3

Tools for Software Evolution

*Men are only as good as their technical development allows them to be.—**George Orwell,*** **Inside the Whale**

Games, statistical tests, and genetic algorithms all rely on random numbers. Unfortunately, the built-in **rand** function is entirely inadequate in circumstances where thousands—or even millions—of random values need to be generated. A run of the algorithm from Chapter 2 may use a hundred thousand or more random values. If the random number "generator" produces repetitive or cyclical values, the algorithm is unlikely to produce satisfactory results.

Random Numbers

A random number is just that: a number whose value cannot be predicted in advance of its existence. While the human mind has been known to be unpredictable, it isn't very good at generating a completely unrelated set of numbers. Try creating a list of twenty random integers selected from the range one through one hundred, inclusive. Are you sure that your numbers are really random, and not simply fragments of old telephone numbers or checkbook balances? And wouldn't it be tedious if you had to generate a thousand, or a million random numbers?

Computers are supposed to be good at reducing tedious numeric operations. Unfortunately, computers perform calculations via algorithms, and truly random numbers cannot be generated by an algorithm. By definition, an algorithm is a specific sequence of operations that produces a predictable output for a given set of parameters. In the case of random numbers, the last thing we want is something predictable! The best we can do with a computer is create an algorithm that *appears* to generate a random sequence of numbers. The numbers aren't really random—a human with a sharp mind or a calculator could predict the numbers in the sequence by following the algorithm. But the sequence of numbers is very difficult to follow, and a human looking at the values will not be able to see any algorithmic pattern to them. For practical applications, pseudo-random numbers suffice.

What we are striving for is something mathematicians call a *uniform deviate*: a sequence such that every number in a given range has an equal chance of being produced.

"Randomizing" Algorithms

In general, a pseudo-random number generator is initialized with a *seed* value that begins the sequence. A set of mathematical operations is performed on the seed, generating a value that is reported as a pseudo-random number. That return value is then used as the next seed value. Researchers have devoted copious time to inventing and analyzing pseudo-random number generators. The goal of this research has been to produce the most unpredictable sequence of values. Designing a good random number generator involves solving two problems:

- Increasing the size of the repetition cycle. As the algorithm is applied, the seed will eventually return to its starting value, and the values start repeating themselves. An algorithm that repeats after generating a million numbers is more useful than a generator that repeats itself after only a hundred values.
- Avoiding predictability. A random number generator that always returns values with the same last digit is worthless. In general, *any* patterns in the output render a generator useless for stochastic computing.

While many fancy and complicated algorithms can generate pseudo-random numbers, the most commonly-used algorithm is also one of the simplest. First introduced by D. Lehmer in 1951, the *linear congruential* method involves only two mathematical operations. The formula is:

$$N_{i+1} = aN_i + c \pmod{m}$$

N is your "random" number, and each successive value (known as a *seed*) is based on the previous one. Each selection of *a*, *c*, and m produces a sequence of values that will eventually cycle back to the starting value of *N*. The equation's factors determine the "randomness" of values and the number of iterations that can be performed before numbers start to repeat. The maximum repetition period is *m*, but not every combination of *a*, *c*, and *m* will produce a maximal period—and most factor sets produce useless sequences. For example, if a=1, c = 1, and m = 1, the algorithm will simply count by ones!

Standard C uses the following linear congruential generator in implementing the **rand** and **srand** functions:

```
static unsigned long next = 1;

int rand(void)
{
    next = next * 1103515245 + 12345;
    return ((unsigned int) (next / 65536UL) % 0x32767UL);
}

void srand(unsigned int seed)
{
    next = seed;
}
```

The Standard C algorithm is a slight elaboration on the basic linear congruential algorithm, in that it uses a **long** for the **seed**, but returns only an **int**. The code above assumes 32-bit **long**s and 16-bit **int**s.

So why not use **rand**? Because the algorithm is inadequate for many applications. And what's wrong with a linear congruential random number generator? Nothing, so long as your random numbers don't need to be very unpredictable and the repetition of those values is not important to your work. The output of **rand** is limited, providing values that only lie between 0 and 32,767, inclusive. In other words, the Standard C generator will produce only a few thousand values before repeating itself—a fatal problem for genetic algorithms that rely on vast quantities of random values. Aside from their numerical limitations, **rand** and **srand** have several faults from a software engineering standpoint:

- A program must explicitly call **srand** to initialize the **seed**. If **srand** isn't called, the default value of **seed** will be used, and every execution of the program will generate the same sequence of pseudo-random numbers.
- Since **srand** and **rand** are two separate functions, **seed** is defined as a global variable. Good programmers avoid global variables, even when those that can be hidden using the **static** keyword.
- Since there is only one **seed** value, only one sequence of pseudo-random numbers is generated in a program. Often, I like to have separate random number generators for different parts of a program.
- The ANSI **rand** function returns values between 0 and **UINT_MAX**. In most cases, I want to retrieve random values that are within a specific range, say from 1 to 100, or between 0.0 and 1.0.
- I might want to obtain random numbers that aren't **long**s. A templatized class could provide the flexibility to generate random values for any type.

Other problems exist with the Standard **rand**. Producing a random floating-point value requires a program to divide the result of **rand** by the constant **RAND_MAX** (as I did in Chapter 2). Even worse, some mathematically-inept compiler vendors try to improve on **rand**, using cute little byte-swapping tricks that only reduce the period of repetition! Statistically, even the best linear congruential generators suffer from convergence in their numeric sequences, and the ANSI generator is not the theoretical best.

A Minimal Standard

A theoretical best does exist, as the result of research by S. K. Park and K. W. Miller. For the multiplicative algorithm to be effective, a and m can only take on a very few values; m most certainly must be prime, for example. Park & Miller identified the values a = 16807, m = 2147483647, and c = 0 as producing the most statistically-random values for 32-bit signed (usually **long**) integers.

> Note: For producing 16-bit values, a good pair of numbers is a = 171, m = 30269, c = 0. Park & Miller also suggested other acceptable values for a in 32-bit algorithms: 42871 and 69621.

One more topic to cover: overflow in multiplication. Obviously, if N is large enough, multiplying by another large value will exceed the maximum value of a **long**, causing an arithmetic overflow before the modulus by m. To prevent overflow, we can use an *approximate factorization* of m, based on the formula known as Schrage's Method:

$$q = m / a; \quad r = m \bmod a, \quad m = aq + r$$

Foundation for a Hierarchy

I began with an abstract base class, **Generator**, defined in the namespace **Coyote::UniformDeviate**. The nested namespace keeps the "internal" uniform deviate identifiers from conflicting with any other names I've declared in my general **Coyote** namespace. **Generator** declares the attributes of any "random number generator", regardless of algorithm.

```
namespace Coyote
{
    namespace UniformDeviate
    {
        //-------------------------------------------------
        // class Generator - declaration
        class Generator
        {
        protected:
            // sets default seed argument from system time
            static long set_seed_from_time()
            {
                return (long)time(NULL);
            }

        public:
            // constructor
            Generator(long initSeed = set_seed_from_time());

            // destructor
            virtual ~Generator();

            // set seed value
            virtual void set_seed(long newSeed = set_seed_from_time());

            // get the current "random" value
            virtual long get_deviate();

            // calculate next seed value
            virtual void next_value() = 0;

        protected:
            long m_seed;  // the seed for generator 1
        };

        inline void UniformDeviate::Generator::set_seed(long initSeed)
        {
            m_seed = initSeed;
        }

        // get the current "random" value
        inline long UniformDeviate::Generator::get_deviate()
        {
            return m_seed;
        }
    }
}
```

The private **set_seed_from_time** method automatically initializes the constructor's seed parameter with the current system time. You can, of course, supply a specific seed when constructing a **Generator** object; any time the generator is run with a specific seed, it will return the same sequence of values, a useful technique when you require reproducible results (as in, for example, a scientific paper).

I defined the **Generator** constructor and destructor in an implementation file. In general, constructors and destructor should not be defined as inline functions in a header; compilers automatically generate calls to these functions under many circumstances, and code can quickly bloat if every instance of creation is inline.

```
// constructor
UniformDeviate::Generator::Generator(long initSeed)
```

```
{
    set_seed(initSeed);
}

// destructor
UniformDeviate::Generator::~Generator()
{
    // does nothing in this base class
}
```

My basic implementation of **Generator** is **MinimalStandard**, a template class with arguments that define values for *a* and *m*. By default, I use the Park-Miller numbers, but later in the chapter, I'll be building better algorithms by combining **MinimalStandard** objects with different *a* and *m* values. The compiler will interpret these parameter values as manifest constants, allowing a good compiler to generate efficient code for **next_value**.

```
namespace Coyote
{
    namespace UniformDeviate
    {
        //-------------------------------------------------
        // class MinimalStandard - declaration
        template <long A = 16807L, long M = LONG_MAX>
             class MinimalStandard : public Generator
        {
        public:
            // constructor
            MinimalStandard(long initSeed = set_seed_from_time());

            // destructor
            virtual ~MinimalStandard();

            // calculate next seed value
            virtual void next_value();

            // interrogators
            long get_M() { return M; }
            long get_A() { return A; }

        protected:
            const long Q; // quotient  (used in Schrage's method)
            const long R; // remainder (used in Schrage's method)
        };

        //-------------------------------------------------
        // class MinimalStandard - definition

        // constructor
        template <long A, long M>
             MinimalStandard<A,M>::MinimalStandard(long initSeed)
            : Generator(initSeed), Q(M/A), R(M%A)
        {
            // nothing here
        }

        // destructor
        template <long A, long M>
            MinimalStandard<A,M>::~MinimalStandard()
        {
            // nothing here, either
        }
```

```
            // move to next seed
            template <long A, long M>
                void MinimalStandard<A,M>::next_value()
            {
                // compute seed = (a * seed) % m, using Schrage's method
                long k = m_seed / Q;

                m_seed = A * (m_seed - k * Q) - R * k;

                if (m_seed < 0)
                    m_seed += M;
            }
        }
    }
```

The **MinimalStandard** class is only the beginning; it provides a base from which we can build even more sophisticated algorithms.

Better than Minimal

One problem with the Minimal Standard is that it can suffer from sequences of repetitive bytes or values. For example, certain large values may *always* be followed by very small values. Such problems can be avoided by using the generator to randomize itself. The best-known technique is called a *shuffle*: Create an small array, load it with the first few generated values, then use subsequent invocations of the algorithm to generate an random index into that array; return the indexed value, and replace it in the array with the another random value. Yes, I know it sounds complicated—but really, all we're doing is mixing up the generated values so that they don't appear in the usual sequence, thus avoiding any correlations or predictable sequences.

Adding a shuffle to **MinimalStandard** is easy; I derived a new template class, **GeneralPurpose**, the includes the shuffle while using the algorithmic code it inherits.

```
            //-------------------------------------------------
            // class GeneralPurpose - declaration
            template <long A = 16807L, long M = LONG_MAX>
                class GeneralPurpose : public MinimalStandard<A,M>
            {
            public:
                // constructor
                GeneralPurpose(long initSeed = set_seed_from_time());

                // destructor
                virtual ~GeneralPurpose();

                // set seed value
                virtual void set_seed(long newSeed = set_seed_from_time());

                // get the current "random" value
                virtual long get_deviate();

                // calculate next seed value
                virtual void next_value();

            protected:
                // set seed value
                void init_table(long seed);

                // table factors
                const long TABLE_SIZE;    // size of the table
                const long TABLE_DIV;     // ratio of M / TABLE_SIZE
```

```
    // shuffle table and values used therein
    long * m_table;
    long   m_shuffle;
};

//-------------------------------------------------
// class GeneralPurpose - definition

// constructor
template <long A, long M>
    GeneralPurpose<A,M>::GeneralPurpose(long initSeed)
    : MinimalStandard<A,M>(initSeed),
      TABLE_SIZE(32L),
      TABLE_DIV(1L + (M - 1L) / TABLE_SIZE)
{
    m_table = new long[TABLE_SIZE];
    init_table(initSeed);
}

// destructor
template <long A,long M> GeneralPurpose<A,M>::~GeneralPurpose()
{
    delete [] m_table;
}

// set seed value
template <long A, long M>
    inline void GeneralPurpose<A,M>::set_seed(long newSeed)
{
    init_table(newSeed);
}

template <long A, long M>
    void GeneralPurpose<A,M>::init_table(long seed)
{
    m_seed    = seed;
    m_shuffle = 0;

    // avoid zero or negative seed!
    if (m_seed <= 0)
        m_seed = 299792458L;

    // initialize the table by getting the first few deviates
    for (int i = TABLE_SIZE + 7; i >= 0; --i)
    {
        // get next value in sequence
        MinimalStandard<A,M>::next_value();

        // store it in a table entry
        if (i < TABLE_SIZE)
            m_table[i] = m_seed;
    }

    // select our shuffled-out value
    m_shuffle = m_table[0];
}

// get the current "random" value
template <long A, long M>
    inline long GeneralPurpose<A,M>::get_deviate()
{
    return m_shuffle;
}
```

```
// move to next seed
template <long A,long M> void GeneralPurpose<A,M>::next_value()
{
    // get next value in sequence
    MinimalStandard<A,M>::next_value();

    // shuffle out table value; save current seed in its place
    size_t i = m_shuffle / TABLE_DIV;

    m_shuffle  = m_table[i];
    m_table[i] = m_seed;
}
```

The Best of the Best

Even the Minimal Standard can show weaknesses when generating millions of values. In a 1988 issue of *Communications of the ACM*, Paul L'Ecuyer suggested a variety of algorithms for the production of reliable, long-period random deviates. By combining two generators based on the Minimal Standard, L'Ecuyer creates a routine that avoids the pitfalls of simpler algorithms. The generator, which I've used below in my **Random** class template, produces uniform random deviates between 0.0 and 1.0.

In a nutshell, L'Ecuyer's algorithm uses an approximate factorization, "shuffling" each result to remove correlation in low-order bits. A single generator of that type will have a repetition period of about 10^8—which, believe it or not, may not be adequate for some very complex genetic algorithms. Running a thousand generations for a population of a hundred chromosomes may require millions of random values. Combining two such generators with a judicious selection of factors the period to approximately 2.3×10^{18}, which should be more than effective in genetic algorithms of any practical scope. The **LEcuyer** class is not a template, but rather a regular class based on instantiations of the **GeneralPurpose** and **MinimalStandard** template classes.

```
//-------------------------------------------------
// class LEcuyer - declaration
class LEcuyer : public Generator
{
public:
    // constructor
    LEcuyer(long initSeed = set_seed_from_time());

    // destructor
    virtual ~LEcuyer();

    // set seed value
    virtual void set_seed(long newSeed = set_seed_from_time());

    // get the current "random" value
    virtual long get_deviate();

    // calculate next seed value
    virtual void next_value();

protected:
    // internal generators
    GeneralPurpose <40014L,2147483563L> m_rand1;
    MinimalStandard<40692L,2147483399L> m_rand2;
};
```

```
        // get the current "random" value
        inline long LEcuyer::get_deviate()
        {
            return m_seed;
        }

    } // end namespace UniformDeviate


//--------------------------------------------------
// class LEcuyer - definition

// constructor
LEcuyer::LEcuyer(long initSeed)
    : m_rand1(initSeed), m_rand2(initSeed)
{
    m_seed = m_rand1.get_deviate();
}

// destructor
LEcuyer::~LEcuyer()
{
    // nothing here
}

// set seed value
void LEcuyer::set_seed(long newSeed)
{
    m_rand1.set_seed(newSeed);
    m_rand2.set_seed(newSeed);
    m_seed = m_rand1.get_deviate();
}

// move to next seed
void LEcuyer::next_value()
{
    // get next value in sequence
    m_rand1.next_value();
    m_rand2.next_value();

    // combined values
    m_seed = m_rand1.get_deviate() - m_rand2.get_deviate();

    if (m_seed < 0)
        m_seed += (m_rand1.get_M() - 1);
}
```

Random Templates

The aforementioned classes provide the algorithmic machinery for random number generation, and are all defined in the **Coyote::UniformDeviate** namespace. To define practical random number generators, I created a pair of very simple templates, **Random** (based on **GeneralPurpose**) and **RandomLEcuyer** (a derivative of **LEcuyer**). These templates specialize on a type that is to be randomized. I've created specializations for common types like **int**, **size_t**, **float**, and **double**.

```
    //------------------------------------------------
    // class Random - declaration
    template <typename T> class Random
         : private UniformDeviate::GeneralPurpose<>
    {
```

```
public:
    // constructor
    Random(long initSeed = set_seed_from_time());

    // get value
    T operator () ();
    T operator () (T limit);      // range 0 to < max
    T operator () (T min, T max); // range min to max
};

//--------------------------------------------------
// class Random - definition
template <typename T> Random<T>::Random(long initSeed)
    : UniformDeviate::GeneralPurpose<>(initSeed)
{
    // nothing else to do
}

//--------------------------------------------------
// class Random: <float> specialization

float Random<float>::operator () ();

template <> inline float Random<float>::operator () (float limit)
{
    return limit * Random<float>::operator ()();
}

template <> inline float Random<float>::operator () (float min,
                                                     float max)
{
    return min + (max - min) * Random<float>::operator ()();
}

//--------------------------------------------------
// class Random: <double> specialization

double Random<double>::operator () ();

template <> inline double Random<double>::operator ()
                                  (double limit)
{
    return limit * Random<double>::operator ()();
}

template <> inline double Random<double>::operator () (double min,
                                                       double max)
{
    return min + (max - min) * Random<double>::operator ()();
}

//--------------------------------------------------
// class Random: <int> specialization

template <> inline int Random<int>::operator () ()
{
    next_value();
    return int(get_deviate());
}

template <> inline int Random<int>::operator () (int limit)
{
    next_value();
    return int((get_deviate()) % long(limit));
```

```
}

template <> inline int Random<int>::operator () (int min, int max)
{
    next_value();
    return min + Random<int>::operator ()(max - min + 1);
}

//-------------------------------------------------
// class Random: <size_t> specialization

template <> inline size_t Random<size_t>::operator () ()
{
    next_value();
    return size_t(get_deviate());
}

template <> inline size_t Random<size_t>::operator ()
                                    (size_t limit)
{
    next_value();
    return size_t((get_deviate()) % long(limit));
}

template <> inline size_t Random<size_t>::operator () (size_t min,
                                                       size_t max)
{
    next_value();
    return min + Random<size_t>::operator ()(max - min + 1);
}

//-------------------------------------------------
// class RandomLEcuyer - declaration
template <typename T> class RandomLEcuyer
    : private UniformDeviate::LEcuyer
{
public:
    // constructor
    RandomLEcuyer(long initSeed = set_seed_from_time());

    // get value
    T operator () ();
    T operator () (T limit);      // range 0 to < max
    T operator () (T min, T max); // range min to max
};

//-------------------------------------------------
// class RandomLEcuyer - definition
template <typename T> RandomLEcuyer<T>::RandomLEcuyer
                                  (long initSeed)
    : UniformDeviate::LEcuyer(initSeed)
{
    // nothing else to do
}

//-------------------------------------------------
// class RandomLEcuyer: <float> specialization

float RandomLEcuyer<float>::operator () ();

template <> inline float RandomLEcuyer<float>::operator ()
                                        (float limit)
{
    return limit * RandomLEcuyer<float>::operator ()();
```

```
}

template <> inline float RandomLEcuyer<float>::operator ()
                           (float min, float max)
{
    return min + (max - min) * RandomLEcuyer<float>::operator ()();
}

//-------------------------------------------------
// class RandomLEcuyer: <double> specialization

double RandomLEcuyer<double>::operator () ();

template <> inline double RandomLEcuyer<double>::operator ()
                                            (double limit)
{
    return limit * RandomLEcuyer<double>::operator ()();
}

template <> inline double RandomLEcuyer<double>::operator ()
                                 (double min, double max)
{
    return min + (max-min) * RandomLEcuyer<double>::operator ()();
}

//-------------------------------------------------
// class RandomLEcuyer: <int> specialization

template <> inline int RandomLEcuyer<int>::operator () ()
{
    next_value();
    return int(get_deviate());
}

template <> inline int RandomLEcuyer<int>::operator () (int limit)
{
    next_value();
    return int((get_deviate()) % long(limit));
}

template <> inline int RandomLEcuyer<int>::operator ()
                                 (int min, int max)
{
    next_value();
    return min + RandomLEcuyer<int>::operator ()(max - min + 1);
}

//-------------------------------------------------
// class RandomLEcuyer: <size_t> specialization

template <> inline size_t RandomLEcuyer<size_t>::operator () ()
{
    next_value();
    return size_t(get_deviate());
}

template <> inline size_t RandomLEcuyer<size_t>::operator ()
                                          (size_t limit)
{
    next_value();
    return size_t((get_deviate()) % long(limit));
}

template <> inline size_t RandomLEcuyer<size_t>::operator ()
```

```
                                        (size_t min, size_t max)
    {
        next_value();
        return min + RandomLEcuyer<size_t>::operator ()(max - min + 1);
    }

} // end namespace Coyote
```

The double and float specializations were too complicated to be implemented entirely in a header file. Note the conditional compilation statements that handle GNU C++'s lack of full support for the Standard C++ **numeric_limits** header.

```
//-------------------------------------------------
// class Random: <float> specialization

float Random<float>::operator () ()
{
    static const float factor1 = 1.0F / float(get_M());

    #ifdef __GNUC__
    static const float factor2 = 1.0F - FLT_EPSILON;
    #else
    static const float factor2 = 1.0F - numeric_limits<float>::epsilon();
    #endif

    next_value();

    float temp = factor1 * float(get_deviate());

    if (temp > factor2)
        return factor2;
    else
        return temp;
}

//-------------------------------------------------
// class Random: <double> specialization

double Random<double>::operator () ()
{
    static const double factor1 = 1.0 / double(get_M());

    #ifdef __GNUC__
    static const double factor2 = 1.0 - DBL_EPSILON;
    #else
    static const double factor2 = 1.0 - numeric_limits<double>::epsilon();
    #endif

    next_value();

    double temp = factor1 * double(get_deviate());

    if (temp > factor2)
        return factor2;
    else
        return temp;
}

//-------------------------------------------------
// class RandomLEcuyer: <float> specialization

float RandomLEcuyer<float>::operator () ()
{
```

```
    static const float factor1 = 1.0F / float(LEcuyer::m_rand1.get_M());

    #ifdef __GNUC__
    static const float factor2 = 1.0F - FLT_EPSILON;
    #else
    static const float factor2 = 1.0F - numeric_limits<float>::epsilon();
    #endif

    next_value();

    float temp = factor1 * float(get_deviate());

    if (temp > factor2)
        return factor2;
    else
        return temp;
}

//-------------------------------------------------
// class RandomLEcuyer: <double> specialization

double RandomLEcuyer<double>::operator () ()
{
    static const double factor1 = 1.0 / double(LEcuyer::m_rand1.get_M());

    #ifdef __GNUC__
    static const double factor2 = 1.0 - DBL_EPSILON;
    #else
    static const double factor2 = 1.0 - numeric_limits<double>::epsilon();
    #endif

    next_value();

    double temp = factor1 * double(get_deviate());

    if (temp > factor2)
        return factor2;
    else
        return temp;
}
```

As we increase the complexity of the algorithms, they get slower; thus the **MinimalStandard** is faster than **GeneralPurpose**, with L'Ecuyer algorithm the slowest of all. The nature of a stochastic algorithm determines how "random" our random numbers really need to be; for example, a checkers program is likely to work well with **GeneralPurpose**, while a complex genetic algorithm may require the long repetition cycle of **LEcuyer**.

Roulette Wheels

I introduced the concept of roulette wheel selection in Chapter 2. To recap: this technique simulates a gambler's roulette wheel in which the sections represent probabilities that a value will be chosen. In the case of genetic algorithms, each segment of the wheel represents the reproductive chance for a chromosome as reflected by its fitness. Several of my applications use roulette wheels, which is an obvious indicator that a class is in order—or, in this case, a template.

```
    //-------------------------------------------------
    // RouletteWheel exception type
    class RouletteException : public GAException
    {
    public:
```

```
    RouletteException() : GAException("Inavlid roulette wheel index") { }
};

//-------------------------------------------------
// class RouletteWheel: declaration
template <class T> class RouletteWheel
{
public:
    // creation constructor
    RouletteWheel(const T * weights, size_t size);

    // copy constructor
    RouletteWheel(const RouletteWheel<T> & rw);

    // assignment operator
    void operator = (const RouletteWheel<T> & rw);

    // destructor
    ~RouletteWheel();

    // change the weight of an entry
    T change_weight(size_t i, T weight);

    // interrogation
    size_t get_size() { return SIZE; }
    float get_weight(size_t i);

    // retrieve a random index
    size_t get_index();

protected:
    // number of weights in this wheel
    size_t m_size;

    // array of m_weights
    T * m_weights;

    // total weight of all indexes
    T m_totalWeight;

    // shared random deviate generator
    Coyote::Random<float> m_devgen;

private:
    // internal copy function
    void copy(const RouletteWheel<T> & rw);
};
```

By defining **RouletteWheel** as a template, I allow it to support fitness values of any numeric type, as specified by the argument **T**. When created, a **RouletteWheel** must be supplied a pair of parameters identifying an array of **T** fitness values and a number of elements in that array—stored, respectively, in the allocated array **m_weights** and the variable **m_size**. The value **m_totalweights** contains the total of all fitness values in **m_weights**, and **m_devgen** is a random number generator used to "spin" the wheel.

The constructor copies and sums the array of fitness values; if the pointer is **NULL**, the constructor creates a new array in which all elements contain an equal weight of one. Note that the constructor does not scale the incoming values; it does, however, use the utility function **abs** to convert negative weights to positive values.

```
// creation constructor
template <class T> RouletteWheel<T>::RouletteWheel(const T * weights,
                                                   size_t size)
{
    size_t i;

    m_size    = size;
    m_weights = new T[size];
    m_totalWeight = T(0);

    if (m_weights == NULL)
    {
        for (i = 0; i < size; ++i)
        {
            m_weights[i]   = T(1);
            m_totalWeight += T(1);
        }
    }
    else
    {
        for (i = 0; i < size; ++i)
        {
            m_weights[i]   = abs(weights[i]);
            m_totalWeight += abs(weights[i]);
        }
    }
}
```

In general, you'll want to ensure that your weights array contains only positive values, and that the sum of all weights is greater than zero.

The destructor simply frees memory allocated to the array of weights.

```
// destructor
template <class T> RouletteWheel<T>::~RouletteWheel()
{
    delete [] m_weights;
}
```

I've defined the copy constructor and assignment operator as inline functions containing calls to the utility function **copy**.

```
template <class T> void RouletteWheel<T>::copy(const RouletteWheel<T> & rw)
{
    m_size    = rw.m_size;
    m_weights = new T[m_size];

    m_totalWeight = rw.m_totalWeight;

    memcpy(m_weights,rw.m_weights,sizeof(T) * m_size);
}

// copy constructor
template <class T> inline RouletteWheel<T>::RouletteWheel
                               (const RouletteWheel<T> & rw)
{
    copy(rw);
}

// assignment operator
template <class T> inline void RouletteWheel<T>::operator =
                                  (const RouletteWheel<T> & rw)
{
```

```
    copy(rw);
}
```

The **change_weight** method alters a single weight at a given index within the wheel. This allows dynamic changes to the table.

```
template <class T> T RouletteWheel<T>::change_weight(size_t i, T weight)
{
    if (i >= m_size)
        throw RouletteException();

    m_totalWeight -= m_weights[i];
    m_totalWeight += weight;

    T res = m_weights[i];
    m_weights[i] = weight;

    return res;
}
```

The interrogation method **get_weight** returns the weight values for a specified index.

```
// interrogator
template <class T> inline float RouletteWheel<T>::get_weight(size_t i)
{
    return (i < m_size) ? m_weights[i] : T(-1);
}
```

The **get_index** method returns a randomly-selected index based on the current weights in **m_weights**.

```
template <class T> size_t RouletteWheel<T>::get_index()
{
    T choice = T(m_devgen() * m_totalWeight);
    size_t i = 0;

    while ((i < m_size) && (choice > m_weights[i]))
    {
        choice -= m_weights[i];
        ++i;
    }

    return i;
}
```

Floating-Point Reproduction

The majority of genetic algorithms work on pure bit strings, converting those strings to the desired types for fitness testing. In Lawrence Davis' book *Handbook of Genetic Algorithms*, he transforms a 44-bit string into two floating point values via a series of operations. I've seen similar techniques elsewhere, and I find them a bit cumbersome.

In theory, a GA should have no knowledge of the format of the data it is modifying; however, natural chromosomes do encode some structure in their sequence. Crossover appears to take place in specific positions along the chromosome. And while mutation doesn't care about the chromosome's structure, but its does affect that structure. In context of a computer program, the structure of a chromosome isn't so important as the ability to logically modify its bits through crossover and mutation.

I decided to build tools for the mutation and crossover of encoded floating-point values of types **float** and **double**. The code that follows assumes we are working with 32-bit floats and 64-bit IEEE doubles, which, in my experience, the norm in Intel-based C and C++ compilers.

Data Types

Floating-point numbers contain scaled values that may have a fractional part. The `float` and `double` types implement the single-precision and `double`-precision floating-point formats defined by the Institute of Electrical and Electronic Engineers (IEEE) standard 754-1985. A **float** is a 32-bit value, and a **double** is a 64-bit. These bits in a floating-point value are divided into three components: A sign bit, an exponent, and a mantissa. Figure 3-1 shows the internal format of the **float** and **double** types. *s* indicates the sign bit; *exp* is an abbreviation for exponent.

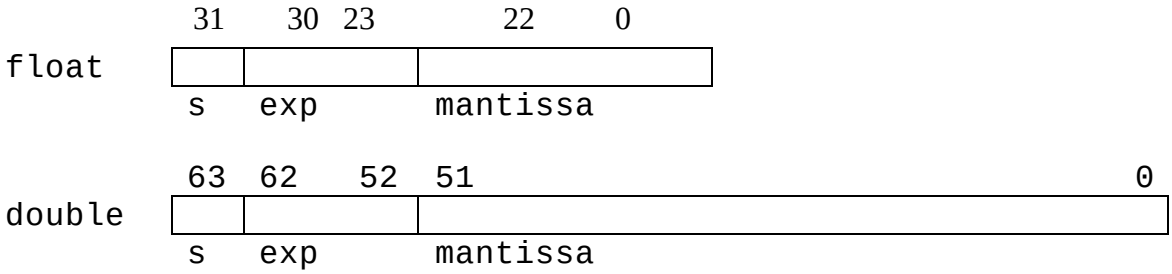

Figure 3-1 Format of IEEE float and double

The highest-order bit in a floating-point value is the sign bit. If the sign bit is one, the value is negative; if the sign bit is zero, the value is positive. In a **float**, the exponent occupies 8 bits and the mantissa uses the remaining 23 bits. A **double** has a 52-bit mantissa and an 11-bit exponent. In addition, the mantissa of **float** and **double** values has an implicit high-order bit of 1.

The mantissa holds a binary fraction greater than or equal to 1 (because of the implied high bit being one) and less than 2. The number of bits in the mantissa affects the accuracy of the floating-point value. A **float** has 6 decimal digits of accuracy, and a **double** (with its longer mantissa) is accurate to 15 decimal digits. Since the mantissa is a binary fraction, and it can't always exactly reflect a decimal value you've tried to store in it. For example, there is no binary fraction that can exactly represent the values 0.6 or 1/3. Floating-point numbers represent an approximation of a decimal value; this is where rounding errors come from.

The exponent is a binary number representing the number of binary digits the mantissa is shifted left (for a positive actual exponent) or right (for a negative actual exponent). The exponent is a biased value; you calculate the actual exponent value by subtracting a bias value from the exponent stored in the value. The bias for a **float** is 127; the bias for a **double** is 1023. Thus, a **float** value with an exponent of 150 would represent a number with an exponent of 23. The constants **FLT_MIN**, **FLT_MAX**, **DBL_MIN**, and **DBL_MAX** define the minimum and maximum values for floating point numbers, in the Standard C header file **float.h**. Most C++ compilers define those constants as

```
#define FLT_MIN  1.17549e-38
#define FLT_MAX  3.40282e+38

#define DBL_MIN  2.22507385850720e-308
#define DBL_MAX  1.79769313486232e+308
```

Two other relevant **float.h** constants are **FLT_EPSILON** and **DBL_EPSILON**, which represent the smallest possible difference between two **float** and **double** values.

```
#define FLT_EPSILON  1.19209e-07
#define DBL_EPSILON  2.22044604925031e-16
```

In Standard C++, the **numeric_limits** template (defined in the <limits> header) is specialized to describe the characteristics of each numeric data type. For the purposes at hand, the relevant members of **numeric_limits** are

Bestiary

Floating-point numbers can take on some unusual values. It's possible for a floating-point number to represent positive and negative infinity, for example. Or, a floating-point value may be in a special format that doesn't represent a valid number. Any routines that randomly change floating-point numbers must avoid generating these unusual values.

A floating-point value represents *infinity* when the bits in the exponent are all one and the bits in the mantissa are all zero. When both the mantissa and exponent are zero, the floating-point number is zero. Infinity, as well as zero, can have a sign. Positive and negative zero operate identically in calculations and comparisons.

When is a number not a number? When its exponent is all ones and its mantissa contains any set of bits that is not all zeros (which would indicate an infinity). A value in this format is known as a *NaN* (*N*ot *a N*umber). The sign bit for a NaN is irrelevant.

So what is the point of knowing these strange floating-point values? For the most part, C++ compilers do not support the use and processing of unusual floating-point values. To maximize portability, we want to do is avoid the creation of unusual numbers through floating-point reproduction. And in looking at the above, we can see an obvious commonality between the troublesome NaNs and infinities: both types have exponents filled with ones.

Mutation in Parts

A floating-point value contains three components that can be changed during mutation and crossover: the sign bit, exponent, and mantissa. Changing the exponent and sign have the most dramatic affect on a floating-point value, since the change of one bit can dramatically alter the magnitude of a number. Assuming that all bits have an equal chance of mutation, we get the following probabilities that a random bit change will affect a specific component:

	float	double
sign bit	3.1%	1.6%
exponent	25.0%	17.1%
mantissa	71.9%	81.3%

Depending on the application, I've found that those fixed percentages don't always allow for the creation of effective mutations. The exponent, in particular, is so likely to be changed that numbers often fluctuate wildly within a population after mutation. I decided to create a system for the roulette-wheel selection of the component to be mutated, allowing me to weight mutation in favor of changing the mantissa.

I create a class named **FloatBreeder**, which defines the parameters of mutation and crossover for **float** and **double** types.

```
class FloatBreeder
{
public:
    FloatBreeder(float sweight = 5.0F,
                 float eweight = 5.0F,
                 float mweight = 90.0F);

    float  mutate(float  f);
    double mutate(double d);

    float  crossover(float  f1, float  f2);
    double crossover(double d1, double d2);

protected:
    const float m_total_weight;
    const float m_sign_weight;
    const float m_exp_weight;

    static Coyote::Random<float> m_devgen;
};
```

When creating a **FloatBreeder** object, you'll need to supply three floating-point values representing the relative chances of changing the parts of a floating-point number.

```
FloatBreeder::FloatBreeder(float sweight, float eweight, float mweight)
    : m_total_weight(sweight + eweight + mweight),
      m_sign_weight(sweight),
      m_exp_weight(eweight)
{
    // intentionally blank
}
```

The **mutate** functions use those values in selecting the sections of **float** and **double** values to be mutated.

```
float FloatBreeder::mutate(float f)
{
    // mask for exponent bits
    static const long FExpt = 0x7F800000L;

    long x, n, mask;

    // choose section to mutate
    float mpick = m_devgen() * m_total_weight;

    // copy float to long for manipulation
    memcpy(&x,&f,sizeof(long));

    // if all exponent bits on (invalid #), return original
    if ((x & FExpt) == FExpt)
        return f;

    // mutate
    if (mpick < m_sign_weight)
    {
        // flip sign
        mask = 0x80000000L;

        if (x & mask)
```

```
            x &= ~mask;
        else
            x |= mask;
    }
    else
    {
        mpick -= m_sign_weight;

        if (mpick < m_exp_weight)
        {
            // mutate exponent while number is valid
            do  {
                n    = x;
                mask = 0x00800000L << int(m_devgen() * 8.0F);

                if (n & mask)
                    n &= ~mask;
                else
                    n |= mask;
            }
            while ((n & FExpt) == FExpt);

            x = n;
        }
        else
        {
            // flip bit in mantissa
            mask = 1L << int(m_devgen() * 23.0F);

            if (x & mask)
                x &= ~mask;
            else
                x |= mask;
        }
    }

    // done!
    float res;
    memcpy(&res,&x,sizeof(float));
    return res;
}

double FloatBreeder::mutate(double d)
{
    // mask for exponent bits
    static const long DExpt = 0x7FF00000UL;

    long x[2], n, mask, bit;

    // choose section to mutate
    double mpick = m_devgen() * m_total_weight;

    // copy double to pair of longs for manipulation
    memcpy(x,&d,2 * sizeof(long));

    if (mpick < m_sign_weight)
    {
        // flip sign
        mask = 0x80000000L;

        if (x[1] & mask)
            x[1] &= ~mask;
        else
            x[1] |= mask;
```

```
        }
        else
        {
            mpick -= m_sign_weight;

            if (mpick < m_exp_weight)
            {
                // mutate exponent while number is valid
                do  {
                    n = x[1];
                    mask = 0x00100000L << int(m_devgen() * 11.0F);

                    if (n & mask)
                        n &= ~mask;
                    else
                        n |= mask;
                }
                while ((n & DExpt) == DExpt);

                x[1] = n;
            }
            else
            {
                bit = long(m_devgen() * 52.0F);

                if (bit > 31L)
                {
                    bit -= 32L;
                    mask = 1L << (int)bit;

                    if (x[1] & mask)
                        x[1] &= ~mask;
                    else
                        x[1] |= mask;
                }
                else
                {
                    // flip bit in mantissa
                    mask = 1L << (int)bit;

                    if (x[0] & mask)
                        x[0] &= ~mask;
                    else
                        x[0] |= mask;
                }
            }
        }

        // done
        double res;
        memcpy(&res,x,sizeof(double));
        return res;
    }
```

The **Mutate** functions use a bitmask to examine the bits in a value's exponent, ensuring that that any output value is not a NaN or infinity.

My experiments advise me to limit the mutability of the exponent to under 15 percent, keeping the sign bit mutation rate at about two or three percent. You don't have to take my word for it; the next chapter implements a genetic algorithm for which you can set the weights for each

component of floating-point values. That allows you to test my results and explore your own ideas.

Crossover

Floating-point crossover is a simple operation, implemented as two member functions named **crossover**:

```
float FloatBreeder::crossover(float f1, float f2)
{
    // mask for exponent bits
    static const long FExpt = 0x7F800000L;

    long  l1, l2, lcross, mask;
    float fcross;

    // store values in longs
    memcpy(&l1,&f1,sizeof(long));
    memcpy(&l2,&f2,sizeof(long));

    do  {
        // create mask
        mask   = 0xFFFFFFFFL << size_t(m_devgen() * 32.0F);

        // generate offspring
        lcross = (l1 & mask) | (l2 & (~mask));
    }
    while ((lcross & FExpt) == FExpt);

    // copy result to float and return
    memcpy(&fcross,&lcross,sizeof(float));

    return fcross;
}

double FloatBreeder::crossover(double d1, double d2)
{
    // mask for exponent bits
    static const long DExpt = 0x7FF00000L;

    long   l1[2], l2[2], lcross[2], mask, bit;
    double fcross;

    // store values in longs
    memcpy(l1,&d1,sizeof(double));
    memcpy(l2,&d2,sizeof(double));

    do  {
        // calculate bit position for flip
        bit = size_t(m_devgen() * 64.0F);

        if (bit > 31) // if flip in high-order word
        {
            // create mask
            mask   = 0xFFFFFFFFL << int(bit - 32L);

            // duplicate low-order word of first parent
            lcross[0] = l1[0];

            // crossover in high-order word
            lcross[1] = (l1[1] & mask) | (l2[1] & (~mask));
        }
```

```
            else
            {
                // create mask
                mask   = 0xFFFFFFFFL << int(bit);

                // crossover in low-order word
                lcross[0] = (l1[0] & mask) | (l2[0] & (~mask));

                // duplicate high-order word of first parent
                lcross[1] = l1[1];
            }
        }
        while ((lcross[1] & DExpt) == DExpt);

        // copy and return
        memcpy(&fcross,lcross,sizeof(double));
        return fcross;
    }
```

Why no long double?

What follows is an editorial comment; you can skip it if you like.

I didn't implement the mutation and crossover operations for **long double**s because I don't use that type in my programs. On a PC, the 80-bit **long double** type represents the internal floating-point format used by the numeric coprocessor. A **long double** has 18 digits of accuracy; it is used internally by the math coprocessor so that the results of calculations can be rounded to produce a very accurate 15 digits of precision in a **double**. The extra three digits in a **long double** provide improved accuracy; they should be viewed *very* suspiciously by a numerical programmer since the coprocessor never means for them to be considered or used.

In my view, current implementations of **long double** are nothing more than frivolous attempts at adding bullets to the compiler advertisement. I'd be far happier if C and C++ compiler vendors would implement a full suite of functions for manipulating **float** values, as required by Standard C++ and the forthcoming C9X. Double precision values already exceed the accuracy needs of most scientific and engineering tasks; for most calculations, **float** is quite adequate.

And for those folks who wonder why some of programmers—including myself—stick with dusty old FORTRAN: It's because FORTRAN is *still* the only language that provides full intrinsic support for single and double precision floating-point and complex numbers.

Onward

Okay, enough grousing! The tools above are components of the designs in subsequent chapters, where I implement complex genetic algorithms. In Chapter 4, I'll implement an experimental environment for testing the efficacy of advanced genetic algorithms in solving complex problems.