8.15 — Global random numbers (Random.h)

ALEX JANUARY 22, 2024

What happens if we want to use a random number generator in multiple functions or files? One way is to create (and seed) our PRNG in our `main()` function, and then pass it everywhere we need it. But that's a lot of passing for something we may only use sporadically, and in many different places. It would add a lot of clutter to our code to pass such an object around.

Alternately, you could create a static local `std::mt19937` variable in each function that needs it (static so that it only gets seeded once). However, it's overkill to have every function that uses a random number generator define and seed its own local generator, and the low volume of calls to each generator may lead to lower quality results.

What we really want is a single PRNG object that we can share and access anywhere, across all of our functions and files. The best option here is to create a global random number generator object (inside a namespace!). Remember how we told you to avoid non-const global variables? This is an exception.

Here's a simple, header-only solution that you can #include in any code file that needs access to a randomized, self-seeded `std::mt19937`:

Random.h:

```
#ifndef RANDOM_MT_H
#define RANDOM_MT_H

#include <chrono>
#include <random>

// This header-only Random namespace implements a self-seeding Mersenne Twister
// It can be included into as many code files as needed (The inline keyword avoids ODR violations)
// Freely redistributable, courtesy of learncpp.com (https://www.learncpp.com/cpp-tutorial/global-random-numbers-random-h/)
namespace Random
{
    // Returns a seeded Mersenne Twister
    // Note: we'd prefer to return a std::seed_seq (to initialize a std::mt19937), but std::seed can't be copied, so it
can't be returned by value.
    // Instead, we'll create a std::mt19937, seed it, and then return the std::mt19937 (which can be copied).
    inline std::mt19937 generate()
    {
        std::random_device rd{};

        // Create seed_seq with clock and 7 random numbers from std::random_device
        std::seed_seq ss{
            static_cast<std::seed_seq::result_type>(std::chrono::steady_clock::now().time_since_epoch().count()),
                rd(), rd(), rd(), rd(), rd(), rd(), rd() };

        return std::mt19937{ ss };
    }

    // Here's our global std::mt19937 object.
    // The inline keyword means we only have one global instance for our whole program.
    inline std::mt19937 mt{ generate() }; // generates a seeded std::mt19937 and copies it into our global object

    // Generate a random int between [min, max] (inclusive)
    inline int get(int min, int max)
    {
        return std::uniform_int_distribution{min, max}(mt);
    }

    // The following function templates can be used to generate random numbers
    // when min and/or max are not type int
    // See https://www.learncpp.com/cpp-tutorial/function-template-instantiation/
    // You can ignore these if you don't understand them

    // Generate a random value between [min, max] (inclusive)
    // * min and max have same type
    // * Return value has same type as min and max
    // * Supported types:
    // *    short, int, long, long long
    // *    unsigned short, unsigned int, unsigned long, or unsigned long long
    // Sample call: Random::get(1L, 6L);             // returns long
    // Sample call: Random::get(1u, 6u);             // returns unsigned int
    template <typename T>
    T get(T min, T max)
    {
        return std::uniform_int_distribution<T>{min, max}(mt);
    }

    // Generate a random value between [min, max] (inclusive)
    // * min and max can have different types
    // * Must explicitly specify return type as template type argument
    // * min and max will be converted to the return type
    // Sample call: Random::get<std::size_t>(0, 6);  // returns std::size_t
    // Sample call: Random::get<std::size_t>(0, 6u); // returns std::size_t
    // Sample call: Random::get<std::int>(0, 6u);    // returns int
    template <typename R, typename S, typename T>
    R get(S min, T max)
    {
        return get<R>(static_cast<R>(min), static_cast<R>(max));
    }
}

#endif
```

And a sample program showing how it is used:

main.cpp:

```cpp
#include "Random.h" // defines Random::mt, Random::get(), and Random::generate()
#include <iostream>

int main()
{
    // We can use Random::get() to generate random numbers

    std::cout << Random::get(1, 6) << '\n';   // returns int between 1 and 6
    std::cout << Random::get(1u, 6u) << '\n'; // returns unsigned int between 1 and 6

    // The following uses a template type argument
    // See https://www.learncpp.com/cpp-tutorial/function-template-instantiation/
    std::cout << Random::get<std::size_t>(1, 6u) << '\n'; // returns std::size_t between 1 and 6

    // We can access Random::mt directly if we have our own distribution

    // Create a reusable random number generator that generates uniform numbers between 1 and 6
    std::uniform_int_distribution die6{ 1, 6 }; // for C++14, use std::uniform_int_distribution<> die6{ 1, 6 };

    // Print a bunch of random numbers
    for (int count{ 1 }; count <= 10; ++count)
    {
        // We can also directly access Random::mt
        std::cout << die6(Random::mt) << '\t'; // generate a roll of the die here
    }

    std::cout << '\n';

    return 0;
}
```

Normally, defining variables and functions in a header file would cause violations of the one-definition rule (ODR) when that header file was included into more than one source file. However, we've made our `mt` variable and supporting functions `inline`, which allows us to have duplicate definitions without violating the ODR so long as those definitions are all identical. Because we're #including those definitions from a header file (rather than typing them manually, or copy/pasting them), we can ensure they are identical. Inline functions and variables were added to the language largely to make doing this kind of header-only functionality possible.

The other challenge that we have to overcome is in how we initialize our global `Random::mt` object, as we want it to be self-seeding so that we don't have to remember to explicitly call an initialization function for it to work correctly. Our initializer must be an expression. But in order to initialize a `std::mt19937`, we need several helper objects (a `std::random_device` and a `std::seed_seq`) which must be defined as statements. This is where a helper function comes in handy. A function call is an expression, so we can use the return value of a function as an initializer. And inside the function itself, we can have any combination of statements that we need. Thus, our `generate()` function creates and returns a fully-seeded `std::mt19937` object (seeded using both the system clock and `std::random_device`) that we use as the initializer to our global `Random::mt` object.

Once "Random.h" has been included, we can use it in one of two ways:

- We can call `Random::get()` to generate a random number between two values (inclusive).
- We can access the `std::mt19937` object directly via `Random::mt` and do whatever we want with it.

Next lesson
8.x Chapter 8 summary and quiz

Back to table of contents

Previous lesson
8.14 Generating random numbers using Mersenne Twister