COMP 161 - Lecture Notes - 13 - Randomized Procedures

Spring 2014

In these notes we discuss issues with designing procedures that have a randomized outcome.

Randomized Procedures

Computers are machines. Machines are predictable and *deterministic*. Thus far, all of our procedures have reflected this fact and we've been able to clearly and explicitly test them for correctness. Functions always produce the same output for specific input values. Effect-based procedures will produce the same effect given the same input.

Randomized procedures introduce the element of chance[1] and produce outcomes from a range of possibilities. This makes them trickier to test and requires us to figure out how to make the machine do something non-mechanical: choose at random. For these notes we'll consider a function to simulate the roll of a 20 sided dice.

[1] or at least the illusion of it

```
/**
 * rollD20 produces a number in [1,20] at random, i.e. carries
 *  out a d20 roll.
 * @param none
 * @return integer in [1,20]
 * @pre PRNG has been seeded with srand
 * @post none
 */
 int rollD20(void);
```

This all seems pretty straightforward albeit new. We don't need inputs, because the result is determined by the computers built-in randomization system called the *Pseudo-Random Number Generator*[2]. This system does not produce true randomness, only a mathematical approximation of randomness. Properly using the PRNG requires that we first *seed* it with a number. This number determines the initial state of the generator. Given the same seed over and over, the PRNG will produce the exact same value. Seeded once, and then used repeatedly, the PRNG will produce a sequence of seemingly random integers. So, seeding the PRNG is something we typically do once during the program and prior to calling our randomized procedure. Thus we list it as a precondition for our procedure. We'll return to the PRNG when we talk implementation, for now, let's talk testing.

[2] PRNG

Testing Randomness

Testing is all about verifying expectations. Our first problem is that our expected outcome for *rollD20* is not a specific number but a number from a range of numbers. That's OK. We can just test that our outcome is within that range as opposed to a specific number. In practice, we might write some special predicates to check this for us us, but we'll use simple boolean expressions for *rollD20* because as it's an easy range to test. Before we write the test itself, notice that this expression wouldn't work as a test expression:

```
rollD20() >= 1 && rollD20() <= 20
```

Each of the two calls to *rollD20* might produce different numbers! So, we'll need to first save the results of a single call to *rollD20* to a variable, then check that variable. Let's begin with a single test. The statement *srand(time(0))* seeds the random number generator.

```
TEST(rollD20,all){
  srand(time(0));

  int d20(0);

  d20 = rollD20();
  EXPECT_TRUE(d20 >= 1 && d20 <= 20);

}
```

At this point we have a general strategy for testing the outcome of random procedures: *test that the random result is in the expected range of possibilities rather than test for a specific outcome*. Clearly one test is insufficient though. How do we know it wasn't a fluke? How do we know it doesn't just compute that same value over and over? If we want a stronger assurance that *rollD20* works, then we should test it repeatedly. How many times is a question best answered with statistics, for now we'll just say that the more the better, and we'll just use 50 or more as a good number for now. Let's use a loop to repeat that test 100 times.

```
TEST(rollD20,all){
  srand(time(0));

  int d20(0);

  for(int i(0); i < 100; i++){
    d20 = rollD20();
    EXPECT_TRUE(d20 >= 1 && d20 <= 20);
  }

}
```

The gTest framework will now alert us if any of our 100 tests fail. Alternatively, we could use iteration to accumulate the results of our tests as a boolean.

```
TEST(rollD20,all){
  srand(time(0));

  int d20(0);

  // accum is true if all the tests run so
  //  far have passed
  bool accum(true);

  for(int i(0); i < 100; i++){
    d20 = rollD20();
    // use and to accumulate the new result
    accum = accum && (d20 >= 1 && d20 <= 20);
  }
  EXPECT_TRUE(accum);
}
```

Let's say our 100 tests all pass. What do we know? How correct is the procedure really? It turns out all we know is that when run 100 times, all our outcomes were in the integer interval $[1, 20]$. We don't know anything about which numbers we got, and so, we don't know if our procedure simulates a fair dice where all outcomes are equally likely. Testing for this is much tougher and requires some really interesting statistics. We also don't really know that our 101^{st} call will pass. Again, we could use statistics and find out that the odds of it failing after the first 100 passed are really, really low. For now, we'll have to be satisfied with knowing that we're unlikely to get a number outside of $[1, 20]$ from *rollD20*.

In the end, serious testing of randomized procedures requires knowledge of some really interesting probability and statistics. Randomized procedures are common enough in computing that seeking out this knowledge would not be a waste of your time. That being said, when we encounter randomized procedures we'll stick to the testing process we used with *rollD20*: verify that the procedure produces a result within the expected range of results on a large number or repeated executions. If we do this, then we're probably not too far off our goal.

Using the PRNG

All modern programming languages provide a mechanism for getting numbers from a PRNG. In most cases, you can get a random integer in $[0, max]$ where *max* is some system specified value. In others you can get a random double from the interval $[0, 1)$. Occasionally, the language will provide a library of different procedures based on these characteristics. The newer C++11 standard provides a library called *random*[3] that is probably worth checking out. These notes will use the traditional procedures provided by C libraries.

[3] `http://www.cplusplus.com/reference/random/`

The C standard library[4] provides the following procedures and constants:

[4] cstdlib

- *RAND_MAX* is a named constant for the maximum value returned by the PRNG.
- *rand* is a procedure for getting a value in the interval $[0, RAND\_MAX]$ from the PRNG.
- *srand* is the procedure used to seed the PRNG.

As we saw earlier, we'll also want to make use of a procedure from the the C time library[5].

[5] ctime

- *time* is a procedure for getting the current time, in milliseconds from 1/1/1970, as measured by the system.

Seeding the PRNG

As discussed earlier, to effectively use the PRNG, we must first seed it. Failure to do so will result in predictable, non-random results. Typically seeding happens once during the course of the program, and so, we put the call to *srand* at the start of our program's *main* procedure, and at the start of our gTest TEST blocks for our randomized procedure. To avoid reusing the same seed value every time you run your program we seed the PRNG with the current time.

```
srand(time(0));
```

That statement should show up near the start of your *main* procedure and at the start of your *TEST* for your randomized procedure.[6].

[6] this means both files must include ctime and cstdlib

```
int main(int argc, char *argv[]){
  // seed the PRNG
  srand(time(0));

  // the rest of the main procedure code
  return 0;
}

TEST(rollD20,all){
  //Seed the PRNG
  srand(time(0));

  // tests for rollD20

}
```

Using rand

The *cstdlib* procedure *rand* takes no inputs and returns a number in the interval $[0, RAND\_MAX]$. We'll explore how to use it to get the following:

1. A random integer in $[a, b]$ for $a < b$.
2. A random double in $[0, 1]$

Let's start with a way of getting our random integer from $[1, 20]$ that is simple to implement but not quite fair[7]. We'll then figure out a slightly more involved, but mathematically sound, way to get a fair die.

[7] not all the outcomes are equally likely

The remainder operator[8] is an easy way to restrict the output of *rand* from $[0, RAND\_MAX]$ to $[0, m)$ for any integer $m \geq 0$. For positive integers a and m, $a \% m$ will always be a number in the interval $[0, m)$. That set of integers are the only possible remainders when dividing a by m. The problem is we want $[1, 20]$ not $[0, 20)$ or $[0, 21)$ which we'd get with *rand() % 20* and *rand() % 21* respectively. The trick is to first figure out *how many possible outcomes we'll get* and then *generate that sized interval starting at 0*. In our case, there are 20 integers in $[1, 20]$ so we'd want to generate $[0, 19]$ or equivalently $[0, 20)$ using the expression *rand() % 20*. If, however, we wanted $[4, 12]$, then we'd need to generate 9 numbers so, $[0, 8]$ or $[0, 9)$ using the expression *rand() % 9*. In general, if $a \leq b$ then there are $b - a + 1$ numbers in $[a, b]$ which is the same number of numbers as $[0, b - a + 1)$, the interval we get from *rand() % b-a+1*.

[8] %

We now can generate the right number of random outcomes, but the numbers are shifted down to start at 0. All we need to do is shift them back up. If I take any number in $[0, 19]$ and add 1 to it, then I'm guaranteed to get something from $[1, 20]$. Putting this all together, we can generate a number from $[1, 20]$ with the following expression:

```
(rand() % 20) + 1
```

In general, we can get $[a, b]$ with *(rand() %* $(b - a + 1)$*)* + a.

If, you're not concerned about every possible outcome being equally as likely to occur, then this method works just fine. If however, you want fair dice, which is often the case when you're running a simulation that relies of equally possible random events, then this method is not what you should use. Sometimes you get fair results, sometimes you do not. It depends on the values of a and b. A better method for randomly generating something from the interval $[a, b]$ requires a random double from $[0, 1)$.

Pick a decimal valued number at random from $[0, 1)$ and multiply it by 20. What do you get? If your random number is 0, then you have 0. As that number gets closer and closer to 1 you'll get numbers closer and closer to 20 but never quite reach 20. So, given a random double r in $[0, 1)$ and some number n, $r * n$ would effectively give us a random double in $[0, n)$. If we then round these numbers down to the nearest integer, then we'll get random integers in $[0, n)$. From there we know how to get random integers in $[a, n + a)$. This method produces more uniformly random[9] results than using the remainder operator. Let's see what this looks like as C++ statements:

[9] meaning every outcome is equally likely

```
// get random [0,1)
double r = double(rand())/double(RAND_MAX + 1.0);

// get random [0,20), round down/truncate, then offset by 1 for [1,20]
int(r * 20.0) + 1
```

Notice that to get a $[0, 1)$ double value we took a random number from $[0, RAND\_MAX]$ and divided by $RAND\_MAX + 1$. This guarantees that we'll never get or exceed 1.0. The only other trick used was letting the natural int to double conversion to do our truncation, or rounding down.

Implementing rollD20

Let's do two implementations of *rollD20* one for each of our random integer generation techniques. First, the more mathematically sound version.

```
int rollD20(void){
 // get random [0,1)
 double r = double(rand())/double(RAND_MAX + 1.0);

 // get random [0,20), round down/truncate, then offset by 1 for [1,20]
  return int(r * 20.0) + 1
}
```

Now the less mathematically sound but probably OK for situations where you're idea of random is not precise version.

```
int rollD20(void){
return (rand() % 20) + 1;
}
```

You've now seen randomized functional procedures. The same principles apply to randomized procedures that produce side-effects. The difference is, as always, that we'll have to test for effect and not value. This means looking to see of one of the expected mutation or I/O effects occurred.

Random words and other things

We know how to get random integers and doubles from a given interval, but we're working on a program that requires random words, not numbers. The typical gut reaction to random word generation is to generate random letters in such a way that the end result is a word. This is very difficult to pull off. An easier way to get random words is choose a word at random from a collection of words. If we place n words in a file, each word separated by whitespace, then we simply do the following:

1. generate an integer i in $[1, n]$ at random
2. get the i^{th} word from the file

Getting the i^{th} word from a file will require either some iteration or recursion. If we had the words in an array or array-like word, then we could simply select the i^{th} word, which is at index $i - 1$, with *operator[]*.

So how do we test procedures generating random words? The same way you do for numbers. Check to see that the word you got is one of the expected words, then repeat that enough times to rule out dumb-luck. Comparing your word to your set of expected word can get pretty involved if your set is large. I suggest you consider a helper predicate function.

This all leads to a general strategy for random items, of any kind, from a finite collection of size n:

1. Enumerate the collection, i.e. assign each item a number from $[0, n)$
2. Generate a random integer from $[0, n)$
3. Get the item corresponding to that number

The trick is now coming up with items for your collection.