Advanced Programming 1 - 89-210 - Ex1

November 13, 2016

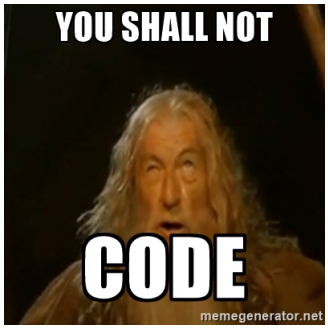
Due date: 27.11.16

Part I
System Design

General

In this project, we will create a system for managing Transportation. We will store information about our drivers, from personal information like age, professional information like years of experience.

In the first exercise, we will focus on designing and future planning.



(at least for the first part)

System Characterization

1. Every driver must have (at least) the following information: a driver id, age, marital status (from: SINGLE, MARRIED, DIVORCED and WIDOWED), years of experience, average satisfaction (based on averaged user experience - each driving score can take values between 1 to 5), taxi cab information (which is, an instance of the taxi). The creation of the driver doesn't involve also a cab. It will be attached to the driver later. The driver start as Tabula Rasa - 0 satisfaction and 0 customers. The rest of the information is part of the initializing part (as part of the input).

2. Each trip information should consist of (at least): a ride id, total meters passed at the current point, starting and ending point, number of passengers and tariff. The current meters needs to be updated on every point which the car moves, starting by 0. Every block (will be discussed later) is one meter.

3. Two kinds of cabs:

   (a) Standard cabs: cab id, number of kilometers passed, car manufacturer (from those: HONDA, SUBARO, TESLA and FIAT) and color (from: RED, BLUE, GREEN, PINK and WHITE) and tariff.

   (b) Luxury cabs. The same as above, they can drive faster and are more expensive. Those cabs move twice as fast, so every movement of those cabs are 2 blocks (excluding the case where the distance from the last block is one block away).
   The tarrif of the cabs, is the coefficient of the trip.

4. Passenger - will contain a source and destination point. Will have a method which produces a random score between 1 to 5 at the end of the trip which represents the satisfaction of the pasnegers regading the ride.

5. Taxi center, which contains the information about all its' employees (drivers), their location, the cabs and the trips. The taxi center should be able to answer calls from customers (part of the input that can arrive during the flow of the program), and send them a taxi, as fast as they can.

6. You will get a map (2D form - <x,y>) in which your world will be represented. The map is a part of the input of the program. It is exactely the same map for all drivers. You should have some logic, held by a class which will parse the input, and create the required map for all drivers.

7. We would like to be able to answers some statistics. Current answers:

   (a) Each driver location

8. You will need a class which will handle the main flow, parse the input and translate it to classes creation. The exact input is currently not important (and would be given to you in the following exercises), you only need to keep this in mind, and to design this class general flow.

Things to notice when planning

- Things always change, so plan accordingly.
- This is a classic object oriented environment, you are already familiar with some design patterns - use them!
- The fact you're not too proficient with the language is not important. You program for already one year (at least) and should be able to design a good environment even if you still don't know how excately to program it.

System operations:

- Getting the world representation grid (2 dimensional coordinates). It will get (m, n) parameters for the city size, and a list of points <x,y> which represents obstacles insize the map. Those obstacles can not be passed by taxi cabs
- Adding a new driver
- Adding a new taxi
- Connecting between a driver and a cab
- Receiving a taxi call, and assigning it to the closest driver
- The receiving driver, should be able to calculate the best way to arrive to his passenger destination
- Printing a driver location

Part II
Code

For getting warmed up, you will write some code. This code will be used in the following exercises as well.

Task 1

Write a Point class. The world we're speaking of is two dimensional, so we could initialize a point in that world like this:

```
Point p = Point(0,0);
```

The two variables will be only integers. The first parameter represents the *x axis*, and the second: *y axis*.

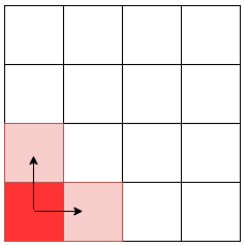
Task 2

Write the BFS Algorithm which will search a path using the Point class we've just created.

You're allowed to use any data structure which comes within *std* (e.g. *vector*, *list*, *deque*, *stack* etc...) but not any external library which alreay implements BFS.

You can read about those here: `www.cplusplus.com/reference/stl`

and a reminder of the BFS algorithm: `https://en.wikipedia.org/wiki/Breadth-first_search`



This is an example of a grid. The starting point is the red box, and the pink boxes are its' neighbors. The way that search algorithms travels a grid is the same as in graphs. Only here, the neighbors are the adjacent grid points excluding diagonals (actually it depends on the system , in our exercise, it will indeed without the diagonals points). For example, if we start at the red box (0,0) the neighbors are (0,1) and (1,0). So actually it's not any different from what you already know about search in graphs. Your BFS algorithm will get two points: source and destination, and will need to navigate through the grid to reach the destination point.

Submission demo:

How to design the classes etc. is up to you, and for the input output:

The parameters of the grid and the grid source and destincation point will be given by the input (the first thing to do is to get them). You will first get the size of the grid, then starting and ending points. You need to print the path from the source point to the destination (including). The format is the following:

```
sizeX_sizeY,startX_startY,endX_endY
```

For Example:

```
$./bfs.out
(input:)
3_3,0_0,2_2
(output:)
(0,0)
(0,1)
(0,2)
(1,2)
(2,2)
```

and there are example files as part of the exercise.

Implementation Notes:

- From every point, you have at most 4 neighbors, and at least 2. Yes, this means you can not go in diagonal directions.
- Although you are asked only for two basic classes implementation, note that a Point is a very basic object, and is not aware of it's location in space (the grid). This is a heavy clue for implementing another class with some kind of relation to the basic Point class.
- The printing operation has to be done using operator overloading of objects (« overloading).
- Even that in our exercise we will be dealing with a 2D world, it doesnt always has to be like that, and the BFS algorithm doesn't need to care about this. We would like to de-couple the BFS algorithm from our project's internal logic. One way to deal with it is to use a 3rd class to mediate between them.
- You can assume that the grid won't be bigger than 10 on 10.
- You can assume the input will be valid - no 0 size axis, no negatives numbers etc...
- (0,0) is in the buttom, leftmost corner. The x and y axes are as always
- The order in which the neighbors should be extracted from every point is clockwise, starting from 9 oclock (9, 12, 3, 6).

Part III

C++ - comparing it to JAVA

or: "Why the hell should I use C++?"

In the third part - and last one - you will implement two simple applications, once in C++ and once in JAVA. As you will talk in the lecture, there are some critical differences between C++ and JAVA (or, in general, between languages which mange the memory themselves). And just for this reason, you will implement two identical programs (which differ only by the language), two major differences between the languages.

Speed & Memory.

Implementation

Write a program which receives by the program arguments a size (you can assume it is a power of two - 2^n). This number represents the number of leaves of the tree. This tree, is a binary, complete tree. What the program does is, travels along the tree in a DFS way. On all the tree!

The objective is to check in what numbers, you start to see a difference between the two programs.

Notes

- The two programs should be as identical as possible
- When you run a java program, it's allocated with a default memory size. You can change this size. This is in contrast to the memory a C++ program can allocate. For changing the ammount of size in your java program, use the flag: "-Xmx8G". 8 stands for the ammount of memory you assign to the program. So, for example:

```
java -Xmx8G -jar your_program.jar 16
```

- The way you implement, doesn't really matter. Of course it should be elegant and efficient as possible, but if both of the programs are the same, and you see the performence differences (which is the main purpose of this part), it is ok. So for example, if you use the *Point* class from the second part, or simple do some recursive call, doesn't matter for this part of the exercise.

Submission:

The exercise is splitted into 3 parts, and the third part also contain 2 parts. Each part has a dedicated 'exercise' name in the submit system. So zip each part separately and submit accordingly. Each part which contains a makefile, should be added into the top folder hierarchy, and the rest however your files are aranged.

1. A PDF file, containing all classes with their corresponding methods and members using UML format `en.wikipedia.org/wiki/Unified_Modeling_Language`. If you are not familiar with UML yet, here is a good tutorial: `www.tutorialspoint.com/uml/`. The submission **has** to be in UML format, by any online tool you wish to use and not by hand writting. There are many online tools for doing so, like: `www.draw.io/` and: `www.gliffy.com/`. (Of course you can chose any tool you wish and is best for you. You can also use any other tool for this, as ppt, word etc. or their equivalent software in other operating systems).
2. In the second part you need to submit all of the relevant code. It must compile on the planet (or u2). Aside of the code include a makefile which compiles the code to a single executable file name 'a.out' (the default). There are a few automatic tests already online, so you will be notified by mail if you passed the automatic test.
3. The third part also needs to be compiled via makefile. Submit all of the code in a zip with the compiling makefile (should generate a.out file). The java code will be compiled automatically, so no makefile is needed here. For this part you'll need all the source files and a manifest file named: manifest.txt in which you'll mention the main class name. For example:

   ```
   // manifest.txt
   Main-Class: Main
   ```

   (The main class name is 'Main')
   Alongside the source code, submit a text file, and mention 2 statistics:

(a) The number when you first started to notice difference in performence.

(b) The number where the java couldn't take it anymore (over one minute runtime).

Beside each size, write down also the time which took for each program to run (if not failing). There will be a time limit of one minute, so for your purposes, over one minute of running means that it didn't finish.

Besides that, please follow the general submission notes file. There are some more guidelines regarding the submission (like adding a file with your personal information - id and name).

Notes

- Don't be late.
- Submit your work to the submit system, with your personal user account.
- For the first part, do not write any code. We mainly focus on planning and designing.
- Please specify inside the UML the state of every method and member (public, protected and private).
- There will be changes in the characterization of the system and there will be more requirements in the following exercises. Think of what could change, and what could be added, and plan your work accordingly, using all methods and design pattern you already know.
- If you use the Clion IDE, you will need to use CMake as the building environments. Although it is a very handy building SW, it is more compilcated than makefile - which you're already familiar with. You're free to decide which compilation tool to use (on your developing process. The submit receives only makefiles).
- The better the work you do in this exercise, the less work you'll have in two exercises from this one (the implementation exercise). It will pay off!