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

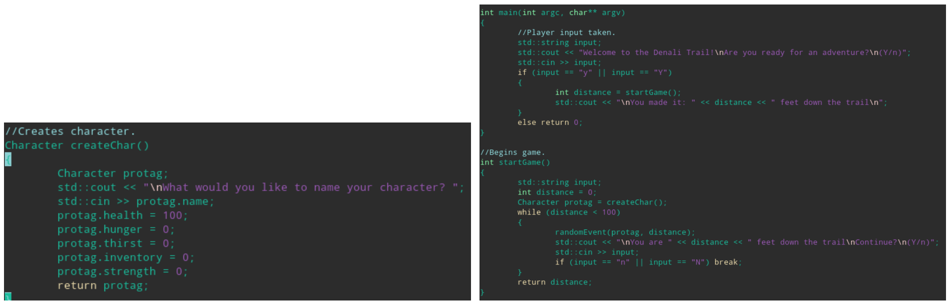
This game is greatly inspired by the Oregon Trail, but it will be visually designed around Denali.

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 weill 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.



```
//Creates character.
Character createChar()
{
        Character protag;
        std::cout << "\nWhat would you like to name your character? ";
        std::cin >> protag.name;
        protag.health = 100;
        protag.hunger = 0;
        protag.thirst = 0;
        protag.inventory = 0;
        protag.strength = 0;
        return protag;
}
```

```
int main(int argc, char** argv)
{
        //Player input taken.
        std::string input;
        std::cout << "Welcome to the Denali Trail!\nAre you ready for an adventure?\n(Y/n)";
        std::cin >> input;
        if (input == "y" || input == "Y")
        {
                int distance = startGame();
                std::cout << "\nYou made it: " << distance << " feet down the trail\n";
        }
        else return 0;
}

//Begins game.
int startGame()
{
        std::string input;
        int distance = 0;
        Character protag = createChar();
        while (distance < 100)
        {
                randomEvent(protag, distance);
                std::cout << "\nYou are " << distance << " feet down the trail\nContinue?\n(Y/n)";
                std::cin >> input;
                if (input == "n" || input == "N") break;
        }
        return distance;
}
```

Figure 1: Millard Artifacts

3 Project Iteration 3: Initial Prototype

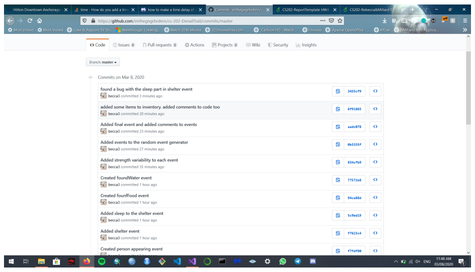
3.1 Contribution: Millard A. Arnold V

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 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


Commits on Mar 8, 2020

found a bug with the sleep part in shelter event
becca3 committed 3 minutes ago — 3455cf9

added some items to inventory. added comments to code too
becca3 committed 20 minutes ago — 6f91865

Added final event and added comments to events
becca3 committed 23 minutes ago — aadc078

Added events to the random event generator
becca3 committed 27 minutes ago — 0b5335f

Added strength variability to each event
becca3 committed 35 minutes ago — 834cfb9

Created foundWater event
becca3 committed 1 hour ago — 77572e8

Created founfFood event
becca3 committed 1 hour ago — 94ca88d

Added sleep to the shelter event
becca3 committed 1 hour ago — 5c9bd19

Added shelter event
becca3 committed 1 hour ago — f7615c4

Created person appearing event
— f7fdf80

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 founfFood 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

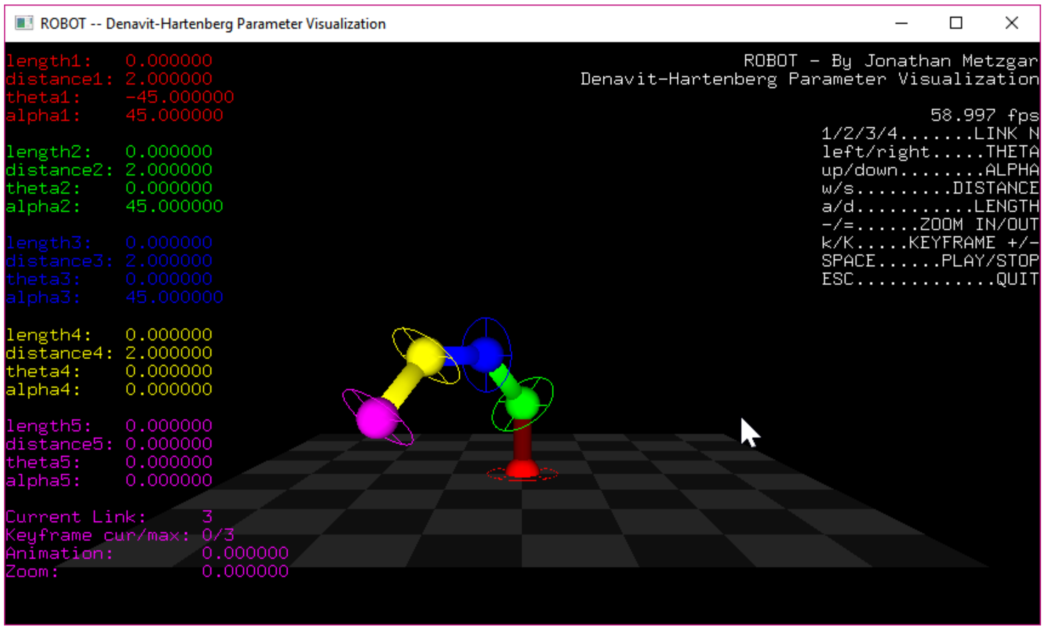
A screenshot of our program is displayed in Figure 3. In this section, talk about the results of your program.

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.



```
length1:    0.000000                                   ROBOT - By Jonathan Metzgar
distance1:  2.000000                          Denavit-Hartenberg Parameter Visualization
theta1:     -45.000000
alpha1:     45.000000                                                      58.997 fps
                                                                1/2/3/4.......LINK N
length2:    0.000000                                            left/right.....THETA
distance2:  2.000000                                            up/down........ALPHA
theta2:     0.000000                                            w/s.........DISTANCE
alpha2:     45.000000                                           a/d...........LENGTH
                                                                -/=......ZOOM IN/OUT
length3:    0.000000                                            k/K.....KEYFRAME +/-
distance3:  2.000000                                            SPACE......PLAY/STOP
theta3:     0.000000                                            ESC.............QUIT
alpha3:     45.000000

length4:    0.000000
distance4:  2.000000
theta4:     0.000000
alpha4:     0.000000

length5:    0.000000
distance5:  0.000000
theta5:     0.000000
alpha5:     0.000000

Current Link:       3
Keyframe cur/max:   0/3
Animation:          0.000000
Zoom:               0.000000
```

Figure 3: A Screenshot of our finished application.

7.4 Program Source Code

The source code is licensed under the WRITETHENAMEOFYOURLICENSEHERE license.

7.5 page design header

```
#ifndef PAGE_DESIGN_H
#define PAGE_DESIGN_H

#include "random.hpp"
#include <string>

//Struct to hold player variables.
struct Character
{
  std::string name;
  int thirst;
  int hunger;
  int strength;
  int health;
  long long inventory;
};

void randomEvent(Character& protag, int&distance);
#endif
```

7.6 page design Source

```
#include "page_design.h"
#include <vector>
#include <iostream>
#include <ctime>

using Random=effolkronium::random_static;

//Bear attacks player.
void bear(Character& protag, int& distance)
{
  system("CLS");
  std::cout << "You have been attacked by a bear.\t";
  protag.health -= 10;
  protag.strength -= 10;
  distance += 5;
}

//Player gets lost.
void lost(Character& protag, int& distance)
{
  system("CLS");
  std::cout << "You get lost for a while and then eventually find your way.";
  protag.hunger += 10;
  protag.thirst += 10;
  protag.strength -= 5;
```

```cpp
  distance += 1;
}

//Player finds a knife along the route.
void foundKnife(Character& protag, int& distance)
{
  system("CLS");
  std::cout << "You found a Knife.";
  protag.health += 5;
  protag.inventory += 1;
  distance += 1;
}

//Player falls ill.
void illness(Character& protag, int& distance)
{
  system("CLS");
  std::cout << "Developing symptoms of illness, maybe an infection.";
  protag.health -= 5;
  protag.strength -= 10;
  distance += 1;
}

//Player attacked by wolf.
void wolf(Character& protag, int& distance)
{
  system("CLS");
  std::cout << "A wolf appears, you run as fast as you can to escape!";
  protag.health -= 10;
  protag.hunger += 5;
  protag.thirst += 10;
  protag.strength -= 10;
  distance += 5;
}

//Player runs across another person.
void person(Character& protag, int& distance)
{
  system("CLS");
  std::cout << "A person appears. May or may not be a friendly. You run just incase.
  protag.thirst += 5;
  distance += 5;
}

//Player comes across a shelter.
void shelter(Character& protag, int& distance)
{
  system("CLS");
  std::cout << "You come across a shelter. Let's rest.";
  sleep(2);
  protag.health += 20;
  protag.strength += 20;
  distance += 0;
}

//Player finds some food.
void foundFood(Character& protag, int& distance)
{
  system("CLS");
  std::cout << "You found a cache of food.";
  protag.hunger -= 10;
  protag.health += 5;
  protag.strength += 10;
  protag.inventory += 3;
  distance += 1;
}

//Player finds some water.
void foundWater(Character& protag, int& distance)
{
  system("CLS");
  std::cout << "You found a source of water.";
  protag.thirst -= 10;
  protag.health += 5;
  protag.strength += 10;
  protag.inventory += 2;
  distance += 1;
}

//Player eats a poison berry.
void poison(Character& protag, int& distance)
{
  system("CLS");
  std::cout << "You found a berry and ate it. It was poisonous.";
  protag.hunger += 5;
  protag.health -= 5;
  protag.strength -= 5;
  distance += 1;
}

//Places all events into a vector and generates one randomly.
void randomEvent(Character& protag, int& distance)
{
  std::vector<std::string> events;
  events.push_back("bear");
  events.push_back("lost");
  events.push_back("foundKnife");
  events.push_back("illness");
  events.push_back("wolf");
  events.push_back("person");
  events.push_back("shelter");
  events.push_back("foundFood");
  events.push_back("foundwater");
  events.push_back("poison");

  //Shuffles events and chooses one at random.
  Random::shuffle(events);
  if (events[0] == "bear")
  {
    bear(protag, distance);
  }
  else if (events[1] == "lost")
  {
    lost(protag, distance);
  }
  else if (events[2] == "foundKnife")
  {
    foundKnife(protag, distance);
  }
  else if (events[3] == "illness")
  {
    illness(protag, distance);
  }
  else if (events[4] == "wolf")
  {
    wolf(protag, distance);
  }
  else if (events[5] == "person")
  {
    person(protag, distance);
  }
  else if (events[6] == "shelter")
  {
    shelter(protag, distance);
  }
  else if (events[7] == "foundFood")
  {
    foundFood(protag, distance);
  }
  else if (events[8] == "foundWater")
  {
    foundWater(protag, distance);
  }
  else if (events[9] == "poison")
  {
    poison(protag, distance);
  }
}
```

7.7 random library header

```cpp
/*
______  ___   _   _______ ________  ___
| ___ \/ _ \ | \ | |  _  \  _  |  \/  | Random for modern C++
| |_/ / /_\ \|  \| | | | | | | | .  . |
|    /|  _  || . ` | | | | | | | |\/| | version 1.3.1
| |\ \| | | || |\  | |/ /\ \_/ / |  | |
\_| \_\_| |_/\_| \_/___/  \___/\_|  |_/ https://github.com/effolkronium/random

Licensed under the MIT License <http://opensource.org/licenses/MIT>.
Copyright (c) 2019 effolkronium

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files( the "Software" ), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions :

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
*/

#ifndef EFFOLKRONIUM_RANDOM_HPP
#define EFFOLKRONIUM_RANDOM_HPP

#include <random>
#include <chrono> // timed seed
#include <type_traits>
#include <cassert>
#include <initializer_list>
#include <utility> // std::forward, std::declval
#include <algorithm> // std::shuffle, std::next, std::distance
#include <iterator> // std::begin, std::end, std::iterator_traits
#include <limits> // std::numeric_limits
#include <ostream>
#include <istream>

namespace effolkronium {

    namespace details {
        /// Key type for getting common type numbers or objects
        struct common{ };

        /// True if type T is applicable by a std::uniform_int_distribution
        template<typename T>
        struct is_uniform_int {
            static constexpr bool value =
                   std::is_same<T,              short>::value
                || std::is_same<T,                int>::value
                || std::is_same<T,               long>::value
                || std::is_same<T,          long long>::value
                || std::is_same<T,     unsigned short>::value
                || std::is_same<T,       unsigned int>::value
                || std::is_same<T,      unsigned long>::value
                || std::is_same<T, unsigned long long>::value;
        };

        /// True if type T is applicable by a std::uniform_real_distribution
        template<typename T>
        struct is_uniform_real {
            static constexpr bool value =
                   std::is_same<T,       float>::value
                || std::is_same<T,      double>::value
                || std::is_same<T, long double>::value;
        };

        /// True if type T is plain byte
        template<typename T>
        struct is_byte {
            static constexpr bool value =
                   std::is_same<T,   signed char>::value
                || std::is_same<T, unsigned char>::value;
        };

        /// True if type T is plain number type
        template<typename T>
        struct is_supported_number {
            static constexpr bool value =
                   is_byte        <T>::value
                || is_uniform_real<T>::value
                || is_uniform_int <T>::value;
        };

        /// True if type T is character type
        template<typename T>
        struct is_supported_character {
            static constexpr bool value =
                   std::is_same<T, char>::value
                || std::is_same<T, wchar_t>::value
                || std::is_same<T, char16_t>::value
                || std::is_same<T, char32_t>::value;
        };

        /// True if type T is iterator
        template<typename T>
        struct is_iterator {
        private:
            static char test( ... );

            template <typename U,
                typename = typename std::iterator_traits<U>::difference_type,
                typename = typename std::iterator_traits<U>::pointer,
                typename = typename std::iterator_traits<U>::reference,
                typename = typename std::iterator_traits<U>::value_type,
                typename = typename std::iterator_traits<U>::iterator_category
            > static long test( U&& );
        public:
            static constexpr bool value = std::is_same<
                decltype( test( std::declval<T>( ) ) ), long>::value;
        };

    } // namespace details

    /// Default seeder for 'random' classes
    struct seeder_default {
        /// return seed sequence
        std::seed_seq& operator() ( ) {
            // MinGW issue, std::random_device returns constant value
            // Use std::seed_seq with additional seed from C++ chrono
            return seed_seq;
        }
    private:
        std::seed_seq seed_seq{ {
                static_cast<std::uintmax_t>( std::random_device{ }( ) ),
                static_cast<std::uintmax_t>( std::chrono::steady_clock::now( )
                                             .time_since_epoch( ).count( ) ),
        } };
    };

    /**
    * \brief Base template class for random
    *        with static API and static internal member storage
    * \note it is NOT thread safe but more efficient then
    *                           basic_random_thread_local
    * \param Engine A random engine with interface like in the std::mt19937
    * \param Seeder A seeder type which return seed for internal engine
    *                                              through operator()
    */
    template<
        typename Engine,
        typename Seeder = seeder_default,
        template<typename> class IntegerDist = std::uniform_int_distribution,
        template<typename> class RealDist = std::uniform_real_distribution,
        typename BoolDist = std::bernoulli_distribution
    >
    class basic_random_static {
    public:
        basic_random_static( ) = delete;

        /// Type of used random number engine
        using engine_type = Engine;

        /// Type of used random number seeder
        using seeder_type = Seeder;

        /// Type of used integer distribution
        template<typename T>
        using integer_dist_t = IntegerDist<T>;

        /// Type of used real distribution
        template<typename T>
        using real_dist_t = RealDist<T>;

        /// Type of used bool distribution
        using bool_dist_t = BoolDist;

        /// Key type for getting common type numbers or objects
        using common = details::common;

        /**
        * \return The minimum value
        * potentially generated by the random-number engine
        */
        static constexpr typename Engine::result_type min( ) {
            return Engine::min( );
        }

        /**
        * \return The maximum value
        * potentially generated by the random-number engine
        */
        static constexpr typename Engine::result_type max( ) {
            return Engine::max( );
        }

        /// Advances the internal state by z times
        static void discard( const unsigned long long z ) {
            engine_instance( ).discard( z );
        }

        /// Reseed by Seeder
        static void reseed( ) {
            Seeder seeder;
            seed( seeder( ) );
        }

        /**
        * \brief Reinitializes the internal state
        * of the random-number engine using new seed value
        * \param value The seed value to use
        *        in the initialization of the internal state
        */
        static void seed( const typename Engine::result_type value =
                          Engine::default_seed ) {
            engine_instance( ).seed( value );
        }
```

```
        /**
        * \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
```

```
    */
    template<typename CharT, typename Traits>
    static void deserialize( std::basic_istream<CharT, Traits>& ist ) {
        ist >> engine_instance( );
    }

    /**
    * \brief Generate a random integer number in a [from; to] range
    *        by std::uniform_int_distribution
    * \param from The first limit number of a random range
    * \param to The second limit number of a random range
    * \return A random integer number in a [from; to] range
    * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
    * \note Prevent implicit type conversion
    */
    template<typename T>
    static typename std::enable_if<details::is_uniform_int<T>::value
        , T>::type get( T from = std::numeric_limits<T>::min( ),
                        T to = std::numeric_limits<T>::max( ) ) {
        if( from < to ) // Allow range from higher to lower
            return IntegerDist<T>{ from, to }( engine_instance( ) );
        return IntegerDist<T>{ to, from }( engine_instance( ) );
    }

    /**
    * \brief Generate a random real number in a [from; to] range
    *        by std::uniform_real_distribution
    * \param from The first limit number of a random range
    * \param to The second limit number of a random range
    * \return A random real number in a [from; to] range
    * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
    * \note Prevent implicit type conversion
    */
    template<typename T>
    static typename std::enable_if<details::is_uniform_real<T>::value
        , T>::type get( T from = std::numeric_limits<T>::min( ),
                        T to = std::numeric_limits<T>::max( ) ) {
        if( from < to ) // Allow range from higher to lower
            return RealDist<T>{ from, to }( engine_instance( ) );
        return RealDist<T>{ to, from }( engine_instance( ) );
    }

    /**
    * \brief Generate a random byte number in a [from; to] range
    * \param from The first limit number of a random range
    * \param to The second limit number of a random range
    * \return A random byte number in a [from; to] range
    * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
    * \note Prevent implicit type conversion
    */
    template<typename T>
    static typename std::enable_if<details::is_byte<T>::value
```

```
        , T>::type get( T from = std::numeric_limits<T>::min( ),
                        T to = std::numeric_limits<T>::max( ) ) {
        // Choose between short and unsigned short for byte conversion
        using short_t = typename std::conditional<std::is_signed<T>::value,
            short, unsigned short>::type;

        return static_cast<T>( get<short_t>( from, to ) );
    }

    /**
    * \brief Generate a random common_type number in a [from; to] range
    * \param Key The Key type for this version of 'get' method
    *        Type should be '(THIS_TYPE)::common' struct
    * \param from The first limit number of a random range
    * \param to The second limit number of a random range
    * \return A random common_type number in a [from; to] range
    * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
    * \note Allow implicit type conversion
    * \note Prevent implicit type conversion from singed to unsigned types
    *       Why? std::common_type<Unsigned, Signed> chooses unsigned value,
    *                 then Signed value will be converted to Unsigned value
    *                       which gives us a wrong range for random values.
    *                           https://stackoverflow.com/a/5416498/5734836
    */
    template<
        typename Key,
        typename A,
        typename B,
        typename C = typename std::common_type<A, B>::type
    >
    static typename std::enable_if<
           std::is_same<Key, common>::value
        && details::is_supported_number<A>::value
        && details::is_supported_number<B>::value
        // Prevent implicit type conversion from singed to unsigned types
        && std::is_signed<A>::value != std::is_unsigned<B>::value
        , C>::type get( A from = std::numeric_limits<A>::min( ),
                        B to = std::numeric_limits<B>::max( ) ) {
        return get( static_cast<C>( from ), static_cast<C>( to ) );
    }

    /**
    * \brief Generate a random character in a [from; to] range
    *        by std::uniform_int_distribution
    * \param from The first limit number of a random range
    * \param to The second limit number of a random range
    * \return A random character in a [from; to] range
    * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
    * \note Prevent implicit type conversion
    */
    template<typename T>
    static typename std::enable_if<details::is_supported_character<T>::value
```

```cpp
            , T>::type get(T from = std::numeric_limits<T>::min(),
                T to = std::numeric_limits<T>::max()) {
            if (from < to) // Allow range from higher to lower
                return static_cast<T>(IntegerDist<std::int64_t>{ static_cast<std::in
            return static_cast<T>(IntegerDist<std::int64_t>{ static_cast<std::int64_
        }

        /**
        * \brief Generate a bool value with specific probability
        *                          by std::bernoulli_distribution
        * \param probability The probability of generating true in [0; 1] range
        *        0 means always false, 1 means always true
        * \return 'true' with 'probability' probability ('false' otherwise)
        */
        template<typename T>
        static typename std::enable_if<std::is_same<T, bool>::value
            , bool>::type get( const double probability = 0.5 ) {
            assert( 0 <= probability && 1 >= probability ); // out of [0; 1] range
            return BoolDist{ probability }( engine_instance( ) );
        }

        /**
        * \brief Return random value from initilizer_list
        * \param init_list initilizer_list with values
        * \return Random value from initilizer_list
        * \note Should be 1 or more elements in initilizer_list
        * \note Warning! Elements in initilizer_list can't be moved:
        *               https://stackoverflow.com/a/8193157/5734836
        */
        template<typename T>
        static T get( std::initializer_list<T> init_list ) {
            assert( 0u != init_list.size( ) );
            return *get( init_list.begin( ), init_list.end( ) );
        }

        /**
        * \brief Return random iterator from iterator range
        * \param first, last - the range of elements
        * \return Random iterator from [first, last) range
        * \note If first == last, return last
        */
        template<typename InputIt>
        static typename std::enable_if<details::is_iterator<InputIt>::value
            , InputIt>::type get( InputIt first, InputIt last ) {
            const auto size = std::distance( first, last );
            if( 0 == size ) return last;
            using diff_t = typename std::iterator_traits<InputIt>::difference_type;
            return std::next( first, get<diff_t>( 0, size - 1 ) );
        }

        /**
        * \brief Return random iterator from Container
        * \param container The container with elements
        * \return Random iterator from container
        * \note If container is empty return std::end( container ) iterator
        */
        template<typename Container>
        static auto get( Container& container ) ->
            typename std::enable_if<details::is_iterator<
                decltype(std::begin(container))>::value
                , decltype(std::begin(container))
            >::type {
            return get( std::begin( container ), std::end( container ) );
        }

        /**
        * \brief Return random pointer from built-in array
        * \param array The built-in array with elements
        * \return Pointer to random element in array
        */
        template<typename T, std::size_t N>
        static T* get( T( &array )[ N ] ) {
            return std::addressof( array[ get<std::size_t>( 0, N - 1 ) ] );
        }

        /**
        * \brief Return value from custom Dist distribution
        *        seeded by internal random engine
        * \param Dist The type of custom distribution with next concept:
        *        http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
        * \param args The arguments which will be forwarded to Dist constructor
        * \return Value from custom distribution
        */
        template<typename Dist, typename... Args>
        static typename Dist::result_type get( Args&&... args ) {
            return Dist{ std::forward<Args>( args )... }( engine_instance( ) );
        }

        /**
        * \brief Return value from custom 'dist' distribution
        *        seeded by internal random engine
        * \param dist The custom distribution with next concept:
        *        http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
        * \param args The arguments which will be forwarded to Dist constructor
        * \return Value from custom 'dist' distribution
        */
        template<typename Dist>
        static typename Dist::result_type get( Dist& dist ) {
            return dist( engine_instance( ) );
        }

        /**
        * \brief Reorders the elements in the given range [first, last)
        *        such that each possible permutation of those elements
        *        has equal probability of appearance.
        * \param first, last - the range of elements to shuffle randomly
        */
```

```
    template<typename RandomIt>
    static void shuffle( RandomIt first, RandomIt last ) {
        std::shuffle( first, last, engine_instance( ) );
    }

    /**
    * \brief Reorders the elements in the given container
    *        such that each possible permutation of those elements
    *        has equal probability of appearance.
    * \param container - the container with elements to shuffle randomly
    */
    template<typename Container>
    static void shuffle( Container& container ) {
        shuffle( std::begin( container ), std::end( container ) );
    }

    /// return internal engine by copy
    static Engine get_engine( ) {
        return engine_instance( );
    }

    /// return internal engine by ref
    static Engine& engine() {
        return engine_instance();
    }
protected:
    /// get reference to the static engine instance
    static Engine& engine_instance( ) {
        static Engine engine{ Seeder{ }( ) };
        return engine;
    }
};

/**
* \brief Base template class for random
*        with thread_local API and thread_local internal member storage
* \note it IS thread safe but less efficient then
*                           basic_random_static
* \param Engine A random engine with interface like in the std::mt19937
* \param Seeder A seeder type which return seed for internal engine
*                                             through operator()
*/
template<
    typename Engine,
    typename Seeder = seeder_default,
    template<typename> class IntegerDist = std::uniform_int_distribution,
    template<typename> class RealDist = std::uniform_real_distribution,
    typename BoolDist = std::bernoulli_distribution
>
class basic_random_thread_local {
public:
    basic_random_thread_local( ) = delete;

```

```
        /// Type of used random number engine
        using engine_type = Engine;

        /// Type of used random number seeder
        using seeder_type = Seeder;

        /// Type of used integer distribution
        template<typename T>
        using integer_dist_t = IntegerDist<T>;

        /// Type of used real distribution
        template<typename T>
        using real_dist_t = RealDist<T>;

        /// Type of used bool distribution
        using bool_dist_t = BoolDist;

        /// Key type for getting common type numbers or objects
        using common = details::common;

        /**
        * \return The minimum value
        * potentially generated by the random-number engine
        */
        static constexpr typename Engine::result_type min( ) {
            return Engine::min( );
        }

        /**
        * \return The maximum value
        * potentially generated by the random-number engine
        */
        static constexpr typename Engine::result_type max( ) {
            return Engine::max( );
        }

        /// Advances the internal state by z times
        static void discard( const unsigned long long z ) {
            engine_instance( ).discard( z );
        }

        /// Reseed by Seeder
        static void reseed( ) {
            Seeder seeder;
            seed( seeder( ) );
        }

        /**
        * \brief Reinitializes the internal state
        * of the random-number engine using new seed value
        * \param value The seed value to use
        *        in the initialization of the internal state
        */
        static void seed( const typename Engine::result_type value =
                          Engine::default_seed ) {
            engine_instance( ).seed( value );
        }

        /**
        * \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
    */
    template<typename CharT, typename Traits>
    static void deserialize( std::basic_istream<CharT, Traits>& ist ) {
        ist >> engine_instance( );
    }

    /**
    * \brief Generate a random integer number in a [from; to] range
    *        by std::uniform_int_distribution
    * \param from The first limit number of a random range
    * \param to The second limit number of a random range
    * \return A random integer number in a [from; to] range
    * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
    * \note Prevent implicit type conversion
    */
    template<typename T>
    static typename std::enable_if<details::is_uniform_int<T>::value
        , T>::type get( T from = std::numeric_limits<T>::min( ),
                        T to = std::numeric_limits<T>::max( ) ) {
        if( from < to ) // Allow range from higher to lower
            return IntegerDist<T>{ from, to }( engine_instance( ) );
        return IntegerDist<T>{ to, from }( engine_instance( ) );
    }

    /**
    * \brief Generate a random real number in a [from; to] range
    *        by std::uniform_real_distribution
    * \param from The first limit number of a random range
    * \param to The second limit number of a random range
    * \return A random real number in a [from; to] range
    * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
    * \note Prevent implicit type conversion
    */
    template<typename T>
    static typename std::enable_if<details::is_uniform_real<T>::value
        , T>::type get( T from = std::numeric_limits<T>::min( ),
                        T to = std::numeric_limits<T>::max( ) ) {
        if( from < to ) // Allow range from higher to lower
            return RealDist<T>{ from, to }( engine_instance( ) );
        return RealDist<T>{ to, from }( engine_instance( ) );
    }

    /**
    * \brief Generate a random byte number in a [from; to] range
    * \param from The first limit number of a random range
    * \param to The second limit number of a random range
    * \return A random byte number in a [from; to] range
    * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
    * \note Prevent implicit type conversion
    */
    template<typename T>
```

```
        static typename std::enable_if<details::is_byte<T>::value
            , T>::type get( T from = std::numeric_limits<T>::min( ),
                            T to = std::numeric_limits<T>::max( ) ) {
            // Choose between short and unsigned short for byte conversion
            using short_t = typename std::conditional<std::is_signed<T>::value,
                short, unsigned short>::type;

            return static_cast<T>( get<short_t>( from, to ) );
        }

        /**
        * \brief Generate a random common_type number in a [from; to] range
        * \param Key The Key type for this version of 'get' method
        *        Type should be '(THIS_TYPE)::common' struct
        * \param from The first limit number of a random range
        * \param to The second limit number of a random range
        * \return A random common_type number in a [from; to] range
        * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
        * \note Allow implicit type conversion
        * \note Prevent implicit type conversion from singed to unsigned types
        *       Why? std::common_type<Unsigned, Signed> chooses unsigned value,
        *                 then Signed value will be converted to Unsigned value
        *                       which gives us a wrong range for random values.
        *                           https://stackoverflow.com/a/5416498/5734836
        */
        template<
            typename Key,
            typename A,
            typename B,
            typename C = typename std::common_type<A, B>::type
        >
        static typename std::enable_if<
                std::is_same<Key, common>::value
            && details::is_supported_number<A>::value
            && details::is_supported_number<B>::value
            // Prevent implicit type conversion from singed to unsigned types
            && std::is_signed<A>::value != std::is_unsigned<B>::value
            , C>::type get( A from = std::numeric_limits<A>::min( ),
                            B to = std::numeric_limits<B>::max( ) ) {
            return get( static_cast<C>( from ), static_cast<C>( to ) );
        }

        /**
        * \brief Generate a random character in a [from; to] range
        *        by std::uniform_int_distribution
        * \param from The first limit number of a random range
        * \param to The second limit number of a random range
        * \return A random character in a [from; to] range
        * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
        * \note Prevent implicit type conversion
        */
        template<typename T>
```

```
    static typename std::enable_if<details::is_supported_character<T>::value
        , T>::type get(T from = std::numeric_limits<T>::min(),
            T to = std::numeric_limits<T>::max()) {
        if (from < to) // Allow range from higher to lower
            return static_cast<T>(IntegerDist<std::int64_t>{ static_cast<std::in
        return static_cast<T>(IntegerDist<std::int64_t>{ static_cast<std::int64_
    }

    /**
    * \brief Generate a bool value with specific probability
    *                         by std::bernoulli_distribution
    * \param probability The probability of generating true in [0; 1] range
    *        0 means always false, 1 means always true
    * \return 'true' with 'probability' probability ('false' otherwise)
    */
    template<typename T>
    static typename std::enable_if<std::is_same<T, bool>::value
        , bool>::type get( const double probability = 0.5 ) {
        assert( 0 <= probability && 1 >= probability ); // out of [0; 1] range
        return BoolDist{ probability }( engine_instance( ) );
    }

    /**
    * \brief Return random value from initilizer_list
    * \param init_list initilizer_list with values
    * \return Random value from initilizer_list
    * \note Should be 1 or more elements in initilizer_list
    * \note Warning! Elements in initilizer_list can't be moved:
    *               https://stackoverflow.com/a/8193157/5734836
    */
    template<typename T>
    static T get( std::initializer_list<T> init_list ) {
        assert( 0u != init_list.size( ) );
        return *get( init_list.begin( ), init_list.end( ) );
    }

    /**
    * \brief Return random iterator from iterator range
    * \param first, last - the range of elements
    * \return Random iterator from [first, last) range
    * \note If first == last, return last
    */
    template<typename InputIt>
    static typename std::enable_if<details::is_iterator<InputIt>::value
        , InputIt>::type get( InputIt first, InputIt last ) {
        const auto size = std::distance( first, last );
        if( 0 == size ) return last;
        using diff_t = typename std::iterator_traits<InputIt>::difference_type;
        return std::next( first, get<diff_t>( 0, size - 1 ) );
    }

    /**
    * \brief Return random iterator from Container
```

```
* \param container The container with elements
* \return Random iterator from container
* \note If container is empty return std::end( container ) iterator
*/
template<typename Container>
static auto get( Container& container ) ->
    typename std::enable_if<details::is_iterator<
        decltype(std::begin(container))>::value
        , decltype(std::begin(container))
    >::type {
    return get( std::begin( container ), std::end( container ) );
}

/**
* \brief Return random pointer from built-in array
* \param array The built-in array with elements
* \return Pointer to random element in array
*/
template<typename T, std::size_t N>
static T* get( T( &array )[ N ] ) {
    return std::addressof( array[ get<std::size_t>( 0, N - 1 ) ] );
}

/**
* \brief Return value from custom Dist distribution
*        seeded by internal random engine
* \param Dist The type of custom distribution with next concept:
*        http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
* \param args The arguments which will be forwarded to Dist constructor
* \return Value from custom distribution
*/
template<typename Dist, typename... Args>
static typename Dist::result_type get( Args&&... args ) {
    return Dist{ std::forward<Args>( args )... }( engine_instance( ) );
}

/**
* \brief Return value from custom 'dist' distribution
*        seeded by internal random engine
* \param dist The custom distribution with next concept:
*        http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
* \param args The arguments which will be forwarded to Dist constructor
* \return Value from custom 'dist' distribution
*/
template<typename Dist>
static typename Dist::result_type get( Dist& dist ) {
    return dist( engine_instance( ) );
}

/**
* \brief Reorders the elements in the given range [first, last)
*        such that each possible permutation of those elements
*        has equal probability of appearance.
* \param first, last - the range of elements to shuffle randomly
```

```
    */
    template<typename RandomIt>
    static void shuffle( RandomIt first, RandomIt last ) {
        std::shuffle( first, last, engine_instance( ) );
    }

    /**
    * \brief Reorders the elements in the given container
    *        such that each possible permutation of those elements
    *        has equal probability of appearance.
    * \param container - the container with elements to shuffle randomly
    */
    template<typename Container>
    static void shuffle( Container& container ) {
        shuffle( std::begin( container ), std::end( container ) );
    }

    /// return internal engine by copy
    static Engine get_engine( ) {
        return engine_instance( );
    }

    /// return internal engine by ref
    static Engine& engine() {
        return engine_instance();
    }
protected:
    /// get reference to the thread local engine instance
    static Engine& engine_instance( ) {
        thread_local Engine engine{ Seeder{ }( ) };
        return engine;
    }
};

/**
* \brief Base template class for random
*        with local API and local internal member storage
* \note it IS thread safe but less efficient then
*                           basic_random_static
* \param Engine A random engine with interface like in the std::mt19937
* \param Seeder A seeder type which return seed for internal engine
*                                             through operator()
*/
template<
    typename Engine,
    typename Seeder = seeder_default,
    template<typename> class IntegerDist = std::uniform_int_distribution,
    template<typename> class RealDist = std::uniform_real_distribution,
    typename BoolDist = std::bernoulli_distribution
>
class basic_random_local {
public:
    /// Type of used random number engine
```

```
        using engine_type = Engine;

        /// Type of used random number seeder
        using seeder_type = Seeder;

        /// Type of used integer distribution
        template<typename T>
        using integer_dist_t = IntegerDist<T>;

        /// Type of used real distribution
        template<typename T>
        using real_dist_t = RealDist<T>;

        /// Type of used bool distribution
        using bool_dist_t = BoolDist;

        /// Key type for getting common type numbers or objects
        using common = details::common;

        /**
        * \return The minimum value
        * potentially generated by the random-number engine
        */
        static constexpr typename Engine::result_type min( ) {
            return Engine::min( );
        }

        /**
        * \return The maximum value
        * potentially generated by the random-number engine
        */
        static constexpr typename Engine::result_type max( ) {
            return Engine::max( );
        }

        /// Advances the internal state by z times
        void discard( const unsigned long long z ) {
            m_engine.discard( z );
        }

        /// Reseed by Seeder
        void reseed( ) {
            Seeder seeder;
            seed( seeder( ) );
        }

        /**
        * \brief Reinitializes the internal state
        * of the random-number engine using new seed value
        * \param value The seed value to use
        *        in the initialization of the internal state
        */
        void seed( const typename Engine::result_type value =
                          Engine::default_seed ) {
            m_engine.seed( value );
        }
        /**
        * \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>
        void seed( Sseq& seq ) {
            m_engine.seed( seq );
        }

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

        /**
        * \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
        */
        bool is_equal( const Engine& other ) {
            return m_engine == 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>
        void serialize( std::basic_ostream<CharT, Traits>& ost ) {
            ost << m_engine;
        }

        /**
        * \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
        */
        template<typename CharT, typename Traits>
        void deserialize( std::basic_istream<CharT, Traits>& ist ) {
            ist >> m_engine;
        }

        /**
        * \brief Generate a random integer number in a [from; to] range
        *        by std::uniform_int_distribution
        * \param from The first limit number of a random range
        * \param to The second limit number of a random range
        * \return A random integer number in a [from; to] range
        * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
        * \note Prevent implicit type conversion
        */
        template<typename T>
        typename std::enable_if<details::is_uniform_int<T>::value
            , T>::type get( T from = std::numeric_limits<T>::min( ),
                            T to = std::numeric_limits<T>::max( ) ) {
            if( from < to ) // Allow range from higher to lower
                return IntegerDist<T>{ from, to }( m_engine );
            return IntegerDist<T>{ to, from }( m_engine );
        }

        /**
        * \brief Generate a random real number in a [from; to] range
        *        by std::uniform_real_distribution
        * \param from The first limit number of a random range
        * \param to The second limit number of a random range
        * \return A random real number in a [from; to] range
        * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
        * \note Prevent implicit type conversion
        */
        template<typename T>
        typename std::enable_if<details::is_uniform_real<T>::value
            , T>::type get( T from = std::numeric_limits<T>::min( ),
                            T to = std::numeric_limits<T>::max( ) ) {
            if( from < to ) // Allow range from higher to lower
                return RealDist<T>{ from, to }( m_engine );
            return RealDist<T>{ to, from }( m_engine );
        }

        /**
        * \brief Generate a random byte number in a [from; to] range
        * \param from The first limit number of a random range
        * \param to The second limit number of a random range
        * \return A random byte number in a [from; to] range
        * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
        * \note Prevent implicit type conversion
        */
        template<typename T>
        typename std::enable_if<details::is_byte<T>::value
            , T>::type get( T from = std::numeric_limits<T>::min( ),
                            T to = std::numeric_limits<T>::max( ) ) {
            // Choose between short and unsigned short for byte conversion
            using short_t = typename std::conditional<std::is_signed<T>::value,
                short, unsigned short>::type;

            return static_cast<T>( get<short_t>( from, to ) );
        }

        /**
        * \brief Generate a random common_type number in a [from; to] range
        * \param Key The Key type for this version of 'get' method
        *        Type should be '(THIS_TYPE)::common' struct
        * \param from The first limit number of a random range
        * \param to The second limit number of a random range
        * \return A random common_type number in a [from; to] range
        * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
        * \note Allow implicit type conversion
        * \note Prevent implicit type conversion from singed to unsigned types
        *       Why? std::common_type<Unsigned, Signed> chooses unsigned value,
        *                 then Signed value will be converted to Unsigned value
        *                       which gives us a wrong range for random values.
        *                           https://stackoverflow.com/a/5416498/5734836
        */
        template<
            typename Key,
            typename A,
            typename B,
            typename C = typename std::common_type<A, B>::type
        >
        typename std::enable_if<
               std::is_same<Key, common>::value
            && details::is_supported_number<A>::value
            && details::is_supported_number<B>::value
            // Prevent implicit type conversion from singed to unsigned types
            && std::is_signed<A>::value != std::is_unsigned<B>::value
            , C>::type get( A from = std::numeric_limits<A>::min( ),
                            B to = std::numeric_limits<B>::max( ) ) {
            return get( static_cast<C>( from ), static_cast<C>( to ) );
        }

        /**
        * \brief Generate a random character in a [from; to] range
        *        by std::uniform_int_distribution
        * \param from The first limit number of a random range
        * \param to The second limit number of a random range
        * \return A random character in a [from; to] range
        * \note Allow both: 'from' <= 'to' and 'from' >= 'to'
        * \note Prevent implicit type conversion
        */
        template<typename T>
        typename std::enable_if<details::is_supported_character<T>::value
```

```
        , T>::type get(T from = std::numeric_limits<T>::min(),
            T to = std::numeric_limits<T>::max()) {
        if (from < to) // Allow range from higher to lower
            return static_cast<T>(IntegerDist<std::int64_t>{ static_cast<std::in
        return static_cast<T>(IntegerDist<std::int64_t>{ static_cast<std::int64_
    }

    /**
    * \brief Generate a bool value with specific probability
    *                          by std::bernoulli_distribution
    * \param probability The probability of generating true in [0; 1] range
    *        0 means always false, 1 means always true
    * \return 'true' with 'probability' probability ('false' otherwise)
    */
    template<typename T>
    typename std::enable_if<std::is_same<T, bool>::value
        , bool>::type get( const double probability = 0.5 ) {
        assert( 0 <= probability && 1 >= probability ); // out of [0; 1] range
        return BoolDist{ probability }( m_engine );
    }

    /**
    * \brief Return random value from initilizer_list
    * \param init_list initilizer_list with values
    * \return Random value from initilizer_list
    * \note Should be 1 or more elements in initilizer_list
    * \note Warning! Elements in initilizer_list can't be moved:
    *               https://stackoverflow.com/a/8193157/5734836
    */
    template<typename T>
    T get( std::initializer_list<T> init_list ) {
        assert( 0u != init_list.size( ) );
        return *get( init_list.begin( ), init_list.end( ) );
    }

    /**
    * \brief Return random iterator from iterator range
    * \param first, last - the range of elements
    * \return Random iterator from [first, last) range
    * \note If first == last, return last
    */
    template<typename InputIt>
    typename std::enable_if<details::is_iterator<InputIt>::value
        , InputIt>::type get( InputIt first, InputIt last ) {
        const auto size = std::distance( first, last );
        if( 0 == size ) return last;
        using diff_t = typename std::iterator_traits<InputIt>::difference_type;
        return std::next( first, get<diff_t>( 0, size - 1 ) );
    }

    /**
    * \brief Return random iterator from Container
    * \param container The container with elements
```

```
        * \return Random iterator from container
        * \note If container is empty return std::end( container ) iterator
        */
        template<typename Container>
        auto get( Container& container ) ->
            typename std::enable_if<details::is_iterator<
                decltype(std::begin(container))>::value
                , decltype(std::begin(container))
            >::type {
            return get( std::begin( container ), std::end( container ) );
        }

        /**
        * \brief Return random pointer from built-in array
        * \param array The built-in array with elements
        * \return Pointer to random element in array
        */
        template<typename T, std::size_t N>
        T* get( T( &array )[ N ] ) {
            return std::addressof( array[ get<std::size_t>( 0, N - 1 ) ] );
        }

        /**
        * \brief Return value from custom Dist distribution
        *        seeded by internal random engine
        * \param Dist The type of custom distribution with next concept:
        *        http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
        * \param args The arguments which will be forwarded to Dist constructor
        * \return Value from custom distribution
        */
        template<typename Dist, typename... Args>
        typename Dist::result_type get( Args&&... args ) {
            return Dist{ std::forward<Args>( args )... }( m_engine );
        }

        /**
        * \brief Return value from custom 'dist' distribution
        *        seeded by internal random engine
        * \param dist The custom distribution with next concept:
        *        http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution
        * \param args The arguments which will be forwarded to Dist constructor
        * \return Value from custom 'dist' distribution
        */
        template<typename Dist>
        typename Dist::result_type get( Dist& dist ) {
            return dist( m_engine );
        }

        /**
        * \brief Reorders the elements in the given range [first, last)
        *        such that each possible permutation of those elements
        *        has equal probability of appearance.
        * \param first, last - the range of elements to shuffle randomly
        */
        template<typename RandomIt>
        void shuffle( RandomIt first, RandomIt last ) {
            std::shuffle( first, last, m_engine );
        }

        /**
        * \brief Reorders the elements in the given container
        *        such that each possible permutation of those elements
        *        has equal probability of appearance.
        * \param container - the container with elements to shuffle randomly
        */
        template<typename Container>
        void shuffle( Container& container ) {
            shuffle( std::begin( container ), std::end( container ) );
        }

        /// return internal engine by copy
        Engine get_engine( ) const {
            return m_engine;
        }

        /// return internal engine by ref
        Engine& engine() {
            return m_engine;
        }
    protected:
        /// return engine seeded by Seeder
        static Engine make_seeded_engine( ) {
            // Make seeder instance for seed return by reference like std::seed_seq
            return Engine{ Seeder{ }( ) };
        }
    protected:
        /// The random number engine
        Engine m_engine{ make_seeded_engine( ) };
    };

    /**
    * \brief The basic static random alias based on a std::mt19937
    * \note It uses static methods API and data with static storage
    * \note Not thread safe but more prefomance
    */
    using random_static = basic_random_static<std::mt19937>;

    /**
    * \brief The basic static random alias based on a std::mt19937
    * \note It uses static methods API and data with thread_local storage
    * \note Thread safe but less perfomance
    */
    using random_thread_local = basic_random_thread_local<std::mt19937>;

    /**
    * \brief The basic static random alias based on a std::mt19937
    * \note It uses non static methods API and data with auto storage
    * \note Not thread safe. Should construct on the stack at local scope
    */
    using random_local = basic_random_local<std::mt19937>;

} // namespace effolkronium

#endif // #ifndef EFFOLKRONIUM_RANDOM_HPP
```

7.8 main

```
#include "page_design.h"
#include <iostream>
#include <string>

int startGame();
Character createChar();

int main(int argc, char** argv)
{
  //Player input taken.
  std::string input;
  std::cout << "Welcome to the Denali Trail!\nAre you ready for an adventure?\n(Y/n)
  std::cin >> input;
  if (input == "y" || input == "Y")
  {
    int distance = startGame();
    std::cout << "\nYou made it: " << distance << " feet down the trail";
  }
  else return 0;
}

//Begins game.
int startGame()
{
  std::string input;
  int distance = 0;
  Character protag = createChar();
  while (distance < 100)
  {
    randomEvent(protag, distance);
    std::cout << "You are " << distance << " feet down the trail\nContinue?\n(Y/n)";
    std::cin >> input;
    if (input == "n" || input == "N") break;
  }
  return distance;
}

//Creates character.
Character createChar()
{
  Character protag;
  std::cout << "\nWhat would you like to name your character? ";
  std::cin >> protag.name;
  protag.health = 100;
  protag.hunger = 0;
  protag.thirst = 0;
  protag.inventory = 0;
  protag.strength = 0;
  return protag;
}
```

References