effolkronium/random

笔记本:	C++		
创建时间:	2018/11/19 15:17	**更新时间:**	2018/11/19 15:21
URL:	https://github.com/effolkronium/random		

README.md

Random for modern C++ with convenient API

Design goals

There are few ways to get working with random in C++:

- **C style**

```
srand( time(NULL) ); // seed with time since epoch
auto random_number = rand() % (9 - 1)) + 1; // get a pseudo-random integer between 1 and 9
```

- Problems
  - should specify seed
  - should write your own distribution algorihtm
  - There are no guarantees as to the quality of the random sequence produced.
- **C++11 style**

```
std::random_device random_device; // create object for seeding
std::mt19937 engine{random_device()}; // create engine and seed it
std::uniform_int_distribution<> dist(1,9); // create distribution for integers with [1; 9] range
auto random_number = dist(engine); // finally get a pseudo-randomrandom integer number
```

- Problems
  - should specify seed
  - should choose, create and use a chain of various objects like engines and distributions
  - mt19937 use 5000 bytes of memory for each creation (which is bad for performance if we create it too frequently)
  - uncomfortable and not intuitively clear usage
- **effolkronium random style**

```
// auto seeded
auto random_number = Random::get(1, 9); // invoke 'get' method to generate a pseudo-random integer in [1; 9] range
// yep, that's all.
```

- Advantages
  - **Intuitive syntax**. You can do almost everything with random by simple 'get' method, like getting simple numbers, bools, random object from given set or using custom distribution.
  - **Trivial integration**. All code consists of a single header file `random.hpp`. That's it. No library, no subproject, no dependencies, no complex build system. The class is written in vanilla C++11. All in all, everything should require no adjustment of your compiler flags or project settings.
  - **Usability**. There are 3 versions of random:
    - *random_static* which has static methods and static internal state. It's not thread safe but more efficient
    - *random_thread_local* which has static methods and thread_local internal state. It's thread safe but less efficient
    - *random_local* which has non static methods and local internal state. It can be created on the stack at local scope

Supported compilers

- GCC 4.9 - 7.0 (and possibly later)
- Clang 3.7 - 4.0 (and possibly later)
- Microsoft Visual C++ 2015
- Microsoft Visual C++ 2017

Integration

CMake

- As subproject

```
add_subdirectory(random) # path to the 'random' library root
... # create target
target_link_libraries(${TARGET} effolkronium_random) # add include path to a compiler
```

- As external project

First of all, build or|and install this project:

```
cd "path_to_root_of_the_library"
mkdir build
cd build
cmake -G"Visual Studio 15 2017" ..
cmake --build . --target install --config Release
ctest -C Release
```

Then, find the package by a cmake

```
find_package(effolkronium_random REQUIRED)
... # create target
target_link_libraries(${TARGET} effolkronium_random)
```

Manually

The single required source, file `random.hpp` is in the `include/effolkronium` directory.

Then

All you need to do is add

```
#include "effolkronium/random.hpp"

// get base random alias which is auto seeded and has static API and internal state
using Random = effolkronium::random_static;
```

to the files you want to use effolkronium random class. That's it. Do not forget to set the necessary switches to enable C++11 (e.g., `-std=c++11` for GCC and Clang).

Five-minute tutorial

Number range

Returns a pseudo-random number in a [first; second] range.

```
auto val = Random::get(-1, 1) // decltype(val) is int
```

```
// specify explicit type
auto val = Random::get<uint8_t>(-1, 1) // decltype(val) is uint8_t
```

```
// you able to use range from greater to lower
auto val = Random::get(1.l, -1.l) // decltype(val) is long double
```

```
auto val = Random::get(1.f, -1) // Error: implicit conversions are not allowed here.
```

Common type number range

Choose common type of two range arguments by std::common_type.

```
auto val = Random::get<Random::common>(1, 0.f) // decltype(val) is float
```

```
auto val = Random::get<Random::common>(0ul, 1ull) // decltype(val) is unsigned long long
```

```
auto val = Random::get<Random::common>(1.2l, 1.5f) // decltype(val) is long double
```

```
auto val = Random::get<Random::common>(1u, -1) // Error: prevent conversion from signed to unsigned.
```

Character range

Returns a pseudo-random character in a [first; second] range.

```
auto val = Random::get('a', 'z')
```

```
auto val = Random::get(L'㉠', L'㉣')
```

```
auto val = Random::get<wchar_t>()
```

Bool

Generate true with [0; 1] probability

```
auto val = Random::get<bool>(0.7) // true with 70% probability
```

```
auto val = Random::get<bool>() // true with 50% probability by default
```

```
auto val = Random::get<bool>(-1) // Error: assert occurred! Out of [0; 1] range
```

Random value from std::initilizer_list

Return random value from values in a std::initilizer_list

```
auto val = Random::get({1, 2, 3}) // val = 1 or 2 or 3
```

Random iterator

Return random iterator from iterator range or container. Iterator must be at least Input iterator. If a std::distance(first, last) == 0, return the 'last' iterator. If container is empty, return std::end(container) iterator.

```
std::array<int, 3> array{ {1, 2, 3} };
```

- Iterator range

```
auto randomIt = Random::get( array.begin(), array.end() );
```

- Container

```
auto randomIt = Random::get( array );
```

Random element from array

Return pointer to random element in built-in array

```
int array [] = {1, 2, 3};
auto randomPtr = Random::get( array );
```

Shuffle

Reorders the elements in a given range or in all container ref

```
std::array<int, 3> array{ {1, 2, 3} };
```

- Iterator range

```
Random::shuffle( array.begin( ), array.end( ) )
```

- Container

```
Random::shuffle( array )
```

Custom distribution

Return result from operator() of a distribution with internal random engine argument

- Template argument

```
// 1.f and 2.f will be forwarded to std::gamma_distribution constructor
auto result = Random::get<std::gamma_distribution<>>( 1.f, 2.f );
```

- Argument by reference

```
std::gamma_distribution<> gamma{ 1.f, 2.f };
auto result = Random DOT get( gamma ); // return result of gamma.operator()( engine_ )
```

Custom Seeder

Specify seed by which random engine will be seeded at construction time:

- Number

```
struct MySeeder {
unsigned operator() () {
return 42u;
}
};
// Seeded by 42
using Random = effolkronium::basic_random_static<std::mt19937, MySeeder>;
```

- Seed sequence

Because we can't copy std::seed_seq, the 'random' library destroy seeder instance after engine seeding. So it's safe to return seed by reference.

```
struct MySeeder {
// std::seed_seq isn't copyable
std::seed_seq& operator() () {
return seed_seq_;
}
std::seed_seq seed_seq_{ { 1, 2, 3, 4, 5 } };
};
```

```
// Seeded by seed_seq_ from MySeeder
using Random = effolkronium::basic_random_static<std::mt19937, MySeeder>;
```

- Reseed

Seed an internal random engine by a newly created Seeder instance

```
Random::reseed( );
```

Thread local random

It uses static methods API and data with thread_local storage which is fully **thread safe** (but less perfomance)

```
using Random = effolkronium::random_thread_local

// use in the same way as random_static. Thread safe
std::thread first{ [ ] { Random::get( ); } };
std::thread second{ [ ] { Random::get( ); } };
```

Local random

It uses non static methods API and data with auto storage which can be created on the stack at local scope

```
#include "effolkronium/random.hpp"

using Random_t = effolkronium::random_local

int main( ) {
Random_t localRandom{ }; // Construct on the stack
// access throughout dot
auto val = localRandom.get(-10, 10);
} // Destroy localRandom and free stack memory
```

Seeding

ref

Set new seed for an internal random engine.

```
Random::seed( 10 ); // 10 is new seed number

std::seed_seq sseq{ 1, 2, 3 };
Random::seed( sseq ); // use seed sequence here
```

Min value

ref

Returns the minimum value potentially generated by the internal random-number engine

```
auto minVal = Random::min( );
```

Max value

ref

Returns the maximum value potentially generated by the internal random-number engine

```
auto maxVal = Random::max( );
```

get without arguments

ref

Returns the random number in [Random::min( ), Random::max] range

```
auto val = Random::get( );
// val is random number in [ Random::min( ), Random::max ] range
```

Get engine

Returns copy of internal engine.

```
auto engine = Random::get_engine( );
```

Discard

ref

Advances the internal engine's state by a specified amount. Equivalent to calling Random::get() N times and discarding the result.

```
Random::discard( 500 );
```

IsEqual

ref

Compares internal pseudo-random number engine with other pseudo-random number engine.

```
Random::Engine otherEngine;
bool isSame = Random::is_equal( otherEngine );
```

Serialize

ref

Serializes the internal state of the internal pseudo-random number engine as a sequence of decimal numbers separated by one or more spaces, and inserts it to the output stream. The fill character and the formatting flags of the stream are ignored and unaffected.

```
std::stringstream strStream;
Random::serialize( strStream ); // the strStream now contain internal state of the Random internal engine
```

Deserialize

ref

Restores the internal state of the internal pseudo-random number engine from the serialized representation, which was created by an earlier call to 'serialize' using a stream with the same imbued locale and the same CharT and Traits. If the input cannot be deserialized, internal engine is left unchanged and failbit is raised on input stream.

```
std::stringstream strStream;
```

```
Random::serialize( strStream );

// ...

Random::deserialize( strStream ); // Restore internal state of internal Random engine
```