5.9 — Random number generation

BY ALEX ON DECEMBER 7TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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.

Rule: You should only seed your random number generators once. Seeding them more than once will cause the results to be less random or non-random.

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

```
#include <iostream>

unsigned int PRNG()
{
    // our initial starting seed is 5323
    static unsigned int seed{ 5323 };

    // Take the current seed and generate a new value from it
    // Due to our use of large constants and overflow, it would be
    // hard for someone to casually predict what the next number is
    // going to be from the previous one.
    seed = 8253729 * seed + 2396403;

    // Take the seed and return a value between 0 and 32767
    return seed % 32768;
}

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

```

        // If we've printed 5 numbers, start a new row
        if (count % 5 == 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. 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. 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:

```
#include <iostream>
#include <cstdlib> // for std::rand() and std::srand()

int main()
{
    std::srand(5323); // set initial seed value to 5323

    // Print 100 random numbers
    for (int count{ 1 }; count <= 100; ++count)
```

```
    {
        std::cout << std::rand() << '\t';

        // 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 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 srand() with a call to std::time(nullptr) (or std::time(0) if your compiler is pre-C++11). 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:

```
#include <iostream>
#include <cstdlib> // for std::rand() and std::srand()
#include <ctime> // for std::time()

int main()
{
    std::srand(static_cast<unsigned int>(std::time(nullptr))); // set initial seed value to sy

    for (int count{ 1 }; count <= 100; ++count)
    {
        std::cout << std::rand() << '\t';

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

    return 0;
}
```

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:

```
// Generate a random number between min and max (inclusive)
// Assumes std::srand() has already been called
// Assumes max - min <= RAND_MAX
int getRandomNumber(int min, int max)
{
    static constexpr double fraction { 1.0 / (RAND_MAX + 1.0) };  // static used for efficiency
    // evenly distribute the random number across our range
    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 * fraction is still 0. If rand() returns RAND_MAX, then RAND_MAX * fraction is RAND_MAX / (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 (max - min + 1). For example, if max = 8 and min = 5, (max - 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 (min - max + 1), we now have a floating point number between 0 (inclusive) and (max - 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 (max - 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:

```
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 lets 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, fraction = 1 / (9 + 1) = 0.1.

max - min + 1 is still 7.

Calculating all possible outcomes:

```
0 + static_cast(7 * (0 * 0.1))) = 0 + static_cast(0) = 0
0 + static_cast(7 * (1 * 0.1))) = 0 + static_cast(0.7) = 0
0 + static_cast(7 * (2 * 0.1))) = 0 + static_cast(1.4) = 1
0 + static_cast(7 * (3 * 0.1))) = 0 + static_cast(2.1) = 2
0 + static_cast(7 * (4 * 0.1))) = 0 + static_cast(2.8) = 2
0 + static_cast(7 * (5 * 0.1))) = 0 + static_cast(3.5) = 3
0 + static_cast(7 * (6 * 0.1))) = 0 + static_cast(4.2) = 4
0 + static_cast(7 * (7 * 0.1))) = 0 + static_cast(4.9) = 4
0 + static_cast(7 * (8 * 0.1))) = 0 + static_cast(5.6) = 5
0 + static_cast(7 * (9 * 0.1))) = 0 + static_cast(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 eventually begin to repeat itself. As mentioned before, PRNGs are deterministic, and given an input number, a PRNG will produce the same output number every time. Consider what happens when a PRNG generates a number it has previously generated. From that point forward, it will begin to duplicate the sequence between the first occurrence of that number and the next occurrence of that number over and over. The length of this sequence 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).

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.

Random numbers in C++11

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):

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

int main()
{
    // Initialize our mersenne twister with a random seed based on the clock
    std::mt19937 mersenne{ static_cast<std::mt19937::result_type>(std::time(nullptr)) };

    // Create a reusable random number generator that generates uniform numbers between 1 and
    std::uniform_int_distribution die{ 1, 6 };
    // If your compiler doesn't support C++17, use this instead
    // std::uniform_int_distribution<> die{ 1, 6 };

    // Print a bunch of random numbers
    for (int count{ 1 }; count <= 48; ++count)
    {
        std::cout << die(mersenne) << '\t'; // generate a roll of the die here

        // If we've printed 6 numbers, start a new row
        if (count % 6 == 0)
            std::cout << '\n';
    }

    return 0;
}
```

Author's note

Before C++17, you need to use 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.

C++11 style random numbers across multiple functions

The above example create 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).

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

namespace MyRandom
{
    // Initialize our mersenne twister with a random seed based on the clock (once at system s
    std::mt19937 mersenne{ static_cast<std::mt19937::result_type>(std::time(nullptr)) };
}

int getRandomNumber(int min, int max)
{
    std::uniform_int_distribution die{ min, max }; // we can create a distribution in any func
    return die(MyRandom::mersenne); // and then generate a random number from our global gener
}

int main()
{
    std::cout << getRandomNumber(1, 6) << '\n';
    std::cout << getRandomNumber(1, 10) << '\n';
    std::cout << getRandomNumber(1, 20) << '\n';

    return 0;
}
```

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:

```
#include <iostream>
#include "random.hpp"

// get base random alias which is auto seeded and has static API and internal state
using Random = effolkronium::random_static;

int main()
{
    std::cout << Random::get(1, 6) << '\n';
    std::cout << Random::get(1, 10) << '\n';
    std::cout << Random::get(1, 20) << '\n';

    return 0;
}
```

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's are two functions that exhibit the issue:

```
int rollDie()
{
    std::srand(std::time(nullptr));
    return getRandomNumber(1, 6); // using definition of getRandomNumber above
}

int getRandomNumber()
{
    std::mt19937 mersenne{ static_cast<std::mt19937::result_type>(std::time(nullptr)) };
    std::uniform_int_distribution rand{ 1, 52 };
    return rand(mersenne);
}
```

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 best 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.

5.10 -- std::cin, extraction, and dealing with invalid text input

Index

5.8 -- Break and continue

C++ TUTORIAL, GAME PROGRAMMING C++, PRNG, PROGRAMMING, RAND, RANDOM, RNG, SRAND, TUTORIAL | PRINT THIS POST

443 comments to 5.9 — Random number generation

« Older Comments 1 … 4 5 6

Charan

December 26, 2019 at 9:39 pm · Reply

Hey,How do I get effolkronium's library?My compiler through a 'fatal error' saying "random.hpp:no such file or directory".

kavin

January 16, 2020 at 10:39 am · Reply

Same error for me too. I think u have to include it via github or something like that. I guess it's beyond the scope of this lesson.

chai

December 23, 2019 at 11:20 am · Reply

```
int main()
{
    // Initialize our mersenne twister with a random seed based on the clock
    std::mt19937 mersenne{ static_cast<std::mt19937::result_type>(std::time(nullptr)) };

    // Create a reusable random number generator that generates uniform numbers between 1 a
    std::uniform_int_distribution die{ 1, 6 };// Argument for class template is missing

    // Print a bunch of random numbers
    for (int count{ 1 }; count <= 48; ++count)
    {
        std::cout << die(mersenne) << '\t'; // Identifier die is undefined

        // If we've printed 6 numbers, start a new row
        if (count % 6 == 0)
```

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

    return 0;
}
```

There are two red squiggly lines that showed up in my Visual Studio after cutting and pasting the example code. One said that:- Argument for class template"std::uniform_int_distribution" is missing.
The other is "die" is undefined.
Any idea how to correct this?

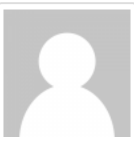
nascardriver
December 24, 2019 at 2:52 am · Reply

The error is correct if you didn't enable C++17 or higher. Enable C++17 or higher in your project settings. If you can't, change line 7 to

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

chai
December 24, 2019 at 8:06 am · Reply

awesome! Thanks.



Ged
December 13, 2019 at 9:23 am · Reply

Just finished everything till OOP. So came back to RNG to ask a few questions.

1. When we use the library <cstdlib>,it declares our RAND_MAX as a MACRO to be typically a value of 32767, so that is why we don't need to declare it ourselves?

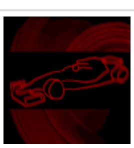
2. Why do we need to use static_cast<int> if we are returning an int, won't it chop off everything when it returns the value?

```
int getRandomNumber(int min, int max)
{
    static constexpr double fraction{ 1.0 / (RAND_MAX + 1.0) };
    return min + static_cast<int>((max - min + 1) * (std::rand() * fraction));
}
```

3. Understanding the last RNG (Mersenne Twister) is a bit hard and as you mentioned in order to fully understand it you need to know classes and templates. So not understanding it fully is ok for now?

4. Is it even worth trying to write your own RNG or it's easier to download 3rd party libraries that requires you to write a single line for a good RNG? If a 3rd party client will always be a better option then what if you are writing a complex program, will you need a lot of 3rd party libraries in order for your code to be much shorter?

5. Is there a chapter in the future that will show us how to import the libraries into your program?



nascardriver
December 13, 2019 at 9:44 am · Reply

1. Yes.

2. Yes, the code does the same without the cast. There might be the unlikely event that double is implemented in a way such that it can't hold all values an int can hold. In that case, the two versions might behave differently.

3. Yes.

4. Use the standard functions unless you need something specialized.

5. Appendix A. The process differs between compilers and IDEs. If yours isn't in the appendix, google will find instructions.

Elis
November 13, 2019 at 4:34 am · Reply

As part of a "Hi-lo game" I used the following randomization process. Just using the randomize function would always yield the same result in lower ranges. E.g. loNumber == 3 and hiNumber == 9 would always yield 7 e.t.c.
With this I managed to get pretty good results.
Since this runs the randomize function many times in a row is it a necessity to place the seed elsewhere than in the function itself (as below)?

```
int randomize(int hiNumber, int loNumber)
{
    static const double fraction{ 1.0 / (RAND_MAX + 1.0) };
    int random = (loNumber + static_cast<int>((hiNumber - loNumber + 1) * (std::rand() * fr
    return random;
}

int generateNumber(int hiNumber, int loNumber)
{
    std::srand(static_cast<int>(std::time(nullptr)));                //seed
    int random{ randomize(hiNumber, loNumber) };                    //random number for tim
    int returnValue{};
    for (int count{ 1 }; count <= ((random % 50) + 5); ++count)            /*get a random va
                                          random nr of times(between 5 and 50)*/
        returnValue = randomize(hiNumber, loNumber);

    return returnValue;
}
```

As always thanks for the great tutorials!

nascardriver
November 13, 2019 at 4:38 am · Reply

Yes. Never seed more than once. Seeding again with the same seed will reset the sequence, so you'll get the same "random" number again.

Ged
November 6, 2019 at 11:08 am · Reply

I don't understand these lines fully. Would appreciate if you could explain a bit more.

```
    // Initialize our mersenne twister with a random seed based on the clock
    std::mt19937 mersenne(static_cast<std::mt19937::result_type>(std::time(nullptr))); // Wh

    // Create a reusable random number generator that generates uniform numbers between 1 an
    std::uniform_int_distribution<> die(1, 6); // how does uniform_int_distribution even wor
```

nascardriver
November 7, 2019 at 2:14 am · Reply

You won't be able to understand them fully until you get to classes and templates.
`std::my19937` wants a `std::mt19937::result_type`, but `std::time` returns an unspecified type, so we need to cast it.

> how does uniform_int_distribution even work
There's no general answer, as the standard only says _what_ `std::uniform_int_distribution` does, but not _how_. If you want to understand how it works, you'll have to read up on discrete probability, finish learning C++, and then try to read a standard implementation.

> why do we have <>
They're not necessary since C++17, you can write `std::uniform_int_distribution die{ 1, 6 };`. Inside the <>, you can specify the integer type you want to have as a result, eg.
`std::uniform_int_distribution die{ 1, 6 };`

> how does die(1,6)
You'll need to know about classes and constructors. Basically, `die` stores the numbers you give it (min and max) and uses them later when it generates a number.

Tushar
November 3, 2019 at 9:02 am · Reply

```
#include<iostream>
unsigned int PRNG(){

    static unsigned int seed=9843;
    seed= seed*8343454 + 3555643;

    return seed%34567;
}
int main(){
    for(int i=1;i<100;i++){
        std::cout<<PRNG()<<"\t";
        if(i%5==0){
            std::cout<<"\n";
        }
    }

}
```

Output:

```
990	3213	1565	28503	21640
5362	29035	4686	32618	27599
7500	3776	13427	15361	8288
30298	17693	11683	32746	9669
10489	9238	19735	17318	20090
18670	12737	24603	871	13768
22541	22541	22541	22541	22541
22541	22541	22541	22541	22541
22541	22541	22541	22541	22541
22541	22541	22541	22541	22541
22541	22541	22541	22541	22541
```

```
22541 22541 22541 22541 22541
22541 22541 22541 22541 22541
22541 22541 22541 22541 22541
22541 22541 22541 22541 22541
22541 22541 22541 22541 22541
22541 22541 22541 22541 22541
22541 22541 22541 22541 22541
22541 22541 22541 22541 22541
22541 22541 22541 22541
```

Why are the numbers being repeated?

Luiz
November 3, 2019 at 12:17 pm · Reply

It's going to take someone more experienced than me to explain but I managed to fix it by doing the following:

snip

```
unsigned int PRNG() {

    static unsigned int seed = 9843;
    seed = seed * 8343454 + 3555643;
    seed %= 34567; // Seed wasn't being assigned to it's module
    return seed;
}
```

I assigned seed to it's module by 34567 everytime

It's probably got to do with undefined behaviour via overflow

Tushar
November 3, 2019 at 7:49 pm · Reply

Okay so this has reduced repetition of a single number though its producing repeated pattern

Output:

```
3381 2929 3321 745 1801
2633 1993 3401 2505 1737
1225 2889 713 1097 73
3401 2505 1737 1225 2889
713 1097 73 3401 2505
1737 1225 2889 713 1097
73 3401 2505 1737 1225
2889 713 1097 73 3401
2505 1737 1225 2889 713
1097 73 3401 2505 1737
1225 2889 713 1097 73
3401 2505 1737 1225 2889
713 1097 73 3401 2505
1737 1225 2889 713 1097
73 3401 2505 1737 1225
2889 713 1097 73 3401
2505 1737 1225 2889 713
1097 73 3401 2505 1737
```

1225 2889 713 1097 73
3401 2505 1737 1225 2889
You can see the pattern starting with 713 1097 73... 2889.

Luiz
November 4, 2019 at 3:12 am · Reply

There's no error within the code, it is all caused by the math and overflowing of the unsigned int. The operations your number go through in your pseudo random generator have created a pattern. If you want to try to create one that's less predictable you could try adding more operations of different sorts or such. I reckon this is also true for the reason as to why 22541 was being repeated on the first version of your code.

Luiz
November 4, 2019 at 3:16 am · Reply

Also as it is said in this lesson: The PRNG is perfectly uniform, so you're better off using a different method to generate random numbers

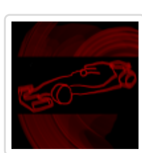
Tushar
November 5, 2019 at 1:21 am · Reply

```
#include<iostream>
#include<cstdlib>
#include<time.h>

unsigned int PRNG(int start,int end,int exceed){
    unsigned int seed=6784;
    for(int i=start;i<end;i++){
        seed+=seed /end *(time(0)/1000) + exceed;
        std::cout<<seed /100000<<"\t";
        if(i%5==0){
            std::cout<<"\n\n";
        }
    }
    return seed;
}
int main(){

   PRNG(1,100,10);
}
```

Doesn't generate repeated number nor a pattern!
Is it correct?



nascardriver
November 5, 2019 at 2:32 am · Reply

`end` and `std::time(nullptr)` are constant, you're basically doing

```
seed += ((seed * k) + exceed);
```

just more expensive. The modulus that you removed is now performed by the wrap-around of `seed`.

Take your original code and change the constant factor to an odd number, that should get rid of the repetition.

```
#include <iostream>

unsigned int PRNG()
{
  static unsigned int seed{ 9843 };
  seed = seed * 8343453 + 3555643;
  //            ^^^^^^^ odd

  return seed % 34567;
}

int main()
{
  for (int i{ 1 }; i <= 1000; ++i)
  {
    std::cout << PRNG() << '\t';
    if ((i % 5) == 0)
    {
      std::cout << "\n";
    }
  }

  return 0;
}
```

I don't know why this works, Mike observed something similar here https://www.learncpp.com/cpp-tutorial/59-random-number-generation/comment-page-6/#comment-427692

Mike
October 16, 2019 at 3:18 pm · Reply

"Consider what happens when a PRNG generates a number it has previously generated. From that point forward, it will begin to duplicate the sequence between the first occurrence of that number and the next occurrence of that number over and over. The length of this sequence is known as the period."

Is this behavior only with large ranges? Because when I do it with a range of 1-6 (roll of die) or other similar low ranges, the numbers often get repeated many times, yet I haven't noticed a pattern from any of them. Based on what Alex stated above, shouldn't a pattern start right after the first number is repeated?

Other than the much larger range, how are these different than the example that Alex provided?

nascardriver
October 17, 2019 at 1:04 am · Reply

When you generate a number from 1 to 6, you're using a way larger number and scale it down. When you see two 1s, they're probably not both based on the same number.
Consider a simple `std::rand` generator

```
int die{ (std::rand() % 6) + 1 };
```

`die` is 1 for 1, 7, 13, 19, etc.

Mike

October 17, 2019 at 9:25 am · Reply

Thanks for the feedback!
So if I understood you correctly, scaling down from a larger number, like maybe the RAND_MAX of 32767, to a significantly smaller range of numbers, like the roll of a die, will likely contribute to a longer or better period?

nascardriver

October 18, 2019 at 1:27 am · Reply

Scaling down the number doesn't affect the period. The period is caused by the implementation of the PRNG, which you're not changing.

hellmet

October 16, 2019 at 5:24 am · Reply

I have a few questions with the examples here.

```
...
    Explain what's happening here please...
std::mt19937 mersenne(static_cast<std::mt19937::result_type>(std::time(nullptr)));

    std::uniform_int_distribution<> die(1, 6);  <- What is die here? Looks like variabl

// Print a bunch of random numbers
for (int count = 1; count <= 48; ++count)
{
    std::cout << die(mersenne) << "\t"; <- die used like a function here?
...
```

What is 'mersenne' here? I think it's the uniform initialization of
What is the type of 'die' here? It looks like a variable but later used as a function?

nascardriver

October 16, 2019 at 5:34 am · Reply

\`mersenne\` is an \`std::mt19937\`, a mersenne twister. It's a random number generator, seeded with the current time.
\`die\`'s type is \`std::uniform_int_distribution\`, you don't need the \`<>\`. It's a variable, but its \`operator()\` has been overloaded so it can be used as if it was a function (Covered later).
\`die\` uses \`mersenne\` to generate a random number. Then it turns the random number into a number in the range of 1 to 6 with all numbers being equally likely to occur.

hellmet

October 16, 2019 at 7:05 am · Reply

Ahh I see, variable with () is an overloaded operator, cool I'll get back to this page once that is covered; and the Mersenne is a variable that's being uniform initialized with time that's being static cast to a std::mt199...::result type. Got it, Thanks a lot!

nascardriver

October 16, 2019 at 7:30 am · Reply

> the Mersenne is a variable that's being uniform initialized
It's not uniform initialized. Uniform initialization (Brace initialization) uses curly braces

```
int i{ 123 }; // Brace initialization (Type safe)
int i(123); // Direct initialization (Not so type safe).
```

Brace initialization is a form of direct initialization, but with higher type safety and it can be used to initialize lists.

hellmet
October 16, 2019 at 10:21 am · Reply

My bad! Got the names mixed up. So kind of you point that out too! Thank you!

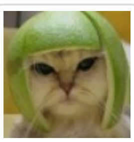
Jon
October 7, 2019 at 2:32 am · Reply

I hope you don't mind a couple of comments related to pedagogy...

I wonder if it's worth mentioning, at least in passing, cryptographically strong random and why you shouldn't use that for anything but crypto.

My take is that this lesson, while quite good, is a big jump from the last one where we just learned about the for loop. We still haven't been taught about angle brackets and such. Personally, I'm not bothered by the pacing here but I've been programming for decades and I'm just brushing up on modern C++.

Thank you so much for the lessons and please don't take this as criticism, it's just food for thought.



Alex
October 8, 2019 at 4:00 pm · Reply

I never mind feedback, criticism or otherwise. It's part of how we learn and grow.

> I wonder if it's worth mentioning, at least in passing, cryptographically strong random and why you shouldn't use that for anything but crypto.

Curious why this is relevant to the lesson.

> My take is that this lesson, while quite good, is a big jump from the last one where we just learned about the for loop

What parts do you find to be big jumps? The heavy algorithmic focus, or something more language-syntax specific?

> We still haven't been taught about angle brackets and such

Somewhere prior to this point I mention that angle brackets are how we parameterize types. I'll see if I can find and reinforce that, because it's an important point prior to being able to describe how they actually function (covered in the lessons on templates).

Mike
October 1, 2019 at 11:27 am · Reply

When debugging, how do you watch or monitor a value that keeps changing, yet doesn't have a variable assigned to it, like this PRNG statement that I took from a For Loop? Obviously, the value is there and it keeps changing, but even hovering your mouse over it while debugging will not display any results.

std::cout << std::rand() << '\t';

Alex

October 2, 2019 at 4:41 pm · Reply

One thing that works sometimes is placing a breakpoint on the entry point to wherever the intermediate value (e.g. the result of std::rand) is being sent. In this case, that would be to operator<<, which is likely going to hit everything.

Probably the easiest thing to do is assign the value you're interested in to a variable so it persists beyond the life of the expression, and examine it immediately after.

```
    int temp;
    std::cout << (temp = std::rand()) << '\t';
```

Anybody else have any better tips here?

nascardriver

October 3, 2019 at 12:36 am · Reply

I didn't reply because this depends on the debugger and IDE and I suppose Mike doesn't use gdb.
In gdb, you can "b"reak on the `std::cout ...` line, "step" into `std::rand`, then "finish" and it prints the return value.
I've seen an integrated debugger that shows return values, but I don't think it was for cpp. Though I'm sure one exist for cpp as well.

If the source code can be modified, the line can be split

```
std::cout
      << std::rand()
      << '\t';
```

then break on the `\t` line and "p"rint $rax if the architecture uses rax for return values and it's an integer.

Personally I do

```
int temp{ std::rand() };

// Debugging without a debugger.
std::cout << temp << '\n';

// Doesn't make a lot of sense with this example, imagine this
// line was something else
std::cout << temp << '\n';
```

Mike

October 1, 2019 at 10:46 am · Reply

I notice that if I alter the following line in the first code snippet from Ch5.9, that after 13 results, the remaining 87 results are always 32767. At first, I was confused by this, but then I realized, it must be because during each iteration of @PRNG(), the @seed value is always increasing. So at some point, the

@seed value is greater than 32768, and so the difference is wrapped. But at some point the amount that is now wrapped is even larger than the 32768, which apparently it can't wrap around again, and thus keeps displaying 32768. So does this mean values can only wrap around once, or is something else at play here?

My primary question is though, why when using the seed values Alex provided in the code snippet, this behavior does not happen? And his numbers are much larger than the one's I used. I even changed the count to 1000, yet the 32768 never repeats. Is he using some special magic numbers to make this happen?

original code from Ch5.9, first snippet:
seed = 8253729 * seed + 2396403;

my alteration:
seed = 2 * seed + 1;

nascardriver
October 2, 2019 at 4:33 am · Reply

The multiplication by 2 is the problem. Once you generate `seed - 1`, you'll always get `seed - 1` as the result, because

```
// @a is the previous seed
// @n is what we modulate by (In the quiz 32768).
// we know
(a % n) = (n - 1)

// Now we update the seed (a = 2a + 1).
((2a + 1) % n)
= ((2a % n) + (1 % n)) % n
= (((2a % n) + 1) % n
= (((2 * (a % n)) % n) + 1) % n
= (((2 * (n - 1)) % n) + 1) % n
= ((n - 2) + 1) % n
= (n - 1) % n
= (n - 1) // we're back where we started
```

You'll observe the same behavior when you replace 2 with any even number (I didn't prove that, but seems to work out).

> using the seed values Alex provided in the code snippet, this behavior does not happen
He uses an odd number for the multiplication. Add or subtract one and it'll get stuck too.

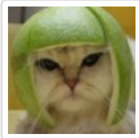
Mike
October 2, 2019 at 7:14 am · Reply

Interesting, it makes since now. It's also good to know the additional wrapping was not the issue, though for a nascent learner like myself, it seemed logical.

SullenSecret
September 17, 2019 at 10:07 pm · Reply

I'm sorry, but I feel like many parts of this lesson need further explanations. I had more trouble with it than usual due to certain details simply not being explained. It's as if this topic is disliked by the author.

Alex
September 19, 2019 at 4:02 pm · Reply



Can you be more specific about which parts were hard for you to understand? Thanks!

Raiyan

September 7, 2019 at 2:49 pm · Reply

What's the difference between

```
std::mt19937 mersenne { static_cast<std::mt19937::result_type>(std::time(nullptr)) }
```

and

```
std::mt19937 mersenne { static_cast<unsigned int>(std::time(nullptr)) }
```

.

nascardriver

September 8, 2019 at 12:48 am · Reply

`std::mt19937::result_type` is an `std::uint_fast32_t`, a 32-bit wide integer.
`unsigned int` is at least 16 bits wide.
By casting to `unsigned int` you're discarding 16 bits, greatly decreasing the number of potential seeds.
On top of that, `std::mt19937` doesn't want to have an `unsigned int`, so an implicit cast has to be performed to convert the `unsigned int` to `std::mt19937::result_type`.
Using `unsigned int` here is unfounded.

alfonso

August 18, 2019 at 1:38 am · Reply

```
#include <iostream> // for standard I/O
#include <ctime> // for std::time ()
#include <random> // for mersenne twister

namespace MyRandom {
    std::mt19937_64 seed {
        static_cast <std::mt19937_64::result_type> (std::time (nullptr))
    };
}

double randomReal (double min, double max) {
    std::mt19937_64 seed {
        static_cast <std::mt19937_64::result_type> (std::time (nullptr))
    };
    std::uniform_real_distribution <> realNumber (min, max);

    return realNumber (seed); // CASE 1
    // return realNumber (MyRandom::seed); // CASE 2
}

int main () {
    std::cout << randomReal (1.0, 3.0) << '\n';
    std::cout << randomReal (1.0, 3.0) << '\n';
    std::cout << randomReal (1.0, 3.0) << '\n';
    return 0;
}
```

I do not understand why. If I use CASE 1, the returned "random" number is always the same. But if I use CASE 2 (with global (namespaced) seed) the result is as expected, random.

nascardriver
August 18, 2019 at 1:47 am · Reply

See the paragraph "Help! My random number generator isn't generating random numbers at all!" at the end of this lesson.

alfonso
August 18, 2019 at 11:53 pm · Reply

"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."

Ok, but why? This is yet another observation not quite an explanation. The answer may reside in the PRNG implementation.

Here because the program runs way below a second, three successive calls to std::time () should return the same timestamp value so seed should (?) have the same value. In the other case, the global seed has the same value at each randomReal () call.

There are many whys in my head, a lot of confusion too. For example, why reseeding is bad when reseeding should give even more randomness and so on. I will go deep in the mersenne implementation, if not now for sure in the near future. Thank you!

nascardriver
August 19, 2019 at 2:34 am · Reply

A random number generator generates a predictable, seemingly random, sequence of numbers. The seed changes what this sequence looks like or where it starts. If you use the same seed twice, you'll get the same sequence twice.

> reseeding should give even more randomness
Random number generators can't be improved by changing the seed.

alfonso
August 19, 2019 at 11:47 pm · Reply

Yes, I would expect that an unchangeable seed to give me similar results. But here in both cases the seed did not changed between successive randomReal () calls. You can verify this using a code line:

```
std::cout << "Seed: " << seed << '\n';
```

just before line 17 and you will see the same seed value for each randomReal () call.

Though, when using global case it generates random different results. This I want to understand, why Alex MUST use a global variable. Of course, I can see the different output of programs, but I still do not understand why the global variable affects different the final output. Again, in both cases the seed does not change between successive randomReal () calls. The reason must be somewhere else.

nascardriver

August 20, 2019 at 12:08 am · Reply

> the seed did not changed between successive randomReal () calls
That's the point. You're creating a new twister and seeding it with the same seed as last time every time you call `randomReal`. This will produce the same sequence and the first call to `realNumber` will give you the same number as it did last time.
With the global variable, the twister is only seeded once.

You might want to have a look at how simple pseudo random number generators are implemented (Don't look at the mersenne twister, unless you're up for some math and bits).

alfonso
August 20, 2019 at 11:19 pm · Reply

So it depends how the seed is used by mersenne twister. The seed is the same in all cases, but the seed is a sequence of many numbers and how that seed is used makes the difference. Seeding again it will use the same seed (sequence of numbers) in the same way. Seeding once it will use the same seed but in different ways between successive calls. So it looks like just scratching the surface and with your help. I can handle some math and bits but I can't handle class templates yet. Thank you!

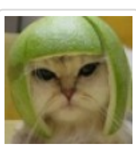
nascardriver
August 21, 2019 at 4:58 am · Reply

> the seed is a sequence of many numbers
Not quite. The seed only initializes the generator and determines what the sequence will look like. Re-using a seed is like resetting the generator.

> I can handle some math and bits but I can't handle class templates yet.
You don't need to look at the cpp implementation. Mersenne twister is not language-specific.
https://en.wikipedia.org/wiki/Mersenne_Twister#Algorithmic_detail

Parsa
August 16, 2019 at 3:57 pm · Reply

How do you set the value of RAND_MAX since RANDMAX is a constant and is not modifiable?



Alex
August 16, 2019 at 9:19 pm · Reply

I think you just answered your own question. :)

You can't. If you need a random library that supports a wider range, use Mersenne Twister or something else.

« Older Comments 1 … 4 5 6