YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 427a: Object-Oriented Programming *Handout #5 (rev. 1)*
Professor M. J. Fischer *September 25, 2011*

Remarks on Problem Set 2

As is often the case in programming, the conceptually hard part of the project becomes the easiest part of the coding job, whereas the most difficult to code is the boring part. This problem is no exception. The conceptually interesting part, how to generate non-independent coin flips, is described in the original assignment and is not difficult to code. The hardest part to code and get correct is managing the several parameters that control the simulation and providing flexible means for testing the code.

1 Managing the parameters

The problem assignment specified that the program should accept 5 command line arguments: `n t tau p s`. Command line arguments are supplied to your program via the arguments to `main(int argc, char* argv[])`. The elements of `argv[]` are C-style strings and must be converted to the appropriate internal type. n, t, and τ are integers, p is a double, and s is an unsigned integer.

1.1 Converting the strings

There are two ways to do the conversion. One is to use the C-style functions `strtol()` and `strtod()`. The other is to use the C++ `stringstream` package. Each method has its advantages and disadvantages. In both cases, the most tedious part is the detection of bad data.

The `strxxx()` functions return a pointer through the second parameter that points to the last character read from the string. If it's not the null terminator character `'\0'`, then it means that the argument contained bad characters. For example, `strtol()` will convert "24q" to the integer 24 but will set its second argument to point to the "q" character which of course does not belong there. Using the simple `atoi()` would not allow you to detect this bad data.

A `stringstream` works just like any other I/O stream except that the characters go to and from a string rather than an I/O device. Thus, if `stringstream ss` contains the string "24q", then one could write `ss >> n;` to read the integer 24 and store it into the `int` variable `n`. As with a file read, the next character to be read from the stream is the "q". If the read has attempted to go beyond the end of the string, then `ss.eof()` becomes true, just as it would for a file. The bits `good()`, `bad()`, and `fail()` also have their usual meanings.

1.2 Extended command line

A small extension to the assignment is to make the seed parameter optional. If omitted, then the seed should be set using `time(0)`, the value of the system clock. This means that each time you run the program, you will get a different seed. This is convenient if you want to perform several runs and observe the statistical behavior. However, for repeatability, you want to be able to specify the seed. This extension lets you get both behaviors without recompiling.

The assignment talks about making `RandBit` operate in two different modes: normal mode and testing mode. It does not talk about how to invoke those modes. The presumption is that it operates in normal mode unless you do something special to put it in test mode. Of course you could have a compile-time switch and recompile, but another way to handle this is to have an optional file-name parameter at the end. If present, `RandBit` is put into test mode and it reads its bits from the file, a character at a time. If absent, then it operates as before.

Combining these two extensions, we now have the following command line argument pattern: `n t tau p [s [filename]]`. Thus, to specify the file name, you must also supply a seed, although in that case, neither *p* nor *s* will be used.

There are many other ways of using the command line to specify desired program behavior. Switches (arguments beginning with '`-`') are one way, but their presence can complicate command line processing.

One fairly simple use of a switch is to turn on debugging mode. Debugging mode can be used to turn on additional printouts, useful for tracing the operation of your program. If the first command line argument is "`-d`", then debugging mode is enabled; if the switch is absent, then debugging mode is turned off. The remaining arguments are processed as usual. This leads to the following allowable argument pattern: `[-d] n t tau p [s [filename]]`. In all cases, what appears within square brackets is optional.

1.3 Code structure

Where should the code that processes the command line arguments go? One answer is that it should go into `main()`. However, command line processing itself can get pretty long and complicated, and I like to keep `main()` short and simple. A better way is to have a `Simulator` constructor that takes the same `argc` and `argv` arguments as does `main()`. Then the simulator is constructed with the declaration

```
Simulator sim(argc, argv);
```

This doesn't have to be the only constructor in case one wants other ways of setting the simulation parameters, so it isn't limiting in the sense of precluding other possible extensions.

2 Enumerated type definitions

It is possible to put an `enum` definition inside of a class, and there is good reason to do so. Namely, it helps to avoid name collisions. For example, in this assignment, you are asked to define an `enum` type for the outcome of a coin toss. The natural way to do this is

```
enum Outcome heads, tails;
```

The question then arises where to put this definition. Because it is closely related with the `Coin` class, it naturally belongs in the same header file. However, if it is made as a top-level definition, then type `Outcome` and the enum constants `heads` and `tails` all have global scope.

One can instead put the enum definition inside a `public` section of the `Coin` class. As with other names declared inside of a class, they are accessible directly within the class.

From outside the class, they require qualification. For example, to test if the outcome of a toss of coin `c` is heads, one could write

```
    if (c.outcome() == Coin::heads) {...}
```

3 Memory cleanup

You might decide to give class `RandBit` two constructors – one for the normal mode, that takes the dependency parameter p as a parameter, and one for the test mode, that takes a string filename as a parameter. You might be in a situation where one of the constructors allocates dynamic memory using `new` and stores a pointer to it in a data member, say `x`. The other constructor, which doesn't use `x`, should set it to the null pointer `nullptr` (which is new in `C++` 0x).[1]

Now, the question is, how do you write the destructor? Even though a class can have multiple constructors, it can have only one destructor. In this case, it is perfectly all right to simply write `delete x` in the destructor. If `x` is set to null, `delete` will do nothing; otherwise, it will free memory as desired.

As usual, you should test your program with `valgrind` to make sure that it is free of memory management problems.

4 Sample output

Below is output from six runs of my program. The first three are for 10,000 trials with debugging mode turned off. The first is with an automatically chosen seed; the second with the seed 1234, and the third with the bits coming from file `bits.in`. This is a file with 1000 bits in it consisting of ten repetitions of the pattern: 40 0-bits, 1 1-bit, then 59 0-bits. These bits will cause the first 40 coin flips to be heads, the 41st flip will be tails, and then all succeeding flips will remain tails. Because there are only ten repetitions of that pattern, the file runs out of bits long before the 10,000 trials have completed.

The second three runs are for only 5 trials with the debugging mode turned on. Here you can see much better what is going on.

```
>  coins 100 10000 40 0.6
---------------------------------------------------------------
Michael J. Fischer
CPSC 427a/527a
Mon Sep 26 2011 18:05:28
---------------------------------------------------------------
number coin flips in experiment: 100
number of trials: 10000
threshold: 40
using random bits with dependency parameter 0.6 and seed 1317074728
Estimator = 0.0424
---------------------------------------------------------------
Normal termination.
```

[1]For compilers that do not support the new standard, you can use the old `C` macro `NULL` instead.

```
>  coins 100 10000 40 0.6 1234
---------------------------------------------------------------
Michael J. Fischer
CPSC 427a/527a
Mon Sep 26 2011 18:05:28
---------------------------------------------------------------
number coin flips in experiment: 100
number of trials: 10000
threshold: 40
using random bits with dependency parameter 0.6 and seed 1234
Estimator = 0.038
---------------------------------------------------------------
Normal termination.

>  coins 100 10000 40 0.6 1234 bits.in
---------------------------------------------------------------
Michael J. Fischer
CPSC 427a/527a
Mon Sep 26 2011 18:05:28
---------------------------------------------------------------
number coin flips in experiment: 100
number of trials: 10000
threshold: 40
taking bits from bit-file 'bits.in'
Insufficient number of bits in 'bits.in'

>  coins -d 100 5 40 0.6
---------------------------------------------------------------
Michael J. Fischer
CPSC 427a/527a
Mon Sep 26 2011 18:05:28
---------------------------------------------------------------
number coin flips in experiment: 100
number of trials: 5
threshold: 40
using random bits with dependency parameter 0.6 and seed 1317074728
Testing RandBit
1 1 0 1 1 0 1 0 1 0 0 0 0 0 0 1 1 1 0 1 0 0 1 1
1 0 0 1 1 0 0 1 0 0 1 0 1 1 1 0 1 0 1 0 0 1 0 1 0
0 1 1 0 0 0 0 1 1 0 1 0 0 0 1 0 0 1 0 1 0 1 1 1 0
0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 0 1 0 0 0 0 1 1
Testing Coin flips
T H H T H H T T H H H H H H H T H T T H H H T H
T T T H T T T H H H T T H T H H T T H H H T T H H
H T H H H H H T H H T T T T H H H T T H H T H T T
T T T T T T T H T H H H H H T T H H T T T T T H T

outcome=52
```

```
outcome=59
outcome=39
outcome=48
outcome=47
Estimator = 0.2
---------------------------------------------------------------
Normal termination.

>  coins -d 100 5 40 0.6 1234
---------------------------------------------------------------
Michael J. Fischer
CPSC 427a/527a
Mon Sep 26 2011 18:05:28
---------------------------------------------------------------
number coin flips in experiment: 100
number of trials: 5
threshold: 40
using random bits with dependency parameter 0.6 and seed 1234
Testing RandBit
0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0 1 0 1 1 0
0 1 1 1 0 1 1 0 0 0 0 1 1 0 1 1 0 1 0 0 1 0 1 0 0
1 1 1 1 1 0 0 1 0 1 1 1 1 0 0 0 0 1 0 0 1 0 1 1 0
0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 0
Testing Coin flips
H H H H H H H H T H H H H H T T T T H H T T H T T
T H T H H T H H H H H T H H T H H T T T H H T T T
H T H T H H H T T H T H T T T T T H H H T T H T T
T T T T T H T T T T T T H H H T T T T T T H T T T

outcome=46
outcome=55
outcome=47
outcome=45
outcome=53
Estimator = 0
---------------------------------------------------------------
Normal termination.

>  coins -d 100 5 40 0.6 1234 bits.in
---------------------------------------------------------------
Michael J. Fischer
CPSC 427a/527a
Mon Sep 26 2011 18:05:28
---------------------------------------------------------------
number coin flips in experiment: 100
number of trials: 5
threshold: 40
taking bits from bit-file 'bits.in'
```

```
Testing RandBit
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Testing Coin flips
H H H H H H H H H H H H H H H H H H H H H H H H H
H H H H H H H H H H H H H H H T T T T T T T T T T
T T T T T T T T T T T T T T T T T T T T T T T T T
T T T T T T T T T T T T T T T T T T T T T T T T T

outcome=40
outcome=40
outcome=40
outcome=40
outcome=40
Estimator = 0
---------------------------------------------------------------
Normal termination.
```