1) Several data types above are “made up” types that others invented using “typedef”. To get the types right **look up the function prototypes**:
   a. In unix, try “man atoi”
   b. In an IDE they may pop up when you enter or hover over the function name, or you might highlight them and invoke help.

   When I search for “time” I get this (and more):

   ```
   time_t time ( time_t * timer );
   Get current time
   Get the current calendar time as a time_t object.
   The function returns this value, and if the argument is not a null pointer, the value is also set to the object pointed by timer.
   **Parameters**
   timer
   Pointer to an object of type time_t, where the time value is stored.
   Alternatively, this parameter can be a null pointer, in which case the parameter is not used, but a time_t object is still returned by the function.
   ```


   where I clicked through to “Random number generator initiation” to find:

   **Function:** gsl_rng * gsl_rng_alloc (const gsl_rng_type * T)

   This function returns a pointer to a newly-created instance of a random number generator of type T. For example, the following code creates an instance of the Tausworthe generator,

   ```
   gsl_rng * r = gsl_rng_alloc (gsl_rng_taus);
   ```

   The generator is automatically initialized with the default seed, gsl_rng_default_seed. This is zero by default but can be changed either directly or by using the environment variable GSL_RNG_SEED (see Random number environment variables).

2) Here is a program to **generate random numbers** and store them in a file:

   ```
   // Purpose: write a size and that many random normals to stdout
   // compile: g++ -c -Wall genGaus.cpp
   // link: g++ -o genGaus genGaus.o -l gsl -lgslcblas
   // run: genGaus size [mean=0.0 [sd=1.0]]
   ```
#include <iostream>
#include <ctime>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h> /* generating RVs */

/* declare the random generator as globally available */

int main(int argc, char *argv[]) {
    gsl_rng *rng = gsl_rng_alloc(gsl_rng_default); /* start the RNG */

    /* set a "random" seed */
    time_t seed = time(NULL);
    gsl_rng_set(rng, static_cast<unsigned long int>(seed));

    // Assure at least 1 argument
    if (argc<2) {
        cout << "usage: " << argv[0] << " size [mean [sd]]" << endl;
        exit(1);
    }

    // Interpret argument as "size"
    int size = atoi(argv[1]);
    if (size<1) {
        cout << "size must be at least 1" << endl;
        exit(1);
    }

    // Get other arguments
    double mean=0.0, sd=1.0;
    if (argc>2) mean = atof(argv[2]);
    if (argc>3) sd = atof(argv[3]);
    if (sd<=0.0) {
        cout << "sd must be positive" << endl;
        exit(1);
    }

    // Write size to stdout
    cout << size << "\n";

    // generate Gaussians and write to stdout
    for (int i=0; i<size; i++) {
        cout << gsl_ran_gaussian(rng, sd) + mean << "\n";
    }

    gsl_rng_free(rng); /* free the RNG */
    return(0);
}
3) **GSL random number generation**

   a. Functions that generate random numbers have an argument that is a pointer to a “gsl_rng” object. The `gsl_rng_alloc()` function creates such an object and returns a pointer to the heap memory location where the new object now lives. The argument to `gsl_rng_alloc()` is a predefined value selecting the RNG type. The predefined value “gsl_rng_default” selects the “gsl_rng_mt19937” generator unless the environmental variable GSL_RNG_TYPE is set otherwise.

   b. If you start generating (pseudo)random numbers immediately after creating your random number generator, you will get the same values each time you run your program. This is because an RNG really generates a fixed sequence of numbers starting at the “seed”. Normally you will want to set a “random” seed. Caution: if you repeatedly run a quick program and your computer’s time resolution is long (e.g., 1/60th second on many Windows PCs), you may get repeat runs with the same output. In the above code we use the typical practice: convert the current time to an unsigned integer and pass it to `gsl_rng_set` to get a “random” start to the RNG.

   c. Sometimes it is useful to be able to repeat a pseudo-random sequence. For example, in a simulation study of 1000 data sets generated and then analyzed, we could save the 1000 seeds in a small data file, and then when we find that something weird happened on run 234, we can regenerate that data set by setting that seed again, and then we can look for what unexpected problem our analysis program ran into.

   d. To generate random numbers from specific distributions look in the GNU GSL documentation, e.g.:

   ```
   double gsl_ran_gaussian (const gsl_rng * r, double sigma)
   ```

   This function returns a Gaussian random variate, with mean zero and standard deviation `sigma`. The probability distribution for Gaussian random variates is,

   \[ p(x) \, dx = \frac{1}{\sqrt{2\pi \sigma^2}} \exp \left(-x^2 / 2\sigma^2\right) \, dx \]

   for \( x \) in the range \(-\infty \) to \(+\infty\). Use the transformation \( z = \mu + x \) on the numbers returned by `gsl_ran_gaussian` to obtain a Gaussian distribution with mean \( \mu \). This function uses the Box-Muller algorithm which requires two calls to the random number generator \( r \).

4) **The gsl_vector package** is a better/safer way to work with vectors.

   a. The descriptions are at:

   ```
   ```

   b. These functions are written in C, not C++. Note the following quote from the “C++ compatibility” section of the manual:

   “The library header files automatically define functions to have extern "C" linkage when included in C++ programs. This allows the functions to be called directly from C++.”

   3
c. The main new data types you will want to use are `gsl_vector` which is a vector of doubles and `gsl_vector_int` which is a vector of integers. These are really structures containing:
   - The size of the data (called “size”)
   - The data itself as a standard C array
   - Other information

d. Normally you define `pointers` to these structures, e.g.,

   ```c
   gsl_vector * ptrToDVector;  /* Use better names! */
   gsl_vector_int * ptrToIVector;
   ```

   and then use

   ```c
   ptrToDVector = gsl_vector_alloc(N);
   ptrToIVector = gsl_vector_int_alloc(N);
   ```

   to dynamically allocate a vector of length N from the heap. You can return the memory to the heap using `gsl_vector_free(ptrToVector)` or `gsl_vector_int_free(ptrToVector)`. The pointer itself still “lives”, so you can reuse it to point to a new vector in the future with another `gsl_vector_alloc()` statement.

e. Since the actual vector code is not in the C++ standard library, you must link in the gsl libraries by adding `-lgsl -lglscblas` to the link command.

f. You may use `ptrToVector->size` to check the size of a vector at some future time.

g. `gsl_vector_get(ptrToVector, index)` is used to reference a value in the vector and `gsl_vector_set(ptrToVector, index, value)` is used to set a value. Corresponding functions for “int” also exist. These functions do range checking. To get a small speedup after full testing, you can turn off range checking by including `#define GSL_RANGE_CHECK_OFF` in any .cpp or header file or by adding `-D GSL_RANGE_CHECK_OFF` to the G++ compile runstring.

h. To pass the data in a `gsl_vector` to a function that expects a standard double array, use `gsl_vector_ptr(ptrToVector, 0)`.

i. Other useful functions:
   - Initialization
     - `gsl_vector_set_all(ptrToVector, value)`
     - `gsl_vector_set_zero(ptrToVector)`
   - Binary format read/write (ptrToFILE may be “stdin” or “stdout”)
     - `gsl_vector_fwrite(ptrToFILE, ptrToVector)`
     - `gsl_vector_fread(ptrToFILE, ptrToVector)`
• Text format read/write
  o \texttt{gsl\_vector\_fprintf(ptrToFILE, ptrToVector, "%f")}
    [use \texttt{\%d for int version}]
  o \texttt{gsl\_vector\_fscanf(ptrToFILE, ptrToVector)}

• Copy and swap
  o \texttt{gsl\_vector\_memcpy(ptrToVect2, ptrToVec1)}
  o \texttt{gsl\_vector\_swap(ptrToVectA, ptrToVecB)}

• Math operations
  o \texttt{gsl\_vector\_add(ptrToVec, ptrToOtherVec)}
  o \texttt{also sub/mul/div (result replaces 1st vector)}
  o \texttt{gsl\_vector\_scale(ptrToVec, multiplier)}
  o \texttt{gsl\_vector\_add\_constant(ptrToVec, constant)}

• Min/max
  o \texttt{gsl\_vector\_min(ptrToVec)}
  o \texttt{gsl\_vector\_max(ptrToVec)}
  o \texttt{gsl\_vector\_minmax(ptrToVec, ptrToMin, ptrToMax)}
  o \texttt{gsl\_vector\_min\_index(ptrToVec)}
  o \texttt{gsl\_vector\_max\_index(ptrToVec)}
  o \texttt{gsl\_vector\_minmax\_index(ptrToVec, ptrToMinIndex, ptrToMaxIndex)}

• Properties
  o \texttt{gsl\_vector\_isnull(ptrToVec)}
  o \texttt{gsl\_vector\_ispos(ptrToVec)}
  o \texttt{gsl\_vector\_isneg(ptrToVec)}
  o \texttt{gsl\_vector\_isnonneg(ptrToVec)}

• Dot product
  o \texttt{gsl\_blas\_ddot(gsl\_vector\_ptr(ptrToVecA, 0),
                              gsl\_vector\_ptr(ptrToVecB, 0), ptrToResult)}
    [#include <gsl/gsl_blas.h>]

5) **Command line (runstring)**

An example of a command line for the program is “genGaus 20 5.2 2.0”.

The arguments to main are: \texttt{main(int argc, char *argv[ ])}

“\texttt{argc}” is the count of the arguments (space delimited) including the program name.
“\texttt{argv}” is a pointer to (the first of) an array of character vectors. So \texttt{argv[0]} is a pointer to
the C-style string “genGaus”, \texttt{argv[1]} points to “20”, etc.

You can use, e.g., \texttt{atoi()} and \texttt{atof()} to convert a string of ASCII characters into an integer
or double.

You cannot change any elements of these arrays because they are declared “const”.


6) **Redirection of input and output**

Unix and command-line Windows or “shell()” in R allow a file or the output of another program to substitute for keyboard input and/or to capture terminal output. The syntax is:

```
myprog <altInput.txt >altOutput.txt
```

Also:

```
myprog1 | myprog2 > altOutput.txt
```

uses the output of myprog1 as the input for myprog2.

You can use, e.g.,

```
MCMC fname 10000 | tail -n20
```

to run MCMC with arguments “fname” and “10000” but limit the screen output to only the last 20 lines of your program output.

In R try `shell("dir>dir.txt")`.

Here we can use `genGaus 1000 10.5 > G1000.dat` to store our random numbers in a file.

7) **Operator precedence**