seed);*
- *void randomize(void);*

It's easy to see where some of these items connect to the ThinAir interface. Obviously, *MaxRandom()* should be implemented using *RAND_MAX*. *Number()* should just be a call to *rand()*. *LimitedNumber()* should be defined as a call to *random()*. Finally, *IntSeed()* is defined as a call to *srand()*. The *randomize()* function does not appear to have any use for our purposes so we'll drop it for now.

Before we get to the rest of the class, a few words are in order about abstract PRNG classes. If this had been an abstract class, we would not have implemented the *Number()* or *LimitedNumber()* member functions. We might not have implemented *IntSeed()* either. We also would not implement the *ReCreate()* function described later. In general, none of these functions can have a reasonable implementation for an abstract class.

Okay, now that the preliminaries are out of the way, let's get down to the part of the design that separates ThinAir PRNGs from the crowd. The unimplemented functions are the constructor, *ReCreate()*, *Reset()*, *CreateToken()*, and *SetToken()*.

Let's start with the constructor. The constructor needs to save any object specific data and prepare the generator for use. The only object-specific data we have for this generator is the initial seed value. The constructor needs a single argument and the class needs a private data member to store that seed. As stated in the ANSI Standard ([7]), the default value for the constructor argument is 1.

In keeping with our need-to-know policy on object data, we must create accessor functions (*Seed()* and *SetSeed(size_rand seed)*) for this piece of data. Both of these methods are declared *protected* because only derived classes should need to access them.

Now we have enough information to build the *ReCreate()* and *Reset()* functions. *ReCreate()* uses the *new* operator to create a new CompilerRandGen, passing it the private seed value. *Reset()* simply calls *srand()* with the private seed value.

Most implementations of the *rand()* function retain 32 bits of state even though the seed values are limited to 16 bits. In order to manage the Token classes we have to add another data item to the CompilerRandGen. The *myCount* data member stores the number of times *rand()* is called. We now need a protected data access member *Count()*.

In order to define the *CreateToken()* and *SetToken()* functions, we must first define the CompilerRandGenToken token class as a derivative of RandGenTokenImpl (found in **randgen.h** and **randgen.cpp**). The only piece of information the token stores is the initial seed. This makes the implementation straightforward. (See **randcomp.cpp** for the details.) Now *CreateToken()* is defined as *new*ing an object of type CompilerRandGenToken and passing it to a brand new RandGenToken object which the function returns. *SetToken()* accepts a RandGenToken reference, from which it extracts the CompilerRandGenToken. From this token it sets its internal data, calls *rand()* the appropriate number of times and exits.

As with all of the PRNG classes, we handle a little coding hygiene to make the classes complete. As discussed in Section 13.1, we need to prevent copying and assignment of objects of this class. To this end, we declare the assignment operator and copy constructor *private* and don't implement them.

The complete class interfaces for CompilerRandGen and CompilerRandGenToken are shown below. The implementations can be found in **randcom.cpp**.

```
class  CompilerRandGen : public RandomGenerator
 {
private:
  size_rand   mySeed;

private:
  // protect the = operator and copy constructor so they cannot be used.
  CompilerRandGen& operator=( CompilerRandGen const & );
  CompilerRandGen( CompilerRandGen const & );

protected:
  size_rand      Seed( void )     const { return mySeed; }
  void           SetSeed( size_rand seed ) { mySeed = seed; }
  unsigned long  Count( void )    const { return myCount; }
  void           RunTo( unsigned long cnt );

public:
  explicit CompilerRandGen( size_rand seed=1 ) : mySeed(seed), myCount(0L)
   { srand((unsigned)seed); }
  virtual   ~CompilerRandGen() { }

  virtual DIFFRET(CompilerRandGen)  *ReCreate( void ) const;

  virtual RandGenToken  CreateToken( void ) const;
  virtual int     SetToken( const RandGenToken &Token );

  virtual void    Reset( void );

  virtual  void        IntSeed( size_rand seed );
  virtual  size_rand   MaxRandom( void ) const;
  virtual  size_rand   Number( void );
 } ;


class  CompilerRandGenToken : public RandGenTokenImpl
 {
public:
  virtual   ~CompilerRandGenToken( void ) { }

  virtual _STRING  CreateInitString( void ) const;
```

```
  virtual int   Initialize( istream &InitStream );
  virtual CompilerRandGenToken   *Clone() const
   {
    return new CompilerRandGenToken( mySeed, myCount );
   }

  void       Print( ostream &os ) const
   {
    os << '(' << mySeed << ',' << myCount << ')' << flush;
   }
  void       Read( istream &is )
   {
    char  dummy;
    is >> dummy >> mySeed >> dummy >> myCount >> dummy;
   }

private:
  friend class CompilerRandGen;
  CompilerRandGenToken( size_rand Seed, unsigned long Count )
   : mySeed(Seed), myCount(Count)
   { }
  size_rand       mySeed;
  unsigned long   myCount;
 } ;
```

Acknowledgements

I'd like to thank Rick Hoselton, who started me on this crazy idea. I also need to thank Jerry Dwyer and Skip Carter, who each gave permission to implement generators they had written in the ThinAir library. Last, but definitely not least, my wife, Debbie Campbell, proofread and massaged this document so that you don't have to read it as I wrote it.

References

[1] James O. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, Reading, MA, USA, 1992.

[2] Jerry Dwyer. Quick and portable random number generators. *C/C++ Users Journal*, pages 33–44, June 1995.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, USA, 1995.

[4] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 10 January 1981.

[5] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, MA, USA, 1996.

[6] Random chapter. http://csep1.phy.ornl.gov/rn/rn.html, December 1994.

[7] Herbert Schildt. *The Annotated ANSI C Standard*. Osborne McGraw-Hill, Berkeley, CA, 94710, USA, 1990. Contains the entire ANSI/ISO 9899-1990 Standards document.

[8] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., New York, NY 10158-0012, USA, second edition, 1996.

[9] Specification for a secure hash standard (shs). Federal Information Processing Standards Publication, January 1992.

[10] Andreas Weingartener. Nonlinear congruential pseudorandom number generators. Master's thesis, Universität Salzburg, June 1994.