CPE 101
Fall 2009
Project 3

Due Date

- **Friday, October 23 by 11:59pm– remember, no late projects accepted!**

Objectives

- More practice making decisions in C.
- To practice writing loops in C
- To experience some of the issues and subtleties involved with solving problems that include repetitive tasks.
- To learn how to write functions and call functions that you have written.

Ground Rules

- Your program must not use any global data.
- You may only use standard library functions printf and scanf found in stdio.h.
- Your program must implement the required functions as specified and no others.
- Your program must use the required functions appropriately.
- Your program must mimic the sample program's behavior ***exactly*** and in all cases.

Orientation

You will be writing a program that simulates landing the Lunar Module (LM) from the Apollo space program on the moon. The simulation picks up when the retrorockets cutoff and the pilot/astronaut takes over control of the LM. The user of the simulator specifies the initial amount of fuel and initial altitude. The LM starts at a user-specified altitude with an initial velocity of zero meters per second and has a user-specified amount of fuel on board. The manual thrusters are off and the LM is in free-fall – meaning that lunar gravity is causing the LM to accelerating toward the surface according to the gravitational constant for the moon. The pilot can control the rate of decent using the thrusters of the LM. The thrusters are controlled by entering integer values between 0 and 9 which represent the rate of fuel flow. A value of 5 maintains the current velocity, 0 means free-fall, 9 means maximum thrust – all other values are a variation of those possibilities and can be calculated with a provided equation. To make things interesting (and to reflect reality) the LM has a limited amount of fuel – if you run out of fuel before touching down you have lost control of the LM and it freefalls to the surface. The goal is to land the LM on the surface with a velocity between 0 and -1 meters per second, inclusive, using the least amount of fuel possible. A landing velocity between -1 meters per second and -10 meters per second (exclusive) means you survive but the LM is damaged and will not be able to leave the surface of the moon – you will be able to enjoy the surface of the moon as long as your oxygen lasts but you will not leave the moon alive. Velocities faster than (less) than or equal to -10 meters per second mean the LM sustains severe damage and you die immediately.

The initial conditions provided below where chosen so that an optimal landing should use approximately 300 liters of fuel.

Note that you will be provided with a sample executable to play with at a later date. Please see class emails for announcement of the sample executables location.

FYI: This program is modeled after a version that was popular on HP-28S programmable calculators in the 1970s. It is interesting to note that the desktop computers you are working on cost about the same amount (unadjusted for inflation) as the calculators did but are quite a bit more powerful (more memory, storage, computational power, not to mention more sophisticated display, keyboard, mouse, et cetera). This program took approximately 90 seconds to load on the calculator – virtually instantaneous on the machines you are running on today. Credits to Dick Nungester of HP for the provided equations.

Section 0: Develop Incrementally…

There are many reasonable ways to approach developing this program. Immediately beginning to write code is not a recommended approach! The first and most important thing to do is to understand what the program is required to do. To do so, run the sample program, read the entire specification, discuss the project with your instructor, as necessary, until you feel you understand what is required of the program. You may want to write some pseudo-code for parts of this program before you begin to write actual C code. Once you have done this much, the writing of the code should be almost trivial (well, eventually it will be!).

When you do begin writing code, you should decide on some incremental steps you can code and test. There are many reasonable ways to approach this problem incrementally. One suggestion is to implement the required data types and functions specified below before you write the main simulation-loop in main. Try writing each function and testing it individually – this is called ***unit testing***. Once you are confident that each function is working correctly you should be in a good position to tackle the simulation-loop. Even while implementing the simulation loop you should write code incrementally. The size of the incremental steps with vary with your experience but developing this way should save time and make the process more pleasant.

Section 1: Required Values and Equations…

You will need variables to represent the following values:

1. The ***gravitational constant*** for the moon 1.62.

2. The ***altitude*** of the Lunar Module (LM) is expressed in meters, its initial value is provided by the user, and its type must be ***double***.

3. The ***velocity*** of the LM is expressed in meters per second, its initial value is zero, and its type must be ***double*** (note: values greater than zero mean you are moving away from the lunar surface).

4. The ***fuel level*** of the LM expressed in liters, its initial value is provided by the user, and its type must be ***int***.

5. The ***rate of fuel consumption*** of the LM expressed in liters per second, its initial value is zero, and its type must be ***int***.

6. The ***acceleration*** of the LM in meters per second^2, its initial value is zero, and its type must be ***double*** (note: values greater than zero means slowing decent or accelerating away from the lunar surface).

7. The ***elapsed time*** in seconds, initial value is zero, and its type must be ***int*** (note: it measures the time LM has been under manual control).

Decide on ***meaningful*** names that ***meet your instructor's style guidelines*** for all of the variables described above.

The program assumes a time interval of one second between each display of the LM's status and request for throttle input from the user. To calculate new values for a new time period you need to use the old values from the old time period ***with the exception of acceleration***. Be sure you examine the following equations carefully and notice that you ***must calculate the acceleration before you calculate altitude and velocity***.

CAUTION: Pay attention to the data types involved in these equations and be sure you are not losing any necessary precision due to C's rules for dealing with mixed-type expressions.

1. $\text{acceleration}_{\text{time-period 1}} = \text{gravitational constant} * ((\text{rate of fuel consumption}_{\text{time-period 0}} / 5) - 1)$

2. $\text{altitude}_{\text{time-period 1}} = \text{altitude}_{\text{time-period 0}} + \text{velocity}_{\text{time-period 0}} + (\text{acceleration}_{\text{time-period 1}} / 2)$

3. $\text{velocity}_{\text{time-period 1}} = \text{velocity}_{\text{time-period 0}} + \text{acceleration}_{\text{time-period 1}}$

4. $\text{fuel level}_{\text{time-period 1}} = \text{fuel level}_{\text{time-period 0}} - \text{rate of fuel consumption}_{\text{time-period 0}}$

Section 2: Required Function Specifications…

You are required to develop the following functions. Think about what values these functions will need to consume (their input) and their return type (output). Please name them reasonable names.

Implement a function which prompts a user for a fuel value (i.e. a positive integer value). It must display an error message if the user enters a negative or zero value and re-prompt for a valid value. It is not expected to handle non-integer input. See the sample program for the exact text of the prompt and any error message(s).

Implement a function which prompts the user for a real value between 1 and 9999, inclusive which represents the initial altitude of the LM. It must display an error message if the user enters a value outside this range and re-prompt for a valid value. It is not expected to handle non-numeric input. See the sample program for the exact text of the prompt and any error message(s).

Implement a function which will display the state of the lunar lander module. This function should display 1) the elapsed time the LM has been flown; 2) the LM's altitude; 3) the LM's velocity; 4) the amount of fuel on the LM; 5) the current rate of fuel usage. See the sample program for the exact text and format of the display.

Implement a function which gets the current fuel rate from the user. This function should take in a parameter representing the current amount of fuel in the LM. The function prompts the user for an integer value and makes sure it is between 0 and 9, inclusive. The function returns the lesser of the user-entered value or the amount of fuel remaining on the LM (value passed in as a parameter). See the sample program for the exact text and formatting of the prompt and any error message(s).

Implement a function which computes the current acceleration. Think about what parameters (input) information this function will need in order to update the acceleration. This function calculates and returns the new acceleration based on the provided inputs and the equation provided above.

Implement a function which computes the current altitude of the LM. Again, think about what parameters you will need to compute this value. The function calculates and returns the new altitude based on the provided inputs and the equation provided above.

Implement a function called which computes the current velocity. Again, think about what parameters you will need to compute this value. The function calculates and returns the new velocity based on the provided inputs and the equation provided above.

Implement a function to update the current status of the LM's fuel. Again, think about what parameters you will need to compute this value. The function calculates the remaining fuel. Note: This is a trivial function as long as your getFuelRate function behaves correctly.

Implement a function which displays the LM's landing state. Think about what parameters you will need to compute this value. This function, as its name implies, displays the status of the LM upon landing. There are three possible outputs depending of the velocity of the LM at landing, they are:

> Status at landing –The eagle has landed!
> Status at landing – Enjoy your oxygen while it lasts!
> Status at landing – Ouch – that hurt!

Print the first message if the final velocity is between 0 and -1 meters per second, inclusive.
Print the second message if the final velocity is -1 > velocity > -10 meters per second, exclusive
Print the third message if the final velocity is < -10 meters per second, inclusive.

See the sample program for the exact text and formatting.

Section 3: The main function…

There are many advantages to writing functions. These include code reuse and, hopefully, improved readability and maintainability. The purpose of writing the functions described in the previous section is to greatly simplify the code in the main function and, in some cases, take advantage of code reuse. This program's main should contain the primary variable declarations and initialization described in Section 1, and the simulator-loop that advances the LM from time period zero to landing, and calls to the functions specified in Section 2 along with minimal code to "glue" it together. One of the challenges in this project will be understanding the overall structure that has been provided to you. Don't hesitate to ask you instructor questions early in the process to help make sure you understand what you are suppose to be doing. You will probably go down some dead ends and need to remove and/or rewrite code. This is part of the process so enjoy the ride!

Section 4: Testing against input

We highly recommend testing your program using an input file. You can redirect input from standard input (the console) to instead come from a file. You can do this by using the < on vogon. For example:

```
12:01pm vogon ~>a.out < input_file.txt
```

where input_file.txt is the name of a text file that contains the input for your program. Example input files will be provided at a later date.

In addition, you can re-direct output to a file as well, using > on vogon. For example:

```
12:01pm vogon ~>a.out < input_file.txt > output_file.txt
```

where output_file.txt is the name of a text file that contains the output generated by your program. Example output files will be provided at a later date.

Finally, you can use the **diff** command on vogon to display any differences in two files. For example:

```
12:01pm vogon ~>diff file1.txt file2.txt
```

will print out any differences between file1.txt and file2.txt. Once you have written your program, run it with the sample input files and save the output and use the diff command to make sure all your calculations are as expected.

Section 5: Handing in Your Source Electronically…

1. Move the necessary file(s) to your vogon account using your favorite FTP client program.

2. Log on to vogon using your favorite Shell client program.

3. Change directory (cd-command) to the directory containing the file(s) to hand in.

4. Be sure to compile and test your code on vogon using the required compiler flags (-Wall –ansi – pedantic) one last time just before turning the files in.

5. Use the following handin command

    handin zwood csc101p3 moonlander.c

6. You should see messages that indicate handin occurred without error. You can (and should) always verify what has been handed in by executing the following command:

    ```
    handin zwood csc101p3
    ```