

CS 202 Group Project

The Denali Trail

Rebecca Morgan
Millard Arnold V

March 9, 2020

Source Code Link:

<https://github.com/imthegngrbrdmn/cs-202-DenailTrail.git>

Git Commit Messages

<https://github.com/imthegngrbrdmn/cs-202-DenailTrail/commits/master>

This project took approximately 02 hours to complete.

1 Pitch

The Oregon Trail was a popular game in its time and is still relevant today. Many people understand the multitude of references to it even if they have never played the game. People understand that it was common to die of dysentery and this has even evolved into memes about dying from dissin' Terry. Here in Alaska, we have the Denali mountain range, so Rebecca & I thought it would be interesting to riff off of the Oregon Trail and make the Denali Trail. On the Denali trail, the player will hike up the mountain while facing obstacles and attempting to survive by eating, drinking, and finding a way to stay warm and rest. Only the strongest players will be successful in completing the Denali trail and surviving the great journey.

2 Project Iteration 2: Design

2.1 Overall Design

The design of our program is to be easily played by anyone and it will have a clean appearance that makes it clear what is going on and how the player should proceed.

2.2 Prior Art

This game is greatly inspired by the Oregon Trail, but it will be visually designed around Denali.

2.3 Technical Design

We plan on implementing the SIGIL Library in order to have pictures which will add some interesting features and make the player more interested in the action of the game. The game will have a struct for the player in order to keep track of their thirst, hunger, health, and strength. We will have a header for the generic way that each "room" will be composed. We will have a vector for options when things need options, and we will store inventory using bitwise operations on an int treating each bit as a specific item, and this will all be handled through its own file.

2.4 Required Libraries

We are implementing the SIGIL Library in order to add pictures into the game to increase the immersion and ability for the player to visualise their circumstance in-game.

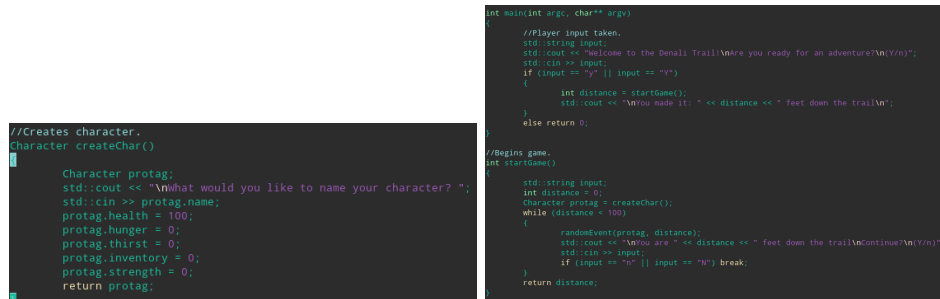


Figure 1: Millard Artifacts

3 Project Iteration 3: Initial Prototype

3.1 Contribution: Millard A. Arnold V

I was charged with creating a functional template and implementing the Random library. I started with a function to handle the game itself so that the main could simply handle the beginning where the user is asked if they want to start the game, and the end where the user is given their result. I had the randomEvent function handle each event while startGame handled the time in between these events. The randomEvent function is where we use the shuffle function from the Random library. We will be incorporating more from this library at a later time as we continue to approach a final product. I also made the createChar function to allow the player to customize their character. We will be improving upon this later.

3.2 Contribution: Rebecca Morgan

I was tasked with adding numerous events to the random event generator. I then put in the details of what would happen in each event such as health increases or decreases, hunger and thirst variations etc. I then went through all the code and added comments to describe what was going on in each part.

FoundError.png

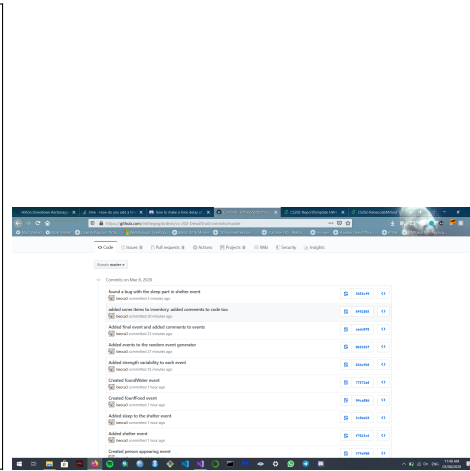


Figure 2: Rebecca Artifacts

3.3 Git Commit Messages

Date	Message
2020-02-06	Initial commit
2020-02-27	Set up Visual Studio Solution
2020-02-27	Initial Main
2020-02-27	Beginning and bare minimum of page design
2020-03-06	Testing SIGIL
2020-03-06	Some small changes. Still figuring out SIGIL
2020-03-06	Added sl.h
2020-03-06	Using random as lib not SIGIL
2020-03-06	Visual Studio Stuff
2020-03-06	Created Events and handles them well.
2020-03-08	Added found weapon event
2020-03-08	Created illness event
2020-03-08	Added wolf appearing event
2020-03-08	Created person appearing event
2020-03-08	Added shelter event
2020-03-08	Added sleep to the shelter event
2020-03-08	Created foundFood event
2020-03-08	Created foundWater event
2020-03-08	Added strength variability to each event
2020-03-08	Added events to the random event generator
2020-03-08	Added final event and added comments to events
2020-03-08	added some items to inventory. added comments to code too
2020-03-08	found a bug with the sleep part in shelter event
2020-03-08	Fixed formatting errors and removed some problems.

4 Project Iteration 4

Same as previous section...Write about the goals achieved for your program and the next goals you set for the next iteration.

- 4.1 Member 1 comments**
- 4.2 Member 2 comments**
- 4.3 Git Commit Messages**

5 Project Iteration 5

Same as previous section...Write about the goals achieved for your program and the next goals you set for the next iteration.

- 5.1 Member 1 comments**
- 5.2 Member 2 comments**
- 5.3 Git Commit Messages**

6 Project Iteration 5: The Shipping Project

This is the last iteration for development of your project before. Write about the goals each of you achieved for your finished program.

- 6.1 Member 1 comments**
- 6.2 Member 2 comments**
- 6.3 Git Commit Messages**

7 Project Iteration 6: Results & Post Mortem

7.1 Results

A screenshot of our program is displayed in Figure 3. In this section, talk about the results of your program.

7.2 Post Mortem

In this section, you will write a paragraph in about 100 words about what went right and what went wrong for your implementation. What lessons were learned or best practices identified?

7.3 Sample Output

In this section, write about the sample output of your program. For example, Figure 3 is a screenshot of a ROBOT parameter visualization screen in our program.

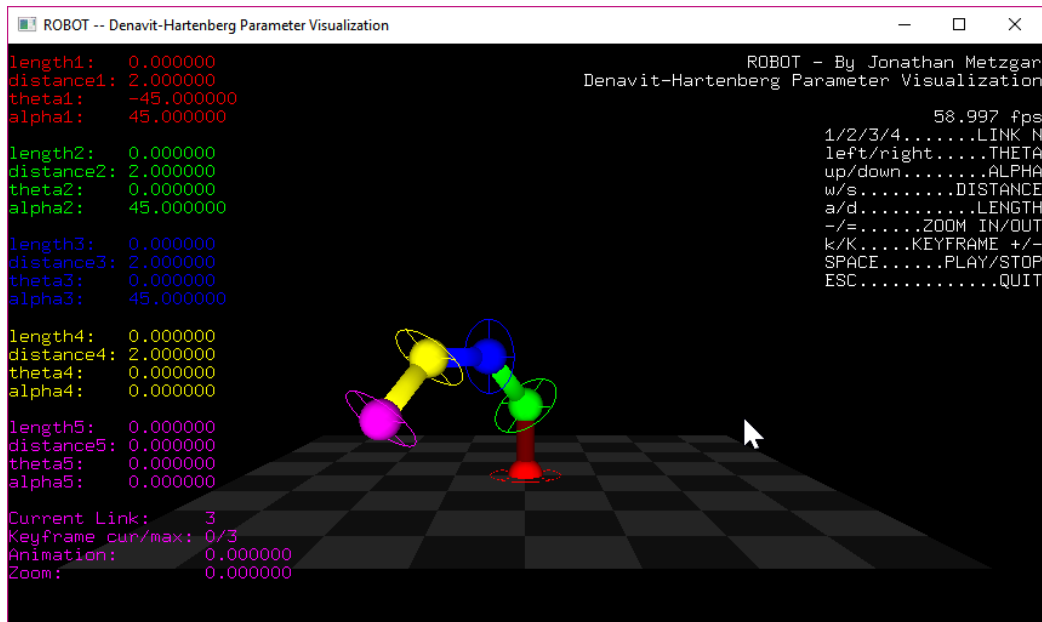


Figure 3: A Screenshot of our finished application.

7.4 Program Source Code

The source code is licensed under the WRITETHENAMEOFY-OURLICENSEHERE license.

7.5 page design header

```
1 #ifndef PAGE_DESIGN_H
2 #define PAGE_DESIGN_H
3
4 #include "random.hpp"
5 #include <string>
6
7 //Struct to hold player variables.
8 struct Character
9 {
10     std::string name;
11     int thirst;
12     int hunger;
13     int strength;
14     int health;
15     long long inventory;
16 };
17
18 void randomEvent(Character& protag, int&distance);
19 #endif
```

7.6 page design Source

```
1 #include "page_design.h"
2 #include <vector>
3 #include <iostream>
4 #include <ctime>
5
6 using Random=effolkronium::random_static;
7
8 //Bear attacks player.
9 void bear(Character& protag, int& distance)
10 {
11     system("CLS");
12     std::cout << "You have been attacked by a bear.\t";
13     protag.health -= 10;
14     protag.strength -= 10;
15     distance += 5;
16 }
17
18 //Player gets lost.
19 void lost(Character& protag, int& distance)
20 {
21     system("CLS");
22     std::cout << "You get lost for a while and then eventually find your way.";
23     protag.hunger += 10;
24     protag.thirst += 10;
25     protag.strength -= 5;
```



```

26     distance += 1;
27 }
28
29 //Player finds a knife along the route.
30 void foundKnife(Character& protag, int& distance)
31 {
32     system("CLS");
33     std::cout << "You found a Knife.";
34     protag.health += 5;
35     protag.inventory += 1;
36     distance += 1;
37 }
38
39 //Player falls ill.
40 void illness(Character& protag, int& distance)
41 {
42     system("CLS");
43     std::cout << "Developing symptoms of illness, maybe an infection.";
44     protag.health -= 5;
45     protag.strength -= 10;
46     distance += 1;
47 }
48
49 //Player attacked by wolf.
50 void wolf(Character& protag, int& distance)
51 {
52     system("CLS");
53     std::cout << "A wolf appears, you run as fast as you can to escape!";
54     protag.health -= 10;
55     protag.hunger += 5;
56     protag.thirst += 10;
57     protag.strength -= 10;
58     distance += 5;
59 }
60
61 //Player runs across another person.
62 void person(Character& protag, int& distance)
63 {
64     system("CLS");
65     std::cout << "A person appears. May or may not be a friendly. You run just incase.
66     protag.thirst += 5;
67     distance += 5;
68 }
69
70 //Player comes across a shelter.
71 void shelter(Character& protag, int& distance)
72 {
73     system("CLS");
74     std::cout << "You come across a shelter. Let's rest.";
75     sleep(2);
76     protag.health += 20;
77     protag.strength += 20;

```

```

78     distance += 0;
79 }
80
81 //Player finds some food.
82 void foundFood(Character& protag, int& distance)
83 {
84     system("CLS");
85     std::cout << "You found a cache of food.";
86     protag.hunger -= 10;
87     protag.health += 5;
88     protag.strength += 10;
89     protag.inventory += 3;
90     distance += 1;
91 }
92
93 //Player finds some water.
94 void foundWater(Character& protag, int& distance)
95 {
96     system("CLS");
97     std::cout << "You found a source of water.";
98     protag.thirst -= 10;
99     protag.health += 5;
100    protag.strength += 10;
101    protag.inventory += 2;
102    distance += 1;
103 }
104
105 //Player eats a poison berry.
106 void poison(Character& protag, int& distance)
107 {
108     system("CLS");
109     std::cout << "You found a berry and ate it. It was poisonous.";
110     protag.hunger += 5;
111     protag.health -= 5;
112     protag.strength -= 5;
113     distance += 1;
114 }
115
116 //Places all events into a vector and generates one randomly.
117 void randomEvent(Character& protag, int& distance)
118 {
119     std::vector<std::string> events;
120     events.push_back("bear");
121     events.push_back("lost");
122     events.push_back("foundKnife");
123     events.push_back("illness");
124     events.push_back("wolf");
125     events.push_back("person");
126     events.push_back("shelter");
127     events.push_back("foundFood");
128     events.push_back("foundwater");

```

```

129 events.push_back("poison");
130
131 //Shuffles events and chooses one at random.
132 Random::shuffle(events);
133 if (events[0] == "bear")
134 {
135     bear(protag, distance);
136 }
137 else if (events[1] == "lost")
138 {
139     lost(protag, distance);
140 }
141 else if (events[2] == "foundKnife")
142 {
143     foundKnife(protag, distance);
144 }
145 else if (events[3] == "illness")
146 {
147     illness(protag, distance);
148 }
149 else if (events[4] == "wolf")
150 {
151     wolf(protag, distance);
152 }
153 else if (events[5] == "person")
154 {
155     person(protag, distance);
156 }
157 else if (events[6] == "shelter")
158 {
159     shelter(protag, distance);
160 }
161 else if (events[7] == "foundFood")
162 {
163     foundFood(protag, distance);
164 }
165 else if (events[8] == "foundWater")
166 {
167     foundWater(protag, distance);
168 }
169 else if (events[9] == "poison")
170 {
171     poison(protag, distance);
172 }
173 }

```

7.7 random library header

```
1  /*
2
3  [RANDOM] Random for modern C++
4  [RANDOM] version 1.3.1
5  [RANDOM] https://github.com/effolkronium/random
6
7  Licensed under the MIT License <http://opensource.org/licenses/MIT>.
8  Copyright (c) 2019 effolkronium
9
10 Permission is hereby granted, free of charge, to any person obtaining a copy
11 of this software and associated documentation files( the "Software" ), to deal
12 in the Software without restriction, including without limitation the rights
13 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
14 copies of the Software, and to permit persons to whom the Software is
15 furnished to do so, subject to the following conditions :
16
17 The above copyright notice and this permission notice shall be included in all
18 copies or substantial portions of the Software.
19
20 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
21 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
22 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
23 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
24 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
25 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
26 SOFTWARE.
27 */
28
29 #ifndef EFFOLKRONIUM_RANDOM_HPP
30 #define EFFOLKRONIUM_RANDOM_HPP
31
32 #include <random>
33 #include <chrono> // timed seed
34 #include <type_traits>
35 #include <cassert>
36 #include <initializer_list>
37 #include <utility> // std::forward, std::declval
38 #include <algorithm> // std::shuffle, std::next, std::distance
39 #include <iterator> // std::begin, std::end, std::iterator_traits
40 #include <limits> // std::numeric_limits
41 #include <ostream>
42 #include <istream>
43
44 namespace effolkronium {
45     namespace details {
46         /// Key type for getting common type numbers or objects
47         struct common{ };
48
49         /// True if type T is applicable by a std::uniform_int_distribution
50         template<typename T>
```

```

54 struct is_uniform_int {
55     static constexpr bool value =
56         std::is_same<T, short>::value
57         || std::is_same<T, int>::value
58         || std::is_same<T, long>::value
59         || std::is_same<T, long long>::value
60         || std::is_same<T, unsigned short>::value
61         || std::is_same<T, unsigned int>::value
62         || std::is_same<T, unsigned long>::value
63         || std::is_same<T, unsigned long long>::value;
64 };
65
66 /// True if type T is applicable by a std::uniform_real_distribution
67 template<typename T>
68 struct is_uniform_real {
69     static constexpr bool value =
70         std::is_same<T, float>::value
71         || std::is_same<T, double>::value
72         || std::is_same<T, long double>::value;
73 };
74
75 /// True if type T is plain byte
76 template<typename T>
77 struct is_byte {
78     static constexpr bool value =
79         std::is_same<T, signed char>::value
80         || std::is_same<T, unsigned char>::value;
81 };
82
83 /// True if type T is plain number type
84 template<typename T>
85 struct is_supported_number {
86     static constexpr bool value =
87         is_byte<T>::value
88         || is_uniform_real<T>::value
89         || is_uniform_int<T>::value;
90 };
91
92 /// True if type T is character type
93 template<typename T>
94 struct is_supported_character {
95     static constexpr bool value =
96         std::is_same<T, char>::value
97         || std::is_same<T, wchar_t>::value
98         || std::is_same<T, char16_t>::value
99         || std::is_same<T, char32_t>::value;
100 };
101
102 /// True if type T is iterator
103 template<typename T>
104 struct is_iterator {
105 private:

```

```

106         static char test( ... );
107
108         template <typename U,
109                 typename = typename std::iterator_traits<U>::difference_type,
110                 typename = typename std::iterator_traits<U>::pointer,
111                 typename = typename std::iterator_traits<U>::reference,
112                 typename = typename std::iterator_traits<U>::value_type,
113                 typename = typename std::iterator_traits<U>::iterator_category
114         > static long test( U&& );
115     public:
116         static constexpr bool value = std::is_same<
117             decltype( test( std::declval<T>( ) ) ), long>::value;
118     };
119
120 } // namespace details
121
122 /// Default seeder for 'random' classes
123 struct seeder_default {
124     /// return seed sequence
125     std::seed_seq& operator() ( ) {
126         // MinGW issue, std::random_device returns constant value
127         // Use std::seed_seq with additional seed from C++ chrono
128         return seed_seq;
129     }
130 private:
131     std::seed_seq seed_seq{ {
132         static_cast<std::uintmax_t>( std::random_device{ }( ) ),
133         static_cast<std::uintmax_t>( std::chrono::steady_clock::now( )
134                                     .time_since_epoch( ).count( ) ),
135     } };
136 };
137
138 /**
139 * \brief Base template class for random
140 *         with static API and static internal member storage
141 * \note it is NOT thread safe but more efficient then
142 *         basic_random_thread_local
143 * \param Engine A random engine with interface like in the std::mt19937
144 * \param Seeder A seeder type which return seed for internal engine
145 *               through operator()
146 */
147 template<
148     typename Engine,
149     typename Seeder = seeder_default,
150     template<typename> class IntegerDist = std::uniform_int_distribution,
151     template<typename> class RealDist = std::uniform_real_distribution,
152     typename BoolDist = std::bernoulli_distribution
153 >
154 class basic_random_static {
155 public:
156     basic_random_static( ) = delete;
157
158     /// Type of used random number engine

```

```

159     using engine_type = Engine;
160
161     /// Type of used random number seeder
162     using seeder_type = Seeder;
163
164     /// Type of used integer distribution
165     template<typename T>
166     using integer_dist_t = IntegerDist<T>;
167
168     /// Type of used real distribution
169     template<typename T>
170     using real_dist_t = RealDist<T>;
171
172     /// Type of used bool distribution
173     using bool_dist_t = BoolDist;
174
175     /// Key type for getting common type numbers or objects
176     using common = details::common;
177
178     /**
179     * \return The minimum value
180     * potentially generated by the random-number engine
181     */
182     static constexpr typename Engine::result_type min( ) {
183         return Engine::min( );
184     }
185
186     /**
187     * \return The maximum value
188     * potentially generated by the random-number engine
189     */
190     static constexpr typename Engine::result_type max( ) {
191         return Engine::max( );
192     }
193
194     /// Advances the internal state by z times
195     static void discard( const unsigned long long z ) {
196         engine_instance( ).discard( z );
197     }
198
199     /// Reseed by Seeder
200     static void reseed( ) {
201         Seeder seeder;
202         seed( seeder( ) );
203     }
204
205     /**
206     * \brief Reinitializes the internal state
207     * of the random-number engine using new seed value
208     * \param value The seed value to use
209     * in the initialization of the internal state
210     */
211     static void seed( const typename Engine::result_type value =
212         Engine::default_seed ) {
213         engine_instance( ).seed( value );
214     }

```

```

215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268

/**
 * \brief Reinitializes the internal state
 * of the random-number engine using new seed value
 * \param seq The seed sequence
 * to use in the initialization of the internal state
 */
template<typename Sseq>
static void seed( Sseq& seq ) {
    engine_instance( ).seed( seq );
}

/// return random number from engine in [min(), max()] range
static typename Engine::result_type get( ) {
    return engine_instance( )( );
}

/**
 * \brief Compares internal pseudo-random number engine
 * with 'other' pseudo-random number engine.
 * Two engines are equal, if their internal states
 * are equivalent, that is, if they would generate
 * equivalent values for any number of calls of operator()
 * \param other The engine, with which the internal engine will be compared
 * \return true, if other and internal engine are equal
 */
static bool is_equal( const Engine& other ) {
    return engine_instance( ) == other;
}

/**
 * \brief 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 stream ost. The fill character
 * and the formatting flags of the stream are
 * ignored and unaffected.
 * \param ost The output stream to insert the data to
 */
template<typename CharT, typename Traits>
static void serialize( std::basic_ostream<CharT, Traits>& ost ) {
    ost << engine_instance( );
}

/**
 * \brief 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 ist
 * \param ost The input stream to extract the data from

```



```

269 */
270 template<typename CharT, typename Traits>
271 static void deserialize( std::basic_istream<CharT, Traits>& ist ) {
272     ist >> engine_instance( );
273 }
274
275 /**
276 * \brief Generate a random integer number in a [from; to] range
277 *         by std::uniform_int_distribution
278 * \param from The first limit number of a random range
279 * \param to The second limit number of a random range
280 * \return A random integer number in a [from; to] range
281 * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
282 * \note Prevent implicit type conversion
283 */
284 template<typename T>
285 static typename std::enable_if<details::is_uniform_int<T>::value
286     , T>::type get( T from = std::numeric_limits<T>::min( ),
287     T to = std::numeric_limits<T>::max( ) ) {
288     if( from < to ) // Allow range from higher to lower
289         return IntegerDist<T>{ from, to }( engine_instance( ) );
290     return IntegerDist<T>{ to, from }( engine_instance( ) );
291 }
292
293 /**
294 * \brief Generate a random real number in a [from; to] range
295 *         by std::uniform_real_distribution
296 * \param from The first limit number of a random range
297 * \param to The second limit number of a random range
298 * \return A random real number in a [from; to] range
299 * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
300 * \note Prevent implicit type conversion
301 */
302 template<typename T>
303 static typename std::enable_if<details::is_uniform_real<T>::value
304     , T>::type get( T from = std::numeric_limits<T>::min( ),
305     T to = std::numeric_limits<T>::max( ) ) {
306     if( from < to ) // Allow range from higher to lower
307         return RealDist<T>{ from, to }( engine_instance( ) );
308     return RealDist<T>{ to, from }( engine_instance( ) );
309 }
310
311 /**
312 * \brief Generate a random byte number in a [from; to] range
313 * \param from The first limit number of a random range
314 * \param to The second limit number of a random range
315 * \return A random byte number in a [from; to] range
316 * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
317 * \note Prevent implicit type conversion
318 */
319 template<typename T>
320 static typename std::enable_if<details::is_byte<T>::value

```

```

321         , T>::type get( T from = std::numeric_limits<T>::min( ),
322                        T to = std::numeric_limits<T>::max( ) ) {
323         // Choose between short and unsigned short for byte conversion
324         using short_t = typename std::conditional<std::is_signed<T>::value,
325            short, unsigned short>::type;
326
327         return static_cast<T>( get<short_t>( from, to ) );
328     }
329
330     /**
331     * \brief Generate a random common_type number in a [from; to] range
332     * \param Key The Key type for this version of 'get' method
333     *         Type should be '(THIS_TYPE)::common' struct
334     * \param from The first limit number of a random range
335     * \param to The second limit number of a random range
336     * \return A random common_type number in a [from; to] range
337     * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
338     * \note Allow implicit type conversion
339     * \note Prevent implicit type conversion from signed to unsigned types
340     *         Why? std::common_type<Unsigned, Signed> chooses unsigned value,
341     *         then Signed value will be converted to Unsigned value
342     *         which gives us a wrong range for random values.
343     *         https://stackoverflow.com/a/5416498/5734836
344     */
345     template<
346         typename Key,
347         typename A,
348         typename B,
349         typename C = typename std::common_type<A, B>::type
350     >
351     static typename std::enable_if<
352         std::is_same<Key, common>::value
353         && details::is_supported_number<A>::value
354         && details::is_supported_number<B>::value
355         // Prevent implicit type conversion from signed to unsigned types
356         && std::is_signed<A>::value != std::is_unsigned<B>::value
357         , C>::type get( A from = std::numeric_limits<A>::min( ),
358                        B to = std::numeric_limits<B>::max( ) ) {
359         return get( static_cast<C>( from ), static_cast<C>( to ) );
360     }
361
362     /**
363     * \brief Generate a random character in a [from; to] range
364     *         by std::uniform_int_distribution
365     * \param from The first limit number of a random range
366     * \param to The second limit number of a random range
367     * \return A random character in a [from; to] range
368     * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
369     * \note Prevent implicit type conversion
370     */
371     template<typename T>
372     static typename std::enable_if<details::is_supported_character<T>::value

```

```

373         , T>::type get(T from = std::numeric_limits<T>::min(),
374             T to = std::numeric_limits<T>::max()) {
375         if (from < to) // Allow range from higher to lower
376             return static_cast<T>(IntegerDist<std::int64_t>{ static_cast<std::int64_t>(from), static_cast<std::int64_t>(to) });
377         return static_cast<T>(IntegerDist<std::int64_t>{ static_cast<std::int64_t>(to), static_cast<std::int64_t>(from) });
378     }
379
380     /**
381     * \brief Generate a bool value with specific probability
382     *         by std::bernoulli_distribution
383     * \param probability The probability of generating true in [0; 1] range
384     *         0 means always false, 1 means always true
385     * \return 'true' with 'probability' probability ('false' otherwise)
386     */
387     template<typename T>
388     static typename std::enable_if<std::is_same<T, bool>::value
389         , bool>::type get( const double probability = 0.5 ) {
390         assert( 0 <= probability && 1 >= probability ); // out of [0; 1] range
391         return BoolDist{ probability }( engine_instance( ) );
392     }
393
394     /**
395     * \brief Return random value from initializer_list
396     * \param init_list initializer_list with values
397     * \return Random value from initializer_list
398     * \note Should be 1 or more elements in initializer_list
399     * \note Warning! Elements in initializer_list can't be moved:
400     *         https://stackoverflow.com/a/8193157/5734836
401     */
402     template<typename T>
403     static T get( std::initializer_list<T> init_list ) {
404         assert( 0u != init_list.size( ) );
405         return *get( init_list.begin( ), init_list.end( ) );
406     }
407
408     /**
409     * \brief Return random iterator from iterator range
410     * \param first, last - the range of elements
411     * \return Random iterator from [first, last) range
412     * \note If first == last, return last
413     */
414     template<typename InputIt>
415     static typename std::enable_if<details::is_iterator<InputIt>::value
416         , InputIt>::type get( InputIt first, InputIt last ) {
417         const auto size = std::distance( first, last );
418         if( 0 == size ) return last;
419         using diff_t = typename std::iterator_traits<InputIt>::difference_type;
420         return std::next( first, get<diff_t>( 0, size - 1 ) );
421     }
422
423     /**
424     * \brief Return random iterator from Container
425     * \param container The container with elements

```

```

426 * \return Random iterator from container
427 * \note If container is empty return std::end( container ) iterator
428 */
429 template<typename Container>
430 static auto get( Container& container ) ->
431     typename std::enable_if<details::is_iterator<
432         decltype(std::begin(container))>::value
433         , decltype(std::begin(container))
434     >::type {
435     return get( std::begin( container ), std::end( container ) );
436 }
437
438 /**
439 * \brief Return random pointer from built-in array
440 * \param array The built-in array with elements
441 * \return Pointer to random element in array
442 */
443 template<typename T, std::size_t N>
444 static T* get( T( &array )[ N ] ) {
445     return std::addressof( array[ get<std::size_t>( 0, N - 1 ) ] );
446 }
447
448 /**
449 * \brief Return value from custom Dist distribution
450 *     seeded by internal random engine
451 * \param Dist The type of custom distribution with next concept:
452 *     http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
453 * \param args The arguments which will be forwarded to Dist constructor
454 * \return Value from custom distribution
455 */
456 template<typename Dist, typename... Args>
457 static typename Dist::result_type get( Args&&... args ) {
458     return Dist{ std::forward<Args>( args )... }( engine_instance( ) );
459 }
460
461 /**
462 * \brief Return value from custom 'dist' distribution
463 *     seeded by internal random engine
464 * \param dist The custom distribution with next concept:
465 *     http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
466 * \param args The arguments which will be forwarded to Dist constructor
467 * \return Value from custom 'dist' distribution
468 */
469 template<typename Dist>
470 static typename Dist::result_type get( Dist& dist ) {
471     return dist( engine_instance( ) );
472 }
473
474 /**
475 * \brief Reorders the elements in the given range [first, last)
476 *     such that each possible permutation of those elements
477 *     has equal probability of appearance.
478 * \param first, last - the range of elements to shuffle randomly
479 */

```

```

480     template<typename RandomIt>
481     static void shuffle( RandomIt first, RandomIt last ) {
482         std::shuffle( first, last, engine_instance( ) );
483     }
484
485     /**
486     * \brief Reorders the elements in the given container
487     *         such that each possible permutation of those elements
488     *         has equal probability of appearance.
489     * \param container - the container with elements to shuffle randomly
490     */
491     template<typename Container>
492     static void shuffle( Container& container ) {
493         shuffle( std::begin( container ), std::end( container ) );
494     }
495
496     /// return internal engine by copy
497     static Engine get_engine( ) {
498         return engine_instance( );
499     }
500
501     /// return internal engine by ref
502     static Engine& engine( ) {
503         return engine_instance();
504     }
505 protected:
506     /// get reference to the static engine instance
507     static Engine& engine_instance( ) {
508         static Engine engine{ Seeder{ }( ) };
509         return engine;
510     }
511 };
512
513 /**
514 * \brief Base template class for random
515 *         with thread_local API and thread_local internal member storage
516 * \note it IS thread safe but less efficient then
517 *         basic_random_static
518 * \param Engine A random engine with interface like in the std::mt19937
519 * \param Seeder A seeder type which return seed for internal engine
520 *               through operator()
521 */
522 template<
523     typename Engine,
524     typename Seeder = seeder_default,
525     template<typename> class IntegerDist = std::uniform_int_distribution,
526     template<typename> class RealDist = std::uniform_real_distribution,
527     typename BoolDist = std::bernoulli_distribution
528 >
529 class basic_random_thread_local {
530 public:
531     basic_random_thread_local( ) = delete;
532

```

```

533     /// Type of used random number engine
534     using engine_type = Engine;
535
536     /// Type of used random number seeder
537     using seeder_type = Seeder;
538
539     /// Type of used integer distribution
540     template<typename T>
541     using integer_dist_t = IntegerDist<T>;
542
543     /// Type of used real distribution
544     template<typename T>
545     using real_dist_t = RealDist<T>;
546
547     /// Type of used bool distribution
548     using bool_dist_t = BoolDist;
549
550     /// Key type for getting common type numbers or objects
551     using common = details::common;
552
553     /**
554     * \return The minimum value
555     * potentially generated by the random-number engine
556     */
557     static constexpr typename Engine::result_type min( ) {
558         return Engine::min( );
559     }
560
561     /**
562     * \return The maximum value
563     * potentially generated by the random-number engine
564     */
565     static constexpr typename Engine::result_type max( ) {
566         return Engine::max( );
567     }
568
569     /// Advances the internal state by z times
570     static void discard( const unsigned long long z ) {
571         engine_instance( ).discard( z );
572     }
573
574     /// Reseed by Seeder
575     static void reseed( ) {
576         Seeder seeder;
577         seed( seeder( ) );
578     }
579
580     /**
581     * \brief Reinitializes the internal state
582     * of the random-number engine using new seed value
583     * \param value The seed value to use
584     * in the initialization of the internal state
585     */
586     static void seed( const typename Engine::result_type value =
587         Engine::default_seed ) {
588         engine_instance( ).seed( value );

```

```

589     }
590
591     /**
592     * \brief Reinitializes the internal state
593     * of the random-number engine using new seed value
594     * \param seq The seed sequence
595     *          to use in the initialization of the internal state
596     */
597     template<typename Sseq>
598     static void seed( Sseq& seq ) {
599         engine_instance( ).seed( seq );
600     }
601
602     /// return random number from engine in [min(), max()] range
603     static typename Engine::result_type get( ) {
604         return engine_instance( ) ( );
605     }
606
607     /**
608     * \brief Compares internal pseudo-random number engine
609     *        with 'other' pseudo-random number engine.
610     *        Two engines are equal, if their internal states
611     *        are equivalent, that is, if they would generate
612     *        equivalent values for any number of calls of operator()
613     * \param other The engine, with which the internal engine will be compared
614     * \return true, if other and internal engine are equal
615     */
616     static bool is_equal( const Engine& other ) {
617         return engine_instance( ) == other;
618     }
619
620     /**
621     * \brief Serializes the internal state of the
622     *        internal pseudo-random number engine as a sequence
623     *        of decimal numbers separated by one or more spaces,
624     *        and inserts it to the stream ost. The fill character
625     *        and the formatting flags of the stream are
626     *        ignored and unaffected.
627     * \param ost The output stream to insert the data to
628     */
629     template<typename CharT, typename Traits>
630     static void serialize( std::basic_ostream<CharT, Traits>& ost ) {
631         ost << engine_instance( );
632     }
633
634     /**
635     * \brief Restores the internal state of the
636     *        internal pseudo-random number engine from
637     *        the serialized representation, which
638     *        was created by an earlier call to 'serialize'
639     *        using a stream with the same imbued locale and
640     *        the same CharT and Traits.
641     *        If the input cannot be deserialized,
642     *        internal engine is left unchanged and failbit is raised on ist

```

```

643 * \param ost The input stream to extract the data from
644 */
645 template<typename CharT, typename Traits>
646 static void deserialize( std::basic_istream<CharT, Traits>& ist ) {
647     ist >> engine_instance( );
648 }
649
650 /**
651 * \brief Generate a random integer number in a [from; to] range
652 *         by std::uniform_int_distribution
653 * \param from The first limit number of a random range
654 * \param to The second limit number of a random range
655 * \return A random integer number in a [from; to] range
656 * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
657 * \note Prevent implicit type conversion
658 */
659 template<typename T>
660 static typename std::enable_if<details::is_uniform_int<T>::value
661     , T>::type get( T from = std::numeric_limits<T>::min( ),
662     T to = std::numeric_limits<T>::max( ) ) {
663     if( from < to ) // Allow range from higher to lower
664         return IntegerDist<T>{ from, to }( engine_instance( ) );
665     return IntegerDist<T>{ to, from }( engine_instance( ) );
666 }
667
668 /**
669 * \brief Generate a random real number in a [from; to] range
670 *         by std::uniform_real_distribution
671 * \param from The first limit number of a random range
672 * \param to The second limit number of a random range
673 * \return A random real number in a [from; to] range
674 * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
675 * \note Prevent implicit type conversion
676 */
677 template<typename T>
678 static typename std::enable_if<details::is_uniform_real<T>::value
679     , T>::type get( T from = std::numeric_limits<T>::min( ),
680     T to = std::numeric_limits<T>::max( ) ) {
681     if( from < to ) // Allow range from higher to lower
682         return RealDist<T>{ from, to }( engine_instance( ) );
683     return RealDist<T>{ to, from }( engine_instance( ) );
684 }
685
686 /**
687 * \brief Generate a random byte number in a [from; to] range
688 * \param from The first limit number of a random range
689 * \param to The second limit number of a random range
690 * \return A random byte number in a [from; to] range
691 * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
692 * \note Prevent implicit type conversion
693 */
694 template<typename T>

```



```

695 static typename std::enable_if<details::is_byte<T>::value
696     , T>::type get( T from = std::numeric_limits<T>::min( ),
697                     T to = std::numeric_limits<T>::max( ) ) {
698     // Choose between short and unsigned short for byte conversion
699     using short_t = typename std::conditional<std::is_signed<T>::value,
700         short, unsigned short>::type;
701
702     return static_cast<T>( get<short_t>( from, to ) );
703 }
704
705 /**
706  * \brief Generate a random common_type number in a [from; to] range
707  * \param Key The Key type for this version of 'get' method
708  *         Type should be '(THIS_TYPE)::common' struct
709  * \param from The first limit number of a random range
710  * \param to The second limit number of a random range
711  * \return A random common_type number in a [from; to] range
712  * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
713  * \note Allow implicit type conversion
714  * \note Prevent implicit type conversion from signed to unsigned types
715  *         Why? std::common_type<Unsigned, Signed> chooses unsigned value,
716  *         then Signed value will be converted to Unsigned value
717  *         which gives us a wrong range for random values.
718  *         https://stackoverflow.com/a/5416498/5734836
719  */
720 template<
721     typename Key,
722     typename A,
723     typename B,
724     typename C = typename std::common_type<A, B>::type
725 >
726 static typename std::enable_if<
727     std::is_same<Key, common>::value
728     && details::is_supported_number<A>::value
729     && details::is_supported_number<B>::value
730     // Prevent implicit type conversion from signed to unsigned types
731     && std::is_signed<A>::value != std::is_unsigned<B>::value
732     , C>::type get( A from = std::numeric_limits<A>::min( ),
733                     B to = std::numeric_limits<B>::max( ) ) {
734     return get( static_cast<C>( from ), static_cast<C>( to ) );
735 }
736
737 /**
738  * \brief Generate a random character in a [from; to] range
739  *         by std::uniform_int_distribution
740  * \param from The first limit number of a random range
741  * \param to The second limit number of a random range
742  * \return A random character in a [from; to] range
743  * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
744  * \note Prevent implicit type conversion
745  */
746 template<typename T>

```

```

747 static typename std::enable_if<details::is_supported_character<T>::value
748     , T>::type get(T from = std::numeric_limits<T>::min(),
749     T to = std::numeric_limits<T>::max()) {
750     if (from < to) // Allow range from higher to lower
751         return static_cast<T>(IntegerDist<std::int64_t>{ static_cast<std::in
752     return static_cast<T>(IntegerDist<std::int64_t>{ static_cast<std::int64_t
753 }
754 /**
755  * \brief Generate a bool value with specific probability
756  *         by std::bernoulli_distribution
757  * \param probability The probability of generating true in [0; 1] range
758  *         0 means always false, 1 means always true
759  * \return 'true' with 'probability' probability ('false' otherwise)
760 */
761 template<typename T>
762 static typename std::enable_if<std::is_same<T, bool>::value
763     , bool>::type get( const double probability = 0.5 ) {
764     assert( 0 <= probability && 1 >= probability ); // out of [0; 1] range
765     return BoolDist{ probability }( engine_instance( ) );
766 }
767 /**
768  * \brief Return random value from initializer_list
769  * \param init_list initializer_list with values
770  * \return Random value from initializer_list
771  * \note Should be 1 or more elements in initializer_list
772  * \note Warning! Elements in initializer_list can't be moved:
773  *         https://stackoverflow.com/a/8193157/5734836
774 */
775 template<typename T>
776 static T get( std::initializer_list<T> init_list ) {
777     assert( 0u != init_list.size( ) );
778     return *get( init_list.begin( ), init_list.end( ) );
779 }
780 /**
781  * \brief Return random iterator from iterator range
782  * \param first, last - the range of elements
783  * \return Random iterator from [first, last) range
784  * \note If first == last, return last
785 */
786 template<typename InputIt>
787 static typename std::enable_if<details::is_iterator<InputIt>::value
788     , InputIt>::type get( InputIt first, InputIt last ) {
789     const auto size = std::distance( first, last );
790     if( 0 == size ) return last;
791     using diff_t = typename std::iterator_traits<InputIt>::difference_type;
792     return std::next( first, get<diff_t>( 0, size - 1 ) );
793 }
794 /**
795  * \brief Return random iterator from Container

```

```

800 * \param container The container with elements
801 * \return Random iterator from container
802 * \note If container is empty return std::end( container ) iterator
803 */
804 template<typename Container>
805 static auto get( Container& container ) ->
806     typename std::enable_if<details::is_iterator<
807         decltype(std::begin(container))>::value
808         , decltype(std::begin(container))
809         >::type {
810     return get( std::begin( container ), std::end( container ) );
811 }
812
813 /**
814 * \brief Return random pointer from built-in array
815 * \param array The built-in array with elements
816 * \return Pointer to random element in array
817 */
818 template<typename T, std::size_t N>
819 static T* get( T( &array )[ N ] ) {
820     return std::addressof( array[ get<std::size_t>( 0, N - 1 ) ] );
821 }
822
823 /**
824 * \brief Return value from custom Dist distribution
825 *         seeded by internal random engine
826 * \param Dist The type of custom distribution with next concept:
827 *         http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
828 * \param args The arguments which will be forwarded to Dist constructor
829 * \return Value from custom distribution
830 */
831 template<typename Dist, typename... Args>
832 static typename Dist::result_type get( Args&&... args ) {
833     return Dist{ std::forward<Args>( args )... }( engine_instance( ) );
834 }
835
836 /**
837 * \brief Return value from custom 'dist' distribution
838 *         seeded by internal random engine
839 * \param dist The custom distribution with next concept:
840 *         http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
841 * \param args The arguments which will be forwarded to Dist constructor
842 * \return Value from custom 'dist' distribution
843 */
844 template<typename Dist>
845 static typename Dist::result_type get( Dist& dist ) {
846     return dist( engine_instance( ) );
847 }
848
849 /**
850 * \brief Reorders the elements in the given range [first, last)
851 *         such that each possible permutation of those elements
852 *         has equal probability of appearance.
853 * \param first, last - the range of elements to shuffle randomly

```

```

854     */
855     template<typename RandomIt>
856     static void shuffle( RandomIt first, RandomIt last ) {
857         std::shuffle( first, last, engine_instance( ) );
858     }
859
860     /**
861     * \brief Reorders the elements in the given container
862     *         such that each possible permutation of those elements
863     *         has equal probability of appearance.
864     * \param container - the container with elements to shuffle randomly
865     */
866     template<typename Container>
867     static void shuffle( Container& container ) {
868         shuffle( std::begin( container ), std::end( container ) );
869     }
870
871     /// return internal engine by copy
872     static Engine get_engine( ) {
873         return engine_instance( );
874     }
875
876     /// return internal engine by ref
877     static Engine& engine( ) {
878         return engine_instance();
879     }
880 protected:
881     /// get reference to the thread local engine instance
882     static Engine& engine_instance( ) {
883         thread_local Engine engine{ Seeder{ }( ) };
884         return engine;
885     }
886 };
887
888 /**
889 * \brief Base template class for random
890 *         with local API and local internal member storage
891 * \note it IS thread safe but less efficient then
892 *         basic_random_static
893 * \param Engine A random engine with interface like in the std::mt19937
894 * \param Seeder A seeder type which return seed for internal engine
895 *               through operator()
896 */
897 template<
898     typename Engine,
899     typename Seeder = seeder_default,
900     template<typename> class IntegerDist = std::uniform_int_distribution,
901     template<typename> class RealDist = std::uniform_real_distribution,
902     typename BoolDist = std::bernoulli_distribution
903 >
904 class basic_random_local {
905 public:
906     /// Type of used random number engine

```

```

907     using engine_type = Engine;
908
909     /// Type of used random number seeder
910     using seeder_type = Seeder;
911
912     /// Type of used integer distribution
913     template<typename T>
914     using integer_dist_t = IntegerDist<T>;
915
916     /// Type of used real distribution
917     template<typename T>
918     using real_dist_t = RealDist<T>;
919
920     /// Type of used bool distribution
921     using bool_dist_t = BoolDist;
922
923     /// Key type for getting common type numbers or objects
924     using common = details::common;
925
926     /**
927     * \return The minimum value
928     * potentially generated by the random-number engine
929     */
930     static constexpr typename Engine::result_type min( ) {
931         return Engine::min( );
932     }
933
934     /**
935     * \return The maximum value
936     * potentially generated by the random-number engine
937     */
938     static constexpr typename Engine::result_type max( ) {
939         return Engine::max( );
940     }
941
942     /// Advances the internal state by z times
943     void discard( const unsigned long long z ) {
944         m_engine.discard( z );
945     }
946
947     /// Reseed by Seeder
948     void reseed( ) {
949         Seeder seeder;
950         seed( seeder( ) );
951     }
952
953     /**
954     * \brief Reinitializes the internal state
955     * of the random-number engine using new seed value
956     * \param value The seed value to use
957     * in the initialization of the internal state
958     */
959     void seed( const typename Engine::result_type value =
960                 Engine::default_seed ) {
961         m_engine.seed( value );
962     }

```

```

963
964 /**
965  * \brief Reinitializes the internal state
966  * of the random-number engine using new seed value
967  * \param seq The seed sequence
968  *          to use in the initialization of the internal state
969  */
970 template<typename Sseq>
971 void seed( Sseq& seq ) {
972     m_engine.seed( seq );
973 }
974
975 /// return random number from engine in [min(), max()] range
976 typename Engine::result_type get( ) {
977     return m_engine( );
978 }
979
980 /**
981  * \brief Compares internal pseudo-random number engine
982  *        with 'other' pseudo-random number engine.
983  *        Two engines are equal, if their internal states
984  *        are equivalent, that is, if they would generate
985  *        equivalent values for any number of calls of operator()
986  * \param other The engine, with which the internal engine will be compared
987  * \return true, if other and internal engine are equal
988  */
989 bool is_equal( const Engine& other ) {
990     return m_engine == other;
991 }
992
993 /**
994  * \brief Serializes the internal state of the
995  *        internal pseudo-random number engine as a sequence
996  *        of decimal numbers separated by one or more spaces,
997  *        and inserts it to the stream ost. The fill character
998  *        and the formatting flags of the stream are
999  *        ignored and unaffected.
1000  * \param ost The output stream to insert the data to
1001  */
1002 template<typename CharT, typename Traits>
1003 void serialize( std::basic_ostream<CharT, Traits>& ost ) {
1004     ost << m_engine;
1005 }
1006
1007 /**
1008  * \brief Restores the internal state of the
1009  *        internal pseudo-random number engine from
1010  *        the serialized representation, which
1011  *        was created by an earlier call to 'serialize'
1012  *        using a stream with the same imbued locale and
1013  *        the same CharT and Traits.
1014  *        If the input cannot be deserialized,
1015  *        internal engine is left unchanged and failbit is raised on ist
1016  * \param ost The input stream to extract the data from

```

```

1017 */
1018 template<typename CharT, typename Traits>
1019 void deserialize( std::basic_istream<CharT, Traits>& ist ) {
1020     ist >> m_engine;
1021 }
1022
1023 /**
1024  * \brief Generate a random integer number in a [from; to] range
1025  *         by std::uniform_int_distribution
1026  * \param from The first limit number of a random range
1027  * \param to The second limit number of a random range
1028  * \return A random integer number in a [from; to] range
1029  * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
1030  * \note Prevent implicit type conversion
1031 */
1032 template<typename T>
1033 typename std::enable_if<details::is_uniform_int<T>::value
1034     , T>::type get( T from = std::numeric_limits<T>::min( ),
1035     T to = std::numeric_limits<T>::max( ) ) {
1036     if( from < to ) // Allow range from higher to lower
1037         return IntegerDist<T>{ from, to }( m_engine );
1038     return IntegerDist<T>{ to, from }( m_engine );
1039 }
1040
1041 /**
1042  * \brief Generate a random real number in a [from; to] range
1043  *         by std::uniform_real_distribution
1044  * \param from The first limit number of a random range
1045  * \param to The second limit number of a random range
1046  * \return A random real number in a [from; to] range
1047  * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
1048  * \note Prevent implicit type conversion
1049 */
1050 template<typename T>
1051 typename std::enable_if<details::is_uniform_real<T>::value
1052     , T>::type get( T from = std::numeric_limits<T>::min( ),
1053     T to = std::numeric_limits<T>::max( ) ) {
1054     if( from < to ) // Allow range from higher to lower
1055         return RealDist<T>{ from, to }( m_engine );
1056     return RealDist<T>{ to, from }( m_engine );
1057 }
1058
1059 /**
1060  * \brief Generate a random byte number in a [from; to] range
1061  *         by std::uniform_int_distribution
1062  * \param from The first limit number of a random range
1063  * \param to The second limit number of a random range
1064  * \return A random byte number in a [from; to] range
1065  * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
1066  * \note Prevent implicit type conversion
1067 */
1068 template<typename T>
1069 typename std::enable_if<details::is_byte<T>::value

```

```

1069         , T>::type get( T from = std::numeric_limits<T>::min( ),
1070                        T to = std::numeric_limits<T>::max( ) ) {
1071         // Choose between short and unsigned short for byte conversion
1072         using short_t = typename std::conditional<std::is_signed<T>::value,
1073            short, unsigned short>::type;
1074
1075         return static_cast<T>( get<short_t>( from, to ) );
1076     }
1077
1078     /**
1079     * \brief Generate a random common_type number in a [from; to] range
1080     * \param Key The Key type for this version of 'get' method
1081     *         Type should be '(THIS_TYPE)::common' struct
1082     * \param from The first limit number of a random range
1083     * \param to The second limit number of a random range
1084     * \return A random common_type number in a [from; to] range
1085     * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
1086     * \note Allow implicit type conversion
1087     * \note Prevent implicit type conversion from signed to unsigned types
1088     *         Why? std::common_type<Unsigned, Signed> chooses unsigned value,
1089     *         then Signed value will be converted to Unsigned value
1090     *         which gives us a wrong range for random values.
1091     *         https://stackoverflow.com/a/5416498/5734836
1092     */
1093     template<
1094         typename Key,
1095         typename A,
1096         typename B,
1097         typename C = typename std::common_type<A, B>::type
1098     >
1099     typename std::enable_if<
1100         std::is_same<Key, common>::value
1101         && details::is_supported_number<A>::value
1102         && details::is_supported_number<B>::value
1103         // Prevent implicit type conversion from signed to unsigned types
1104         && std::is_signed<A>::value != std::is_unsigned<B>::value
1105         , C>::type get( A from = std::numeric_limits<A>::min( ),
1106                       B to = std::numeric_limits<B>::max( ) ) {
1107         return get( static_cast<C>( from ), static_cast<C>( to ) );
1108     }
1109
1110     /**
1111     * \brief Generate a random character in a [from; to] range
1112     *         by std::uniform_int_distribution
1113     * \param from The first limit number of a random range
1114     * \param to The second limit number of a random range
1115     * \return A random character in a [from; to] range
1116     * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
1117     * \note Prevent implicit type conversion
1118     */
1119     template<typename T>
1120     typename std::enable_if<details::is_supported_character<T>::value

```



```

1121         , T>::type get(T from = std::numeric_limits<T>::min(),
1122             T to = std::numeric_limits<T>::max()) {
1123         if (from < to) // Allow range from higher to lower
1124             return static_cast<T>(IntegerDist<std::int64_t>{ static_cast<std::int64_t>(from),
1125                 static_cast<std::int64_t>(to) });
1126     }
1127
1128     /**
1129     * \brief Generate a bool value with specific probability
1130     *         by std::bernoulli_distribution
1131     * \param probability The probability of generating true in [0; 1] range
1132     *         0 means always false, 1 means always true
1133     * \return 'true' with 'probability' probability ('false' otherwise)
1134     */
1135     template<typename T>
1136     typename std::enable_if<std::is_same<T, bool>::value
1137         , bool>::type get( const double probability = 0.5 ) {
1138         assert( 0 <= probability && 1 >= probability ); // out of [0; 1] range
1139         return BoolDist{ probability }( m_engine );
1140     }
1141
1142     /**
1143     * \brief Return random value from initializer_list
1144     * \param init_list initializer_list with values
1145     * \return Random value from initializer_list
1146     * \note Should be 1 or more elements in initializer_list
1147     * \note Warning! Elements in initializer_list can't be moved:
1148     *         https://stackoverflow.com/a/8193157/5734836
1149     */
1150     template<typename T>
1151     T get( std::initializer_list<T> init_list ) {
1152         assert( 0u != init_list.size( ) );
1153         return *get( init_list.begin( ), init_list.end( ) );
1154     }
1155
1156     /**
1157     * \brief Return random iterator from iterator range
1158     * \param first, last - the range of elements
1159     * \return Random iterator from [first, last) range
1160     * \note If first == last, return last
1161     */
1162     template<typename InputIt>
1163     typename std::enable_if<details::is_iterator<InputIt>::value
1164         , InputIt>::type get( InputIt first, InputIt last ) {
1165         const auto size = std::distance( first, last );
1166         if( 0 == size ) return last;
1167         using diff_t = typename std::iterator_traits<InputIt>::difference_type;
1168         return std::next( first, get<diff_t>( 0, size - 1 ) );
1169     }
1170
1171     /**
1172     * \brief Return random iterator from Container
1173     * \param container The container with elements

```

```

1174 * \return Random iterator from container
1175 * \note If container is empty return std::end( container ) iterator
1176 */
1177 template<typename Container>
1178 auto get( Container& container ) ->
1179     typename std::enable_if<details::is_iterator<
1180         decltype(std::begin(container))>::value
1181         , decltype(std::begin(container))
1182     >::type {
1183     return get( std::begin( container ), std::end( container ) );
1184 }
1185
1186 /**
1187 * \brief Return random pointer from built-in array
1188 * \param array The built-in array with elements
1189 * \return Pointer to random element in array
1190 */
1191 template<typename T, std::size_t N>
1192 T* get( T( &array ) [ N ] ) {
1193     return std::addressof( array[ get<std::size_t>( 0, N - 1 ) ] );
1194 }
1195
1196 /**
1197 * \brief Return value from custom Dist distribution
1198 *         seeded by internal random engine
1199 * \param Dist The type of custom distribution with next concept:
1200 *         http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
1201 * \param args The arguments which will be forwarded to Dist constructor
1202 * \return Value from custom distribution
1203 */
1204 template<typename Dist, typename... Args>
1205 typename Dist::result_type get( Args&&... args ) {
1206     return Dist{ std::forward<Args>( args )... }( m_engine );
1207 }
1208
1209 /**
1210 * \brief Return value from custom 'dist' distribution
1211 *         seeded by internal random engine
1212 * \param dist The custom distribution with next concept:
1213 *         http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
1214 * \param args The arguments which will be forwarded to Dist constructor
1215 * \return Value from custom 'dist' distribution
1216 */
1217 template<typename Dist>
1218 typename Dist::result_type get( Dist& dist ) {
1219     return dist( m_engine );
1220 }
1221
1222 /**
1223 * \brief Reorders the elements in the given range [first, last)
1224 *         such that each possible permutation of those elements
1225 *         has equal probability of appearance.
1226 * \param first, last - the range of elements to shuffle randomly
1227 */

```

```

1228     template<typename RandomIt>
1229     void shuffle( RandomIt first, RandomIt last ) {
1230         std::shuffle( first, last, m_engine );
1231     }
1232
1233     /**
1234     * \brief Reorders the elements in the given container
1235     *         such that each possible permutation of those elements
1236     *         has equal probability of appearance.
1237     * \param container - the container with elements to shuffle randomly
1238     */
1239     template<typename Container>
1240     void shuffle( Container& container ) {
1241         shuffle( std::begin( container ), std::end( container ) );
1242     }
1243
1244     /// return internal engine by copy
1245     Engine get_engine( ) const {
1246         return m_engine;
1247     }
1248
1249     /// return internal engine by ref
1250     Engine& engine() {
1251         return m_engine;
1252     }
1253 protected:
1254     /// return engine seeded by Seeder
1255     static Engine make_seeded_engine( ) {
1256         // Make seeder instance for seed return by reference like std::seed_seq
1257         return Engine{ Seeder{ }( ) };
1258     }
1259 protected:
1260     /// The random number engine
1261     Engine m_engine{ make_seeded_engine( ) };
1262 };
1263
1264     /**
1265     * \brief The basic static random alias based on a std::mt19937
1266     * \note It uses static methods API and data with static storage
1267     * \note Not thread safe but more prefomance
1268     */
1269     using random_static = basic_random_static<std::mt19937>;
1270
1271     /**
1272     * \brief The basic static random alias based on a std::mt19937
1273     * \note It uses static methods API and data with thread_local storage
1274     * \note Thread safe but less performance
1275     */
1276     using random_thread_local = basic_random_thread_local<std::mt19937>;
1277
1278     /**
1279     * \brief The basic static random alias based on a std::mt19937
1280     * \note It uses non static methods API and data with auto storage
1281     * \note Not thread safe. Should construct on the stack at local scope

```

```

1282     */
1283     using random_local = basic_random_local<std::mt19937>;
1284
1285 } // namespace effolkronium
1286
1287 #endif // #ifndef EFFOLKRONIUM_RANDOM_HPP

```

7.8 main

```

1 #include "page_design.h"
2 #include <iostream>
3 #include <string>
4
5 int startGame();
6 Character createChar();
7
8 int main(int argc, char** argv)
9 {
10     //Player input taken.
11     std::string input;
12     std::cout << "Welcome to the Denali Trail!\nAre you ready for an adventure?\n(Y/n)";
13     std::cin >> input;
14     if (input == "y" || input == "Y")
15     {
16         int distance = startGame();
17         std::cout << "\nYou made it: " << distance << " feet down the trail";
18     }
19     else return 0;
20 }
21
22 //Begins game.
23 int startGame()
24 {
25     std::string input;
26     int distance = 0;
27     Character protag = createChar();
28     while (distance < 100)
29     {
30         randomEvent(protag, distance);
31         std::cout << "You are " << distance << " feet down the trail\nContinue?\n(Y/n)";
32         std::cin >> input;
33         if (input == "n" || input == "N") break;
34     }
35     return distance;
36 }
37
38 //Creates character.
39 Character createChar()
40 {
41     Character protag;

```

```
42  std::cout << "\nWhat would you like to name your character? ";
43  std::cin >> protag.name;
44  protag.health = 100;
45  protag.hunger = 0;
46  protag.thirst = 0;
47  protag.inventory = 0;
48  protag.strength = 0;
49  return protag;
50 }
```

References