8.13 — Introduction to random number generation

ALEX DECEMBER 28, 2023

The ability to generate random numbers can be useful in certain kinds of programs, particularly in games, statistical modelling programs, and cryptographic applications that need to encrypt and decrypt things. Take games for example -- without random events, monsters would always attack you the same way, you'd always find the same treasure, the dungeon layout would never change, etc... and that would not make for a very good game.

In real life, we often produce randomization by doing things like flipping a coin, rolling a dice, or shuffling a deck of cards. These events aren't actually random, but involve so many physical variables (e.g. gravity, friction, air resistance, momentum, etc...) that they become almost impossible to predict or control, and (unless you're a magician) produce results that are for all intents and purposes random.

However, computers aren't designed to take advantage of physical variables -- your computer can't toss a coin, throw a dice, or shuffle real cards. Modern computers live in a controlled electrical world where everything is binary (0 or 1) and there is no in-between. By their very nature, computers are designed to produce results that are as predictable as possible. When you tell the computer to calculate 2 + 2, you always want the answer to be 4. Not 3 or 5 on occasion.

Consequently, computers are generally incapable of generating truly random numbers (at least through software). Instead, modern programs typically simulate randomness using an algorithm.

In this lesson, we'll cover a lot of the theory behind how random numbers are generated in programs, and introduce some terminology we'll use in future lessons.

Algorithms and state

First, let's take a detour through the concepts of algorithms and states.

An **algorithm** is a finite sequence of instructions that can be followed to solve some problem or produce some useful result.

For example, let's say your boss gives you a small text file containing a bunch of unsorted names (one per line), and asks you to sort the list. Since the list is small, and you don't expect to do this often, you decide to sort it by hand. There are multiple ways to sort a list, but you might do something like this:

- Create a new empty list to hold the sorted results
- Scan the list of unsorted names to find the name that comes first alphabetically
- Cut that name out of the unsorted list and paste it at the bottom of the sorted list
- Repeat the previous two steps until there are no more names on the unsorted list

The above set of steps describes a sorting algorithm (using natural language). By nature, algorithms are reusable -- if your boss asks you to sort another list tomorrow, you can just apply the same algorithm to the new list.

Because computers can execute instructions and manipulate data much more quickly than we can, algorithms are often written using programming languages, allowing us to automate tasks. In C++, algorithms are typically implemented as reusable functions.

Here's a simple algorithm for generating a sequence of numbers where each successive number is incremented by 1:

```
#include <iostream>

int plusOne()
{
    static int s_state { 3 }; // only initialized the first time this function is called

    // Generate the next number

    ++s_state;      // first we modify the state
    return s_state; // then we use the new state to generate the next number in the sequence
}

int main()
{
    std::cout << plusOne() << '\n';
    std::cout << plusOne() << '\n';
    std::cout << plusOne() << '\n';

    return 0;
}
```

This prints:

```
4
5
6
```

This algorithm is pretty simple. The first time we call `plusOne()`, `s_state` is initialized to value `3`. Then the next number in the sequence is generated and returned.

An algorithm is considered to be **stateful** if it retains some information across calls. Conversely, a **stateless** algorithm does not store any information (and must be given all the information it needs to work with whenever it is called). Our `plusOne()` function is stateful, in that it uses the static variable `s_state` to store the last number that was generated. When applied to algorithms, the term **state** refers to the current values held in stateful variables (those retained across calls).

To generate the next number in the sequence, our algorithm uses a two step process:

- First, the current state (initialized from the start value, or preserved from the prior call) is modified to produce a new state.
- Then, the next number in the sequence is generated from the new state.

Our algorithm is considered **deterministic**, meaning that for a given input (the value provided for `start`), it will always produce the same output sequence.

Pseudo-random number generators (PRNGs)

To simulate randomness, programs typically use a pseudo-random number generator. A **pseudo-random number generator (PRNG)** is an algorithm that generates a sequence of numbers whose properties simulate a sequence of random numbers.

It's easy to write a basic PRNG algorithm. Here's a short PRNG example that generates 100 16-bit pseudo-random numbers:

```
#include <iostream>

// For illustrative purposes only, don't use this
unsigned int LCG16() // our PRNG
{
    static unsigned int s_state{ 5323 };

    // Generate the next number

    // We modify the state using large constants and intentional overflow to make it hard
    // for someone to casually determine what the next number in the sequence will be.

    s_state = 8253729 * s_state + 2396403; // first we modify the state
    return s_state % 32768; // then we use the new state to generate the next number in the sequence
}

int main()
{
    // Print 100 random numbers
    for (int count{ 1 }; count <= 100; ++count)
    {
        std::cout << LCG16() << '\t';

        // If we've printed 10 numbers, start a new row
        if (count % 10 == 0)
            std::cout << '\n';
    }

    return 0;
}
```

The result of this program is:

```
23070   27857   22756   10839   27946   11613   30448   21987   22070   1001
27388   5999    5442    28789   13576   28411   10830   29441   21780   23687
5466    2957    19232   24595   22118   14873   5932    31135   28018   32421
14648   10539   23166   22833   12612   28343   7562    18877   32592   19011
13974   20553   9052    15311   9634    27861   7528    17243   27310   8033
28020   24807   1466    26605   4992    5235    30406   18041   3980    24063
15826   15109   24984   15755   23262   17809   2468    13079   19946   26141
1968    16035   5878    7337    23484   24623   13826   26933   1480    6075
11022   19393   1492    25927   30234   17485   23520   18643   5926    21209
2028    16991   3634    30565   2552    20971   23358   12785   25092   30583
```

Each number appears to be pretty random with respect to the previous one.

Notice how similar `LCG16()` is to our `plusOne()` example above! We can pass `LCG16()` an initial value that is used to initialize the state. Then to produce the next number in the output sequence, the current state is modified (by applying some mathematical operations) to produce a new state, and the next number in the sequence is generated from that new state.

As it turns out, this particular algorithm isn't very good as a random number generator (note how each result alternates between even and odd -- that's not very random!). But most PRNGs work similarly to `LCG16()` -- they just typically use more state variables and more complex mathematical operations in order to generate better quality results.

Seeding a PRNG

The sequence of "random numbers" generated by a PRNG is not random at all. Just like our `plusOne()` function, `LCG16()` is also deterministic. Once the state has been initialized, `LCG16()` (and all other PRNGs) will generate the same output sequence.

When a PRNG is instantiated, an initial value (or set of values) called a **random seed** (or **seed** for short) can be provided to initialize the state of the PRNG. When a PRNG has been initialized with a seed, we say it has been **seeded**.

Key insight

All of the values that a PRNG will produce are deterministically calculated from the seed value(s).

Most PRNGs that produce quality results use at least 16 bytes of state, if not significantly more. However, the size of the seed value can be smaller than the size of the state of the PRNG. When this happens, we say the PRNG has been **underseeded**.

Ideally, every bit in the state is initialized from a seed of equal size, and every bit in the seed has been independently determined somehow. However, if a PRNG is underseeded, some number of bits in the state will need to be initialized from the same bits in the seed. If a PRNG is significantly underseeded (meaning the size of the seed is much smaller than the size of the state), the quality of the random results the PRNG produces can be impacted.

What makes a good PRNG? (optional reading)

In order to be a good PRNG, the PRNG needs to exhibit a number of properties:

- The PRNG should generate each number with approximately the same probability.

This is called distribution uniformity. If some numbers are generated more often than others, the result of the program that uses the PRNG will be biased! To check distribution uniformity, we can use a histogram. A histogram is a graph that tracks how many times each number has been generated. Since our histograms are text-based, we'll use a * symbol to represent each time a given number was generated.

Consider a PRNG that generates numbers between 1 and 6. If we generate 36 numbers, a PRNG with distribution uniformity should generate a histogram that looks something like this:

```
1|******
2|******
3|******
4|******
5|******
6|******
```

A PRNG that is biased in some way will generate a histogram that is uneven, like this:

```
1|***
2|******
3|******
4|******
5|******
6|*********
```

or this:

```
1|****
2|********
3|******
4|********
5|******
6|****
```

Let's say you're trying to write a random item generator for a game. When a monster is killed, your code generates a random number between 1 and 6, and if the result is a 6, the monster will drop a rare item instead of a common one. You would expect a 1 in 6 chance of this happening. But if the underlying PRNG is not uniform, and generates a lot more 6s than it should (like the second histogram above), your players will end up getting more rare items than you'd intended, possibly trivializing the difficulty of your game, or messing up your in-game economy.

Finding PRNG algorithms that produce uniform results is difficult.

- The method by which the next number in the sequence is generated shouldn't be predictable.

For example, consider the following PRNG algorithm: `return ++num`. This PRNG is perfectly uniform, but it is also completely predictable -- and not very useful as a sequence of random numbers!

Even sequences of numbers that seem random to the eye (such as the output of `LCG16()` above) may be trivially predictable by someone who is motivated. By examining just a few numbers generated from the `LCG16()` function above, it is possible to determine which constants are used (`8253729` and `2396403`) to modify the state. Once that is known, it becomes trivial to calculate all of the future numbers that will be generated from this PRNG.

Now, imagine you're running a betting website where users can bet $100. Your website then generates a random number between 0 and 32767. If the number is greater than 20000, the customer wins and you pay them double. Otherwise, they lose. Since the customer wins only 12767/32767 (39%) of the time, your website should make tons of money, right? However, if customers are able to determine which numbers will be generated next, then they can strategically place bets so they always (or usually) win. Congrats, now you get to file for bankruptcy!

- The PRNG should have a good dimensional distribution of numbers.

This means the PRNG should return numbers across the entire range of possible results at random. For example, the PRNG should generate low numbers, middle numbers, high numbers, even numbers, and odd numbers seemingly at random.

A PRNG that returned all low numbers, then all high numbers may be uniform and non-predictable, but it's still going to lead to biased results, particularly if the number of random numbers you actually use is small.

- The PRNG should have a high period for all seeds

All PRNGs are periodic, which means that at some point the sequence of numbers generated will begin to repeat itself. The length of the sequence before a PRNG begins to repeat itself is known as the **period**.

For example, here are the first 100 numbers generated from a PRNG with poor periodicity:

```
112     9       130     97      64      31      152     119     86      53
20      141     108     75      42      9       130     97      64      31
152     119     86      53      20      141     108     75      42      9
130     97      64      31      152     119     86      53      20      141
108     75      42      9       130     97      64      31      152     119
86      53      20      141     108     75      42      9       130     97
64      31      152     119     86      53      20      141     108     75
42      9       130     97      64      31      152     119     86      53
20      141     108     75      42      9       130     97      64      31
152     119     86      53      20      141     108     75      42      9
```

You will note that it generated 9 as the 2nd number, again as the 16th number, and then every 14 numbers after that. This PRNG is stuck generating the following sequence repeatedly: 9-130-97-64-31-152-119-86-53-20-141-108-75-42-(repeat).

This happens because PRNGs are deterministic. Once the state of a PRNG is identical to a prior state, the PRNG will start producing the same sequence of outputs it has produced before -- resulting in a loop.

A good PRNG should have a long period for all seed numbers. Designing an algorithm that meets this property can be extremely difficult -- many PRNGs have long periods only for some seeds and not others. If the user happens to pick a seed that results in a state with a short period, then the PRNG won't do a good job if many random numbers are needed.

- The PRNG should be efficient

Most PRNGs have a state size of less than 4096 bytes, so total memory usage typically isn't a concern. However, the larger the internal state, the more likely the PRNG is to be underseeded, and the slower the initial seeding will be (since there's more state to initialize).

Second, to generate the next number in sequence, a PRNG has to mix up its internal state by applying various mathematical operations. How much time this takes can vary significantly by PRNG and also by architecture (some PRNGs perform better on certain architectures than others). This doesn't matter if you only generate random numbers periodically, but can have a huge impact if you need lots of randomness.

There are many different kinds of PRNG algorithms

Over the years, many different kinds of PRNG algorithms have been developed (Wikipedia has a good list here (https://en.wikipedia.org/wiki/List_of_random_number_generators)). Every PRNG algorithm has strengths and weaknesses that might make it more or less suitable for a particular applications, so selecting the right algorithm for your application is important.

Many PRNGs are now considered relatively poor by modern standards -- and there's no reason to use a PRNG that doesn't perform well when it's just as easy to use one that does.

()Randomization in C++ (#random)

The randomization capabilities in C++ are accessible via the `<random>` header of the standard library. Within the random library, there are 6 PRNG families available for use (as of C++20):

Type name	Family	Period	State size*	Performance	Quality	Should I use this?
minstd_rand minstd_rand0	Linear congruential generator	2^31	4 bytes	Bad	Awful	No
mt19937 mt19937_64	Mersenne twister	2^19937	2500 bytes	Decent	Decent	Probably (see next section)
ranlux24 ranlux48	Subtract and carry	10^171	96 bytes	Awful	Good	No
knuth_b	Shuffled linear congruential generator	2^31	1028 bytes	Awful	Bad	No
default_random_engine	Any of above (implementation defined)	Varies	Varies	?	?	No[2]
rand()	Linear congruential generator	2^31	4 bytes	Bad	Awful	No[no]

There is zero reason to use `knuth_b`, `default_random_engine`, or `rand()` (which is a random number generator provided for compatibility with C).

As of C++20, the Mersenne Twister algorithm is the only PRNG that ships with C++ that has both decent performance and quality.

For advanced readers

A test called PracRand (http://pracrand.sourceforge.net/) is often used to assess the performance and quality of PRNGs (to determine whether they have different kinds of biases). You may also see references to SmallCrush, Crush or BigCrush -- these are other tests that are sometimes used for the same purpose.

If you want to see what the output of Pracrand looks like, this website (https://arvid.io/2018/06/30/on-cxx-random-number-generator-quality/) has output for all of the PRNGs that C++ supports as of C++20.

So we should use Mersenne Twister, right?

Probably. For most applications, Mersenne Twister is fine, both in terms of performance and quality.

However, it's worth noting that by modern PRNG standards, Mersenne Twister is a bit outdated (https://en.wikipedia.org/wiki/Mersenne_Twister#Disadvantages). The biggest issue with Mersenne Twister is that its results can be predicted after seeing 624 generated numbers, making it non-suitable for any application that requires non-predictability.

If you are developing an application that requires the highest quality random results (e.g. a statistical simulation), the fastest results, or one where non-predictability is important (e.g. cryptography), you'll need to use a 3rd party library.

Popular choices as of the time of writing:

- The Xoshiro family and Wyrand for non-cryptographic PRNGs.
- The Chacha family for cryptographic (non-predictable) PRNGs.

Okay, now that your eyes are probably bleeding, that's enough theory. Let's discuss how to actually generate random numbers with Mersenne Twister in C++.

Next lesson
8.14 Generating random numbers using Mersenne Twister

Back to table of contents

Previous lesson
8.12 Halts (exiting your program early)