

There is nothing wrong with `rand` per se, but there *is* a lot of confusion about what `rand` does and how it does it. The often results in `rand` being used in situations where it is a very poor choice. In general, `rand` should be avoided except in the most simple of cases where pseudorandom numbers are desired, and even then there are caveats. So the question is, what exactly is the problem with `rand` ? Let us count the ways:

- **Range:** `rand` returns numbers in the range of $[0, \text{RAND_MAX})$, and `RAND_MAX` is specified with a minimum value of 32,767. Portable code cannot assume that the range is any larger than this, so portable code is very limited in the random numbers that can be predictably generated. Compare this with better random number generators that have a range of 32 or 64 bits.
- **Period:** The period of `rand` is implementation defined, but typically falls around $2^{32}-1$. Compare this with the Mersenne Twister algorithm, which is $2^{19937}-1$. You typically want the period of a random number generator to exceed the amount of numbers expected to be generated, because that's the point where the sequence repeats.
- **Quality:** `rand` has a history of poor implementations, where the quality of the numbers generated were either very predictable or not 'random' enough. Notable in these implementations the low order bits were decidedly *not* random, which prompted programmers to only use the high order bits for adjusting ranges (which we'll see shortly).
- **Distribution:** First and foremost, `rand` is *not* required to conform to any standard distribution. Typically, the base random number generator is expected to return values in a uniform distribution. That is, every number in the range is equally likely to be returned for any given request. Even when `rand` claims a uniform distribution, it may not be a terribly good distribution, which greatly affects any attempts to adjust the range.

So when should you use `rand` ? According to the above, `rand` is fine to use when you don't need numbers outside of $[0, 32,767)$, won't generate more numbers than the common period, don't particularly care about the quality of the pseudorandom sequence, and don't need the sequence to conform to a strict distribution. That's a lot of not caring for my tastes, but there are certainly situations where all of them apply.

However, there is another wrench that can be thrown in our gears, because a great many applications need a smaller range than $[0, 32,767)$. For this, the programmer typically adjusts the range by forcing a number returned by `rand` into a smaller range. For example, to roll a 6 sided die, one would prefer to only receive values in the range of $[0, 6)$. This is often done using the remainder operator:

```
int r = rand() % N;
```

Or if both a lower and upper bound are needed:

```
int r = L0 + rand() % (HI - L0);
```

Anyone who does this will be rewarded with a seemingly random sequence and be thrilled that their clever solution worked. Unfortunately, this does not work. The first solution only works when `N` evenly divides into `RAND_MAX`. The second solution isn't any better. The reason is because forcing the range in this way

eliminates any chance of having a uniform distribution. Now, this is okay if you care nothing about some numbers being more probable than others, but to be correct, you must work with the distribution instead of destroy it.

Why does the distribution get destroyed though? It's the pigeonhole principle (http://en.wikipedia.org/wiki/Pigeonhole_principle). If you have a bunch of numbers and try to merge them into a smaller number of buckets, unless the number of buckets is evenly divisible by the range of numbers, some buckets will be given more numbers than other buckets. Thus the distribution of numbers in the buckets is worse than the distribution of numbers in the original range.

Most experienced C programmers will tell you that using remainder with random numbers is bad, but not many of them will be able to tell you why. This is similar to the `goto` issue, where beginners are taught to blindly believe that any code with a `goto` statement is spaghetti code and any use of `goto` is evil. The solution suggested to get around the remainder issue is usually division by `RAND_MAX`. For a range of `[0, N)`, you would do something like this:

```
int r = rand() / (RAND_MAX / N + 1);
```

With the lower and upper bound support looking like this:

```
int r = M + rand() / (RAND_MAX / (N - M) + 1);
```

All is well, right? This produces a seemingly random sequence. The seasoned community is happy because you are not using the remainder operator with `rand`, and you are happy because you got your random numbers and foiled the evils of `rand`. But wait! This is not an improvement...at all. This 'solution' does nothing to alleviate distribution problems. In reality, it is a fix for poor implementations of `rand` where the low order bits are very non-random. The remainder operator uses the low order bits primarily while the 'fix' uses the high order bits. It may help, but it doesn't do jack diddly where distribution is concerned.

A *real* fix is to work with the natural distribution rather than try to circumvent it. Simply call `rand` until you get a number that fits within your selected range:

```
int r;

do
{
    r = rand();
} while (r >= HI);
```

Or for both lower and upper bounds:

```
int r;

do
{
    r = rand();
} while (r < LO || r >= HI);
```

Not only does this fix the distribution problem, it also avoids the low order bit problem entirely. The downside to this solution is that the performance dictated by the range. A small range could potentially call `rand` many times. Unfortunately, there is not a workaround for this if you are interested in maintaining the natural distribution of the random number generator. Further, this does *not* fix any problems in the natural distribution of the random number generator. If `rand` does not have a good uniform distribution naturally, the above loops will not improve matters.

Often you'll see recommendations to introduce floating-point to 'fix'...something. I myself made this recommendation in a previous incarnation of this article, and it shames me to admit it. The problem is floating-point creates its own problems, and those problems often exacerbate the problems you're using it to solve. As they say, if you have 1 problem and decide to use floating-point, now you have 2.000001 problems. Please do not use floating-point to try fixing distribution issues with `rand`, it will only result in further heartache.

It should be kept in mind that these distribution problems are consistent with all random number generators that return a ranged integer, not just `rand`. So please do not read this article as me harping on `rand`, rather read it as me harping on the majority of programmers who use it incorrectly. I suppose that's not much better, but it is what it is. :) And note that even if you are using the most awesome random number generator available, adjusting the range *will* require some thought if it needs to be done correctly.

Seeding `rand`

Probably the most vexing problem with how people use `rand` is how the generator is seeded. The usual solution is to get the system time using the `time` function. In theory, this is a great idea because not only does it seed the random number generator with an ever changing value, but it is also portable:

```
srand((unsigned int)time(NULL));
```

Right? Well no, not really. The issue is that `time_t` is a restricted type, and may not be meaningfully converted to `unsigned int`. That is not a terribly big issue as I have yet to see a system where it would fail to work correctly, and I do not know anyone who has either. But subtle lingering portability issues *should* leave a sour taste in the mouth. Fortunately, there is a way to use the result of `time` portably as a seed for `rand`; just hash the bytes of a **`time_t`**:

```
unsigned time_seed()
{
    time_t now = time(0);
    unsigned char *p = (unsigned char *)&now;
    unsigned seed = 0;

    for (size_t i = 0; i < sizeof now; i++)
    {
        seed = seed * (UCHAR_MAX + 2U) + p[i];
    }

    return seed;
}

...

srand (time_seed());
```

A hash will take advantage of the way the system time changes. Even better, the C and C++ standards guarantee that type punning is a portable operation for simple types, and `time_t` is a simple type. So hashing the system time and seeding `rand` with it is a portable solution with desirable properties.

Of course, one might wonder why we should go to all of the trouble when `time_t` is extremely likely to work. The uncertainty of being able to use `time_t` alone should be a good enough reason to avoid directly seeding `rand` with one.

Now for the caveat, and there always seems to be one. :(The resolution for `time` is 1 second. That's not a requirement, but it is the overwhelming choice in implementations. This means that calling `time_seed` multiple times in a second will result in the same seed. Further, because the time is predictable, the seed will also be predictable. Unfortunately, I have yet to find a better way to seed a random number generator in C in a portable manner. I've actually received a lot of feedback on this (thank you! keep it coming), but none of them were portable. The best one was a hash of the `environ` variable in POSIX-based systems, and while it was very interesting, for such systems I would prefer simply using `/dev/random` from the beginning to avoid all of the problems from this article entirely.

Honorable Mention (C++11)

The C++11 standard introduced the `<random>` header, which offers some amazingly effective, yet equally amazingly confusing random number utilities. Ultimately for the average programmer it is rather simple, but you need to recognize that the majority of the classes can be ignored. Typically, the only ones that matter are:

- `std::mt19337` : Mersenne Twister 32 bit implementation.
- `std::mt19337_64` : Mersenne Twister 64 bit implementation.
- `std::random_device` : Implementation-dependent random number generator.
- `std::uniform_int_distribution` : Accurately adjusts numbers from a generator into a uniform integer range.
- `std::uniform_real_distribution` : Accurately adjusts numbers from a generator into a uniform floating-point range.

Using the library is straightforward. You create a random number generator (an engine object), create a distribution object, then apply the engine object to the distribution object to get the random numbers you want using `operator()` of the distribution object. Since `std::random_device` is notably slower than the Mersenne Twister, my preference is to use it only for seeding the Mersenne Twister algorithm in cases where Mersenne Twister is the primary engine:

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

int main()
{
    std::random_device seed;
    std::mt19337 gen(seed());
    std::uniform_int_distribution<int> dist(0, 100);

    for (int i = 0; i < 10; i++)
    {
        std::cout << dist(gen) << '\n';
    }
}
```

Obviously, if you're writing code in C, this option is not available. But for C++ programmers, you should probably forget that `rand` exists and use the new `<random>` library.

Conclusion

There is really nothing wrong with `rand`, despite what you may be told. The problem is with the people who use it, and how they use it incorrectly and without having all of the facts. Another problem is that one can use `rand` incorrectly and still get seemingly random results, so the incorrect use appears to work as expected.

Now for the big question: Does any of this matter to you? Yes, of course it does. A keen understanding of how things work will give you insight into writing good code. Will you encounter these problems in everyday programming? Probably not. Pseudorandom numbers are typically expected to be sufficiently random for the application but no more. As such, most people who use `rand` will either not see, or not care, about the discussed issues. In fact, these issues are generally not noticeable except on a very large scale or a very small scale.

An article on `rand` isn't complete without the usual caveat that `rand` isn't required to be a strong random number generator. In fact, it's usually pretty weak. For little things that don't need a good random number generator, `rand` will do fine, and whatever option you use for adjusting ranges will be equally fine. For everything else, a much stronger algorithm with more guarantees, such as the Mersenne Twister, is strongly recommended. In other words, for a dice rolling school project, `rand` is just peachy. But for a slot machine in Las Vegas, you need the very best you can get, and `rand` is not it.