Centrum Wiskunde & Informatica

C++: `<random>`

WISM454 Laboratory Class Scientific Computing, Jan-Willem Buurlage

April 1, 2019

The `<random>` standard library

- There is a standard library for generating random numbers, `<random>`
- It is enlightning to compare the design of that library, with the library we have been developing in this course.
- Today, I will give an overview of this library.

High-level overview

- Just as in our RNG library, there are two main components:
    - Engines, for generating random integer sequences uniformly at random
    - Distributions, for transforming these integer sequences into statistical distributions.
- In addition, the standard library has support for
    1. non-deterministic RNGs (hardware entropy source)
    2. engine adaptors.

RNG engines

- There are three engines available
  - `linear_congruential_engine`
  - `mersenne_twister_engine`
  - `subtract_with_carry_engine`
- These are templates, i.e. the parameters for these engines are taken as compile-time arguments.

linear_congruential_engine

```
template<class T, T a, T c, T m>
class linear_congruential_engine;
```

- Here, `T` is some unsigned integer type, and `a`, `c`, and `m` are compile time constants of this types that are the parameters for the LCRNG.

```
auto engine =
  std::linear_congruential_engine<uint32_t,5, 3, 11>();
engine.seed(1);
std::cout << engine() << "\n"; // 5 * 1 + 3 (mod 11) = 8
std::cout << engine() << "\n"; // 5 * 8 + 3 (mod 11) = 10
...
```

Predefined RNGs

- One of the benefits of using compile time arguments, is that parameter choices define a type rather than an instance.

```
using minstd_rand0 = std::linear_congruential_engine<
    std::uint_fast32_t, 16807, 0, 2147483647>;
using minstd_rand = std::linear_congruential_engine<
    std::uint_fast32_t, 48271, 0, 2147483647>;
// Park--Miller (and variant)

using mt19937_64 = std::mersenne_twister_engine<
                          std::uint_fast64_t,
                          64, 312, 156, 31,
                          0xb5026f5aa96619e9, 29,
                          0x5555555555555555, 17,
                          0x71d67fffeda60000, 37,
                          0xfff7eee000000000, 43,
                          6364136223846793005>;
```

Predefined RNGs (cont.)

```
auto engine = std::minstd_rand();
engine.seed(12345);
// or.. std::minstd_rand(12345);

f(engine());
```

- Note that this is an alternative (for predefining RNGs) to subclassing (inheriting from `lcrng` base class).

RNG distributions

- Many distributions available
    - `uniform_int_distribution`
    - `uniform_real_distribution`
    - `bernoulli_distribution`
    - `binomial_distribution`
    - ...
- The distributions are constructed independent from an engine, but engines are passed when sampling the distribution.

uniform_int_distribution

- For example, let us look at a uniform integer distribution.

```
template< class I = int >
class uniform_int_distribution;

uniform_int_distribution(I a = 0,
    I b = std::numeric_limits<I>::max());

template<typename Generator>
I operator()(Generator& g);
```

Engine + distribution

```
auto engine = std::minstd_rand(12345);
auto distribution = std::uniform_int_distribution(0, 10);
std::cout << distribution(engine) << "\n";
...
```

random_device

- There is a special 'engine' available, random_device.
- This uses a 'hardware entropy source' (if available) to generate random numbers. Not reproducible, and slow, but is often used for seeding a PRNG.

```
auto rd = std::random_device();
auto engine = std::mt19937(rd());
// ... seed MT with 'true random' number
```

RNG adaptors

- Available adaptors:
  - `discard_block_engine`: discards some output
  - `independent_bits_engine`: packs output into blocks
  - `shuffle_order_engine`: shuffle output
- These are independent of the engines that they 'adapt'

Conclusion

- Note that we did not talk about any implementation! Only about the design of the

standard libray.

- The design of `<random>` is much like ours:
  - Parameters chosen at runtime or compile time, and therefore predefined engines through specialization or type aliases.
  - Both libraries keep concepts such as distributions and engines independent.
- Because `<random>` is heavily templated and puts few restrictions on the 'links', it is more difficult to construct in a 'type safe' way, this may be fixed with concepts, see for example:

```
template<typename Generator>
I distribution::operator()(Generator& g);
```

Overview of `<random>`

- There are some components of `<random>` that we did not discuss, such as seed sequences.
- See: `http://en.cppreference.com/w/cpp/numeric/random` for a complete overview.