ENEE 140 Project 1: A Roll of the Dice

Posted: Tuesday, 3 March 2026
Due: This project has two deadlines:

- A partial implementation is due on Friday, 13 March 2026 at 11:59 PM
- A complete implementation is due on Friday, 3 April 2026 at 11:59 PM

Project objectives

1. Become familiar with the process of completing a programming project in C.
2. Declare and use of variables of basic types: `unsigned`, `int`, and `float`.
3. Perform arithmetic operations with these variables.
4. Understand type conversions, overflow and the range of numbers stored using integer variables.
5. Declare global variables.
6. Use `if` and `if-else` statements for conditional processing.
7. Use loops (`while` or `for`) for iterative processing.
8. Implement and invoke functions.
9. Solve a programming problem by breaking it down into smaller problems.
10. Understand how pseudo-random numbers are generated in a computer system.

Project description

In this project, you will write two programs: `enee140_gen_rnd.c` and `enee140_test_rnd.c`. The first program will generate pseudo-random numbers, with several ranges and distributions, and the second program will test the randomness of the numbers you have generated. You will have to implement several C functions in order to provide the functionality of the random number generator (RNG). This program will print out a menu of options, prompt the user to choose an option from the menu and then invoke the corresponding function.

A sequence of numbers generated by a deterministic computer program is not truly random, unlike a sequence of coin tosses or repeated rolls of a pair of dice. Instead, the sequence *appears* to be random, by providing properties (e.g. uniformity) that are adequate for a variety of applications that require random numbers (e.g. simulation, cryptography, gaming). Such numbers are formally called *pseudo-random numbers*; in this handout, we simply call them random numbers, to avoid complicating the description.

The core of most random number generation programs is a function, such as `rand()` from the C standard library, that generates integer numbers uniformly distributed between `0` and a value `RND_MAX`. Other ranges and distributions can be obtained by transforming numbers generated in this manner. The random number generators must be initialized by calling a function to provide a *seed*—a number from which all the subsequent numbers are generated. For the random number generator included in the C library, this is achieved by calling the `srand()` function. While the function that generates the next random number may be invoked *several times* in a program, the seeding function is generally invoked only *once*. Because the random number generators are deterministic computer programs, two sequences generated using the same algorithm and starting from the same seed will be identical (a common mistake is to seed the generator with a constant value before generating each random number—this will result in generating the same number repeatedly). However, the numbers must still be distributed uniformly in the requested range. This property can be tested using a *statistical uniformity test*, which compares the observed frequencies of the numbers generated with the ones that would be expected from a uniform distribution.

User interface

Write a complete program, called `enee140_gen_rnd.c`, that prints a menu, prompts the user for several options, and then generates and prints a sequence of random numbers.

1 Main menu

Your first task is to print out the following menu on the screen:

```
Welcome to the ENEE140 pseudo-random number generator!
1: Print RND_MAX
2: Generate uniformly-distributed positive integers
3: Generate uniformly-distributed positive integers, up to a given limit
4: Generate uniformly-distributed integers, from a given range
5: Generate uniformly-distributed floats, from a given range
6: Generate exponentially-distributed floats
```

Then prompt the user for a choice by printing out:

```
Enter your choice (1-6):
```

You may assume that the user will enter an integer. If the input is a valid number between 1 and 6, you should proceed to the next step. Otherwise, you should print out the following error message and ask user to re-enter a choice:

```
Sorry, that is not a valid option
Enter your choice (1-6):
```

If the user fails to enter a valid choice 3 times in a row, you should print out the following on the screen and terminate the program:

```
You have entered 3 invalid options. Goodbye!
```

2 Algorithm and seed

Your second task is to prompt the user for the algorithm to use when generating random numbers and for the seed. First print out:

```
Select the algorithm to use.
Enter your choice (1-3):
```

You may assume that the user will enter an integer. If the input is a valid number between 1 and 3, you can proceed to the next step. Otherwise, you should print out the following error message and ask user to re-enter a choice:

```
Sorry, that is not a valid option
Enter your choice (1-3):
```

If the user fails to enter a valid choice 3 times in a row, you should print out the following on the screen and terminate the program:

```
You have entered 3 invalid options. Goodbye!
```

Hint: Note that the pattern for prompting the user is the same as in the previous step. To avoid writing the same code twice, you could define a *helper function* that takes one integer parameter (the maximum number of options) and that returns an integer (the user's choice). To terminate the program when the user repeatedly fails to provide a valid option, you may call the function `exit()` from `stdlib.h`.

If the user has entered a valid algorithm number, print out:

```
Select the seed for the random number generator:
```

Read the integer provided by the user and seed the random number generator, by invoking the function described in Step 5. Then proceed to the next step.

3 Additional parameters

Prompt the user for additional parameters needed by the selected option:

1. No additional parameters are needed. Print `RND_MAX` as described in Step 6, then prompt the user for another choice from the main menu (you do not have to read the algorithm and seed again).

2. Prompt the user for the length of the generated sequence:

   ```
   How many numbers should I generate:
   ```

   Read the integer number provided by the user, then generate random numbers by repeatedly invoking the function described in Step 7 and print them as described in Step 4.

3. Prompt the user for the for the length of the sequence and for the maximum number to generate:

   ```
   How many numbers should I generate:
   Enter the maximum number to generate:
   ```

Read two positive integer numbers provided by the user, then generate random numbers by repeatedly invoking the function described in Step 8 and print them as described in Step 4.

4. Prompt the user for the length and range of the sequence:

   ```
   How many numbers should I generate:
   Enter the minimum number to generate:
   Enter the maximum number to generate:
   ```

   Read one positive integer and two signed integer numbers provided by the user, then generate random numbers by repeatedly invoking the function described in Step 9 and print them as described in Step 4.

5. Prompt the user for the length and range of the sequence:

   ```
   How many numbers should I generate:
   Enter the minimum number to generate:
   Enter the maximum number to generate:
   ```

   Read one positive integer and two floating point numbers provided by the user, then generate random numbers by repeatedly invoking the function described in Step 10 and print them as described in Step 4.

6. Prompt the user for the length and mean of the sequence:

   ```
   How many numbers should I generate:
   Enter the mean of the distribution:
   ```

   Read one positive integer and one floating point number provided by the user. If the floating point number is negative, print the error message below and prompt the user again:

   ```
   Error: the mean must be a positive number.
   ```

   After the user has provided a correct mean, generate random numbers by repeatedly invoking the function described in Step 11 and print them as described in Step 4.

4 Print the generated sequence

Print each number number generated followed by a space. If the numbers generated are `float`s, print only two digits after the decimal point. When you are done printing the sequence, print an end-of-line character and exit the program. In other words, the last line of your program's output should be a sequence of random numbers separated by one space.

Random number generation

When random numbers are *uniformly distributed* in a given range, all the values in the range occur with equal probabilities. For example, a coin toss can have two possible outcomes—heads or tails—each with probability $p = \frac{1}{2}$; rolling a die can produce any number from 1 to 6 with probability $p = \frac{1}{6}$.

A popular method for generating uniformly-distributed random numbers on a computer is the *linear congruential* (LC) method. The LC method is governed by three parameters:

- $M = 2^m$, the modulus; $0 < M$.
- A, the multiplier; $0 \leq A < M$.
- B, the increment; $0 \leq B < M$.

A, B, and m are positive numbers, which can be represented as `unsigned` variables in a C program. This method generates a sequence of unsigned numbers X_0, X_1, …. X_0 is initialized with the seed provided by the user. Each invocation of the random number generator will produce a new number X_{i+1} by transforming the last number in the sequence X_i using the following formula:

$$X_{i+1} = (AX_i + B) \bmod M$$

For example, the sequence obtained when $M = 10$ and $X_0 = A = B = 7$ is

```
7, 6, 9, 0, 7, 6, 9, 0 ...
```

Because the X_i sequence is computed modulo M, the numbers are generated in the range between 0 and $M - 1$. While it is possible to have a modulus that is not a power of 2, in this project we will consider that M is always of the form 2^m and we will specify m instead of M. If $m = 32$, the numbers are generated in the whole range on `unsigned` numbers on the Elms machines. In this case, the modulus operation can be omitted because arithmetic operations with `unsigned` operands are performed modulo 2^{32}.

All LC generators enter a periodic *orbit*, i.e. a cycle of numbers that are repeated, as seen in the example above. An RNG should have a period that is as long as possible; some applications require millions of random numbers that do not repeat. The longest possible period is the modulus M, in which case the orbit is a complete permutation: every number in the range is generated exactly once.

In this project, you will implement three algorithms, defined by the following parameters:

Algorithm	m	A	B
1	32	214013	2531011
2	32	1103515245	12345
3	31	1103515245	12345

These parameter choices ensure that all three algorithms have complete orbits.

To ensure that the state of the random number generator is preserved in between invocations, define four *global variables* (variables defined outside of any function, including `main()`), as follows:

```
unsigned X;          // Current value of RNG
unsigned m;          // Modulus exponent: M = 2^m
unsigned A;          // Multiplier
unsigned B;          // Increment
```

You will manipulate these variables in the functions you will implement, as described below, to provide the random number generation functionality.

5 Seed the random number generator

Implement a function with the following declaration:

```
void seed_rnd (unsigned seed, int algorithm);
```

This function should examine the value of the `algorithm` parameter and should initialize the `m`, `A`, and `B` variables with the corresponding values from the table above. The function should then seed the generator with the value of `seed`.

6 Print `RND_MAX`

Implement a function with the following declaration:

```
void get_rnd_max();
```

This function should print the `RND_MAX` that corresponds to the selected algorithm.

Hint: This value depends on the modulus used and is not necessarily the same as `RAND_MAX` from the standard C library.

7 Generate uniformly-distributed integers

Implement a function with the following declaration:

```
unsigned gen_rnd();
```

This function should return an `unsigned` number, from a random sequence generated using the LC method and the parameters that correspond to the selected algorithm.

8 Generate uniformly-distributed integers, up to a given limit

Implement a function with the following declaration:

```
unsigned gen_rnd_limit(unsigned limit);
```

This function should generate an `unsigned` number, by invoking the `gen_rnd()` function, and should transform it so that the maximum number generated is `limit` instead of `RND_MAX`. You may assume that `limit` $\leq$ `RND_MAX`.

Hint: Pay attention to the fact that `RND_MAX + 1` may overflow, for some of the RNG algorithms described above.

9 Generate uniformly-distributed integers, from a given range

Implement a function with the following prototype:

```
int gen_rnd_range(int min_gen, int max_gen);
```

This function should generate an `int` number, by invoking the `gen_rnd()` function, and should transform it so that the minimum number generated is `min_gen` and maximum number generated is `max_gen`. You may assume that `max_gen - min_gen` $\leq$ `RND_MAX`. If `min_gen` $>$ `max_gen`, return 0.

10 Generate uniformly-distributed floats, from a given range

Implement a function with the following prototype:

```
float gen_rnd_float(float min_gen, float max_gen);
```

This function should return a `float` number, from a random sequence of numbers between `min_gen` and `max_gen`. You can achieve this by invoking one of the functions described above and by transforming the numbers generated into a sequence of floating-point numbers X so that `min_gen` $\leq X <$ `max_gen`. If `min_gen > max_gen`, return 0.

Hint: Think about the case when `max_gen - min_gen > RND_MAX`.

11 Generate exponentially-distributed floats

Some applications require random numbers that follow other distributions than the uniform distribution. For example, if events occur continuously and independently at a constant average rate μ (e.g. a radioactive substance that emits one alpha particle every μ seconds, or a computer virus that infects a new host every μ milliseconds) then the time between two successive events has the *exponential distribution* with mean μ. The exponential distribution produces positive numbers, and the probability that a generated number is less than or equal to a given x is: $\Pr[X \leq x] = 1 - e^{-x/\mu}$.

To generate exponentially-distributed numbers on a computer, you first need to generate a sequence of random `float`s $U_0, U_1, U_2, ...$ that are uniformly distributed between 0 and 1. You can transform this sequence into a sequence $X_0, X_1, X_2, ...$ of numbers that are exponentially distributed with mean μ with the following formula:

$$X_i = -\mu \ln U_i$$

Implement a function with the following prototype:

```
float gen_rnd_exp(float mean);
```

This function should return a `float` number, from a random sequence that follows the exponential distribution with mean given by the `mean` parameter. You can achieve this by invoking the `gen_rnd_float` function and by transforming the numbers returned as described above.

Hint: You can compute the natural logarithm $\ln U_i$ using the `log()` function from `math.h`.

Hint: When U_i returns 0, you cannot apply the natural logarithm. Instead, substitute any convenient value ϵ for 0, as the probability of this case is small.

Testing the randomness of the numbers generated

Because the sequence of numbers produced by an LC generator only *appears* to be random, it is important to test their randomness. One of the most popular randomness tests is the χ^2 (read chi-squared) test, which determines whether the occurrence frequencies of the numbers generated are consistent with the uniform distribution.

To perform the χ^2 test, split the range of the numbers generated into 10 bins and count how many numbers *produced by the RNG* fall in each bin. Also compute how many numbers *would be expected* to fall in each bin; if the bins have equal sizes and the random numbers are uniformly distributed, the counts are expected to be the same for all bins. For each bin i, let o_i and e_i be the observed and expected counts of numbers that fall in the bin. Then compute the following quantity:

$$D = \sum_{i=1}^{10} \frac{(o_i - e_i)^2}{e_i}$$

If $D \leq 14.684$, the sequence is consistent with the uniform distribution. If $14.684 < D \leq 21.666$, the sequence looks suspicious. If $D > 21.666$, it is unlikely that the sequence comes from a uniform distribution.

12 Test the randomness of the uniformly-distributed numbers generated

Write a complete program, called `enee140_test_rnd.c`, that tests the randomness of a sequence of random numbers between 0 and 99, generated using the function described in Step 8. The program should read inputs that look like this:

```
10
11 58 97 20 3 46 9 0 35 98
```

The `enee140_test_rnd.c` program should start by reading one integer, which indicates how many numbers there are in the random sequence. The program should then read a sequence of random integers. If fewer numbers are provided than specified (by the first integer on the first line), the program should print the following error message (replace `XXX` with the number of integers expected and `YYY` with the number of integers you were able to read) and exit:

```
XXX numbers are required, but only YYY were provided.
```

If any number provided is outside the range specified the program should print the following error message (replace `XXX` with the incorrect number you read) and exit:

```
XXX is not in the [0, 99] range.
```

If there are no errors, perform the χ^2 test as follows:

- Each number generated falls into one of 10 bins:
  - Numbers between 0 and 9
  - Numbers between 10 and 19
  - Numbers between 20 and 29

  - ...
  - Numbers between 90 and 99
- Count how many of the generated numbers fall in each bin (o_i).
- If you are generating 1,000 uniformly-distributed numbers, 100 numbers are expected to fall in each bin ($e_i = 100$).
- Compute D as described above.
- Print one of the following messages, depending on the value of D:

  ```
  The sequence is consistent with the uniform distribution (D = XXX).
  ```

  or

  ```
  The sequence is suspicious (D = XXX).
  ```

  or

  ```
  The sequence is unlikely to come from a uniform distribution (D = XXX).
  ```

  Replace `XXX` with the computed value of D, using only two digits after the decimal point.

Hint: Perform the χ^2 on sequences of at least 50 numbers.

Hint: You do not need to store all the numbers you read.

13 10 bonus points: χ^2 test for the exponential distribution

The χ^2 test can be used to test whether a sequence of numbers is consistent with any distribution, not only the uniform distribution. Write a complete C program, called `enee140_test_rnd_exp.c`, that performs the χ^2 test on exponentially-distributed numbers, with mean 1, generated by the function described in Step 11. You can perform the test as before, but you will have to choose different bins (the numbers generated in this manner may be greater than 99) and you will also have to determine the expected counts for each bin (the counts for different bins will likely be different, as the numbers are not uniformly distributed in this case). It is also important to choose the bins so that the expected count for each bin is at least 5.

This question is optional; this is an opportunity for you to earn bonus points.

Testing your programs

Complex programs are more likely to have bugs. It is therefore important to test your programs thoroughly, using a broad range of inputs. A *test case* is a set of inputs for which you know the output that your program must produce (you may determine the output manually, by following the steps of the algorithm). A good programming practice is to write test cases *before* writing the program.

You can also use `enee140_test_rnd.c` to test `enee140_gen_rnd.c`. First, compile the two programs from the command line, as follows:

```
gcc -o enee140_gen_rnd -lm enee140_gen_rnd.c
gcc -o enee140_test_rnd -lm enee140_test_rnd.c
```

Then, generate uniformly distributed integers by invoking the first program as follows:

```
./enee140_gen_rnd | tail -n 1 >test_case.txt
```

The `tail` UNIX command prints the last line from the output of the first command, and `>` redirects the output from `tail` into a file. Note that, if you invoke the program in this way, the menu will not be printed (as the output from `enee140_gen_rnd` is redirected into `tail`), so you will have to remember the parameters that you must input to generate the desired sequence.

Open the `test_case.txt` file produced in a text editor and add the length of the sequence on a separate line before the random numbers generated. You can then invoke the second program as follows to test the randomness of the sequence:

```
./enee140_test_rnd <test_case.txt
```

Hint: Don't forget to test your programs with *incorrect inputs*, to see if they produce appropriate error messages.

Project requirements

1. You must program in C and name your program files `enee140_gen_rnd.c`, `enee140_test_rnd.c` (and, optionally, `enee140_test_rnd_exp.c`). Templates for these programs are included at the end of this document (you do not have to use them, but they may provide some hints).
2. Your programs must compile on the GRACE UNIX machines using `gcc enee140_gen_rnd.c` and `gcc enee140_test_rnd.c`.
3. Your programs must be readable to other programmers (e.g. Prof. Dumitraș, the TAs).
4. You must first submit a partial implementation by **Friday, 13 March 2026 at 11:59 PM**, then a complete implementation by **Friday, 3 April 2026 at 11:59 PM**.

Partial implementation

Your partial implementation must correctly print the menu and prompt the users for parameters, as described in Steps 1, 2, and 3. For each menu option, if you have already implemented the random-number generation functionality print the values as described above; otherwise, print a message saying "Functionality not yet implemented". Log into Elms, click on Gradescope in the course menu, then go to Project 1 (partial) to submit your work.

Complete implementation

Your complete implementation must implement all the steps described above correctly. Create a .zip archive containing the programs you wrote, then log into Elms, click on Gradescope in the course menu, then go to Project 1 (complete) to submit your work.

Grading criteria

Correctness:	80%
Good coding style and comments:	20%
Late submission penalty:	-40% for the first 24 hours -100% for more than 24 hours
Program that does not compile on GRACE:	-100%
Wrong file names (other than `enee140_gen_rnd.c`, `enee140_test_rnd.c`, `enee140_test_rnd_exp.c`):	-100%
Bonus (χ^2 test for exponential distribution):	10%

Program templates

You can start from the following templates (also available in the GRACE class public directory, at `public/projects/project1`).

Template for `enee140_gen_rnd.c`

```c
/*
 * enee140_gen_rnd.c
 *
 *   Generate random numbers, in different ranges and
 *   from various distributions.
 *
 */

#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>


/*
 * Public API -- Function prototypes
 */

void seed_rnd (unsigned seed, int algorithm);

void get_rnd_max();

unsigned gen_rnd();

unsigned gen_rnd_limit(unsigned limit);

int gen_rnd_range(int min_gen, int max_gen);

float gen_rnd_float(float min_gen, float max_gen);

float gen_rnd_exp(float mean);


/*
 * State variables of the RNG.
 */

unsigned X;           // Current value of RNG
unsigned m;           // Modulus exponent: M = 2^m
unsigned A;           // Multiplier
unsigned B;           // Increment
```

```c
/*
 * Main function
 */

int
main()
{
        return 0;
}
```

Template for `enee140_test_rnd.c`

```c
/*
 * enee140_test_rnd.c
 *
 *  Test 1000 randomly generated numbers for uniformity,
 *  using a chi-squared test with 10 bins.
 */


#include <stdio.h>



/*
 * Main function
 */

int
main()
{
        return 0;
}
```