# Y2 Computational Physics Laboratory

UNIVERSITY OF BIRMINGHAM



# Exercise 3

## Monte Carlo or Bust: Physics Simulations using Random Numbers

**Aims:**

- To learn how to generate random numbers using C++
- To simulate a one-dimensional random walk and to analyse its probability distribution
- To simulate a Geiger counter and to analyse the counting distribution
- **1 mark given for overall presentation**

**Duration:** 3 weeks

**Hand-in deadline:** **At the end of Session #8**

---

## Introduction

Monte Carlo simulations use computer-generated random numbers to simulate *stochastic* (*i.e.*, random) physical processes, such as the Brownian motion of a particle in a liquid or Higgs-boson production at the Large Hadron Collider. In this exercise, you will use random numbers to simulate (*i*) a particle undergoing a random walk (a simplified model for Brownian motion) and (*ii*) the random clicks of a Geiger counter. Using a `for` loop to repeat the simulations many times, you will also investigate the statistical distributions for the random walk and for particle counting.

## A simple random-number generator

A simple random-number generator (RNG) produces a *pseudorandom* sequence (also known as a *Lehmer* sequence) of integers $i_n$ using the iteration formula

$$i_{n+1} = mi_n + c \pmod{N}, \qquad (1)$$

where $m$ and $c$ are (carefully chosen) fixed integers, and 'mod $N$' means that we evaluate the right-hand side *modulo* $N$ to produce the next number $i_{n+1}$ between 0 and $N$-1. (This amounts to taking the remainder after integer division of $mi_n + c$ by $N$.) The sequence appears to be random, with no obvious correlation between successive numbers. However, the first number, which is called the *seed*, actually determines all the subsequent numbers in the sequence through the iteration formula (1). For that reason, the numbers are said to be 'pseudorandom' rather than random.

The modular arithmetic in (1) can be performed in C++ by using the '%' operator. However, a simpler approach, used by most practical RNGs, exploits the integer overflow that occurs when the right-hand side of equation (1) is greater than the largest 32-bit integer, $2^{32}$-1, which is equivalent to setting $N = 2^{32}$. The iteration statement for a simple RNG in C++ is then simply

```
iran = IM*iran+IC;
```
(2)

Here, `iran` is an `unsigned` integer variable that stores the 32-bit random integer between 0 and $2^{32}$-1, whilst `IM` and `IC` are constants that we declare as `const unsigned IM=1664525` and `const unsigned IC=1013904223`, which are the carefully chosen values recommended by *Numerical Recipes*. (If you are working on your own computer, you should run the program in the test project RandomTest to verify that the simple RNG algorithm works correctly on your machine.)

The following code from sample project Random1 shows the simple RNG in action:

```
#include <iostream>         //Header for C++ output stream 'cout'
#include <conio.h>          //Header for console I/O _getch()
#include <ctime>            //Header for time functions
using namespace std;

int main(void)
{
   const unsigned IM=1664525;             //Integer constant for the RNG algorithm
   const unsigned IC=1013904223;          //Integer constant for the RNG algorithm
   const double zscale=1.0/0xFFFFFFFF; //Scaling factor for random double between 0 and 1
   unsigned iran=9999;                    //Seeds the random-number generator
   double z;                              //Holds a random double between 0 and 1
   const int nsteps=20;                   //Number of steps

   //Uncomment the following line to use a random seed:
   //iran=time(0); //Seeds the RNG from the system time

   //Write to the console using cout:
   cout<<"Seed = "<<iran<<endl; //Prints the RNG seed
   cout<<"Calculating "<<nsteps<< " random numbers..."<<endl<<endl;
   cout<<"i\tiran\t\tz"<<endl; //Prints a column header

   //Generate the random numbers:
   for(int i=1;i<=nsteps;i++)
   {
       z=zscale*double(iran=IM*iran+IC); //RNG algorithm based on Numerical Recipes 'ranqd2'
       cout<<i<<"\t"<<iran<<"\t"<<z<<endl; //Prints to the screen
   }

   cout<<"Press any key to exit...";
   _getch();  //Waits for a key to be pressed

   return 1;
} //End of main()
```

### Points to note:

1. To produce a random number `z` between 0 and 1, we multiply `iran` by the scaling factor `zscale=1.0/0xFFFFFFFF`, where `0xFFFFFFFF` represents $2^{32}$-1 in hexadecimal (base-16) notation.
2. The statement `z=zscale*double(iran=IM*iran+IC)` is a *nested* assignment statement, which instructs the computer to take the result of `iran=IM*iran+IC`, convert it to type `double`, multiply by `zscale`, and then put the result into `z`. Nesting statements in this way (rather than using multiple statements) can produce fast code. This is useful to speed up the RNG, which may be called hundreds of millions of times.
3. However, nesting can be taken to extremes, where a whole program is nested into just a few statements. Although this is fun to do for a challenge, the resulting 'obfuscated' code is fast but virtually unintelligible. Hence, it's best not to nest statements too deeply, so that your code can still be easily understood, debugged and modified by human beings.
4. The above program uses a fixed seed in the initialization statement `unsigned iran=9999`. Hence, every time the program runs, the sequence of 'random' numbers is exactly the same. If you need the sequence to be different, then uncomment the statement `iran=time(0)`, which seeds the RNG with the system time (the number of seconds since 1st January 1970). Try running the sample project Random1 several times with and without a fixed seed.

Project Random2 constructs a histogram for the distribution of 20 million random numbers between 0 and 1. Here, the nested statement `hist[int(nbin*zscale*double(iran=IM*iran+IC))]++` uses the `hist[]` array with `nbin = 20` to count the random numbers in each of 20 bins in the histogram. Run this program to check that the random numbers have a uniform distribution. We would expect one million counts per bin, but random fluctuations cause the actual number of counts to be slightly different for each bin. A useful rule of thumb is that the statistical fluctuations for $N$ counts are $\sim\sqrt{N}$.

Although the random numbers in Random2 are uniformly distributed, there are in fact some subtle serial correlations between them, as explained in *Numerical Recipes*. For real scientific work, you should therefore use a more sophisticated RNG that eliminates correlations, as discussed in the *Additional Material* at the end of this manual. However, for the purposes of this exercise, the simple RNG above is sufficiently random. It also has the virtue of keeping the C++ code fairly simple.

## Problem 1: Simulating a Random Walk

The problem of a particle undergoing a random walk occurs in many different areas of physics, most notably as a simple model for a particle undergoing Brownian motion or diffusion (Feynman 1963). You can produce a one-dimensional random walk without using a computer by repeatedly flipping a coin: if the result is 'heads' you take a step forwards, whilst if it's 'tails' you take a step backwards. However, instead of us flipping a coin, we want the PC to do the work by performing a Monte Carlo simulation of a random walk using the RNG from the previous section.

We have seen that the random floating-point numbers `z` are uniformly distributed between 0 and 1. Hence, to simulate the flipping of a coin, we simply check whether `z < 0.5` ,which will occur, at random, half of the time. We can therefore simulate a random walk with $N$ steps by using a `for` loop containing `if...else` statements of the form

```
if(zscale*double(iran=IM*iran+IC) < 0.5) x++; //Steps forward
else x--;                                     //Steps backward
```

Here, `x` is an integer variable that records the position of the particle. The `if` statement generates a new random number, and the particle steps forwards if the random number is less than 0.5; otherwise, it steps backwards.

Since forward and backward steps are equally likely, we expect the displacement of the particle after a random walk of $N$ steps on average to be zero. However, owing to statistical fluctuations, the actual displacement at the end of a single random walk is, in general, non-zero. This is illustrated in Figure 1 for a random walk with $N = 100$ steps. It is only by repeating the random walk many, many times, that we find that the average (or *mean*) displacement $\bar{x}$ is zero. Moreover, as shown by Feynman, the *mean-square* displacement $\overline{x^2}$ after $N$ steps of unit size is

$$\overline{x^2} = N \,. \tag{3}$$

Hence, the *root mean-square* displacement $x_{RMS}$, which is a useful measure of the typical distance moved after $N$ steps, is

$$x_{RMS} = \sqrt{N} \,. \tag{4}$$

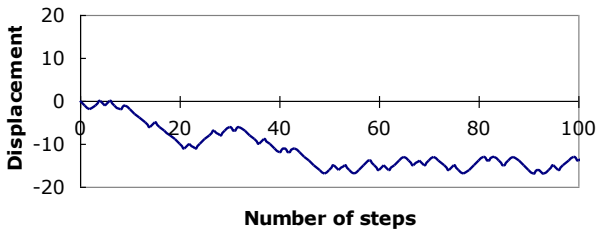


**Figure 1. Random walk with 100 steps**

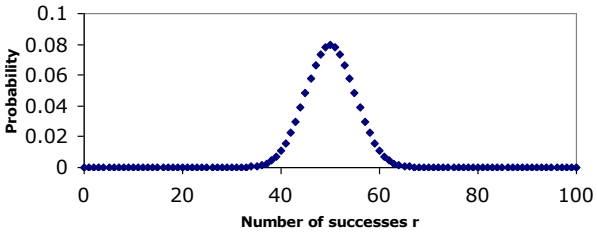


**Figure 2. Binomial distribution for $N = 100$**

If the particle takes $r$ forward steps and $N$-$r$ backward steps, then the displacement after $N$ steps is

$$x = r - (N - r) = 2r - N \,. \tag{5}$$

As discussed by Feynman, the probability $P(r, N)$ of obtaining $r$ successes in $N$ trials is given by the *binomial distribution*. If the probability of success in a single trial is ½, then

$$P(r,N) = {}^{N}C_r / 2^N \,, \qquad \text{where} \quad {}^{N}C_r = \frac{N!}{(N-r)!\,r!} \,. \tag{6}$$

This probability distribution arises because the binomial coefficient ${}^{N}C_r$ represents the number of ways of obtaining $r$ successes in $N$ trials, whilst $2^N$ is the total number of possible outcomes, all of which are equally likely.

Figure 2 shows the binomial distribution from equation (6) for the number of successes $r$ when $N = 100$. It follows from equation (5) that the probability distribution for the displacement $x$ in a random walk is also given by equation (6), in which $r = (x + N)/2$.

### The problem—what's required

1. Write a C++ program to simulate a random walk of 100 steps. Save the results to a text file and paste them into an Excel spreadsheet so that you can plot the random walk, as in Figure 1. Repeat this three times with a random seed.
2. Modify your program to perform one million random walks of 100 steps and construct a histogram showing the distribution of displacements after 100 steps.

### Your solution—what's needed

1. Include clearly commented printouts of both versions of your code. The marker must be able to follow the code easily and understand its logic and function. **[10 marks]**
2. Plot your results from the first program for four random walks on the same Excel chart. Are your results consistent with the value of $x_{RMS}$ expected from equation (4)? **[3 marks]**
3. Plot your histogram from the second program in Excel. On the same chart, plot the theoretical histogram from the binomial distribution, for which Excel has an inbuilt function. **[2 marks]**

## Monte Carlo simulations as a function of time

Atomic and molecular transitions (such as the steps in a random walk) occur at random times but (usually) at a fixed average rate. An example of this is the radioactive decay of $N$ nuclei, which is governed by the differential equation

$$\frac{dN}{dt} = -\lambda N , \qquad (7)$$

where $\lambda$ is the radioactive decay constant. The number of decays in a short interval $\delta t$ is $\lambda N \delta t$. Since the nuclei decay independently of each other, the probability of any given nucleus decaying in time $\delta t$ is therefore just $\lambda \delta t$. This result applies quite generally to stochastic processes: the probability of an event occurring in a short interval $\delta t$ is given by a *rate constant* times $\delta t$.

We can produce a Monte Carlo simulation as a function of time by taking small time steps $\delta t$. For the random walk, the particle steps forwards or backwards if the computer-generated random number $z < (k_+ + k_-)\delta t$, where $k_+$ and $k_-$ are the rate constants for forward and backward steps, and $z$ is uniformly distributed between 0 and 1. Indeed, the particle steps forwards if $z < k_+ \delta t$ and backwards if $k_+ \delta t \le z < (k_+ + k_-)\delta t$. Note, however, that we must keep the interval $\delta t$ sufficiently small such that there is a negligible probability of the particle making two or more steps in time $\delta t$. For an accurate simulation, we might choose $\delta t$ such that the probability of a step in time $\delta t$ is $10^{-2}$. In that case, 99 times out of 100 *nothing* happens! This simple approach is therefore computationally very inefficient.

More efficient Monte Carlo simulations exploit the fact that random events with a fixed rate constant have an *exponential* time distribution. One can see this from the solution to the radioactive-decay equation (7), which is $N = N_0 e^{-\lambda t}$, where $N_0$ is the number of nuclei at $t = 0$. The probability that a given nucleus has not decayed at time $t$ is therefore $N/N_0 = e^{-\lambda t}$, and the probability that it then decays in a time interval $\delta t$ is $\lambda \delta t$. Hence, the probability that the nuclear decay occurs in the time interval $t$ to $t+\delta t$ is $e^{-\lambda t} \lambda \delta t$. It follows that the distribution function for nuclear decays is $\lambda e^{-\lambda t}$. This result applies quite generally to the distribution of times between random events that occur at an average rate $\lambda$, and which are therefore separated by an average time $\tau = 1/\lambda$.

We can generate random times $t$ with the required exponential distribution very simply by means of the transformation $z = e^{-t/\tau}$, where the random number $z$ is uniformly distributed between 0 and 1. Hence, we find

$$t = -\tau \ln z . \qquad (8)$$

Instead of taking many small time steps of size $\delta t$, we can therefore use this equation to determine the time $t$ of the next random event by generating just a single random number $z$.

## Problem 2: Simulating a Geiger Counter

Equation (8) provides us with a simple way of simulating the random clicks of a Geiger counter (or, indeed, any other type of detector) that is counting particles at a fixed rate. As with the steps in the random-walk simulation,

we can use a `for` loop to generate successive clicks of the Geiger counter. The time delay between each click is calculated using a statement inside the loop of the form

```
t = -tau*log(zscale*double(iran=IM*iran+IC)); //Time to next event (s)
```

where `tau` is the average time between clicks. This statement effectively translates equation (8) into C++. (Following a click, a real Geiger counter actually has a short *dead time* during which it cannot detect another particle. It is important to correct for this at high counting rates. However, for simplicity, we do not include the dead time in this simulation.)

To make the simulation more realistic on a Windows PC, we may also add the statements

```
Sleep(1000.*t);                              //Waits for event (delay is in ms)
PlaySound(L"click.wav",0,SND_FILENAME|SND_ASYNC); //Plays the click.wav sound
```

The `Sleep()` function causes the program to pause for a given number of milliseconds, which simulates the random delay[†]. The `PlaySound()` function then plays a sound file to imitate the click of the Geiger counter. (You'll need to unzip the `click.zip` file from Canvas to your project folder[‡], and you'll need to use headphones to listen to the sound on a cluster PC.) Note that these functions are specific to Windows. Your .cpp file must also include the pre-processor statements

```
#include <Windows.h>              //Header file for Windows API functions
#pragma comment(lib,"Winmm.lib")  //Library for playing a wav file
```

### The problem—what's required

1. Write a C++ program to simulate 50 random counts of a Geiger counter assuming an average count rate of 5 $s^{-1}$. Your program should produce a click for each count after an appropriate delay and print the delay time in the console window.

### Your solution—what's needed

1. Include a clearly commented printout of your code. The marker must be able to follow the code easily and understand its logic and function. **[5 marks]**

## Problem 3: Counting statistics and the Poisson distribution

If we repeatedly count the number of clicks of a Geiger counter in time intervals of length $\Delta t$, then we find that the number of counts $r$ fluctuates. The probability $P_r$ of $r$ counts in time $\Delta t$ is given by the *Poisson distribution*

$$P_r = \frac{\mu^r}{r!} e^{-\mu}, \tag{9}$$

where $\mu$ is the mean number of counts in time $\Delta t$. (The Poisson distribution is, in fact, a limiting case of the binomial distribution when the number of trials $N \to \infty$ while the mean $\mu$ remains constant.) The standard deviation $\sigma$ of the Poisson distribution is

$$\sigma = \sqrt{\mu}\,. \tag{10}$$

So, on average, we expect $\mu$ counts in time $\Delta t$ with fluctuations about that average of $\sim\sqrt{\mu}$. These fluctuations about the average are readily apparent in figure 3, which shows the Poisson distribution for $\mu = 5$.

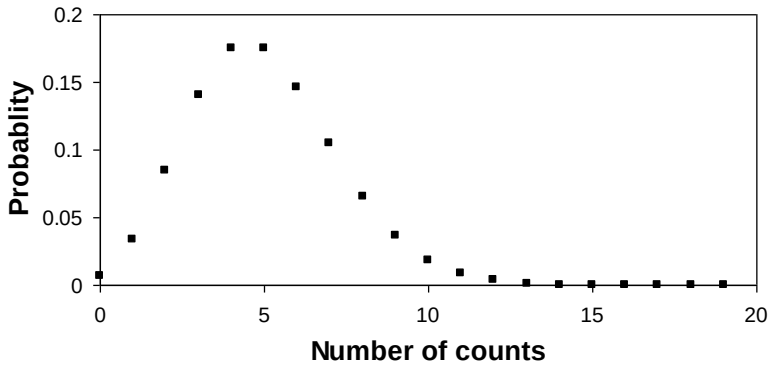


**Figure 3. Poisson distribution for $\mu = 5$**

[†] Owing to the multi-tasking nature of Windows, the actual delay time may be slightly different.
[‡] If you run your program outside Visual Studio, `click.wav` must be in the same folder as your `.exe` file.

If we measure the number of counts $r_i$ for $n$ time intervals of length $\Delta t$, then the average number of counts for the $n$ samples is

$$\bar{r} = \frac{1}{n}\sum_{i=1}^{n} r_i \,. \tag{11}$$

This *sample mean* $\bar{r}$ is our best experimental estimate of the *population mean* $\mu$, which is the mean value that we obtain as $n \to \infty$ The uncertainty in the measured value is the *standard error on the mean* $\sigma_{sem}$, which is

$$\sigma_{sem} = \frac{s}{\sqrt{n-1}}, \tag{12}$$

where the standard deviation $s$ of the sample is determined by

$$s^2 = \overline{r^2} - \bar{r}^2 = \frac{1}{n}\sum_{i=1}^{n}\left(r_i - \bar{r}\right)^2 . \tag{13}$$

Note that when $n = 1$ the RHS of equation (12) is 0/0. This means that we cannot estimate $\sigma_{sem}$ from just a single measurement. We must make *repeated* measurements to provide an objective statistical estimate of $\sigma_{sem}$.

If the number of samples $n$ is large, then we expect $s \approx \sigma$. In that case, we may approximate (12) by

$$\sigma_{sem} \approx \sigma/\sqrt{n} \,. \tag{14}$$

Equations (11) to (14) apply to averaging *any* kind of experimental measurements, not just particle counting. As a general rule, equation (14) demonstrates that to obtain a more accurate measurement of the mean $\mu$, we must take a large number of measurements $n$. The accuracy then improves as $\sqrt{n}$ .

For the particular case of particle counting, we know from the Poisson distribution that $\sigma = \sqrt{\mu}$ . Hence,

$$\frac{\sigma_{sem}}{\mu} \approx \frac{\sigma}{\mu\sqrt{n}} = \frac{1}{\sqrt{n\mu}} . \tag{15}$$

Hence, to improve the accuracy we should increase the total number of counts $n\mu$, which is proportional to the total counting time $T = n\Delta t$. If we choose $T = 10{,}000$ s, then it does not actually matter if take $n = 100$ and $\Delta t = 100$ s or $n = 10{,}000$ and $\Delta t = 1$ s. The total number of counts $n\mu$ is the same for both cases, and we therefore expect the *same* fractional accuracy $1/\sqrt{n\mu}$ in determining the average count rate. This is in accord with the simple rule of thumb that the fluctuation in $N$ counts is $\sim\sqrt{N}$ .

### The problem—what's required

1. Write a C++ program to construct a histogram with `nbin = 20` for the number counts of the Geiger counter in 10,000 intervals of $\Delta t = 1$ s assuming an average count rate of 5 $s^{-1}$. You can determine the number of counts in a time interval `deltat` with a `while` statement of the form
   ```
   while((t-=tau*log(zscale*double(iran=IM*iran+IC)))<deltat)count++;
   ```
2. Use your program to determine the mean number of counts in 1 s and the standard error on the mean.

### Your solution—what's needed

1. Include a clearly commented printout of your code. The marker must be able to follow the code easily and understand its logic and function. **[5 marks]**
2. Plot your histogram in Excel and compare it with that expected from the Poisson distribution, for which Excel has an inbuilt function. **[2 marks]**
3. Write down the sample mean and the standard error on the mean. On the basis of these results, is your simulation consistent with the theoretical count rate of 5 $s^{-1}$? **[2 marks]**

## Further reading


Ammeraal, L. (2000) *C++ for Programmers* (Wiley).
Barford, N.C. (1985) *Experimental Measurements: Precision, Error and Truth* (Wiley).
Feynman, R.P. *et al*. (1963) *The Feynman Lectures on Physics*, Vol. I, Ch. 6 (Addison-Wesley).
Press, W.H. *et al*. (1992) *Numerical Recipes in C*, Ch. 7. (Cambridge)



Neil Thomas
n.thomas@bham.ac.uk

## Additional Material (optional and non-assessed)

The sample project RandomClass uses a C++ class to create a random-number generator. In C++, a *class* is a template for a new type of object that contains both *data* and *methods* (*i.e.*, functions) that operate on the data. The file RandomNumberGenerator.h declares the `RandomNumberGenerator` class as follows:

```
class RandomNumberGenerator
{
   public:
        RandomNumberGenerator(unsigned iseed);    //Constructor
        RandomNumberGenerator();                  //Default constructor
        double Z();          //Returns a random double between 0 and 1.0
   private:
        unsigned iran;                            //Holds random integer
        double ztable[32];                        //Array for shuffle table
        void Init(unsigned iseed);                //Initializes the shuffle table
};
```

The `public` section of the class declares data and methods that are publicly accessible from outside the class. In this case, there are no publicly accessible data members, but there are three public methods. The first method `RandomNumberGenerator(unsigned iseed)` is a function that has the same name as the class itself. It is called a *constructor*, because this method is called when a `RandomNumberGenerator` object is created. The constructor performs initialization, such as seeding the RNG with the value of the parameter `iseed`. A second ('overloaded') version of the constructor `RandomNumberGenerator()` is also provided that does not take a parameter. This *default constructor* seeds the RNG using the system time. The third public method is a function `Z()` that returns the value of a random double between 0 and 1. Its operation is explained below.

The `private` section of the class declares data and methods that can only be accessed from within the class itself. The first piece of data here is `iran`, which holds the current value of the integer in the pseudorandom sequence. Users of the `RandomNumberGenerator` class cannot access `iran` directly and cannot therefore interfere with the correct operation of the RNG. This principle of 'data hiding' is widely used in C++ programming to protect critical data and hence to produce more reliable code. Also hidden from users of the `RandomNumberGenerator` class are an array `ztable[32]`, which is used as a 'shuffle table' to remove correlations between random numbers, and an `Init()` method, which is called by the constructor.

The `public` methods of the `RandomNumberGenerator` class provide the interface for users, without allowing them direct access to the critical data contained in the `private` data members, `iran` and `ztable[32]`. In particular, users can call the public `Z()` method, which makes use of both `iran` and `ztable[32]`. The following code for this method is contained in the file RandomNumberGenerator.cpp, which contains the 'implementation' of the `RandomNumberGenerator` class:

```
double RandomNumberGenerator::Z()
{
   int index=iran % 32;//iran modulo 32 -> random index from 0 to 31
   double z=ztable[index];  //Chooses z at random from the shuffle table
   ztable[index]=zscale*double(iran=IM*iran+IC); //Generates a new random number for the shuffle table
   return z; //Returns random z between 0 and 1
} //End of Z()
```

The full name of the `Z()` method is `RandomNumberGenerator::Z()`, which specifies that this is the `Z()` method belonging to the `RandomNumberGenerator` class. The first statement in `Z()` uses the expression `iran % 32` to calculate a random `index` between 0 and 31 from the current value of `iran`. Note that, since `iran` is a data member of the `RandomNumberGenerator` class, its value is always available to member functions such as `Z()`. In contrast, `index` and `z` are local variables that must be declared within `Z()` before they can be used.

Since `index` is a random number between 0 and 31, the statement `z=ztable[index]` extracts a number *at random* from the shuffle table, which was previously filled with random numbers by the `Init()` method. We then use the simple RNG algorithm `zscale*double(iran=IM*iran+IC)` to replace the random number `ztable[index]` that has been removed from the shuffle table. The use of the random `index` with the shuffle table ensures that the order in which random numbers are extracted from the table is quite different from the order in which they were generated. The shuffle table therefore effectively removes correlations between numbers in the pseudorandom Lehmer sequence.

As explained in *Numerical Recipes*, one should always use a shuffle table when the quality of the RNG is important, as in scientific research. (An alternative, more modern, approach is to use a 'Mersenne twister' algorithm. This, however, is somewhat more complicated than the shuffle-table method.)

The file RandomClass.cpp demonstrates how to use the `RandomNumberGenerator` class. It includes the pre-processor statement

```
#include "RandomNumberGenerator.h"
```

which provides the compiler with the declaration of the `RandomNumberGenerator` class. In the `main()` procedure, we can then create a `RandomNumberGenerator` 'object' called `R`, seeded with the number `999`, through the declaration

```
RandomNumberGenerator R = RandomNumberGenerator(999);
```

The compiler recognises that `RandomNumberGenerator` is the name of a new type (analogous to basic types such as `int` and `double`), and it creates an object of that type by invoking the object's constructor. The following statement then calls the `Z()` method to obtain a random number from object `R`:

```
z1 = R.Z(); //Calls Z() method for object R using direct addressing
```

The "." operator here is known as the *direct* member operator.

Object `R` above is created 'automatically' by the compiler, which allocates storage to it. We can also create a `RandomNumberGenerator` object 'dynamically' at run-time through a statement of the form

```
RandomNumberGenerator *p = new RandomNumberGenerator(); //p points to a dynamically allocated RNG
```

Here, operator `new` returns a pointer to a `RandomNumberGenerator` object, created in this case using the default constructor. Variable `p` is therefore a *pointer* that holds the memory address of the new `RandomNumberGenerator` object. The following statement calls the `Z()` method to obtain a random number from this object:

```
z2 = p->Z(); //Calls Z() method for object *p using a pointer
```

The "->" operator here is known as the *indirect* member operator. Note that, since we created this object dynamically, it is important to delete it with `delete p` before the program terminates.

Classes are the foundation of *object-oriented programming* (OOP), whereby a program becomes a collection of 'objects' that exchange 'messages' by calling each other's methods. This allows the programmer to build sophisticated objects that have a user interface through their public methods and (usually) with their inner workings safely hidden away in private data. Microsoft provides powerful class libraries (*Microsoft Foundation Class* and the *.NET Framework*) that contain classes representing all of the objects such as buttons, menus, windows, *etc*., that are required in a Windows program. The .NET Framework makes it particularly simple to write a Windows program in Visual Studio as a 'Windows Forms Application'.

Although C++ can be used for OOP, it is actually a hybrid language that preserves the non-object oriented features of the original C language. More modern languages, such as Java and C#, are entirely object oriented. Indeed, C# draws on both C++ and Java and includes many features that make programming easier and more reliable. Using C# with the .NET Framework is therefore an easy, but powerful, way to write Windows programs.