COMP 161 - Lecture Notes - 12 - Randomized Procedures

April 4, 2015

In these notes we discuss issues with designing procedures that utilize randomized data.

Randomness in Computing

Early on in the history of modern computing, people figured out that there was often some advantage to utilizing randomness as resource in the design of procedures. Monte Carlo methods[1] work by repeating a computation of a large sample of randomly generated inputs. Optimization algorithms often use randomness to escape a local minimum or maximum. In cryptography, randomness lends itself to stronger guarantees of security.

[1] http://en.wikipedia.org/wiki/Monte_Carlo_method

Of course even if there weren't some computational advantage to the use of randomness, we'd still like computers to have access to randomness as lots of real world problems and applications rely on randomness. Games that use dice or cards often require randomness. Random processes exist in nature and we might wish to simulate or emulate these processes. Sometimes we're not concerned with randomness as much as we are uncertainty. We can use probability to quantify uncertainty, but to simulate this uncertainty we'd need some level of randomness.

Whether it's a problem solving device or a part of a problem, we often need randomness in our programs. What we'd like is for our computer to have a fair coin. If it did, then any time we needed some randomness we could simply instruct the computer to flip the coin. Just like we've constructed a wealth of data from raw binary numbers, we can construct a whole host of interesting random phenomenon from a single coin. If you want a random *int* value, you could simply flip the coin 32 times. A random *char* would need 8 flips. In practice, we don't deal in random bits, but instead in random numbers[2]. So when we want randomness in our programs we look for libraries that provide access to a random number generator. Unfortunately, computers don't *really* have a built in random number generator.

[2] Typically random unsigned integers

The hardware on which modern computers are build is deterministic and therefore has no way to produce randomness[3]. We've found some clever means of harvesting the apparent randomness in nature, but these are often slow or hard to implement for general use[4]. Historically, people would employ tables of randomly generated numbers such that whenever a new number was needed they'd

[3] no coins in there

[4] http://en.wikipedia.org/wiki/Random_number_generation#Generation_methods

just grab the next one from the table. This practice of fetching a predetermined random value lends some intuition to the most common source of randomness in modern computers, the PSUEDO RANDOM NUMBER GENERATOR

Psuedo Random Number Generation

Imagine a giant book full of tables of seemingly random sequences of numbers. The book has a clear table of contents such that you can flip to any one table easily. To the owner of the book there's nothing really random going on here. The ith entry of the jth table is always the same. From this perspective this book is totally deterministic. We could simply write it to a computer chip and have the same data accessible in the same way we it was in the book.

Now consider the outside observer that does not have access to the book. Let's say they know the book's owner will read off one number from the book. Well there's no way to know what that number will be and given that the tables themselves contain random sequences, there's no way to predict the next number having heard one or more already. That is given the ith entry from a table, there's no way to predict the $(i+1)$th entry. In a perfect world we'd like there to be no way to predict any one entry given any number of of the other entries from the table.

Now lets put a computational spin on this. The combination of the book's owner and the book is one system such that the table number j and the current entry on the table i defines the *state* of the system. We can modify the state by choosing a different table or by reading off a number from the current table. Numbers are read in order, so when we read number i the next number to be read is $i+1$. In the event that we run out of numbers on the current table, we'll simply start over again and begin reading numbers from the beginning of the same table. This repetition is only a problem if our table size is smaller or near in size to the number of random numbers we expect to need[5]. Notice that given this book, we have a totally deterministic means of generating random numbers.

[5] repeating the table causes a pattern to form. patterns are antithesis of random

The problem is that for the book metaphor to work, we'd need to set aside a large amount of space for the tables. What we really want is some mathematical function that computes the tables. We need some $\{$ such that $\{(j,i)$ returned the ith number from the jth table. It's not the least bit obvious that we should even attempt to find such a function. It would seem like the ability to calculate the tables via some formula would imply a pattern in the numbers. Our imagined tables are random. They should have no pattern. Yet by the 40s people had discovered easy to compute functions that when given a

SEED value produced a stream of numbers that generally appear to be random to the outside observer. Given the same seed value, they'll continue to produce the same sequence, just like our tables. We call these functions PSEUDO RANDOM NUMBER GENERATORS, or PRNGs. A PRNG does not provide true randomness. Certain PRNGs do a much better job at obfuscating the underlying pattern than others. So, when simulating true randomness matters, then you'll need to known a bit more about the landscape of PRNGs available to you. For the things we're doing in this class, we're OK with whatever default PRNG the system provides and the level of randomness that PRNG offers to us.

Given a PRNG we have access to as many sequences of random numbers as there are possible seed values for the PRNG. This is great, except that we often don't want the exact numbers given to us by our PRNG and we have no way to directly modify the possible values our PRNG computes. To get around this we typically turn to statistics, which provides us with a whole vocabulary for describing the form and function of randomness in terms of DISTRIBUTIONS. So, the other thing we look for libraries to provide is a way of turning PRNG output into numbers from a specific statistical distribution. So, before we translate all of this over to C++, I want to plug your local Probability and Statistics course. Out best means of understanding and quantifying randomness is through probability and statistics. Making good use of randomness is becoming increasingly common in computing. It's well worth your time to familiarize yourself with the basic principles of probability and statistics.

PRNGs in C++

The changes to C++ brought about by the new C++11 standard included a new library called *random*. This library provides several different PRNGs and a set of procedures for converting PRNG output to well studied probability distributions, including uniform distributions. Prior to C++11 people used the C standard library PRNG and either wrote their own methods transforming its output to a specific distribution or sought out non-standard C++ libraries. Let's see how to use this library to get uniformly distributed random integers and doubles.

All PRNGs are represented as random number engine objects. To get an instance of one of these objects we declare a variable of type *std::default_random_engine*[6]. Like all variables, we'd like to initialize generator. To initializing a PRNG means passing it a seed value. The basic way to seed a PRNG[7] is to give it an unsigned integer. We can either do this when we declare the variable or after the fact. Here you

[6] Other generator types are available, but this will work OK for our purposes

[7] and the only way we'll explore

see both options.

```
std::default_random_engine generator; //default seed
generator.seed(5); //seed with 5

std::default_random_engine gen(0);  //seed value 0
```

The *min* and *max* methods allow you to determine the minimum and maximum value possible with the PRNG.

```
std::default_random_engine gen(0);

cout << "The default PRNG produces integers from [" << gen.min() << ","
     << gen.max() <<"]\n";
```

To extract a random number from the PRNG object we do something we've never done before- *we use the object as if it were a procedure*. Let's get and print 15 numbers, 5 per line.

```
std::default_random_engine gen(0);

for(int i{0} ; i < 15 ; ++i)

  cout << gen();

  if( i % 5 == 4 ){
    cout << '\n';
  }
  else{
    cout << ' ';
  }
}
```

Here we see the expression *gen()* being used to extract random numbers from our PRNG *gen*. This is weird for two reasons. First we expect the thing before a set of parenthesis to be a procedure name or control structure keyword[8]. In this case its the name of a variable. The second source of weirdness is the fact that the "procedure" takes no inputs.

[8] if, while, for

We can use our PRNG object as a procedure because C++ gives us the ability to treat parenthesis as operators[9]. We call this operator the APPLICATION OPERATOR. Object classes with a defined application operator are typically called FUNCTORS. Functors can be a bit confusing at first. They blur the line between data and procedures by virtue of functioning in both worlds. We'll get our feet wet with functors with the random library. Later we'll learn to write our own functors.

[9] operator() in the documentation

No argument procedures seem a bit strange coming from a purely functional world. However, considering what we know about PRNGs and our PRNG gen, we can start to piece together why this is a perfect case for a no argument procedure. When we constructed *gen* we seeded the PRNG. The user of *gen()* simply wants a random number. In terms of our book metaphor, it makes no sense to force them to choose a table[10] and entry number from that table. In fact, if they did, the process would cease to be random because they'd simply be selecting entry *i* from table *j* and table which is always the same. So, in the end, the seed and the "location" of the next entry should be hidden away from the user in order to ensure the interface we wanted all along-give me a random number.

[10] seed

Of course, we deterministically selected the seed value for or PRNG. This is a bit problematic. Every time we run our code to print 15 random values we'll get the same 15 values- the first 15 value in the sequence generated by seed 0. What would be great was if we generated a random seed, but that causes a bit of a paradox. Instead, we could settle for a different seed every time we run the program. You could keep the previous seed value in a file[11] and then when the program runs it reads that value from the file and then adds one to it. This would let you count through seeds. This approach is a bit cumbersome however. A better source for numbers that are different every time you run your program is time itself.

[11] Files are the only way to save program state from one execution to the next

The standard approach to generating unique seeds across program executions is to use the system time. Computers tend to see time as the number of milliseconds that have passed since some fixed point in time called the EPOCH[12]. What we want to do is get this number and use it as our seed. This means that unless we call our program more than once in the same millisecond, we'll have a different seed every time we run it.

[12] `http://en.wikipedia.org/wiki/Unix_time`

To get the time since the Epoch in C++ we can use the C++11 *chrono* library. By chaining together a few method calls we can get time since the epoch measured in milliseconds[13].

[13] we'll trace through these method calls in class. It's a great exercise in reading library documentation

```
// ms since epoch
unsigned int seed1 = std::chrono::system_clock::now().time_since_epoch().count();

// use time as seed
std::default_random_engine gen(seed1);

// do some random stuff
```

We did not use the curly braces for initialization because we're actually converting between *long int* and *unsigned int*. In this case, if the conversion changes the number, we don't particularly care. The

actual time isn't important, just the fact that it's different than the last time we ran the program. If you use the curly braces you'll get a narrowing conversion warning[14].

[14] go try it!

Getting the current time in milliseconds and initializing a variable with it is a bit cumbersome, we could break it up. While doing so, we could make use of the C++11 *auto* feature. This lets the compiler determine variable types for us. This is a good place for *auto* because the whole statement is pretty much boilerplate whenever we want to seed a PRNG and so the types are not essential to the problem. When using *auto*, it's typical to use assignment based initialization rather than curly braces.

```
// get the time
auto clk = std::chrono::system_clock::now();
auto now_in_ms = clk.time_since_epoch().count();

// create and seed the PRNG
std::default_random_engine gen(now_in_ms);
```

Using the current time gets us a pretty good approximation to a random seed and we'll use it for most applications. However, it's important to note the fact that we can pass an explicit seed value and get a specific sequence of numbers. We'll see that this is extremely helpful when it comes to testing code involving randomness.

Uniformly Distributed Randomness in C++11

When people tend to think of randomness they think of the UNIFORM DISTRIBUTION. A single flip of a coin and the roll of a die are classic examples of discrete, uniform randomness. You're just as likely to get heads as you are tails. On a six sided die, all six sides are equally likely. In terms of a PRNG, we're talking about getting some integer from the interval $[a, b]$ where all possible outcomes are equally likely. We can imagine something similar for doubles. The key difference is that there are, in theory, an infinite number of possible numbers in the interval $[0, 1]$. Setting that issue aside, what we really want is no favoritism between possible outcomes. In more formal terms, the probability of each possible outcome is the same, or uniform[15]. This is the essence of uniformly distributed possibilities and what most people probably associated with randomness.

[15] Probability is exactly what we're distributing when we talk about statistical distributions

Distributions in the C++ library work by first declaring a distribution object with the parameters of that distribution. Then, given a source of randomness[16], we can use that distribution object as a functor for retrieving random values from that distribution. In the case of uniformly distributed numbers from a set range, we first declare the

[16] a PRNG

distribution and supply the min and max value of the range. Here we see distributions for integers and real values.

```
// simulating a six sided die
std::uniform_int_distribution<int> distribution{1,6};

// getting random reals in [0,1]
std::uniform_real_distribution<double> distribution{0.0,1.0};
```

These types utilize TEMPLATES. This is another new and extremely useful feature of C++ that the random library exposes us to.

Recall that C++ allows us to use many different types for mathematical integers. The standard is the *int* but we could also use *long*, *short*, and *byte*. All of these also have *unsigned* versions as well. For real valued[17] numbers, we typically use *double*, but we could also use the less precise *float*. This means that if we want random integers, we must also choose the type C++ uses to represent these integers and similarly for reals. This is what templates do for you. The type *std::uniform_int_distribution<int>* provides uniformly distributed integers as *int* values where *std::uniform_int_distribution<short>* would provide them as short integers. Similarly *std::uniform_real_distribution<double>* provides real values as doubles as opposed to some other type. The sub-type[18] is called the TEMPLATE ARGUMENTS. The definition for the distribution has a TEMPLATE PARAMETER that acts as a type variable. What we're seeing is writing definitions with variable types and not just variable values.

[17] things you write with a decimal

[18] int in *std::uniform_int_distribution<int>*

We can now produce PRNGs and distributions. All that's left is to put the two together. In order to do this we use the distribution as a functor that takes a single argument, a PRNG. Let's generate and print 30 random numbers from 1 to 6[19] with 6 numbers per line.

[19] 30 rolls of a six sided die

```
// Setup the PRNG
auto clk = std::chrono::system_clock::now();
auto now_in_ms = clk.time_since_epoch().count();
std::default_random_engine gen{now_in_ms};

// Declare and initialize
std::uniform_int_distribution<int> dist{1,6};


for(int i{0} ; i < 30 ; ++i ){
  // dist(gen) produces a uniformly distributed random int from [1,6]
  cout << dist(gen);

  if( i % 6 == 5 ){
    cout << '\n';
```

```
  }
  else{
    cout << ' ';
  }
}
```

If the look of *dist(gen)* bothers you, then the documentation for the random class points you in the direction of some more mysterious[20] code that lets you hide passing of the PRNG and replace [21] with something that looks like the no argument PRNG call we saw in the previous section. The procedure *std::bind* is in the *functional* library.

[20] for now

[21] dist(gen)

```
auto clk = std::chrono::system_clock::now();
auto now_in_ms = clk.time_since_epoch().count();
std::default_random_engine gen{now_in_ms};
std::uniform_int_distribution<int> dist{1,6};

// dice now acts like dist(gen)
auto dice = std::bind ( dist, gen );

// roll 3D6 for your wisdom stat
int wisdom = dice()+dice()+dice();
```

Binding your PRNG to the distribution is certainly optional, but it is nice to hide the PRNG away. It also allows us to establish a uniform signature[22] for sources of randomness that we can later exploit for designing randomized procedures.

[22] void → number

Random Permutations and std::shuffle

Sometimes we don't want a sequence of random numbers, but instead we just want some set of data in a random order. The mathematical term for this is a RANDOM PERMUTATION. Since C++11, the library *algorithm* provides the procedure *std::shuffle* that, given a source of randomness[23], will reorder data in a collection.

[23] i.e. a seeded PRNG

The std::shuffle algorithm is a mutator. Like many algorithm library procedures, it makes use of ITERATOR objects, a type of object that acts as an abstract pointer to an element in the collection. We're currently working with strings and have been using the integer value of character indexes to "point" to specific locations in the string. Iterators to those locations do the same thing but do it with respect to the specific string in question. Things may make a bit more sense after seeing a few examples. For the following examples, you can assume that *gen* is a default PRNG.

The first argument to std::shuffle is an iterator to the first item to be shuffled. The second argument is an iterator to the item after

the last item to be shuffled. So if *first* is the first argument and *last* is the second argument, then std::shuffle will shuffle everything in *[first,last)*. The third argument to std::shuffle is a PRNG, or some source of uniform randomness.

We can use the procedure *begin* to get an iterator to the first item in a string[24] and *end* to get an iterator to just off the end such that the following call to std::shuffle sorts the whole string.

[24] or any C++ collection

```
std::string str{"hello world!"};

// shuffle all of str using gen as our source of randomness
std::shuffle(begin(str),end(str),gen);
```

If *begin*(*str*) and *end*(*str*) were integer indexes they'd be 0 and *str.length*().

To shuffle part of a string we can add and subtract from our iterators to step forward and backward, respectively, from the start and end of our string.

```
std::string str{"hello world!"};

// shuffle all but the first and last 3 characters
std::shuffle(begin(str)+3,end(str)-3,gen);
```

Randomized Procedure Design

The std::shuffle procedure and the use of the application operator[25] with distribution objects[26] gives us a good design cue for how to design a randomized procedure in C++. Both procedures use an external source of randomness passed as an argument to the procedure. The highly stateful nature of PRNGs is also a good indicator that our random number generating argument is passed by reference. This makes the design of randomized procedures similar to that of I/O procedures. Both designs utilize another resource in the system that is initialized outside of the procedure and then passed by reference to the procedure.

[25] operator()

[26] dist(gen)

There is another important advantage to passing the PRNG as an argument to the procedure: testing. If the PRNG is local to our randomized procedure, then we're most likely going to want to use something like the system time to seed it. This is fine for the actual application when we want the outward appearance of randomness. It also means the the randomness of the PRNG and the logic of what we're doing with random values are inseparable. If, instead, we initialize and seed the PRNG externally, then we can use a seed value

for which we know the behavior. We can save the first n values produced by the PRNG and evaluate the expected behavior of the procedure with respect to those values. Put another way, we can remove randomness from the equation and test the expected results in a completely deterministic way.

Let's look at the following procedure.

```
/*
  Simulate rolling n m-sided dice and get the
  total of all n dice.
  @param n number of dice
  @param m number of sides per dice
  @param prng instance of the system's uniform random number generator
  @return total from rolling n m-sided dice
  @pre prng has been seeded. m > 0.
  @post prng produced n new random numbers
 */
unsigned int ndm(unsigned int n, unsigned int,
 std::default_random_engine& prng);
```

To test this we'll pick a seed value, in this case 1, and generate some tables relative to this seed[27].

[27] The program RandTabMaker.cpp was used for this purpose and the data is in randData.txt

```
First 20 numbers from seed 1:
16807       282475249   1622650073  984943658   1144108930
470211272   101027544   1457850878  1458777923  2007237709
823564440   1115438165  1784484492  74243042    114807987
1137522503  1441282327  16531729    823378840   143542612

First 20 numbers from [1,4] with seed 1:
1  1  4  2  3  1  1  3  3  4  2  3  4  1  1  3  3  1  2  1

First 20 numbers from [1,6] with seed 1:
1  1  5  3  4  2  1  5  5  6  3  4  5  1  1  4  5  1  3  1

First 20 numbers from [1,8] with seed 1:
1  2  7  4  5  2  1  6  6  8  4  5  7  1  1  5  6  1  4  1

First 20 numbers from [1,10] with seed 1:
1  2  8  5  6  3  1  7  7  10 4  6  9  1  1  6  7  1  4  1

First 20 numbers from [1,12] with seed 1:
1   2   10  6   7   3   1   9   9   12  5   7   10  1   1   7   9
1   5   1

First 20 numbers from [1,20] with seed 1:
```

```
1   3   16  10  11  5   1   14  14  19  8   11  17  1   2   11  14
1   8   2
```

Now that we know what numbers the PRNG will spit out we can predict exactly what would happen with our procedure with a freshly seeded PRNG.

```
std::default_random_engine prng;

prng.seed(1);
EXPECT_EQ(0,ln12::ndm(0,6,prng));

prng.seed(1);
EXPECT_EQ(1,ln12::ndm(1,1,prng));

prng.seed(1);
EXPECT_EQ(5,ln12::ndm(5,1,prng));

prng.seed(1);
EXPECT_EQ(1,ln12::ndm(1,4,prng));

prng.seed(1);
EXPECT_EQ(2,ln12::ndm(2,4,prng));

prng.seed(1);
EXPECT_EQ(11,ln12::ndm(5,4,prng));

prng.seed(1);
EXPECT_EQ(1,ln12::ndm(1,12,prng));

prng.seed(1);
EXPECT_EQ(60,ln12::ndm(10,12,prng));
```

We can no work out the design for the procedure itself. We'll go with iteration and accumulate the total as we "roll" the dice.

```
unsigned int ndm(unsigned int n, unsigned int m,
std::default_random_engine& prng){

    std::uniform_int_distribution<unsigned int> d{1,m};

    unsigned int total{0};
    for( unsigned int i{0}; i < n ; ++i){
      total += d(prng);
    }
```

```
    return total;
}
```