Wojciech Olech (2017)

Wstęp

Wiele osób nadal korzysta z przestarzałego std::rand do generowania liczb pseudolosowych w języku C++. Z racji że w C++11 pojawiła się biblioteka <random>, która oferuje znacznie więcej lepszych (pod wieloma względami) metod generowania liczb pseudolosowych, postanowiłem empirycznie sprawdzić jeden z podstawowych parametrów jaki może interesować potencjalnego programistę - szybkość generowania.

Do testów wykorzystałem biblioteki do mikrobenchmarków od Google. Zrobiłem testy dla wszystkich dostępnych w STLu silników generowania liczb z użyciem jednolitej dystrybucji liczb całkowitych oraz rzeczywistych. Efekty przedstawiają sie następująco:

Wyniki

```
Run on (4 X 2394 MHz CPU s)
10/08/17 19:42:52
------------------------------------------------------------------------------
--------------------
Benchmark                                                      Time
CPU Iterations
------------------------------------------------------------------------------
--------------------
old_rand                                                    14 ns
14 ns   49777778
old_rand_int_range/1234/12345678                            14 ns
15 ns   44800000
old_rand_real_range/1234/12345678                           15 ns
15 ns   49777778
random_uniform_int_dist<std::minstd_rand0>/1234/12345678     8 ns
8 ns   74666667
random_uniform_int_dist<std::minstd_rand>/1234/12345678      7 ns
7 ns   89600000
random_uniform_int_dist<std::mt19937>/1234/12345678         10 ns
10 ns   74666667
random_uniform_int_dist<std::mt19937_64>/1234/12345678      22 ns
22 ns   29866667
random_uniform_int_dist<std::ranlux24_base>/1234/12345678   21 ns
20 ns   37333333
random_uniform_int_dist<std::ranlux48_base>/1234/12345678   25 ns
21 ns   29866667
random_uniform_int_dist<std::ranlux24>/1234/12345678       139 ns
138 ns    4977778
random_uniform_int_dist<std::ranlux48>/1234/12345678       287 ns
289 ns    2488889
random_uniform_int_dist<std::knuth_b>/1234/12345678         25 ns
25 ns   28000000
random_uniform_real_dist<std::minstd_rand0>/1234/12345678   15 ns
```

```
15 ns   44800000
random_uniform_real_dist<std::minstd_rand>/1234/12345678          13 ns
13 ns   56000000
random_uniform_real_dist<std::mt19937>/1234/12345678              12 ns
12 ns   56000000
random_uniform_real_dist<std::mt19937_64>/1234/12345678           19 ns
18 ns   40727273
random_uniform_real_dist<std::ranlux24_base>/1234/12345678        27 ns
25 ns   26352941
random_uniform_real_dist<std::ranlux48_base>/1234/12345678        17 ns
18 ns   40727273
random_uniform_real_dist<std::ranlux24>/1234/12345678            238 ns
241 ns    2986667
random_uniform_real_dist<std::ranlux48>/1234/12345678            541 ns
544 ns    1120000
random_uniform_real_dist<std::knuth_b>/1234/12345678              49 ns
49 ns   11200000
```

Wnioski

Generowanie liczb z użyciem starego std::rand wynosiło w każdym przypadku średnio 15ns. Nie jest to zły wynik pod względem szybkości, ale ta metoda losowania jest w tym momencie wysoce niewygodna i na dodatek nie jest bezpieczna (nie polecam używać std::rand do jakiejkolwiek kryptografii). Przykładem jest choćby to, w jaki sposób musiałem zaimplementować losowanie liczb w zakresie dla liczb całkowitych

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

oraz rzeczywistych

```
min + static_cast<double>(std::rand()) / static_cast<double>(RAND_MAX / (max -
min));
```

Więcej na temat niewygodności i niebezpieczności std::rand można usłyszeć tutaj.

A z drugiej strony mamy do dyspozycji bibliotekę <random> i kilka różnych silników oraz dystrybucji do zabawy. Na przykład std::minstd_rand, które jest prawie 2 razy szybsze od std::rand (ale nie jest bezpieczne!). Jeśli algorytm liniowego generatora z jakichś powodów nam nie pasuje, mamy do dyspozycji silnik std::mt19937 oparty o algorytm Mersenne Twister, który dostarcza wysokiej jakości liczby pseudolosowe w dość szybkim czasie - jak widać, dla liczb 32-bitowych jest o 1/3 szybszy od std::rand (a o liczbach 64-bitowych nie ma co mówić, bo std::rand generuje zazwyczaj maksymalnie liczby 32-bitowe).

A jak się używa takiego silnika? Ano w ten sposób, na przykładzie std::mt19937 i jednolitej dystrybucji liczb całkowitych z użyciem std::uniform_int_distribution:

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

int main() {
    // Najpierw pobierzmy sobie jakiś seed, na przykład aktualny czas, z użyciem biblioteki <chrono>. Możemy też użyć std::random_device, ale na niektórych systemach zauważyłem że nie działa prawidłowo.
    unsigned seed = static_cast<unsigned>(std::chrono::high_resolution_clock::now().time_since_epoch().count());

    // Następnie, inicjalizujemy silnik
    std::mt19937 engine(seed);
    // Oraz nasz generator. W naszym przykładzie będziemy losować liczby z zakresu [1, 100]
    std::uniform_int_distribution<int> gen(1, 100);

    // I teraz możemy sobie coś wylosować
    std::cout << "Wylosowalem " << gen(engine) << std::endl;
}
```

Proste? Proste. Czytelne. Bezpieczniejsze i szybsze. Jedyne co może nas zniechęcić to to magiczne generowanie seeda - można tutaj użyć na przykład std::time(0), ale wolałem trzymać sie bibliotek C++owych.

Projekt benchmarku dostępny jest tutaj. Przed użyciem należy pobrać i skompilować bibliotekę do benchmarków Google.