Problem Set 5

Due before midnight on Monday, November 12, 2018.

1 Assignment Objective

This problem set continues the development begun in Problem Set 4 of a simulator for a population of simple agents attempting to reach consensus on a choice value. The PS4 assignment handout describes two different such agent algorithms: *fickle* and *follow the crowd*. In PS4, you implemented a simulator for a collection of *fickle* agents.

In this assignment, you will add a number of new features to the PS4 consensus program. Just as in real-life, refactoring code to handle new requirements is easy in places and harder in others.

2 Assignment Goals

- Learn to use polymorphism.
- Experience the effect of refactoring a big class (`Simulator`) into two related but smaller classes.
- Learn how to explore a rich parameter space, and gain insight into the behavior of random processes.

3 Problem

Here is an overview of the new features and required changes:

1. `Agent` will become a pure abstract class. Recall that that means all functions are virtual, and all are abstract except for the virtual destructor. The public `Agent` functions are the same as before: `update()` and `choice()`.
2. `Fickle` is a new class publicly derived from `Agent`. It is pretty much the same as the `Agent` class from PS4. `Crowd` is another new class publicly derived from `Agent`. It implements the *follow-the-crowd* algorithm described in the PS4 assignment handout.
3. The `Simulator` of PS4 did two distinct jobs:
   (a) It set up the population of agents to be simulated.
   (b) It ran the simulation.

   In this assignment, you will separate these two tasks. A new class `Population` will create and manage the agents. The revised `Simulator` will take a `Population` reference as a parameter and run the simulation, as before, until consensus is reached.
4. `Population` will maintain the aggregated array of `Agent` that was previously in `Simulation`. However, since `Agent` is now the base class for two different derived classes, the array elements will be `Agent` pointers. Each agent will be created using either `new Fickle( val )` or `new Crowd( val )`, depending on which kind of agent

is desired. Here `val` is the initial choice value for that agent. `Population` will retain custody of all of these agents, so its destructor must take care to delete them all.

5. The method for constructing agents is different from PS4. Each agent is randomly assigned to one of the two concrete agent types, `Fickle` or `Crowd`. The initial value `v` is randomly chosen from the set $\{0, 1\}$. Both of these random choices are biased according to new command line parameters. `probFickle` is a real number in the semi-open interval $[0, 1)$ and specifies the probability that an agent is chosen to be of type `Fickle` rather than of type `Crowd`. `probOne` similarly is a real number in the range $[0, 1)$ and specifies the probability that an agent's initial choice is 1 rather than 0.
6. `Population` has several public functions in addition to constructors, destructor, and print:
   (a) `int size() const` returns the number of agents.
   (b) `void sendMessage(int sender, int receiver)` simulates a single communication step from sender to receiver.
   (c) `bool consensusReached()` returns `true` iff consensus has been reached.
   (d) `int consensusValue()` returns the consensus value if consensus has been reached; otherwise it returns -1.
7. The `Simulator` constructor now only takes a single parameter of type `Population&`. Its run function has signature `void run()`. To obtain the results of the simulation, the caller can call two new public functions: `numRounds()` and `consensusValue()`. Since the simulator is doing the simulation, it knows how many rounds it has used. On the other hand, only `Population` knows the consensus value, so this is a case where delegation should be used.
8. `main.cpp` changes considerably. It takes different command line arguments and it prints different output than PS4.
   (a) The new command line arguments are

   ```
   numAgents probFickle probOne [seed]
   ```

   where `seed` is optional as before. `numAgents` is again the total number of agents. `probFickle` and `probCrowd` are the probabilities discussed in item 5 above.
   (b) All output should go to `cout`. `banner()` and `bye()` should be used as usual. The output from a run has three parts: The initial parameters, the statistics of the population after the random generation process, and the results of the simulation. See `sample.out` for an example of the new output format.
9. The resulting executable file should be called `consensus2` to distinguish it from the PS4 command name.

An important part of this assignment is to test your program on reasonable test cases, to submit the test case inputs and corresponding outputs, and to report on what you observe. For example, you should run your code on extreme cases such as 0 agents, 1 agent, probabilities of 0.0 and 1.0, and so forth.

Try to gain some insight about the extent to which *follow-the-crowd* agents do better than *fickle* agents. For example, what do you observe with a modest size population (say 1000) with different values for `probFickle`, say, 0.0, 0.01, 0.5, 0.99, 1.0.

4 Programming Notes

1. `random()` is now used in two different parts of the code.
   (a) `Simulator::run()` uses `uRandom()` as before in order to choose first a sender and then a receiver for a communication step. `uRandom()` of course is based on `random()`.
   (b) The `Population` constructor needs random values when constructing each agent. First it uses randomness to choose an initial choice value for the agent. Then it uses it to choose the agent type. In both cases, it chooses a `double` in the semi-open interval $[0, 1)$ and compares that number with the desired probability in order to make its decision. For example, to generate the choice value, test if the random number is less than the desired probability of choosing 1. If it is, then choose 1, else choose 0.
2. To choose a random real in $[0, 1)$, you can use my code

```
double Population::
dRandom() {
  return random()/(RAND_MAX+1.0); // result is double in [0,1)
}
```

   By default, the type of "1.0" is `double`, so the coercion rules force the addition and then the division to both be performed using double arithmetic. If you change "1.0" to "1", it won't compile without warnings, and it won't work correctly.
3. In order to duplicate my output, you will need to use the random number generator in exactly the same way as my program does. In particular, you will need to choose the sender before the receiver, and you will need to choose the initial consensus value for an agent before choosing the agent type. Of course, you will also need to start with the same seed.
4. The submission guidelines are the same as in previous assignments. Submit all files needed to compile your project along with a `Makefile`. Include a `notes.txt` file, a file of sample inputs and a file of the corresponding outputs.
5. Note that the grading rubric for this assignment puts more emphasis on good design, good style and good choice of test data than the previous assignments.

5 Grading Rubric

Your assignment will be graded according to the scale given in Figure 1 (see below).

#	Pts.	Item
1.	4	All relevant standards from previous problem sets are followed regarding submission, identification of authorship on all files, and so forth. A well-formed `Makefile` or `makefile` is submitted that specifies compiler options `-O1 -g -Wall -std=c++17`. Running `make` successfully compiles and links the project and results in an executable file `consensus2`. Each function definition is preceded by a comment that describes clearly what it does.
2.	3	Sample input and output files are submitted that show good coverage of the parameter space, e.g., small inputs, large inputs, edge cases for the probabilities (e.g., 0.0 and 1.0) as well as reasonable intermediate cases.
3.	5	The program shows good style. All functions are clean and concise. Inline initializations, inline functions, and `const` are used where appropriate. Variable names are appropriate to the context. Programs are consistently indented according to the course indenting style. Each class has a separate `.hpp` file and, if needed, a separate `.cpp` file. However, it is acceptable to group the three polymorphic agent classes together in the same `.hpp` and `.cpp` files.
4.	2	Everything is private in all classes except for the specified public interface, any needed special functions (constructors, destructor, move and copy constructors and assignments), and functions needed for debugging.
5.	6	All of the functionality in section 3 is correctly implemented.
	20	Total points.

Figure 1: Grading rubric.