8.13 — Introduction to random number generation

learncpp.com/cpp-tutorial/introduction-to-random-number-generation/

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 output 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{ 0 }; // only initialized the first time this function is
called

    // 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:

```
4339    838     25337   15372   6783    2642    6021    19992   14859   26462
25105   13860   28567   6762    17053   29744   15139   9078    14633   2108
7343    642     17845   29256   5179    14222   26689   12884   8647    17050
8397    18528   17747   9126    28505   13420   32479   23218   21477   30328
20075   26558   20081   3716    13303   19146   24317   31888   12163   982
1417    16540   16655   4834    16917   23208   26779   30702   5281    19124
9767    13050   32045   4288    31155   17414   31673   11468   25407   11026
4165    7896    25291   26654   15057   26340   30807   31530   31581   1264
9187    25654   20969   30972   25967   9026    15989   17160   15611   14414
16641   25364   10887   9050    22925   22816   11795   25702   2073    9516
```

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

Notice how similar `LCG16()` is to our `plusOne()` example above! The state is initially set to value `0`. 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. Given some initial state value (such as `0`), a PRNG will generate the same sequence of numbers each time. If you run the above program 3 times, you'll see it generates the same sequence of values each time.

In order to generate different output sequences, the initial state of a PRNG needs to be varied. The value (or set of values) used to set the initial state of a PRNG is called a **random seed** (or **seed** for short). When the initial state of a PRNG has been set using a seed, we say it has been **seeded**.

Key insight

Because the initial state of the PRNG is set from the seed value(s), all of the values that a PRNG will produce are deterministically calculated from the seed value(s).

The seed value is typically provided by the program using the PRNG. Here's a sample program that requests a seed value from the user and then generates 10 random numbers using that seed value (using our `LCG16()` function).

```
#include <iostream>

unsigned int g_state{ 0 };

void seedPRNG(unsigned int seed)
{
    g_state = seed;
}

// For illustrative purposes only, don't use this
unsigned int LCG16() // our PRNG
{
    // 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.

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

void print10()
{
    // Print 10 random numbers
    for (int count{ 1 }; count <= 10; ++count)
    {
        std::cout << LCG16() << '\t';
    }

    std::cout << '\n';
}

int main()
{
    unsigned int x {};
    std::cout << "Enter a seed value: ";
    std::cin >> x;

    seedPRNG(x); // seed our PRNG
    print10();   // generate 10 random values

    return 0;
}
```

Here are 3 sample runs from this:

```
Enter a seed value: 7
10458   3853    16032   17299   10726   32153   19116   7455    242     549

Enter a seed value: 7
10458   3853    16032   17299   10726   32153   19116   7455    242     549

Enter a seed value: 9876
24071   18138   27917   23712   8595    18406   23449   26796   31519   7922
```

Notice that when we provide the same seed value, we get the same output sequence. If we provide a different seed value, we get a different output sequence.

Seed quality and underseeding

If we want the program to produce different randomized numbers each time it is run, then we need some way to vary the seed each time the program is run. Asking the user to provide a seed value isn't great, since they can just enter the same value each time. The program really needs some way to generate a randomized seed value each time it is run. Unfortunately, we can't use a PRNG to generate a random seed, because we need a randomized seed to generate random numbers. Instead, we'll typically use a seed generation algorithm that is designed to produce seed values. We'll discuss (and implement) such algorithms in the next lesson.

The theoretical maximum number of unique sequences that a PRNG can generate is determined by the number of bits in the PRNG's state. For example, a PRNG with 128 bits of state can theoretically generate up to 2^128 (340,282,366,920,938,463,463,374,607,431,768,211,456) unique output sequences. That's a lot!

However, which output sequence is *actually* generated depends on the initial state of the PRNG, which is determined by the seed. Therefore, practically speaking, the number of unique output sequences a PRNG can *actually* generate is limited by the number of unique seed values the program using the PRNG can provide. For example, if a particular seed generation algorithm can only generate 4 different seed values, then the PRNG will only be able to generate at most 4 different output sequences.

If a PRNG is not provided with enough bits of quality seed data, we say that it is **underseeded**. An underseeded PRNG may begin to produce randomized results whose quality is compromised in some way -- and the more severe the underseeding, the more the quality of the results will suffer.

For example, an underseeded PRNG may exhibit any of the following issues:

- The random sequences generated by consecutive runs may have a high correlation to each other.
- On the generation of the Nth random number, some values will never be able to be generated. For example, a Mersenne Twister that is underseeded in a particular way will never generate the values 7 or 13 as its first output.
- Someone may be able to guess the seed based on the initial random value produced (or the first few random values). That would allow them to then generate all future random numbers that are going to be produced by the generator. This may allow them to cheat or game the system.

For advanced readers

An ideal seed should have the following characteristics:

- The seed should contain at least as many bits as the state of the PRNG, so that every bit in the state of the PRNG can be initialized by an independent bit in the seed.

- Each bit in the seed should independently randomized.
- The seed should contain a good mix of 0 and 1s, and high bits and low bits.
- There should be no bits in the seed that are always 0 or always 1. These “stuck bits” do not provide any value.
- The seed should have a low correlation with previously generated seeds.

In practice, we may compromise on some of these characteristics. Some PRNGs have huge states (e.g. the state of a Mersenne Twister has 19937 bits), and generating quality seeds that large can be difficult. As a result, PRNGs with large states are often designed to be resilient to being seeded with fewer bits. Stuck bits are also common. For example, if we use the system clock as part of our seed, we’ll end up with some number of stuck bits, as the bits that represent larger time units (e.g. years) are effectively stuck.

Developers who aren’t familiar with proper seeding practices will often try to initialize a PRNG using a single 32-bit or 64-bit value (unfortunately, the design of C++’s standard Random library inadvertently encourages this). This will generally result in a significantly underseeded PRNG.

Seeding a PRNG with 64 bytes of quality seed data (less if the PRNGs state is smaller) is typically good enough to facilitate the generation of 8-byte random values for non-sensitive uses (e.g. not statistical simulations or cryptography).

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|****
```

or maybe even 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). 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++

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 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 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. 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++.