LC-3 Basic Input/Output

The LC-3 has the absolute bare minimum of I/O functionality: it can accept a single character input from the keyboard as an ASCII code; and it can output a single character to a plain text console. These two functions are controlled by "BIOS Subroutines" aka "Trap Service Routines" (TSRs) which you can invoke from your code.

Towards the end of the quarter, we will delve into exactly how they work, but for now we just need to know how to use them.

GETC or TRAP x20
: This pauses your program until the user taps a key. The keyboard circuitry generates the corresponding ASCII code, which is captured and stored in R0, where your program can access it.

OUT or TRAP x21
: This outputs the ASCII code currently stored in R0 to the text console as the corresponding character.

PUTS or TRAP x22
: This assumes that R0 *holds the starting address of a c-string* (a null-terminated array of ascii codes). The PUTS routine reads the ascii code stored at the address provided, and displays the corresponding character to the console. It then increments the address in R0, and gets & displays the next ascii code, etc., until the value fetched is 0 *(aka "the null character", or '\0' - i.e. the value x0000)*, at which point the routine terminates.

One of the quickest ways to mess up your program is to use these BIOS routines incorrectly!
So here is a review of what you have already learned in lab - make sure you get it!

First: outputting a single character:

In your data block, you have a location labelled, say, newline, with the pseudo-op `.FILL x0A`
(you may store ascii characters either as a numeric value as above; or as the character itself within single quotes: `.FILL '\n'`*)*
You can load that ascii code into R0 using LD, and output it with the TSR OUT:

```
LD R0, newline    ; LD copies the value stored at the address newline into R0
OUT               ; the TSR OUT writes the ascii code in R0 to console as a character
```

Just remember that the label `newline` *must be within +/- 256 lines of your LD instruction*

Make sure you only attempt to print printable ascii characters! That's [x20, x7E] - look it up.
Any value outside that range will either print nothing, or actually cause OUT to fail *(if so, you will usually see an error message in your simpl Text Window).*

Next: outputting a string *(in the form of a c-string, i.e. a null terminated array of ascii codes)*:

In your data block, you have a location labelled, say, greeting, with the pseudo-op
`.STRINGZ "hello!"` *(note the double quotes!)*
Now, using **LEA**, you can load into R0 the *address corresponding to the label greeting* (i.e. the address of the start of the string, i.e. the address at which 'h', the first character of of the string, is stored).
You can then output the entire string with the TSR PUTS:

```
LEA R0, greeting   ; LEA copies the address corresponding to the label greeting into R0.
                   ; greeting is the starting address of the c-string "hello!"
PUTS               ; The TSR writes to console the entire string starting at greeting
```

If you mess ANY of this up, you will get at best a warning in the simpl Text window; at worst you will crash simpl, and abort the grader.

What can go wrong, you ask?

- If you invoke OUT when you have an *address* in R0, you will be attempting to write something like x301A (an address) to console as though it were an ascii code!
  The OUT BIOS routine will usually abort, with a run-time error message in the Text Window saying that you attempted to output an ascii character "with the high byte set".
- Or If you store an actual number (e.g. #7 = x07) in R0, and then invoke OUT, you will be sending a valid but non-printing ascii code to the console *(look up the ascii table to see what x07 means)*.
  Even though you won't see it when you run simpl, the grader will see it, and mark you as failing that particular test.
- If instead you invoke PUTS when you have an *ascii code* in R0 instead of an address, you will be attempting to read an ascii code from from an address like x0061 - which is protected memory - a guaranteed abort!
- ... and so on.

TL;DR:

- You can store a single character with **.FILL**, followed by *the actual hex value ascii code* like x62, x37, etc., or *a single-quoted character* like 'b' or '7' or '@' or '\n'
  You then **LD** the ascii code directly into R0, and print that single character to console with **OUT**

OR

- You can store a c-string *(null-terminated character array)* with **.STRINGZ**, followed by a double-quoted string like "we're done!\n".
  The label for the string is *an alias for the address of the first character*, i.e. the starting address of the character array.
  You then **LEA** the *starting address* (i.e. the label) of that array into R0, and print the entire string to console with **PUTS**

DO NOT GET THE TWO MODES MIXED UP!