YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 427a: Object-Oriented Programming *Handout #4*

Professor M. J. Fischer *September 19, 2011*

Problem Set 2

Due before midnight on Wednesday, September 28, 2011.

1 Assignment Goals

1. To learn how to define a class.
2. To learn how multiple classes interact.
3. To learn how to use random numbers for simulations.

2 Problem

Many statistical problems are difficult to solve analytically, but they can be approximated through simulations. For example, suppose I toss a fair coin 100 times. The expected number of heads is 50, but because of statistical variation, the actual outcome of such an experiment might be 51 heads, or 45 heads, or even no heads at all! However, these outcomes are not equiprobable. Exactly 50 heads is slightly more likely than 49 or 51, but only a little bit. Below is a partial table of the probability of getting k heads in 100 tosses:

k	Probability
...	...
35	0.000863856
36	0.00155974
37	0.00269793
38	0.00447288
39	0.00711073
40	0.0108439
41	0.0158691
42	0.0222923
43	0.0300686
44	0.0389526
45	0.0484743
46	0.0579584
47	0.0665905
48	0.073527
49	0.0780287
50	0.0795892
51	0.0780287
52	0.073527
53	0.0665905
54	0.0579584
55	0.0484743
56	0.0389526
57	0.0300686
58	0.0222923
59	0.0158691
60	0.0108439
61	0.00711073
62	0.00447288
63	0.00269793
64	0.00155974
65	0.000863856
...	...

Often the question we're interested in is how likely is it that the outcome will deviate significantly from its expected value? For example, what is the probability that the number of heads in an experiment is less than 40? That, too, can be calculated from the complete version of this table—namely, add up the probabilities for k in the range $[0 \ldots 39]$. Doing so, one obtains the result 0.0176001. By symmetry, this is also the probability that the experiment results in more than 60 heads. Hence, with probability 0.9647998, the number of heads in 100 coin tosses will lie in the interval $[40 \ldots 60]$.

When the coin tosses are not independent, it becomes much more difficult to obtain analytical solutions, so simulations may be the only reasonable way to gain understanding of the behavior. For example, suppose the outcome of the next coin toss is influenced by the result of the previous toss. This could happen in the physical world, for example, if the device that tosses the coin is more likely to flip it over an even number of times than an odd number. Perhaps the outcome of the coin toss has probability $p = 0.6$ of being *the same* as the previous toss and 0.4 of being *the opposite*.

Assuming the coin starts with heads, then the first toss is heads with probability $p = 0.6$. The probability that the second toss is heads is then $0.6 * 0.6 + 0.4 * 0.4 = 0.52$. This is because there are two different ways for the second toss to be heads, namely, the two tosses are (heads, heads) and (tails, heads). These in turn are generated by (same, same) and (opposite, opposite) outcomes of the randomizing device, which have probabilities of $0.6*0.6$ and $0.4 * 0.4$, respectively.

It is not a priori clear how big a difference this should make on our problem of estimating the probability that an experiment of 100 flips will result in fewer than 40 heads. We would expect that the number of heads will be slightly greater, but perhaps not significantly more. On the other hand, if p is close to 1.0, then we can expect to get a long streak of heads at the beginning of the sequence, greatly reducing the chance of seeing at most 40 heads in all. The purpose of this assignment is to explore this question of how non-independence in the coin tosses affects the outcome of an experiment of repeated coin tosses.

In this assignment, you are to write a program to estimate the probability q that, in a sequence of n non-independent tosses of a coin, the number of heads is less than a threshold τ. Your program will do this by running a simulation consisting of some large number of trials, say $t = 1000$. Each trial consists of performing an experiment and seeing if the outcome is less than τ. If so, we say the trial is a *success*. Let m be the number of successful trials. You will then estimate q by the quantity $\hat{q} = m/t$, the fraction of successful trials.[1]

3 The Program

Your program should obtain the following parameters that define the problem:

- n, the number of coin tosses in an experiment;
- t, the number of trials to run;
- τ, the threshold;
- p, the probability that two successive coin tosses are the same;
- s, the initial seed for the random number generator.

It should then run t experiments, where each experiment consists of n coin tosses. Let m be the number of experiments where the outcome (number of heads) is less than τ. Your program should then compute and print out the estimator $\hat{q}$.

The parameters should be obtained through five command line arguments. Command line arguments are C-style strings and must be converted to the appropriate types before being used. All parameters are unsigned integers except for p, which should be converted to a `double`. You may use the C-style functions `strtol()` and `strtod()` in `cstdlib` to do the conversion, or you may create a `stringstream` for each parameter, and then use the stream input operator `>>` to convert the data. (See `stringstream` reference.)

[1] There are many different possible estimators for the parameter q. See Wikipedia page on Estimation Theory for an introduction to estimation theory.

This is a relatively simple program and could easily be written in an unstructured way in `C` . However, I want you to focus on using `C++` classes to cleanly model the various aspects of the problem.

3.1 Class `Coin`

You should define a class `Coin` that models a non-independent coin. A coin toss has two possible outcomes that should be represented by an `enum` type with constants `heads` and `tails`. The current outcome of the `Coin` should be stored in a data member that is initialized to `heads` by the `Coin` constructor.

Class `Coin` should have a member function `toss()` that simulates one toss of the coin, and a const member function `outcome` that returns the current outcome.

Let p give the probability that two successive tosses are the same. To generate a coin toss, you need to generate a biased random bit b, where $b = 0$ with probability p and $b = 1$ with probability $1 - p$. Then $b = 0$ means keep the same coin outcome and $b = 1$ means flip the current outcome. We delegate the problem of generating b to another class `RandBit` (see below). Class `Coin` has a data member `rb` that points to an instance of `RandBit`. Then each call on `rb->next()` returns the next bit in sequence.

Finally, class `Coin` should have a member function `print()` that prints a character string representation of the current coin outcome.

3.2 Class `RandBit`

The class `RandBit` is used to generate the sequence of "random" bits needed by `toss()`. The reason for making `RandBit` its own class is so that it can have two different modes of operation. In normal mode, it uses the system random number generator `rand()` to generate successive bits. In testing mode, it can read a sequence of 0's and 1's from a file. This allows unit test programs to control the bits that are used by `Coin` and makes the program deterministic. Which mode it operates in is determined at the time it is constructed.

The dependency parameter p is stored in a data member of `RandBit` for use in its normal mode of operation. To generate b, you should generate a random `double` r in the semi-open interval $[0 \ldots 1)$. Then if $r < p$ take $b = 0$ and otherwise take $b = 1$. To generate a random double, compute `(double)rand()/((double)RAND_MAX + 1.0)`. You will need to include `cstdlib` in your program.

One way to determine the mode to use is to have two constructors for `RandBit`. One constructor takes the double p as its argument and sets the class to operate in normal mode. The other constructor takes a file name as argument and sets the class to operate in file mode. Obviously, some provision must be made for the case that the file runs out of bits.

3.3 Class `Experiment`

You should define a class `Experiment` that models an experiment. It should have a data member n for the number of tosses in the experiment, initialized by the constructor. A member function `run(c)` should run one experiment using the coin `c` and return the number of heads. `c` will be a call-by-value parameter of type `Coin`. This makes it possible to run experiments with different kinds of coins, which is very useful for testing. By making the parameter call-by-value, one starts each experiment with a fresh copy of `c`, which will be in outcome state `heads`, assuming it has never been flipped.

3.4 Class `Simulator`

Finally, you should define a class `Simulator` that carries out the simulation. It should have data members to store all of the parameters defining the problem. It should have a member function `run()` that carries out the simulation, computes $\hat{q}$, and stores it in a data member. A const member function will get the value of $\hat{q}$. Successive calls on `run()` should rerun the simulation from scratch, but of course using different random numbers.

The main program should create a `Simulator` instance, seed the random number generator (using `srand()`), run one (or maybe more) simulations, and print out the results.

4 Testing

The real difficulty of this problem is not writing the code but verifying that it is correct. As stated above, your program reads in a bunch of parameters and prints out a number that you have no way of knowing whether or not it is correct. To make matters worse, the output depends on the results of a simulation, so one expects that rerunning the program with different seeds will produce different answers.

To have confidence in your answer, you will need to create unit tests to verify the different parts of your code. Some tests that you should make are with parameters for which you know what to expect. For example, if $p = 1.0$, then the sequence of coin tosses should consist of all `heads`, and if $p = 0.0$, then the sequence should alternate `tails`, `heads`, `tails`, `heads`, ... If $p = 0.5$, then the coin tosses are independent and unbiased. In each of these cases, it is easy to calculate the success probability q and make sure that the output of your program is reasonable.

Better testing can be done by controlling the sequence of random bits used in the simulation. To control the sequence of bits, put the test sequence into a file, and set the testing mode of `RandBit` to use them. Then you can verify directly, for example, that `Coin` is producing the correct sequence of coin tosses.

5 Deliverables

You should submit this assignment using the `submit` script that you will find in `/c/cs427/bin/` on the Zoo. Remember to submit the following items:

1. Source code and header files.
2. A `makefile` and any other files necessary to build your project. (If you're using Eclipse with the default settings, the makefiles will be found in the directory `Debug`.)
3. The output of several runs of your program using different seeds and different numbers of experiments.
4. Unit test programs along with test input and output that you used to verify that your program was operating correctly.
5. A brief report named `report.txt` (or any other common format such as `.pdf`) describing the design choices you made in implementing the required code extensions. You can also put any other information here that the TA should know about your program.