Random Numbers in C

The C standard library includes a functio[n] named **rand**, which returns a random integer between 0 and RAND_MAX, as defined in the <stdlib.h> header file. The **rand** function is handy for developing simulations, games, and other applications that require randomness, but it is one of the most misunderstood and often misused functions in the standard library. Many books, articles, and speakers have discussed the theory and practice of random number generation, some including harsh criticism of the **rand** function.

There is nothing technically wrong with the **rand** function, but it's important to understand its behavior and limitations, so we can use it wisely. In this article, I'll explain what the **rand** function provides, and common approaches to gaining control over the randomness we're looking for. And, for those of us willing to venture into non-portable code territory, I'll also offer a few non-portable options.

Psuedo-Random Acts of Randomness

Successive calls to the **rand** function actually return what is known as a *pseudo-random sequence*. If we call the **rand** function 100 times in a row within a single run of our program, we'll get what appear to be 100 randomly-chosen numbers. However, if we run our program over again, we'll get the exact same sequence of 100 numbers. In fact, every time we run the program, we'll see the exact same sequence of 100 numbers returned by the 100 calls to **rand**. The function uses an algorithm that generates a sequence of values which does its best to approximate the properties of a truly random sequence. But truly random it clearly is not.

This behavior makes testing our program easier, because we'll always get the exact same sequence of numbers. However, this behavior makes guessing games significantly less challenging for the user; once they figure out the pattern is repeating every time the program is run, they'll be able to win every time. And if we're attempting to program a serious simulation, having the same sequence of events occur on every run of the program can erode the credibility of our simulation.

In Search of a Different Random Sequence

How do we make the **rand** function return something truly random - or at least something that seems more random? This is where the **srand** function enters the picture. The **srand** function, also declared in <stdlib.h>, lets us specify a *seed* for the random number generator, so that later calls to **rand** will return an oddball random value on the first call, and then what appears to be other random values after that. Each unique random seed, provided to the

srand function, essentially kicks off a different sequence of pseudo-random numbers from **rand**.

The good news is that a single call to **srand** at the beginning of the program will affect the sequence of values returned from all subsequent calls to **rand** in that program. The bad news is that, if we pass the exact same value into **srand** every time we run the program, we'll get the same sequence of values out of **rand** every time we run the program. So, unless we can somehow pass a different seed into **srand** every time we run the program, we're really back where we started, with the same sequence repeating every time we run the program.

One approach would be to ask the user for some unsigned integer value, have the program use that number as the seed passed into the **srand** function, and then continue with the program, calling **rand** as needed to get our random numbers. However, if the user always supplies the same seed to the **srand** function, they'll always get the same sequence of values out of subsequent calls to **rand**. Again, that approach puts us back where we started. It puts the burden on the user to select a new seed on each program run, and we know that users will just tend to fall unto a rut by using the same value on every run (and the same password on every web site, etc.). Moreover, if our program is not interactive, there is no user interface, and thus no way for the user to specify the seed. "Ah," I hear you say, "Have the user put the seed in a configuration file," you say. Well, the likelihood of a user changing the configuration file on every run is, at best, slim to none.

What we need to do is pass some truly random value into the **srand** function, so that the seed we're starting with is truly random, and the sequence of values we get out of **rand** will be different every time we run the program.

A Time for Randomness

One popular approach is to grab the current time and use that value as the seed we pass to **srand**. The C libraries contain a function named **time**, declared in <time.h>, which returns the number of seconds elapsed since midnight, January 1, 1970. Here's an example of how to use the **time** function to seed the random number generator in a single call to **srand** at the beginning of the program:

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

This statement grabs the current internal clock time using the **time** function, converts (i.e., casts) it to an unsigned integer, so that it can be passed into **srand**, and uses that value as the seed for the random number generator.

Keep in mind that we only need to call **srand** once, before our first call to **rand**, and then use calls to **rand** to generate as many pseudo-random numbers as we want. Calling **srand** with a new seed will begin a new pseudo-random sequence; calling **srand** with the same seed with start the same sequence over again.

One downside to this approach is that the data type returned by the **time** function, time_t, is not guaranteed to boil down to an unsigned integer, so a cast to unsigned integer might not produce a meaningful result. Implementations are free to define the time_t data type in other ways. In practice, implementations use unsigned integer as the basis for the data type returned. In practice, having worked with many different C compilers and libraries over many years, we have not yet run into any that would fail to do the right thing. But because there are no guarantees governing how time_t might be implemented, the code above is not truly portable. Here is a portable alternative that hashes the individual bytes of the time function's returned value to generate the seed for **srand**:

```
void srandUsingTimeBytes(void)
{
    time_t currentTime = time(NULL);
    unsigned char *pByte = (unsigned char *)&currentTime;
    unsigned seed = 0;

    for (size_t index = 0; index < sizeof(currentTime); index++)
    {
        seed = seed * (UCHAR_MAX + 2U) + pByte[index];
    }

    srand(seed);
}
```

Controlling the Random Range

The **rand** function takes no parameters, and returns an integer. The C standards guarantee that RAND_MAX is at least 32767, but it may be significantly higher than that in a given compiler implementation. If we need to refer to the largest possible random value returned by **rand**, and we want our code to be portable to other environments, use the symbol RAND_MAX instead of hard-coding the actual number.

Using the above techniques, we can achieve good randomness using a single call to **srand** at the beginning of the program. But what if our goal is to have the program select a random value that is in a range smaller than 0 to RAND_MAX? What if we wanted our program to select a random value in the range 1 to 10? In this range, there are exactly 10 possible values. Recall our discussion of the modulo operator (the % operator in an arithmetic expression. It provides us with the remainder of a division operation. Consider the following expression:

```
rand() % 10
```

The value of that expression will be the remainder when the value returned by **rand** is divided by 10. If the value returned by **rand** is divisible by 10, the remainder will be 0. If it is not divisible by 10, the remainder will be in the range 1 through 9. So, the above expression actually does narrow the range for us; it narrows the range down to 0 through 9 inclusive. But we wanted a value in the range 1 through 10. We can easily shift the entire range up by one by, well, adding one:

```
(rand() % 10) + 1
```

The extra parentheses have been added for clarity, but are not required because the modulo operator has higher precedence than the addition operator. The value inside the parentheses will be a number in the range 0 through 9. By adding one to whatever that value is, the whole expression will have a value in the range 1 through 10. So, a statement that captures this value into an integer variable might look like:

```
myRandomValue = (rand() % 10) + 1;
```

If we need to generate random values in a variety of ranges, we can add the following C function to our code to provide a more general solution:

```
int randRange(int minValue, int maxValue)
{
    return ((rand() % (maxValue - minValue + 1)) + minValue);
}
```

and call it with the range of values we're interested in:

```
myRandomValue = randRange(1, 10);
```

which generates a value in the range 1 through 10.

A more in-depth discussion of the problems with random numbers follows.

Read on, if you're thoroughly fascinated by this subject.

The Pigeonhole Principle

Unfortunately, this technique almost always skews the distribution of the pseudo-random sequence, causing some values to be generated much more often than others. Part of the problem is that many implementations produce very non-random low-order bits, and remainder approach focuses on the low-order bits. While there are ways to overcome this problem, they don't address the second issue – the *pigeonhole principle*:

```
Given a set of numbers, if you attempt merge them all into a
smaller number of boxes, unless the number of boxes is evenly
divisible by the range of the original set of numbers, some boxes
will contain more numbers than other boxes. Therefore, the
distribution of numbers in boxes is going to be worse than the
distribution of numbers in the original set.
```

Ultimately, the only sure way to get a distribution that matches the original **rand** function is to call it repeatedly:

```
int randRange(int minValue, int maxValue)
{
    int randomValue;

    do
    {
        randomValue = rand();
    }
    while ((randomValue < minValue) || (randomValue > maxValue));

    return randomValue;
}
```

Clearly, the smaller the requested range, the higher the number of potential iterations in the loop, so performance can suffer significantly.

The Quality of Randomness

The overall quality of a random number generator is affected by the following:

- **Range** - The **rand** function generates values in the range 0 to RAND_MAX, which is guaranteed by the C standard to be at least 32767. Implementations that use this maximum have a smaller range of values to return than implementations that use 32 or 64 bits.
- **Period** - The period of **rand** is left open by the C standard as implementation defined. Real-world implementations typically have a period of about 2^32. The higher the period, the less likely we'll run into a repeating pseudo-random sequence. The period of

the generator should at least exceed the number of random numbers we'll request, or we'll run into repetition.

- **Distribution** - The C standard doesn't require values returned by **rand** to be evenly distributed. Values returned by a good generator are expected to be uniformly distributed. (Think of a good distribution as: "Every value in the range is equally likely to appear in the next call to the function. No value is more likely to appear than any other value." Think of a bad distribution in terms of George Orwell's *Animal Farm*, where "All values are equally likely to occur, but some values are more likely to occur than others.")
- **Predictability** - Some algorithms generate more predictable sequences than others.

The C standard leaves the details up to the C library implementation, and other than stating a minimum range, none of the above factors is specified. There has been so much discussion about the quality of random sequences returned by **rand**, that the C11 standard adds this disclaimer:

> *"There are no guarantees as to the quality of the random sequence produced and some implementations are known to produce sequences with distressingly non-random low-order bits. Applications with particular requirements should use a generator that is known to be sufficient for their needs."*
>
> [ISO/IEC 9899:2011, 7.22.2.1]

None of the techniques discussed earlier do anything to overcome a low quality implementation of **rand**.

Non-Portable Randomness

For some applications where better randomness is required, several non-portable alternatives have arisen. Most modern operating systems provide mechanisms for generating cryptographically secure pseudo-random sequences. But there is a price for using these approaches:

- They are typically significantly slower at generating values than the **rand** function.
- They are non-portable, because they are not part of the C standard library, and because they are tied to low-level functions in the operating system, or to intrinsics provided by the specific compiler implementation.

Hardware-generated random values are also a possibility, but that approach has the disadvantage of reducing source code portability by tying code to a specific hardware device or CPU version.

That said, here are a few non-portable alternatives:

- **The rand_s function (Windows)** - The Windows **rand_s** function uses a low-level operating system function, available on Windows XP and later. For more details on this function, see https://msdn.microsoft.com/en-us/library/sxtz2fa8(v=vs.120).aspx.

- **The CryptGenRandom function (Windows)** - The Windows **CryptGenRandom** function is another possibility, also available on Windows XP and later. It requires including <windows.h>, and linking to advapi32.lib. For more details on this function, see https://msdn.microsoft.com/en-us/library/windows/desktop/aa379942(v=vs.85).aspx.
- **The rdrand instruction (Intel only, Ivy Bridge and later)** - Since early 2012, Intel CPUs (Ivy Bridge platform and later) provide the **rdrand** instruction. It requires dipping into assembly language, or linking to a library that uses this instruction. For more details on this CPU instruction, see https://software.intel.com/en-us/blogs/2011/06/22/find-out-about-intels-new-rdrand-instruction/.
- **The /dev/random generator (Unix/Linux derivatives)** - This mechanism is available on most Unix- and Linux-derived operating systems, although the specific semantics vary across implementations. For a general overview, see https://en.wikipedia.org/wiki//dev/random.
- **The random function (POSIX)** - POSIX-compliant libraries include the **random** function, which is considered better than most **rand** implementations because it guarantees a specific period. For more details, see http://pubs.opengroup.org/onlinepubs/009695399/functions/initstate.html.
- **The ar4random (BSD derivatives)** - This function is available in some BSD (Berkley Software Distribution) derivatives, and is generally considered better than the POSIX **random** function. For more details, see http://www.manpagez.com/man/3/arc4random_uniform/.

Random Conclusion

There is nothing technically wrong with the **rand** function. It is both portable and useful. But it's important to have a clear understanding of what it provides, how to use it effectively, and what alternatives there are if **rand** doesn't meet our needs.