C++11: How to set seed using <random>

Asked 5 years, 11 months ago Active 5 years, 11 months ago Viewed 29k times

23

9

I am exercising the random library, new to C++11. I wrote the following minimal program:

```
#include <iostream>
#include <random>
using namespace std;
int main() {
    default_random_engine eng;
    uniform_real_distribution<double> urd(0, 1);
    cout << "Uniform [0, 1): " << urd(eng);
}
```

When I run this repeatedly it gives the same output each time:

```
>a
Uniform [0, 1): 0.131538
>a
Uniform [0, 1): 0.131538
>a
Uniform [0, 1): 0.131538
```

I would like to have the program set the seed differently each time it is called, so that a different random number is generated each time. I am aware that random provides a facility called seed_seq, but I find the explanation of it (at cplusplus.com) totally obscure:

http://www.cplusplus.com/reference/random/seed_seq/

I'd appreciate advice on how to have a program generate a new seed each time it is called: The simpler the better.

My platform(s):

- Windows 7 : TDM-GCC compiler

c++11 random

Share Edit Follow

edited Dec 28 '15 at 12:07
Mickaël Rémond
8,850 1 20 43

asked Dec 28 '15 at 9:09
Argent
747 1 6 16

2 Answers

Active | Oldest | Votes

23

The point of having a `seed_seq` is to increase the entropy of the generated sequence. If you have a random_device on your system, initializing with multiple numbers from that random device may arguably do that. On a system that has a pseudo-random number generator I don't think there is an increase in randomness, i.e. generated sequence entropy.

Building on that your approach:

If your system does provide a random device then you can use it like this:

```
std::random_device r;
// std::seed_seq ssq{r()};
// and then passing it to the engine does the same
default_random_engine eng{r()};
uniform_real_distribution<double> urd(0, 1);
cout << "Uniform [0, 1): " << urd(eng);
```

If your system does not have a random device then you can use `time(0)` as a seed to the random_engine

```
default_random_engine eng{static_cast<long unsigned int>(time(0))};
uniform_real_distribution<double> urd(0, 1);
cout << "Uniform [0, 1): " << urd(eng);
```

If you have multiple sources of randomness you can actually do this (e.g. 2)

```
std::seed_seq seed{ r1(), r2() };
  default_random_engine eng{seed};
  uniform_real_distribution<double> urd(0, 1);
  cout << "Uniform [0, 1): " << urd(eng);
```

where r1 , r2 are different random devices , e.g. a *thermal noise* or *quantum source* .

Ofcourse you could mix and match

```
std::seed_seq seed{ r1(), static_cast<long unsigned int>(time(0)) };
  default_random_engine eng{seed};
  uniform_real_distribution<double> urd(0, 1);
  cout << "Uniform [0, 1): " << urd(eng);
```

Finally, I like to initialize with an one liner:

```
  auto rand = std::bind(std::uniform_real_distribution<double>{0,1},
              std::default_random_engine{std::random_device()()});
  std::cout << "Uniform [0,1): " << rand();
```

If you worry about the `time(0)` having second precision you can overcome this by playing with the `high_resolution_clock` either by requesting the time since epoch as designated firstly by bames23 below:

```
static_cast<long unsigned int>
(std::chrono::high_resolution_clock::now().time_since_epoch().count())
```

or maybe just play with CPU randomness

```
long unsigned int getseed(int const K)
{

    typedef std::chrono::high_resolution_clock hiclock;

    auto gett= [](std::chrono::time_point<hiclock> t0)
    {
        auto tn = hiclock::now();
        return static_cast<long unsigned int>
(std::chrono::duration_cast<std::chrono::microseconds>(tn-t0).count());
    };

    long unsigned int diffs[10];
    diffs[0] = gett(hiclock::now());
    for(int i=1; i!=10; i++)
    {
        auto last = hiclock::now();
        for(int k=K; k!=0; k--)
        {
            diffs[i]= gett(last);
        }
    }

    return *std::max_element(&diffs[1],&diffs[9]);
}
```

Share Edit Follow

edited May 23 '17 at 10:29
Community Bot
1 1

answered Dec 28 '15 at 11:53
g24l
2,869 11 28

Thanks, g241. After modifying the initialization of default_random_engine to include a call to time(0) program output is non-deterministic. – Argent Dec 28 '15 at 17:03

@Argent , AFAIK on Windows there is no random engine , time(0) initializes the seed to current with second precision, so subsequent runs are seeded differently and achieve your goal . A second is your granularity. if this answer solves your problem please accept it. – g24l Dec 28 '15 at 20:11

1 Windows absolutely has a random engine; the default uses rand_s() which uses RtlGenRandom, which calls into advapi32.dll to generate fairly-good random numbers (without the cost of going all the way to crypto.) – Jon Watte Sep 6 '17 at 19:27

@JonWatte: I suspect g24l is thinking of an issue with MinGW, which, at least on Windows, is producing a 100% repeatable, deterministic number stream (as of 2017, no idea if/when it will be fixed). MSVC provides a usable random engine, and such features are available on Windows in general, but the `libstdc++` MinGW uses doesn't even try to use it. – ShadowRanger Oct 30 '18 at 17:00

Friends don't let friends use MinGW. Visual Studio is a free download. Fixing warnings VS finds that g++ doesn't, is good for your code! – Jon Watte Oct 30 '18 at 18:24

12

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

using namespace std;

int main() {
    std::random_device r;                                   // 1
    std::seed_seq seed{r(), r(), r(), r(), r(), r(), r(), r()}; // 2
    std::mt19937 eng(seed);                                 // 3

    uniform_real_distribution<double> urd(0, 1);

    cout << "Uniform [0, 1): " << urd(eng);
}
```

In order to get unpredictable results from a pseudo-random number generator we need a source of unpredictable seed data. On *1* we create a `std::random_device` for this purpose. On *2* we use a `std::seed_seq` to combine several values produced by `random_device` into a form suitable for seeding a pseudo-random number generator. The more unpredictable data that is fed into the `seed_seq`, the less predictable the results of the seeded engine will be. On *3* we create a random number engine using the `seed_seq` to seed the engine's initial state.

A `seed_seq` can be used to initialize multiple random number engines; `seed_seq` will produce the same seed data each time it is used.

Note: Not all implemenations provide a source of non-deterministic data. Check your implementation's documentation for `std::random_device`.

If your platform does not provide a non-deterministic `random_device` then some other sources can be used for seeding. The article *Simple Portable C++ Seed Entropy* suggests a number of alternative sources:

- A high resolution clock such as `std::chrono::high_resolution_clock` (`time()` typically has a resolution of one second which generally too low)
- Memory configuration which on modern OSs varies due to address space layout randomization (ASLR)
- CPU counters or random number generators. C++ does not provide standardized access to these so I won't use them.
- thread id
- A simple counter (which only matters if you seed more than once)

For example:

```
#include <chrono>
#include <iostream>
#include <random>
#include <thread>
#include <utility>

using namespace std;

// we only use the address of this function
static void seed_function() {}

int main() {
    // Variables used in seeding
    static long long seed_counter = 0;
    int var;
    void *x = std::malloc(sizeof(int));
    free(x);

    std::seed_seq seed{
        // Time
        static_cast<long long>(std::chrono::high_resolution_clock::now()
                               .time_since_epoch()
                               .count()),
        // ASLR
        static_cast<long long>(reinterpret_cast<intptr_t>(&seed_counter)),
        static_cast<long long>(reinterpret_cast<intptr_t>(&var)),
        static_cast<long long>(reinterpret_cast<intptr_t>(x)),
        static_cast<long long>(reinterpret_cast<intptr_t>(&seed_function)),
        static_cast<long long>(reinterpret_cast<intptr_t>(&_Exit)),
        // Thread id
        static_cast<long long>(
            std::hash<std::thread::id>()(std::this_thread::get_id())),
        // counter
        ++seed_counter};

    std::mt19937 eng(seed);
```

```
    uniform_real_distribution<double> urd(0, 1);

    cout << "Uniform [0, 1): " << urd(eng);
}
```

Share Edit Follow

edited Dec 29 '15 at 18:06

answered Dec 28 '15 at 9:12

bames53

81.5k 13 165 233

Could you add more comments within code to elaborate a bit more ? :) – Richard Dally Dec 28 '15 at 9:14

1 bames53: I compiled and ran your code. I continue to get a deterministic output, i.e., the same each time. It may be that my platform is an issue. It is Windows 7, and I am runninging the TDM-GCC compiler. – Argent Dec 28 '15 at 9:39

2 The issue is that libstdc++ on Windows falls back to a deterministic implementation. They haven't yet tried to hook into the Windows' OS facilities for non-determinism (and of course Windows doesn't provide the facilities that libstdc++ uses on *nix platforms). If you build the program with VS2015 or with gcc on Linux then you'll get non-repeated results. If you want to continue to use gcc on Windows then you'll need to replace `random_device` with something like using Windows CryptoAPI. – bames53 Dec 28 '15 at 10:26