**gHW#3 - Flight Simulator Interpreter**

This homework you will do in pairs.
*** We strongly advise everybody to work on github.

**Deadline:**
6.1.19 midnight through the submit.
Submit until 9.1 midnight with 5 points off.
Submit until 10.1 with 10 points off.
Anybody else with **special** requests can email roi and me for a request.

## Resources:

Git: https://www.youtube.com/watch?v=SWYqp7iY_Tc
Video on communication between windows and your virtual machine with the simulator:
https://drive.google.com/file/d/1Hn2pse_LFLGliL5lF6xQm9vHV6xEAYSg/view
(Thanks to the TA Michael from last year)
http://wiki.flightgear.org/New_to_FlightGear
http://wiki.flightgear.org/Autostart

**Installations**: Please read flightgear installations carefully! It explicitly says how to download on which OS and where the files exist once you download onto your machine.
**newest mac users with Catalina, you may have problems with the simulator. I suggest finding a parter without the newest version of Mac OS.
https://www.flightgear.org/download/main-program/
**Those in dualboot on ubuntu, you can install flightgear from the app store (software center) that ubuntu has.**

## Clarifications for the assignment (Please read the assignment pdf carefully! Before reading this doc).

** I've uploaded a new script that changes the rpm and alt values. It works on my end with a solution, so I strongly recommend to try running this script with your homework to catch any mistakes.
Grade break down has been added at the end of the doc.

** keep checking if the current files have been updated (fly.txt)
** note the fly.txt is different than the code provided in the pdf. We believe there is some bug there so better use the fly.txt. If you have the time and curious why those lines of commands don't work, let us know and good for you!

In general, the simulator is very iffy, so let us know if you see some commands that don't work or help, or if you have a better txt files for testing.

More clarifications from Piazza will be here.

## Piazza questions:

- I uploaded a "fake" simulator. Try to run it along with a dummy server and client to see how the connections and values should look like.
- Function command is no longer necessary, but is a bonus of 5 points (check out the .txt example).
- There will be no condition (while or if) command inside another condition command.
- All the commands you've seen in the .txt files (+ 'if') are the only commands you'll see.
- There will be no new "vars" declared inside scopes. (there is a "var x" that a function accepts).
- Xml is updated, it now has all variables we use in the .txt file, so if you use the python script you can see all commands updated.

## A. General clarifications

1. Please work with git, its a great tool for version control and for working in teams.
2. Versions **c++14** and below are allowed for the project.
3. In main, you will expect to receive a file name as a parameter.
    a. ./a.out file_name
    b. If a file name is not provided print an error message.
4. Generic_small represents a file with all the possible values that the simulator has access to.
5. The commands and values in the file may need to be changed in order to make the simulator stable and fly. It depends on which version you are running, etc, so play around with the values.
    a. When testing your code we imitate the simulator as a server, so we don't care about stabilizing anything or making the simulator fly.
6. **The text file we provide you with commands**:
    a. You can assume that all syntax in the txt file is correct.
        i. Of course its always good practice to check.
    b. There is no limit on how long a single line command can be
    c. Each separate command will be in its own line, meaning a new command will always start on a new line.
    d. There will be no numbering in the text file as shown in the PDF
    e. We will always declare a variable with "var" before using a variable.
        i. Meaning you will never see an uninitialized variable
    f. You can assume we **will not** initialize a variable twice.
        i. Both d & e are easy to check for.

g. There doesn't have to be a space between tokens:
    i. var x = 2 + 3 //okay
    ii. var x=2+3 //okay
h. There will always be a space after the "var".
i. "->" and "<-" will always be at the same line of a "var" declaration.
    i. But "var" can be without the arrows, since I can also declare a new var to hold some value.
j. If you see " " (quotes), they will always refer to a string inside sim( ) or inside print( ).
k. Anything else (without " ") will be either a variable, or a number.
l. Connect client command will always receive an IP in the correct form, and will always be: "127.0.0.1"
m. Variable names can only be the usual convention, as was used in HW1.

## 7. Expressions

a. As opposed to HW1, here anything that is ***not*** inside quotes (""), could be an expression.
b. You already know how to handle expressions, try to think of a way to incorporate them when parsing parameters and commands.
    i. There's multiple ways of doing it, it could be a generic function that you pass to it what you parse
    ii. You can also try to incorporate expression class into the command hierarchy.
    iii. And more..
c. Notice the condition expression inside a "while" or an "if" statement is a **BooleanType** expression.
    i. You can add this class to the Expression hierarchy as a new binary operator.
        1. You will have to accomodate the following operators: <,>,<=,>=,==,!=

## 8. Commands

a. You ***do not*** need to handle a "for" loop.
b. A function will only have one parameter (as seen in Eli's example), and its scope will have access to other variables declared previously with "Var".
    i. This will be a bonus feature of 5 points. You don't have to handle it, but if you do (and you should, its a gift), we will test exactly how it looks like in the example .txt I provided.
c. There will only ever be **1 condition** inside an "if" or "while" loop. (example: if (a == b), and ***Not*** if (a == b && b == c)).
    i. There will not be other 'if' or 'while' inside a condition command.
d. Commands will appear as they do in the pdf, meaning **case-sensitive**.
e. A block scope will be characterized by '{'  ' }'.
    i. They will never appear on the same line, since each command has its own line.
    ii. A condition and '{' will always appear on the same line as the commands 'while' and 'if'.
    iii. If a '{' appears, there will always be a '}'.

f. The ‘<-’ and ‘->’ commands are **not transitive**. Meaning if var b -> “abc”, and we do var c = b, it doesn’t Not mean that now c -> “abc”.

g. Both the Open server command and Client command are **blocking calls**. Meaning you must create a thread when creating the server, and wait until you accept the simulator as client. Once you know you have a connection, you can continue to listen to the server on this thread and continue parsing the next line, connect client command. You must also create a thread for connect client command, and wait on the thread until you make a successful connection with the simulator. Once both of those lines have finished, you may continue execution and go on and let the threads run in the background.
   i. All threads use **TCP protocol**.

h. ** important, the simulator server is of type “telnet”. The telnet server accepts information/commands and expects a “carriage return”, meaning a new line to end the command.
   i. For every message you send the simulator, it must end with: "\r\n”
   ii. For example:

```
string message = "set " + path+ " " + to_string(value) + "\r\n";
ssize_t return_val;
// Send message to the server
returl_val = write(sockfd, message.c_str(), message.length());
```

## 9. General Suggestions

a. **Start with understanding the simulator and the environment. (Take a good amount of hours to really understand what’s required, and even try to draw out the classes, understanding who talks to whom)**.

b. Then maybe run the --generic command in the simulator, and write a simple server.cpp that listens on the IP&port that we indicated in the OpenServer command. See what type of information the simulator sends (should be flight data corresponding to generic_small.xml file).

c. Students working with VM, it is strongly suggested to run simulator through windows so your machine doesn’t freeze/crash. (watch the video in the resources).

d. I strongly suggest all students in the class to come up with more .txt files containing commands in order to test your code.

e. At the end of the day, the simulator is just a server & a client. The way we will test your code is by simulating the simulator (ha!), meaning we will create a server that is waiting for commands from your server, and we will send flight information based on the generic_small.xml file using a client server. **I would suggest someone whos feeling generous to implement these two generic sockets to test your code with, so you don’t have to rely on the simulator.**

f. **Make sure you close/delete all allocated resources for the program (files/sockets/memory).

B. **If connection between simulator and your server doesn't work, make sure the following is correct:**

1. The simulator needs to have the following settings written before it launches:

   ```
   --generic=socket,out,10,127.0.0.1,5400,tcp,generic_small
   --telnet=socket,in,10,127.0.0.1,5402,tcp
   ```
2. The file "generic_small.xml" needs to be in the **protocol** folder under **data** of the simulator code (wherever you downloaded it to).
3. Your servers are running on localhost.

## Submission

You will submit all of your code in a zip file.

We will compile with the usual command, except with:

```
g++ -std=c++14 *.cpp -Wall -Wextra -Wshadow -Wnon-virtual-dtor -pedantic -o a.out
-pthread
```

1. You must include a **details.txt** file in your zip with the following format:

   Daniela

   12344

   Roi

   2134235

   - **Failure to include a details.txt with all the required information will lose you 5 points.**

2. README

   a. This file will describe how to compile your code, how to run it, where do files need to reside, everything that someone needs to know when wanting to run your code. (google how to write a README, theres no specific format we want, just for it to be descriptive).

   b. Put it in your github.

   c. Add a link to your github (bonus 5 points)

3. Add well written comments that document your code (5 points)

We will test your code with similar .txt files that we provided you with. The rest of the code is up to you, meaning you **must also provide** a main.cpp.
** we strongly suggest testing your code with the fake_server.py script.

Grade break down:

Providing a README - 5 points (https://www.makeareadme.com/)

Well written documentation comments in code - 10 points

Able to accept a client - 10 points

Able to connect to a server - 10 points
Able to handle expressions - 10 points
Handle command print - 5 points
Handle command sleep - 5 points
Handle while loop condition - 10 points
Handle if condition - 10 points
Able to update and send variable values - 10 points
**Bonus** - link to github, (should have a readme) - 5 points
**Bonus** - able to handle func command - 5 points