

9.5 — Random number generation

 ALEX  AUGUST 24, 2021

The ability to generate random numbers can be useful in certain kinds of programs, particularly in games, statistics modeling programs, and scientific simulations that need to model random events. 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.

So how do we generate random numbers? In real life, we often generate random results by doing things like flipping a coin, rolling a dice, or shuffling a deck of cards. These events involve so many physical variables (e.g. gravity, friction, air resistance, momentum, etc...) that they become almost impossible to predict or control, and 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. Computers live in a controlled electrical world where everything is binary (false or true) 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 random numbers. Instead, they must simulate randomness, which is most often done using pseudo-random number generators.

A **pseudo-random number generator (PRNG)** is a program that takes a starting number (called **a seed**), and performs mathematical operations on it to transform it into some other number that appears to be unrelated to the seed. It then takes that generated number and performs the same mathematical operation on it to transform it into a new number that appears unrelated to the number it was generated from. By continually applying the algorithm to the last generated number, it can generate a series of new numbers that will appear to be random if the algorithm is complex enough.

Best practice

You should only seed your random number generators once. Seeding them more than once will cause the results to be less random or not random at all.

It's actually fairly easy to write a PRNG. Here's a short program that generates 100 pseudo-random numbers:

```

1  #include <iostream>
2  unsigned int PRNG()
3  {
4      // our initial starting seed is 5323
5      static unsigned int seed{ 5323 };
6
7      // Take the current seed and generate a new value from it
8      // Due to our use of large constants and overflow, it would
9      be
10     // hard for someone to casually predict what the next number
11     is
12     // going to be from the previous one.
13     seed = 8253729 * seed + 2396403;
14
15     // Take the seed and return a value between 0 and 32767
16     return seed % 32768;
17 }
18
19 int main()
20 {
21     // Print 100 random numbers
22     for (int count{ 1 }; count <= 100; ++count)
23     {
24         std::cout << PRNG() << '\t';
25
26         // If we've printed 5 numbers, start a new row
27         if (count % 5 == 0)
28             std::cout << '\n';
29     }
30
31     return 0;
32 }

```

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. As it turns out, our algorithm actually isn't very good, for reasons we will discuss later. But it does effectively illustrate the principle of PRNG number generation.

Generating random numbers in C++

C (and by extension C++) comes with a built-in pseudo-random number generator. It is implemented as two separate functions that live in the `cstdlib` header:

`std::srand()` sets the initial seed value to a value that is passed in by the caller. `std::srand()` should only be called once at the beginning of your program. This is usually done at the top of `main()`.

`std::rand()` generates the next random number in the sequence. That number will be a pseudo-random integer between 0 and `RAND_MAX`, a constant in `cstdlib` that is typically set to 32767.

Here's a sample program using these functions:

```
1 #include <iostream>
  #include <cstdlib> // for std::rand() and std::srand()
2
  int main()
  {
    std::srand(5323); // set initial seed value to 5323
3
    // Due to a flaw in some compilers, we need to call std::rand() once here to get "better" random
4    numbers.
5    std::rand();
6
    // Print 100 random numbers
    for (int count{ 1 }; count <= 100; ++count)
    {
7        std::cout << std::rand() << '\t';
8
        // If we've printed 5 numbers, start a new row
        if (count % 5 == 0)
            std::cout << '\n';
    }

    return 0;
  }
```

Here's the output of this program:

```
17421 8558 19487 1344 26934
7796 28102 15201 17869 6911
4981 417 12650 28759 20778
31890 23714 29127 15819 29971
1069 25403 24427 9087 24392
15886 11466 15140 19801 14365
18458 18935 1746 16672 22281
16517 21847 27194 7163 13869
5923 27598 13463 15757 4520
15765 8582 23866 22389 29933
31607 180 17757 23924 31079
30105 23254 32726 11295 18712
29087 2787 4862 6569 6310
21221 28152 12539 5672 23344
28895 31278 21786 7674 15329
10307 16840 1645 15699 8401
22972 20731 24749 32505 29409
17906 11989 17051 32232 592
17312 32714 18411 17112 15510
8830 32592 25957 1269 6793
```

PRNG sequences and seeding

If you run the `std::rand()` sample program above multiple times, you will note that it prints the same result every time! This means that while each number in the sequence is seemingly random with regards to the previous ones, the entire sequence is not random at all! And that means our program ends up totally predictable (the same inputs lead to the same outputs every time). There are cases where this can be useful or even desired (e.g. you want a scientific simulation to be repeatable, or you're trying to debug why your random dungeon generator crashes).

But often, this is not what is desired. If you're writing a game of hi-lo (where the user has 10 tries to guess a number, and the computer tells them whether their guess is too high or too low), you don't want the program picking the same numbers each time. So let's take a deeper look at why this is happening, and how we can fix it.

Remember that each number in a PRNG sequence is generated from the previous number, in a deterministic way. Thus, given any starting seed number, PRNGs will always generate the same sequence of numbers from that seed as a result! We are getting the same sequence because our starting seed number is always 5323.

In order to make our entire sequence randomized, we need some way 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! 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.

The commonly accepted method for doing this is to enlist the system clock. Each time the user runs the program, the time will be different. If we use this time value as our seed, then our program will generate a different sequence of numbers each time it is run!

C comes with a function called `std::time()` that returns the number of seconds since midnight on Jan 1, 1970. To use it, we merely need to include the `ctime` header, and then initialize `std::srand()` with a call to `std::time(nullptr)`. We haven't covered `nullptr` yet, but it's essentially the equivalent of 0 in this context.

Here's the same program as above, using a call to `time()` as the seed:

```
1  #include <iostream>
2  #include <cstdlib> // for std::rand() and std::srand()
   #include <ctime> // for std::time()
3
4  int main()
5  {
6      std::srand(static_cast<unsigned int>(std::time(nullptr))); // set initial seed value to system clock
7
8      // Due to a flaw in some compilers, we need to call std::rand() once here to get "better" random
9      // numbers.
10     std::rand();
11
12     for (int count{ 1 }; count <= 100; ++count)
13     {
14         std::cout << std::rand() << '\t';
15
16         // If we've printed 5 numbers, start a new row
17         if (count % 5 == 0)
18             std::cout << '\n';
19     }
20
21     return 0;
22 }
```

Now our program will generate a different sequence of random numbers every time! Run it a couple of times and see for yourself.

Generating random numbers between two arbitrary values

Generally, we do not want random numbers between 0 and `RAND_MAX` -- we want numbers between two other values, which we'll call min and max. For example, if we're trying to simulate the user rolling a die, we want random numbers between 1 and 6 (pedantic grammar note: yes, die is the singular of dice).

Here's a short function that converts the result of `rand()` into the range we want:

```

1 // Generate a random number between min and max (inclusive)
  // Assumes std::rand() has already been called
  // Assumes max - min <= RAND_MAX
2 int getRandomNumber(int min, int max)
  {
    static constexpr double fraction { 1.0 / (RAND_MAX + 1.0) }; // static used for efficiency, so we
3 only calculate this value once
    // evenly distribute the random number across our range
4    return min + static_cast<int>((max - min + 1) * (std::rand() * fraction));
  }

```

To simulate the roll of a die, we'd call `getRandomNumber(1, 6)`. To pick a randomized digit, we'd call `getRandomNumber(0, 9)`.

Optional reading: How does the previous function work?

The `getRandomNumber()` function may seem a little complicated, but it's not too bad.

Let's revisit our goal. The function `rand()` returns a number between 0 and `RAND_MAX` (inclusive). We want to somehow transform the result of `rand()` into a number between `min` and `max` (inclusive). This means that when we do our transformation, 0 should become `min`, and `RAND_MAX` should become `max`, with a uniform distribution of numbers in between.

We do that in five parts:

1. We multiply our result from `std::rand()` by `fraction`. This converts the result of `rand()` to a floating point number between 0 (inclusive), and 1 (exclusive).
If `rand()` returns a 0, then $0 * \text{fraction}$ is still 0. If `rand()` returns `RAND_MAX`, then $\text{RAND_MAX} * \text{fraction}$ is $\text{RAND_MAX} / (\text{RAND_MAX} + 1)$, which is slightly less than 1. Any other number returned by `rand()` will be evenly distributed between these two points.
2. Next, we need to know how many numbers we can possibly return. In other words, how many numbers are between `min` (inclusive) and `max` (inclusive)?
This is simply $(\text{max} - \text{min} + 1)$. For example, if `max` = 8 and `min` = 5, $(\text{max} - \text{min} + 1) = (8 - 5 + 1) = 4$. There are 4 numbers between 5 and 8 (that is, 5, 6, 7, and 8).
3. We multiply the prior two results together. If we had a floating point number between 0 (inclusive) and 1 (exclusive), and then we multiply by $(\text{max} - \text{min} + 1)$, we now have a floating point number between 0 (inclusive) and $(\text{max} - \text{min} + 1)$ (exclusive).
4. We cast the previous result to an integer. This removes any fractional component, leaving us with an integer result between 0 (inclusive) and $(\text{max} - \text{min})$ (inclusive).
5. Finally, we add `min`, which shifts our result to an integer between `min` (inclusive) and `max` (inclusive).

Optional reading: Why don't we use the modulus operator (%) in the previous function?

One of the most common questions readers have submitted is why we use division in the above function instead of modulus (%). The short answer is that the modulus method tends to be biased in favor of low numbers.

Let's consider what would happen if the above function looked like this instead:

```
1 | return min + (std::rand() %  
   | (max-min+1));
```

Seems similar, right? Let's explore where this goes wrong. To simplify the example, let's say that `rand()` always returns a random number between 0 and 9 (inclusive). For our sample case, we'll pick `min = 0`, and `max = 6`. Thus, `max - min + 1` is 7.

Now let's calculate all possible outcomes:

```
0 + (0 % 7) = 0  
0 + (1 % 7) = 1  
0 + (2 % 7) = 2  
0 + (3 % 7) = 3  
0 + (4 % 7) = 4  
0 + (5 % 7) = 5  
0 + (6 % 7) = 6  
  
0 + (7 % 7) = 0  
0 + (8 % 7) = 1  
0 + (9 % 7) = 2
```

Look at the distribution of results. The results 0 through 2 come up twice, whereas 3 through 6 come up only once. This method has a clear bias towards low results. By extension, most cases involving this algorithm will behave similarly.

Now let's take a look at the result of the `getRandomNumber()` function above, using the same parameters as above (`rand()` returns a number between 0 and 9 (inclusive), `min = 0` and `max = 6`). In this case, $\text{fraction} = 1 / (9 + 1) = 0.1$. `max - min + 1` is still 7.

Calculating all possible outcomes:

```
0 + static_cast<int>(7 * (0 * 0.1)) = 0 + static_cast<int>(0) = 0
0 + static_cast<int>(7 * (1 * 0.1)) = 0 + static_cast<int>(0.7) = 0
0 + static_cast<int>(7 * (2 * 0.1)) = 0 + static_cast<int>(1.4) = 1
0 + static_cast<int>(7 * (3 * 0.1)) = 0 + static_cast<int>(2.1) = 2
0 + static_cast<int>(7 * (4 * 0.1)) = 0 + static_cast<int>(2.8) = 2
0 + static_cast<int>(7 * (5 * 0.1)) = 0 + static_cast<int>(3.5) = 3
0 + static_cast<int>(7 * (6 * 0.1)) = 0 + static_cast<int>(4.2) = 4
0 + static_cast<int>(7 * (7 * 0.1)) = 0 + static_cast<int>(4.9) = 4
0 + static_cast<int>(7 * (8 * 0.1)) = 0 + static_cast<int>(5.6) = 5
0 + static_cast<int>(7 * (9 * 0.1)) = 0 + static_cast<int>(6.3) = 6
```

The bias here is still slightly towards lower numbers (0, 2, and 4 appear twice, whereas 1, 3, 5, and 6 appear once), but it's much more uniformly distributed.

Even though `getRandomNumber()` is a little more complicated to understand than the modulus alternative, we advocate for the division method because it produces a less biased result.

What is a good PRNG?

As I mentioned above, the PRNG we wrote isn't a very good one. This section will discuss the reasons why. It is optional reading because it's not strictly related to C or C++, but if you like programming you will probably find it interesting anyway.

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

First, 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!

For example, let's say you're trying to write a random item generator for a game. You'll pick a random number between 1 and 10, and if the result is a 10, the monster will drop a powerful item instead of a common one. You would expect a 1 in 10 chance of this happening. But if the underlying PRNG is not uniform, and generates a lot more 10s than it should, your players will end up getting more rare items than you'd intended, possibly trivializing the difficulty of your game.

Generating PRNGs that produce uniform results is difficult, and it's one of the main reasons the PRNG we wrote at the top of this lesson isn't a very good PRNG.

Second, the method by which the next number in the sequence is generated shouldn't be obvious or predictable. For example, consider the following PRNG algorithm: `num = num + 1`. This PRNG is perfectly uniform, but it's not very useful as a sequence of random numbers!

Third, the PRNG should have a good dimensional distribution of numbers. This means it should return low numbers, middle numbers, and high 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.

Fourth, 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 second number, and 9 again as the 16th number. The PRNG gets stuck generating the sequence in-between these two 9's repeatedly: 9-130-97-64-31-152-119-86-53-20-141-108-75-42-(repeat).

This happens because PRNGs are deterministic -- given some set of input values, they will produce the same output value every time. This means that once the PRNG encounters a set of inputs it has used before, it 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 -- most PRNGs will have long periods for some seeds and short periods for others. If the user happens to pick a seed that has a short period, then the PRNG won't be doing a good job.

Despite the difficulty in designing algorithms that meet all of these criteria, a lot of research has been done in this area because of its importance to scientific computing.

std::rand() is a mediocre PRNG

The algorithm used to implement std::rand() can vary from compiler to compiler, leading to results that may not be consistent across compilers. Most implementations of rand() use a method called a [Linear Congruential Generator \(LCG\)](#). If you have a look at the first example in this lesson, you'll note that it's actually a LCG, though one with intentionally picked poor constants. LCGs tend to have shortcomings that make them not good choices for most kinds of problems.

One of the main shortcomings of rand() is that RAND_MAX is usually set to 32767 (essentially 15-bits). This means if you want to generate numbers over a larger range (e.g. 32-bit integers), rand() is not suitable. Also, rand() isn't good if you want to generate random floating point numbers (e.g. between 0.0 and 1.0), which is often useful when doing statistical modelling. Finally, rand() tends to have a relatively short period compared to other algorithms.

That said, rand() is perfectly suitable for learning how to program, and for programs in which a high-quality PRNG is not a necessity.

For applications where a high-quality PRNG is useful, I would recommend [Mersenne Twister](#) (or one of its variants), which produces great results and is relatively easy to use. Mersenne Twister was adopted into C++11, and we'll show how to use it later in this lesson.

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 set the random seed (via `std::srand`) to a specific value (e.g. 0) 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 seed using the system clock again to start generating randomized results again.

Better random numbers using Mersenne Twister

C++11 added a ton of random number generation functionality to the C++ standard library, including the Mersenne Twister algorithm, as well as generators for different kinds of random distributions (uniform, normal, Poisson, etc...). This is accessed via the <random> header.

Here's a short example showing how to generate random numbers in C++11 using Mersenne Twister (h/t to user Fernando):

```
1 #include <iostream>
2 #include <random> // for std::mt19937
3 #include <ctime> // for std::time
4
5 int main()
6 {
7     // Initialize our mersenne twister with a random seed based on the clock
8     std::mt19937 mersenne{ static_cast<std::mt19937::result_type>(std::time(nullptr)) };
9
10    // Create a reusable random number generator that generates uniform numbers between 1
11    // and 6
12    std::uniform_int_distribution die{ 1, 6 };
13    // If your compiler doesn't support C++17, use this instead
14    // std::uniform_int_distribution<> die{ 1, 6 };
15
16    // Print a bunch of random numbers
17    for (int count{ 1 }; count <= 48; ++count)
18    {
19        std::cout << die(mersenne) << '\t'; // generate a roll of the die here
20
21        // If we've printed 6 numbers, start a new row
22        if (count % 6 == 0)
23            std::cout << '\n';
24    }
25
26    return 0;
27 }
```

Author's note

Before C++17, you need to add empty brackets to create `die` after the type

```
std::uniform_int_distribution<> die{ 1, 6 }
```

You'll note that Mersenne Twister generates random 32-bit unsigned integers (not 15-bit integers like `std::rand()`), giving a lot more range. There's also a version (`std::mt19937_64`) for generating 64-bit unsigned integers.

Random numbers across multiple functions

The above example creates a random generator for use within a single function. What happens if we want to use a random number generator in multiple functions?

Although you can create a static local `std::mt19937` variable in each function that needs it (static so that it only gets seeded once), it's a little overkill to have every function that needs a random number generator seed and maintain its own local generator. A better option in most cases is to create a global random number generator (inside a namespace!). Remember how we told you to avoid non-const global variables? This is an exception (also note: `std::rand()` and `std::srand()` access a global object, so there's precedent for this).

```
1 #include <iostream>
2 #include <random> // for std::mt19937
3 #include <ctime> // for std::time
4
5 namespace MyRandom
6 {
7     // Initialize our mersenne twister with a random seed based on the clock (once at system startup)
8     std::mt19937 mersenne{ static_cast<std::mt19937::result_type>(std::time(nullptr)) };
9
10    int getRandomNumber(int min, int max)
11    {
12        std::uniform_int_distribution die{ min, max }; // we can create a distribution in any function that
13        // needs it
14        return die(mersenne); // and then generate a random number from our global generator
15    }
16
17    int main()
18    {
19        std::cout << getRandomNumber(1, 6) << '\n';
20        std::cout << getRandomNumber(1, 10) << '\n';
21        std::cout << getRandomNumber(1, 20) << '\n';
22
23        return 0;
24    }
25 }
```

Using a random library

A perhaps better solution is to use a 3rd party library that handles all of this stuff for you, such as the header-only [Effolkronium's random library](#). You simply add the header to your project, #include it, and then you can start generating random numbers via `Random::get(min, max)`.

Here's the above program using Effolkronium's library:

```
1 #include <iostream>
2 #include "random.hpp"
3
4 // get base random alias which is auto seeded and has static API and internal
  state
  using Random = effolkronium::random_static;
5
6 int main()
7 {
8     std::cout << Random::get(1, 6) << '\n';
9     std::cout << Random::get(1, 10) << '\n';
10    std::cout << Random::get(1, 20) << '\n';
11
12    return 0;
13 }
```

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. Make sure you're seeding it with a value that changes each time the program is run (like `std::time(nullptr)`).

Help! My random number generator always generates the same first number!

The implementation of `rand()` in Visual Studio and a few other compilers has a flaw -- the first random number generated doesn't change much for similar seed values. This means that when using `std::time(nullptr)` to seed your random number generator, the first result from `rand()` won't change much in successive runs. However, the results of successive calls to `rand()` aren't impacted, and will be sufficiently randomized.

The solution here, and a good rule of thumb in general, is to discard the first random number generated from the random number generator.

Help! My random number generator isn't generating random numbers at all!

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.

Here are two functions that exhibit the issue:

```
1 // This is the same function we have shown before
2 int getRandomNumber(int min, int max)
3 {
4     static constexpr double fraction { 1.0 / (RAND_MAX + 1.0) };
5     return min + static_cast<int>((max - min + 1) * (std::rand() * fraction));
6 }
7
8 int rollDie()
9 {
10    std::srand(static_cast<unsigned int>(std::time(nullptr)));
11    return getRandomNumber(1, 6);
12 }
13
14 int getOtherRandomNumber()
15 {
16    std::mt19937 mersenne{ static_cast<std::mt19937::result_type>(std::time(nullptr)) };
17    std::uniform_int_distribution rand{ 1, 52 };
18    return rand(mersenne);
19 }
```

In both cases, the random number generator is being seeded each time before a random number is generated. This will cause a similar number to be generated each time.

In the top case, `std::srand()` is reseeding the built-in random number generator before `rand()` is called (by `getRandomNumber()`).

In the bottom case, we're creating a new Mersenne Twister, seeding it, generating a single random number, and then destroying it.

For random results, you should only seed a random number generator once (generally at program initialization for `std::srand()`, or the point of creation for other random number generators), and then use that same random number generator for each successive random number generated.

Warning

The `getOtherRandomNumber()` example is one of the most common mistakes made later in quizzes. You won't notice that `getOtherRandomNumber()` is broken until you start calling it more than once per second (Because the seed only changes once per second). Remember to make random number generators `static` or declare them outside of the function.



Next lesson

9.x Chapter 9 summary and quiz



[Back to table of contents](#)



Previous lesson

9.4 Structs


Leave a comment... Put C++ code between triple-backticks (markdown style): ``Your C++ code``

Name*

Email*

?

Avatars from <https://gravatar.com/> are connected to your provided email address.

Notify me about replies: 

POST COMMENT

DP N N FOUT

Newest ▼

