8.14 — Generating random numbers using Mersenne Twister

learncpp.com/cpp-tutorial/generating-random-numbers-using-mersenne-twister/

In the previous lesson 8.13 -- Introduction to random number generation, we introduced the concept of random number generation, and discussed how PRNG algorithms are typically used to simulate randomness in programs.

In this lesson, we'll take a look at how to generate random numbers in your programs. To access any of the randomization capabilities in C++, we include the `<random>` header of the standard library.

Generating random numbers in C++ using Mersenne Twister

The Mersenne Twister PRNG, besides having a great name, is probably the most popular PRNG across all programming languages. Although it is a bit old by today's standards, it generally produces quality results and has decent performance. The random library has support for two Mersenne Twister types:

- `mt19937` is a Mersenne Twister that generates 32-bit unsigned integers
- `mt19937_64` is a Mersenne Twister that generates 64-bit unsigned integers

Using Mersenne Twister is straightforward:

```
#include <iostream>
#include <random> // for std::mt19937

int main()
{
	std::mt19937 mt{}; // Instantiate a 32-bit Mersenne Twister

	// Print a bunch of random numbers
	for (int count{ 1 }; count <= 40; ++count)
	{
		std::cout << mt() << '\t'; // generate a random number

		// If we've printed 5 numbers, start a new row
		if (count % 5 == 0)
			std::cout << '\n';
	}

	return 0;
}
```

This produces the result:

```
3499211612      581869302       3890346734      3586334585      545404204
4161255391      3922919429      949333985       2715962298      1323567403
418932835       2350294565      1196140740      809094426       2348838239
4264392720      4112460519      4279768804      4144164697      4156218106
676943009       3117454609      4168664243      4213834039      4111000746
471852626       2084672536      3427838553      3437178460      1275731771
609397212       20544909        1811450929      483031418       3933054126
2747762695      3402504553      3772830893      4120988587      2163214728
```

First, we include the <random> header, since that's where all the random number capabilities live. Next, we instantiate a 32-bit Mersenne Twister engine via the statement `std::mt19937 mt`. Then, each time we want to generate a random 32-bit unsigned integer, we call `mt()`.

Tip

Since `mt` is a variable, you may be wondering what `mt()` means.

In lesson 5.9 -- Introduction to std::string, we showed an example where we called the function `name.length()`, which invoked the `length()` function on `std::string` variable `name`.

`mt()` is a concise syntax for calling the function `mt.operator()`, which for these PRNG types has been defined to return the next random result in the sequence. The advantage of using `operator()` instead of a named function is that we don't need to remember the function's name, and the concise syntax is less typing.

Rolling a dice using Mersenne Twister

A 32-bit PRNG will generate random numbers between 0 and 4,294,967,295, but we do not always want numbers in that range. If our program was simulating a board game or a dice game, we'd probably want to simulate the roll of a 6-sided dice by generating random numbers between 1 and 6. If our program was a dungeon adventure, and the player had a sword that did between 7 and 11 damage to monsters, then we'd want to generate random numbers between 7 and 11 whenever the player hit a monster.

Unfortunately, PRNGs can't do this. They can only generate numbers that use the full range. What we need is some way to convert a number that is output from our PRNG into a value in the smaller range we want (with an even probability of each value occurring). While we could write a function to do this ourselves, doing so in a way that produces non-biased results is non-trivial.

Fortunately, the random library can help us here, in the form of random number distributions. A **random number distribution** converts the output of a PRNG into some other distribution of numbers.

As an aside…

For the stats geeks: a random number distribution is just a probability distribution designed to take PRNG values as input.

The random library has many random numbers distributions, most of which you will never use unless you're doing some kind of statistical analysis. But there's one random number distribution that's extremely useful: a **uniform distribution** is a random number distribution that produces outputs between two numbers X and Y (inclusive) with equal probability.

Here's a similar program to the one above, using a uniform distribution to simulate the roll of a 6-sided dice:

```
#include <iostream>
#include <random> // for std::mt19937 and std::uniform_int_distribution

int main()
{
	std::mt19937 mt{};

	// Create a reusable random number generator that generates uniform numbers
between 1 and 6
	std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use
std::uniform_int_distribution<> die6{ 1, 6 };

	// Print a bunch of random numbers
	for (int count{ 1 }; count <= 40; ++count)
	{
		std::cout << die6(mt) << '\t'; // generate a roll of the die here

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

	return 0;
}
```

This produces the result:

```
3	1	3	6	5	2	6	6	1	2
2	6	1	1	6	1	4	5	2	5
6	2	6	2	1	3	5	4	5	6
1	4	2	3	1	2	2	6	2	1
```

There are only two noteworthy differences in this example compared to the previous one. First, we've created a uniform distribution variable (named `die6`) to generate numbers between 1 and 6. Second, instead of calling `mt()` to generate 32-bit unsigned integer random numbers, we're now calling `die6(mt)` to generate a value between 1 and 6.

The above program isn't as random as it seems

Although the results of our dice rolling example above are pretty random, there's a major flaw with the program. Run the program 3 times and see if you can figure out what it is. Go ahead, we'll wait.

Jeopardy music

If you run the program multiple times, you will note that it prints the same numbers every time! While each number in the sequence is random with regards to the previous one, the entire sequence is not random at all! Each run of our program produces the exact same result.

Imagine that you're writing a game of hi-lo, where the user has 10 tries to guess a number that has been picked randomly, and the computer tells the user whether their guess is too high or too low. If the computer picks the same random number every time, the game won't be interesting past the first time it is played. So let's take a deeper look at why this is happening, and how we can fix it.

In the prior lesson (8.13 -- Introduction to random number generation), we covered that each number in a PRNG sequence is in a deterministic way. And that the state of the PRNG is initialized from the seed value. Thus, given any starting seed number, PRNGs will always generate the same sequence of numbers from that seed as a result.

Because we are value initializing our Mersenne Twister, it is being initialized with the same seed every time the program is run. And because the seed is the same, the random numbers being generated are also the same.

In order to make our entire sequence randomized differently each time the program is run, we need to pick a seed that's not a fixed number. The first answer that probably comes to mind is that we need a random number for our seed! That's a good thought, but if we need a random number to generate random numbers, then we're in a catch-22. It turns out, we really don't need our seed to be a random number -- we just need to pick something that changes each time the program is run. Then we can use our PRNG to generate a unique sequence of pseudo-random numbers from that seed.

There are two methods that are commonly used to do this:

- Use the system clock
- Use the system's random device

Seeding with the system clock

What's one thing that's different every time you run your program? Unless you manage to run your program twice at exactly the same moment in time, the answer is that the current time is different. Therefore, if we use the current time as our seed value, then our program will produce a different set of random numbers each time it is run. C and C++ have a long history of PRNGs being seeded using the current time (using the `std::time()` function), so you will probably see this in a lot of existing code.

Fortunately, C++ has a chrono library containing various clocks that we can use to generate a seed value. To minimize the chance of two time values being identical if the program is run quickly in succession, we want to use some time measure that changes as quickly as possible. For this, we'll ask the clock how much time has passed since the earliest time it can measure. This time is measured in "ticks", which is a very small unit of time (usually nanoseconds, but could be milliseconds).

```
#include <iostream>
#include <random> // for std::mt19937
#include <chrono> // for std::chrono

int main()
{
	// Seed our Mersenne Twister using steady_clock
	std::mt19937 mt{ static_cast<std::mt19937::result_type>(
		std::chrono::steady_clock::now().time_since_epoch().count()
		) };

	// Create a reusable random number generator that generates uniform numbers
between 1 and 6
	std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use
std::uniform_int_distribution<> die6{ 1, 6 };

	// Print a bunch of random numbers
	for (int count{ 1 }; count <= 40; ++count)
	{
		std::cout << die6(mt) << '\t'; // generate a roll of the die here

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

	return 0;
}
```

The above program has only two changes from the prior. First, we're including <chrono>, which gives us access to the clock. Second, we're using the current time from the clock as a seed value for our Mersenne Twister.

The results generated by this program should now be different each time it is run, which you can verify experimentally by running it several times.

The downside of this approach is that if the program is run several times in quick succession, the seeds generated for each run won't be that different, which can impact the quality of the random results from a statistical standpoint. For normal programs, this doesn't matter, but for programs that require high quality, independent results, this method of seeding may be insufficient.

Tip

`std::chrono::high_resolution_clock` is a popular choice instead of `std::chrono::steady_clock`. `std::chrono::high_resolution_clock` is the clock that uses the most granular unit of time, but it may use the system clock for the current time, which can be changed or rolled back by users. `std::chrono::steady_clock` may have a less granular tick time, but is the only clock with a guarantee that users cannot adjust it.

Seeding with the random device

The random library contains a type called `std::random_device` that is an implementation-defined PRNG. Normally we avoid implementation-defined capabilities because they have no guarantees about quality or portability, but this is one of the exception cases. Typically `std::random_device` will ask the OS for a random number (how it does this depends on the OS).

```
#include <iostream>
#include <random> // for std::mt19937 and std::random_device

int main()
{
	std::mt19937 mt{ std::random_device{}() };

	// Create a reusable random number generator that generates uniform numbers
between 1 and 6
	std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use
std::uniform_int_distribution<> die6{ 1, 6 };

	// Print a bunch of random numbers
	for (int count{ 1 }; count <= 40; ++count)
	{
		std::cout << die6(mt) << '\t'; // generate a roll of the die here

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

	return 0;
}
```

In the above program, we're seeding our Mersenne Twister with one random number generated from a temporary instance of `std::random_device`. If you run this program multiple times, it should also produce different results each time.

One potential problem with `std::random_device`: it isn't required to be non-deterministic, meaning it *could*, on some systems, produce the same sequence every time the program is run, which is exactly what we're trying to avoid. There was a bug in MinGW (fixed in GCC 9.2) that would do exactly this, making `std::random_device` useless.

However, the latest versions of the most popular compilers (GCC/MinGW, Clang, Visual Studio) support proper implementations of `std::random_device`.

Best practice

Use `std::random_device` to seed your PRNGs (unless it's not implemented properly for your target compiler/architecture).

Q: What does `std::random_device{}()` mean?

`std::random_device{}` creates a value-initialized temporary object of type `std::random_device`. The `()` then calls `operator()` on that temporary object, which returns a randomized value (which we use as an initializer for our Mersenne Twister)

It's the equivalent of the calling the following function, which uses a syntax you should be more familiar with:

```
unsigned int getRandomDeviceValue()
{
   std::random_device rd{}; // create a value initialized std::random_device object
   return rd(); // return the result of operator() to the caller
}
```

Using `std::random_device{}()` allows us to get the same result without creating a named function or named variable, so it's much more concise.

Q: If std::random_device is random itself, why don't we just use that instead of Mersenne Twister?

Because std::random_device is implementation defined, we can't assume much about it. It may be expensive to access or it may cause our program to pause while waiting for more random numbers to become available. The pool of numbers that it draws from may also be depleted quickly, which would impact the random results for other applications requesting random numbers via the same method. For this reason, std::random_device is better used to seed other PRNGs rather than as a PRNG itself.

Only seed a PRNG once

Many PRNGs can be reseeded after the initial seeding. This essentially re-initializes the state of the random number generator, causing it to generate results starting from the new seed state. Reseeding should generally be avoided unless you have a specific reason to do so, as it can cause the results to be less random, or not random at all.

Best practice

Only seed a given pseudo-random number generator once, and do not reseed it.

Here's an example of a common mistake that new programmers make:

```
#include <iostream>
#include <random>

int getCard()
{
    std::mt19937 mt{ std::random_device{}() }; // this gets created and seeded every
time the function is called
    std::uniform_int_distribution card{ 1, 52 };
    return card(mt);
}

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

    return 0;
}
```

In the `getCard()` function, the random number generator is being created and seeded every time the function is called. This is inefficient at best, and will likely cause poor random results.

Mersenne Twister and underseeding issues

The internal state of a Mersenne Twister contains 624 integral values. For `std::mt19937`, these values have type `std::uint_fast32_t`, which could be 32-bit or 64-bit in size. For `std::mt19937_64`, these values have type `std::uint_fast64_t`, which are typically 64-bits each.

In the examples above, where we seed from the clock or std::random_device, our seed is only a single value. This means we're essentially initializing 624 values using a single value, which is significantly underseeding the Mersenne Twister PRNG. The Random library does the best it can to fill in the remaining 623 values with "random" data… but it can't work magic. Underseeded PRNG can generate results that are suboptimal for applications that need the highest quality results. For example, seeding `std::mt19937` with a single 32-bit value will never generate the number `42` as its first output.

So how do we fix this? As of C++20, there's no easy way. But we do have some suggestions.

First, let's talk about `std::seed_seq` (which stands for "seed sequence"). In the prior lesson, we mentioned that ideally we want our seed data to be as many bits as the state of our PRNG, or our PRNG will be underseeded. But in many cases (especially when our PRNG has a large state), we won't have that many bits of randomized seed data.

`std::seed_seq` is a type that was designed to help with this. We can pass it as many randomized values as we have, and then it will generate as many additional unbiased seed values as needed to initialize a PRNG's state. If you initialize `std::seed_seq` with a single value (e.g. from `std::random_device`) and then initialize a Mersenne Twister with the `std::seed_seq` object, `std::seed_seq` will generate 623 values of additional seed data. That's not going to be any better than just directly using a single value. But the real power of `std::seed_seq` is that we can give it more than one piece of data, and the more pieces of random data we can give `std::seed_seq` to work with, the better. So the easiest idea is to simply use `std::random_device` to give `std::seed_seq` more data to work with. If we initialize `std::seed_seq` with 8 values from `std::random_device` instead of 1, then the remaining values generated by `std::seed_seq` should be much better:

```
#include <iostream>
#include <random>

int main()
{
	std::random_device rd{};
	std::seed_seq ss{ rd(), rd(), rd(), rd(), rd(), rd(), rd(), rd() }; // get 8
integers of random numbers from std::random_device for our seed
	std::mt19937 mt{ ss }; // initialize our Mersenne Twister with the
std::seed_seq

	// Create a reusable random number generator that generates uniform numbers
between 1 and 6
	std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use
std::uniform_int_distribution<> die6{ 1, 6 };

	// Print a bunch of random numbers
	for (int count{ 1 }; count <= 40; ++count)
	{
		std::cout << die6(mt) << '\t'; // generate a roll of the die here

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

	return 0;
}
```

This is pretty straightforward so there isn't much reason not to do this at a minimum. The results from seeding a `std::mt_19937` with 8 seed values instead of a single value should be much better.

Q: Why not give std::seed_seq 624 values from `std::random_device`?

You can, but this is likely to be slow, and risks depleting the pool of random numbers that `std::random_device` uses.

You can also use other "random" inputs to `std::seed_seq`. We've already shown you how to get a value from the clock, so you can throw that in easily. Other things that are sometimes used include the current thread id, the address of particular functions, the user's id, the process id, etc… Doing that is beyond the scope of this article, but this article has some context and a link to randutils.hpp that implements this.

An alternate path is to use a different PRNG with a smaller state. Many good PRNGs use 64 or 128 bits of state, which can easily be initialized using `std::seed_seq` filled with 8 calls to `std::random_device`.

Warming up a PRNG

When a PRNG is given a poor quality seed (or underseeded), the initial results of the PRNG may not be high quality. For this reason, some PRNGs benefit from being "warmed up", which is a technique where the first N results generated from the PRNG are discarded. This allows the internal state of the PRNG to be mixed up such that the subsequent results should be of higher quality. Typically a few hundred to a few thousand initial results are discarded. The longer the period of the PRNG, the more initial results should be discarded.

As an aside…

Visual Studio's implementation of `rand()` had (or still has?) a bug where the first generated result would not be sufficiently randomized. You may see older programs that use `rand()` discard a single result as a way to avoid this issue.

The `seed_seq` initialization used by `std::mt19937` performs a warm up, so we don't need to explicitly warm up `std::mt19937` objects.

Random numbers across multiple functions or files (Random.h)

This content was moved to 8.15 -- Global random numbers (Random.h).

Debugging programs that use random numbers

Programs that use random numbers can be difficult to debug because the program may exhibit different behaviors each time it is run. Sometimes it may work, and sometimes it may not. When debugging, it’s helpful to ensure your program executes the same (incorrect) way each time. That way, you can run the program as many times as needed to isolate where the error is.

For this reason, when debugging, it’s a useful technique to seed your PRNG with a specific value (e.g. `5`) that causes the erroneous behavior to occur. This will ensure your program generates the same results each time, making debugging easier. Once you’ve found the error, you can use your normal seeding method to start generating randomized results again.

Random FAQ

Q: Help! My random number generator is generating the same sequence of random numbers.

If your random number generator is generating the same sequence of random numbers every time your program is run, you probably didn’t seed it properly (or at all). Make sure you’re seeding it with a value that changes each time the program is run.

Q: Help! My random number generator keeps generating the same number over and over.

If your random number generator is generating the same number every time you ask it for a random number, then you are probably either reseeding the random number generator before generating a random number, or you’re creating a new random generator for each random number.