CS 171 Computer Programming 1

Winter 2017

Assignment 5: Files, Math, and Loops

Purpose:

After completing this assignment you will have designed, implemented, debugged, compiled and provided documentation for a C++ program that involves loop, streams and math.

The Assignment:

This assignment consists of:

1. An Introduction
2. Make some updates to an old program (Lab 9) to get it ready for the new version! (Problems 1-2)
3. Creating four functions that you will use in your final program (Problems 3-6).
4. A final program that uses the previously created and tested problems (Problem 7).
5. A document showing your analysis of the results (Problem 8).

In addition to the problem specifications, this document contains:

- *What to Submit*
- *Grading Rubric*
- *Academic Honesty*

Make sure you read everything!

Introduction

In this assignment you will develop a lunar lander program that is more substantial than the one you did in lab 9.

The game of Lunar Lander is based on a variety of similarly themed games inspired by the Apollo 11 space mission of July, 1969 in which US astronauts first landed on the Moon.

In our variant, the player is trying to land a spacecraft on the moon safely (as close to 0 ft/sec as possible) **when beginning at a given height (1000 ft)** above the lunar surface **and a given velocity (50 ft/S).** At 1 second intervals, the player is permitted to burn fuel which changes the speed of descent of the spacecraft. **The amount of available fuel is fixed (150 lbs).**

You are to write a program that engages the user in a session that plays the game, updating the values of height and velocity at each turn and stopping when the spacecraft reaches the surface (aka *touchdown*). At this point, a final analysis is printed.

Your main program should do the following:

1. Output instructions for the user
2. Repeatedly
    a. Output current information which includes measurements of height (distance above the lunar surface, in feet), velocity (speed of descent toward the lunar surface, in feet/second), elapsed time (one second expires on each iteration), and remaining fuel.
    b. If there is more fuel to burn prompt the user for a number of units of fuel to burn which cannot be more than 30 units or less than 0 units at any time. If the user requests to burn more fuel than available, just burn what's left. If there is no fuel to burn display an appropriate message (see sample run).
3. This interaction continues as long as the craft is above the lunar surface
4. Once the craft intersects the lunar surface, final computations are made, final touchdown information is output, as is a message based on the final velocity.

On the assignment page there is a sample session (LunarLander-sample.txt), which you can use to determine if your calculations and program logic are working correctly. The instructions, final analysis, calculations and prompts are all based on the original game by Dave Ahl, and transcribed by Joe Morrison.

In addition to subtle difference, three critical change to your Lab 9 implementation include:

1. Breaking the program into subprograms via creation and usage of functions.
2. The ability to read in and write out to generic `istream` and ostream object, respectively, so that I/O can be done via either the console or a file (this will be used in the last part).
3. Correct computations on when the craft hits the surface. In your Lab 4 go at this you may have noticed that when the craft eventually reaches the surface (touchdown), the height may be a negative number, indicating how far the craft would have gone had the surface not been there. That is, it may reach the surface in 25 seconds and stop, but the calculations tell us only how far the craft would have gone in a full second at its current velocity. At a later part of this assignment you will apply an equation that finds the velocity at the time of the actual touchdown.

Problem 1

Take your solution from the second half of Lab 9 (the Lunar Lander one) and make the following basic changes:

1. Fix the initial height (altitude) to be 1000 feet, the initial velocity to be 50 feet/sec and the initial fuel remaining to be 150 units. These values will no longer be input by the user.
2. Set acceleration due to gravity to be 5 feet/s^2
3. In your main loop only prompt the user to enter the amount of fuel to burn if there is fuel left. If there is not output "**** OUT OF FUEL ****". If there is more fuel and the user tries to burn more than available, then only burn what's left.

Problem 2

Next we want to move some of the code into functions. The first function we'll make is prototyped as

```
void reportStatus(ostream &os, double time, double height,
  double velocity, double fuel, string name)
```

In addition to the current `time`, `height`, `velocity` and remaining `fuel`, this takes a reference to an output stream `os` as a parameter so that we can choose to either output to the command line or to an output file.

This function should output information as follows (and as seen in the sample session) where `<name>` is replaced with whatever is passed in via the `name` variable. In this sample session variable `name=APOLLO`:

```
Status of your <name> spacecraft:
Time   : 0 seconds
Height: 1000 feet
Speed  : 50 feet/second
Fuel   : 150
```

Problem 3

Next we'll move the code for updating the height, velocity, and fuel into its own function.

Write a function prototyped as:

```
void updateStatus(double &velocity, double burnAmount, double
     &fuelRemaining, double &height)
```

that uses the current values of `velocity`, `burnAmount`, `fuelRemaining` and `height` to compute the values one second later. That is, if the lander is currently a distance of `height` above the surface, falling at speed of velocity feet/sec, with `fuelRemaining` units of fuel remaining and chooses to burn `burnAmount` units of fuel, what will the new `height`, `velocity`, and amount of fuel remaining be after one second, assuming the lander is falling freely above the surface and has not yet made contact?

The following are the details on how these are computed:

- The amount of fuel is simply the previous amount minus the amount burned.
- For each time step, the `velocity` is increased by `5` (to account for acceleration due to gravity) and decreased by the burn amount (see the example below). Note that this implies that positive `velocity` is in the downward direction. Also note that this implies that with a burn amount of 5 units, the net effect on the velocity is 0.
- The `height` is the old `height` minus the **average** of the old and new velocities (*this is different than what you did before*). This computation of the `height` is a closer approximation than we used in the similar lab exercise to the integration that simulates the behavior of the system governed by a differential equation.

For example, if the craft begins falling at a rate of 50 feet/second from a height of 1000 feet, with 150 units of fuel, and the user chooses to burn 8 units of fuel, there will now be 142 units of fuel remaining, the new velocity will be 47 feet/second, and the new height will be 1000 - (50 + 47)/2 = 951.5.

Problem 4

Next we want to add the ability to get in instructions via some input stream `is`, and output those instructions to some output stream `os`, while replacing all instances of some string `target` from the input stream with the string specified by the `replacement` parameter for the output stream.

The following is the prototype for this function.

```
void introduction(istream &is, ostream &os, string target,
     string replacement)
```

For your testing, to create a set of instructions like those shown in the sample run, you can use the file input.txt which is available on the assignment page and set the target to "`$SPACECRAFT`" and the replacement to "APOLLO".

Towards the top of you `main()` you should first ask the user if they want to see the instructions. If they reply with 'Y' or 'y' output these instructions by calling this function. Otherwise don't output them.

Problem 5

The Lunar Lander program needs to do some corrections at the point of touchdown in order to get accurate final values of height and velocity (from the Introduction this was pointed out as an issue with our first version in lab). It is possible, but very unlikely, that the lunar lander makes impact with the surface precisely one second after the last burn. It is highly likely that the calculations provided by the `updateStatus` procedure will show that height is a negative value (i.e., below the surface) at some point - which means these values need to be readjusted in order to figure out the correct velocity at the time of impact.

As described in the Introduction, the lander may be just above the surface after 25 seconds and then we discover a full second later that it must have hit the surface somewhere between 25 and 26 seconds. Therefore, once we discover this (that we had hit the surface), a recalculation should be done to determine the exact time of landing (as a fraction of a second) and the exact velocity at the exact time of impact.

Your goal is to write a function that determines the final/exact values of the parameters once we detected that impact must have happened. Here's the prototype:

```
void touchdown(double &elapsedTime, double &velocity, double
     &burnAmount, double &height)
```

The computations to update/correct the elapsed time, height, and velocity can be done as follows:

- First, you need to "undo" the last update to determine the *velocity*, *height* and *time* prior to hitting the ground. These calculations are fairly easily done (we'll leave you to figure this out from the equations in `updateStatus`). Basically now you'll have something like *oldtime, oldvelocity,* and *oldheight* for use in the next two steps.
- Next, compute Δ, which is the fraction of a second needed to reach the surface. Do this with the following formula:

$$\Delta = \frac{\sqrt{v^2 + h(10 - 2b)} - v}{5 - b}$$

  where `b` is the burn amount and `v` and `h` are the old (pre-crash) velocity and height. (If $b=5$, the value is simply h/v.)
- Finally, compute

$$elapsedTime = (elapsedTime - 1) + \Delta,$$
$$velocity = v + (5 - b)\Delta$$
$$height = 0$$

- For our purposes you don't need to update/fix the burn rate (who cares!)

Problem 6

The last piece of the puzzle is to write a function that reports the damage based on the final velocity.

This function will be prototyped as:

```
void finalAnalysis(ostream &os, double velocity)
```

The job of this function is to assess the damage to the lunar landing module based on its velocity at the time of impact. You should output a descriptive message to the output stream passed as an argument. The damage values are shown in the following table (from Lunar Lander - Eleven source code by Joe Morrison). With the exception of a perfect landing, all boundary values should be reported with the more severe damage message.

Velocity (ft/sec)	Analysis
0	*Congratulations! A perfect landing!!* *Your license will be renewed...later.*
0–2	*A little bumpy.*
2–5	*You blew it!!!!!!* *Your family will be notified...by post.*
5–10	*Your ship is a heap of junk !!!!!* *Your family will be notified...by post.*
10–30	*You blasted a huge crater !!!!!* *Your family will be notified...by post.*
30–50	*Your ship is a wreck !!!!!* *Your family will be notified...by post.*
>=50	*You totaled an entire mountain !!!!!* *Your family will be notified...by post.*

Problem 7

Ok. All the pieces should be in place. Now time to test them within your own main. From the introduction, the basic program flow should be something like:

1. Ask the user if they want to see the instructions and reads them in from a file and outputs them if they request to see them.
2. Repeatedly
    a. Output current information which includes measurements of height (distance above the lunar surface, in feet), velocity (speed of descent toward the lunar surface, in feet/second), elapsed time (one second expires on each iteration), and remaining fuel.
    b. If there is more fuel to burn prompt the user for a number of units of fuel to burn which cannot be more than 30 units or less than 0 units. If the user requests to burn more fuel than available, just burn what's left. If there is no fuel to burn output an appropriate message.
    c. Update the status.
    d. If we hit the surface, correct the final touchdown values, print out the "Contact" information (see the sample run file), print out the final message, and quit.

Again, use the sample run provided on the assignment page to ensure you have done everything correctly.

Problem 8

So now we want to take advantage of all those functions we designed to have generic `ostream` parameters.

To create a log of the activity, in your `main()` should first ask the user for a file name to log their session to, read in the file, and open it in write mode.

With the exception of your prompt about which file to log to, every time you want to display something, first output it to the command line, then output it to the file stream. Hint: In several instances this may involve calling the same function twice, but passing different `ostream` objects to it.

In order to completely log our session, every time you get input make sure you also output the input you got to the file (again, with the exception of the user entering the file name to log to).

Finally when your session is over, close the file.

Submission

All homework for this course must be submitted electronically using Blackboard Learn. ***Do not e-mail your assignment to a TA or Instructor!*** If you are having difficulty with your Blackboard Learn account, ***you*** are responsible for resolving these problems with a TA, an Instructor, or someone from IRT, before the assignment it due. It is suggested you complete your work early so that a TA can help you if you have difficulty with this process.

For this assignment, you must submit:

Problems 1-6: NOTHING (kind-of)!!

- The work you do here will be in your final program.

Problem 7/8:

- Your C++ source code file that contains all the functions of Programs 1-8. *Just submit your ".cpp" file, NOT your .dsp, .dsw, or similar IDE project files.*
- A PDF document with your User Manual
- A PDF document with your System Manual

Problem 8:

- Session logs for at least 3 runs of your program, submitted as .txt files (so when you respond to the question about what file you want to log to, it would be easiest to say something like: test1.txt)

Grading Rubric:

Note there are 115 points here, so you can get 15 "bonus points"

1. Problem 1 None 10pts
2. Problem 2 None 10pts
3. Problem 3 10pts
4. Problem 4 10pts
5. Problem 5 10pts
6. Problem 6 10pts
7. Problem 7 30pts
8. Problem 8 (all or none) 10pts

In addition, for Problem 7/8 (which includes Problems 1-6)

1. Coding style: consistent adherence to coding standards described in assignment 2 (the previous assignment). Through use of variable names, indentation, and comments it makes itself easy to comprehend. Those who want to adopt variations need justify their divergences and must provide in your written comments a justification that is persuasive and convincing to the grader. (You may wish to talk to the staff about this rather than assuming that you know what they like.) Coding styles reflect what code readers want to read, not just what programmers want to write.

    - 5 points = excellent.
    - 4 points = good but a few improvements are in order
    - 2 points = many improvements needed for code to be readable for a general audience of programmers, or the formatting of the code in the .pdf or .cpp makes it unpleasant and difficult to read (grader's opinion is what matters here - if you have any doubts, you should ask before submitting.).
    - 1 point = missing major principles, should talk to staff about this to do better next time. This is the maximum number of points you can get in this category regardless of style quality if your program does effectively print out the table it was designed to do.

2. Program construction

    - 5 points = excellent. Code is not needlessly complex. Functionality is encapsulate d in the mandated functions and used in that fashion in the program submitted. The program is written in a way that makes extension or changes easy to do. Global constants are used as mandated. Computations should not be needlessly complex, use math to simplify calculational expressions or logic to simplify if statements or complicated Boolean expressions. The grader's judgment is the relevant measure here.
    - 4 points = good, but a few improvements are in order.
    - 2 points = many improvements needed for code to come up to expectation for coding quality.
    - 1 point = many flaws, should talk to staff about this. This is the maximum number of points you can get in this category if your program does effectively print out the table it was designed to do.

3. External Documentation

    This includes content in the User Manual, System Manual as PDFs

    - 5 points = function definitions are commented with full explanation of purpose, parameters, results, and side effects, in the style of section 4.3 of the textbook "Function Comments".
    - 4 points = good, but some significant missing information that would ordinarily be expected
    - 2 points = needs significant improvement. Major miss.
    - 1 point = see the instructional staff to do better next time.

Academic Honesty

You must compose all written material yourself, including answers to book questions. All material taken from outside sources must be appropriately cited. If you need assistance with this aspect of the assignment, see a consultant during consulting hours.

To ensure that assignments are done independently, in addition to human observation, we will be running all assignments through a plagiarism detection system. This program uses compiler techniques which are invariant of syntax and style. It has a very high accuracy rate.