COSC 1252-1254 Programming using C++
Assignment 2

Deliverable #2 **(15 marks)**

Due date: September 9, 2012; 23:59hrs
Late submission: September 14, 2012; 23:59hrs

1 Problem

Having presented the previous deliverable, Parasol Co. wants to test in a simulation that a survivor (or a set of survivors), given certain characteristics, can escape a danger zone. For this, they have designed the second deliverable for you to implement and test, following the indications below.

2 Second deliverable

2.1 Survival Rate Calculator

- Parasol Co. wants you to make an actual object-oriented implementation of the survival rate calculator implemented in the previous deliverable. For this, they require you to implement three classes: `Survivor`, which contains attributes and methods associated to a survivor in a scenario; `InfestedArea`, which encases attributes and methods for the location where the survivor is located; and the `Simulator`, which reads inputs and performs all the calculations involved in this deliverable. (NOTE: if you have implemented a similar mechanism in the previous deliverable but with different class names, skip this step). Having in mind this organisation, distribute the attributes and methods from the previous deliverable as you consider appropriate for an OO application.
- You must extend the `Survivor` class with three new attributes: `name` of type `string`, `healthLevel` and `weaponPower`, both of type `int` and restricted to the range 5–10.
- There are three types of survivors, as described below.
  - The basic `Survivor` (identified by letter `S`, note it is capitalised) implements the original values input by a user.
  - A `TrainedSurvivor`, which is a subtype of a `Survivor`. This subtype (identified by a `T`) has the same attributes than a `Survivor`, but its values are calculated differently, as follows:

$$stamina = input + 15$$

$$behaviourUnderStress = 0.9 \times input$$

where *input* is the original value introduced for that attribute.

- Create another subtype of `Survivor` called `WeakSurvivor`, which is identified by a letter `W`. A weak survivor has different effects in its attributes as follows:

$$ability = 0.7 \times input$$

$$behaviourUnderStress = \frac{input}{4}$$

Again, *input* is the original value introduced for that attribute. For both subtypes of `Survivor`, have the *getter* functions overriding the original functions from their base class. NOTE: be mindful of not allowing these values to exceed the maximum limits established! If this happens, assign the maximum available value and continue executing the program.

- Extend the original `InfestedArea` class with two more attributes: one is `pathDistance`, which represents an integer with a value between 10 and 50 measured in kilometers; the other is `zombieStrength`, which has a value within the range 5–11.

2.2 Simulator

The next part of the deliverable is a simulator that, given an `InfestedArea` and a type of `Survivor`, determines if the survivor can leave the area alive. The mechanics of the simulation are the following:

A survivor (either trained or untrained) has to traverse an infested area, which is represented by a linear path, in order to get out of danger. To advance in the area, the survivor moves a random number indicated by the following equation:

$$stepsAdvanced = random(1-3) \times \frac{updatedStamina}{maxStamina}$$

(remember that, from previous deliverable, $maxStamina = 100$). The value of *updatedStamina* is calculated as:

$$updatedStamina = stamina \times \frac{currentHealth}{healthLevel}$$

While traversing the path, survivors can encounter zombies: the probability of this event to happen is calculated after each *stepsAdvanced* turn. This probability is calculated using the following equation:

$$encounterRate = random() \times \frac{populationSize}{maxPopulationSize} \times stepsAdvanced$$

(remember that, from previous assignment, $maxPopulationSize = 500$ and, in this case, the random number is within the range $0-1$). A value above 0.5 for *encounterRate* means that the survivor has encountered a zombie, while below 0.5 means that the user has safely traversed the *stepsAdvanced* of the path.

If a survivor encounters a zombie, there is a chance that the survivor is either hurt or capable of beating the zombie. This is determined by two calculations: `survivorAttack` and `zombieAttack`. These values are calculated as follows:

$$survivorAttack = random() \times (weaponPower - 0.3 \times ability)$$

$$zombieAttack = random() \times zombieStrength$$

In the case that $survivorAttack < zombieAttack$, then the survivor loses $zombieAttack - survivorAttack$ units of *healthLevel*; otherwise, the survivor keeps advancing the path. The simulation continues until either the survivor has travelled the full distance in the area, or the survivor has lost all of its *healthLevel* units

If a survivor loses all its health units, the simulation must report the distance travelled by the survivor before losing all of its health. If a survivor traverses all the distance of the Environment, then it has succeeded and all the pertinent values must be reported.

2.3 Requirements for the simulation

- Define two types of `InfestedAreas`: one called `Pub` (identified by a `P`) and another called `Hospital` (letter `H`). In a `Pub` area zombies are stronger, thus every *zombieAttack* is boosted by 1 unit. In a `Hospital`, if survivors advance five steps without encountering a zombie, they recover a unit of health (NOTE: don't exceed the maximum value input during a simulation definition!).

- Your simulation must take into account multiple survivors. This will be done via input from a file. When executing your program, you must call an input file as following:

  `./simulator -f inputfile.txt`

  An input file must have the following format:

```
% the number of survivors
3
% attributes of the infested area:  type, populationSize,
% areaSize and zombieAttack
P 400 30 6
% attributes of each survivor per line:  type(S/T/W),
% name, healthLevel, weaponPower, stamina, ability,
% behaviourUnderStress and luckFactor(true/false)
T Shaun 8 7 80 0.7 0.1 true
W Philip 6 5 30 0.4 0.87 false
S Yvonne 9 8 60 0.55 0.52 true
```

  This assumes that all survivors will run under the same `InfestedArea`. NOTE: Your program must check if the input file is in the correct format;

in the example above the separator is a blank space. If the number of parameters per line (e.g. ! = 8 for survivors) or the format of a parameter is unexpected, close the program by notifying in which line the error was detected. It is not necessary to use exceptions for this, a simple boolean flag will do. After each simulation, your program must display the following information (for the sake of space, only two survivors are displayed):

```
./simulator -f inputfile.txt

Infected area          Pub
Population size        400/km2
Area size              20 kilometers
Zombie strength        7

Number of survivors:  3
---SURVIVOR #1---
Trained survivor ID: Shaun
Max.  health           8
Weapon                 7
Stamina                95   %note the difference!
Ability                0.7
Behaviour under stress 0.09 %note the difference!
Luck factor enabled

The survival rate of Shaun is 0.7212 (±0.0830)
After traversing 20 kilometers, Shaun
got out safely from the Pub!
---SURVIVOR #2---
Weak survivor ID: Philip
Max.  health           6
Weapon                 5
Stamina                30   %note the difference!
Ability                0.28 %note the difference!
Behaviour under stress 0.2175
Luck factor disabled

Infected area          Pub
Population size        400/km2
Area size              20 kilometers
Zombie strength        7

The survival rate of Philip is 0.3966
After traversing 8 kilometers on the Pub, Philip
became a zombie!
```

NOTE: See the differente between some values input from file and values displayed in the output, which change due to the types of survivor! The output must be written onto a file, by default called `output.txt`, located in the folder containing the executable file.

- Finally, you must include a class diagram that illustrates the organisation that you defined in your assignment. Only define attributes and methods for the core classes defined at the start, that is `Survivor`, `InfestedArea` and `SurvivalRateSimulator`.

3 General Requirements

- Your deliverable must be submitted using Blackboard. For a guide on how to upload assignments, refer to the next section.
- After the due date, you will have 5 *business* days to submit your assignment as a *late submission*. Late submissions will incur a penalty of 10% per day. After these five days, Blackboard will be closed and you will lose ALL the assignment marks.
- ONLY include the source code of your program, along with its corresponding `Makefile`. No object nor executable files should be submitted. Also, stick to the guidelines specified above (e.g. the name of the executable file). Failure in sticking to the standards will automatically deduct you 2 marks.
- Stick to proper coding styles and OO design principles using C++. Be careful on what you implement, as many solutions available online (even from C++ specialised forums) use C methods!
- You must properly name your attributes, methods and classes, as well as comment your code. As a minimum requirement, your classes, methods and attributes should be properly described in the code. This has a weight in the marking!
- If you made any adjustment, like changing the parameter separators from the input file, include a README file with your assignment describing in detail your concerns (alternatively, use the "comments" section when you submit your assignment via Blackboard). If such information is not included and your assignment cannot be run, you will NOT have a chance to ask for remarking. Also, remember that the markers must NOT edit any line of code in your files to run your program. So avoid hard-coding paths!
- Your assignment must primarily run on any computer from the Sutherland lab (see next section). After compiling, your program should not return any warning or error. Failure in these requirements will automatically deduct 20% out of your assignment marks.

4 Compilation and Execution Environment

You are free to set up your own environment on your desktop/laptop and compile your code from home. However, your source files must compile on any of the Linux machines in the Sutherland lab (the one where most of the tutorials take place) by simply typing `make` in the top directory of your project. What this means is that you should restrict yourself to any libraries available on the Sutherland machines.

Each source file should be compiled using the following flags: `-ansi -Wall -pedantic`. You can connect to the Sutherland machines from *Yallara* using the following command:

```
ssh ndsusername@sutherlandXX
```

and then enter your nds (Novell) password, which is the same as the password you use to login to Yallara; `ndsusername` is your RMIT nds username and XX is a number between 01 and 18 (corresponding to the enumeration of computers in this lab). Note that you cannot ssh to the sutherland machines from outside the university. You can of course come into the sutherland lab (where your tutorials are normally scheduled) and login and compile that way as well.

Copying your files from yallara to Sutherland

You can copy your files from Yallara to any of the Sutherland machines in the following way. Assuming that your source files are on Yallara in a directory called `a2` in the current directory. We can copy the `a2` directory and all files inside it using the following command:

```
scp -r a2 ndsusername@sutherlandXX
```

Please keep in mind that your Sutherland home directory forms part of your overall quota and is not additional space. That is, by storing files on the Sutherland space you will have less space for storage on Yallara!

5 Submission

Click on the Assignment tab in Blackboard, then click over the "Assignment 2" link (an orange link, do not click on the PDF with the specs!).

6 Further Information

Further enquiries about this assignment should be raised in appropriate discussion board in Blackboard. Do not contact lecturers/tutors/lab assistants via direct email, as these will be disregarded with the exception of personal issues; those have to be addressed to both the Head Tutor and Lecturer.

7 Plagiarism

The assignment that you submit must be your own work. No marks will be given for pieces of code that have not been created by you. Plagiarism is a serious offence at RMIT University. This includes direct copy of code from the Internet, other students or resources without a proper reference. While working in groups for an assignment is valid and encouraged, the submission of similar files is considered plagiarism as well. Having similar ideas is not a justification for not creating your own assignment files. Before marking, the markers will run plagiarism-detection tools. Plagiarised assignments will incur in penalties that range from assingment invalidation, to automatic fail of the course or even being expelled from the School if incurring for the second time. Be careful!