C++11 Randomisation Library

The C+11 specification provides more reliable, versatile and less biased random number generators.

Issues with Randomisation

Problems with randomisation include

- biased distributions
- pseudo-random sequences
- getting a seed

A Better Scheme

When using the C++11 randomisation you need to:

1. Choose an engine that generates raw pseudo-random numbers, some engines provide better simulations of randomness than others
2. Choose an initial seed (unless you're debugging)
3. Choose a distribution, the STL randomisations are optimised to produce numbers in a desired range as efficiently as possible

To use: `#include <random>`, all symbols are part of the `std` namespace

```
default_random_engine eng(seed); // randomisation engine
uniform_int_distribution<int> dist(1, 6); // distribution
```

The same engine should be kept in use. Starting a new engine might be problematic as it may give the same sequence of values (unless over a second has past).

The seed should be changed if a new randomisation instance is going to be created.

Common Engines

There are a wide choice of engines in the C++11 randomisation library. The most commonly used engines are:

- `default_random_engine`: default engine for randomisation - choice is implementation dependent, so may get a different engine on a different platform
- `minstd_rand`: version of the `linear_congruential_engine`, fast and takes up little space in memory - small state and liable to repeat
- `minstd_rand0`: similar to `minstd_rand`, usually just as fast but with slightly better results
- `ranlux24_base`: version of `subract_with_carry` engine, reasonable speed but takes up more memory than `minstd_rand`

- `mt19937`: version of the `mersenne_twister` engine, high-quality, high-speed and produces a sequence 2^19937 - 1 long before repeating, takes up more space in memory than other engines, seed value needs to be picked carefully.
- `mt19937_64`: similar to `mt19937` but operates in 64 bits
- `knuth_b`: complex adaptation that applies the `shuffle_order_engine`, high quality but slower than other engines

Common Distributions

- `uniform_int_distribution<type> name(first, last)`: discrete uniform between *first* and *last* of type *type* (`short`, `int`, `long`)
- `uniform_real_distribution<type> name(lower, upper)`: uniform distribution of floating point numbers usually `double` or `float`, where x in [*lower*, *upper*)
- `bernoulli_distribution name(prob=0.5)`: Bernoulli distribution, produces `true` / `false` with probability *prob*
- `binomial_distribution<type1, type2> name(n, prob)`: produces sum of *n* Bernoulli trial, *type1* and *type2* are the types of *n* and *prob* usually `int` and `double`
- `exponential_distribution<type> name(rate)`: Exponential distribution, produces floating point numbers of specified *type*, usually `double`
- `geometric_distribution<type1, type2> name(prob)`: Returns an integer of specified *type1* based on a *prob* of specified *type2*
- `normal_distribution<type> name(mean = 0.0, std_dev = 1.0)`: Produces a floating point number from the Normal distribution
- `poisson_distribution<type> name(rate)`: Takes a *rate* usually `double` and produces an integer

There are many more distributions including `gamma`, `weibull`, `extreme_value`, `chi_squared`, `cauchy`, `fisher_f`, `student_t`, `discrete`, `piecewise_constant`, `piecewise_linear`

Operations on Engines

- `engine.seed()`: Resets an engine to its default initial state
- `engine.seed(seed_val)`: Sets a new seed for an existing engine
- `engine()`: Gets raw output from the engine
- `engine.discard(n)`: Advances random number sequence *n* steps
- `ostream_obj << engine / istream_obj >> engine`: Performs serialisation, read and write the internal value of the engine in text form (i.e. save and reset the state).

Operations on Distributions

- `dist(engine)`: produces a pseudo-random number from the distribution
- `dist.min()`: returns the minimum number the distribution can produce

- `dist.max()`: returns the maximum number the distribution can produce
- `ostream_obj << dist / istream_obj >> dist`: Performs serialisation, read and write the internal value of the distribution in text form (i.e. save and reset the state).